```
Prof. James Bush,
206 Church St. SE,
Minneapolis,
MN 55455

J. E. Dougal,
Rice University,
Houston,
TX 77251

Jack Finch,
104 Ayres Hall,
Knoxville,
TN 37996
```

7.17 Exercises

1. Write suitable type statements for the following quantities:

i) an array to hold the number of counts in each of the 100 bins of a histogram numbered from 1 to 100;

ii) an array to hold the temperature to two significant decimal places at points, on a sheet of iron, equally spaced at 1cm intervals on a rectangular grid 20cm square, with points in each corner (the melting point of iron is 1530° C);

iii) an array to describe the state of 20 on/off switches;

iv) an array to contain the information destined for a printed page of 44 lines each of 70 letters or digits.

2. Explain the difference between the following pair of declarations

```
real :: i = 3.1
```

and

```
real, parameter :: i = 3.1
```

What is the value of `i` in each case?

3. Write type declaration statements which initialize:

i) all the elements of an integer array of length 100 to the value zero.

ii) all the odd elements of the same array to 0 and the even elements to 1.

iii) the elements of a real 10×10 square array to 1.0 .

iv) a character string to the digits 0 to 9.

4. In the following module, identify all the scoping units and list the mappings for implicit typing for all the letters in all of them:

```
module mod
   implicit character(10, 2) (a-b)
   :
contains
   subroutine outer
      implicit none
      :
   contains
      subroutine inner(fun)
         implicit complex (z)
         interface
            function fun(x)
               implicit real (f, x)
               :
            end function fun
         end interface
      end subroutine inner
   end subroutine outer
end module mod
```

5.

i) Write a type declaration statement that declares and initializes a variable of derived type person (Section 2.9).

ii) Either

   a. write a type declaration statement that declares and initializes a variable of type entry (Section 2.13), or

   b. write a type declaration statement for such a variable and a data statement to initialize its non-pointer components.

6. Which of the following are initialization expressions:

i) `kind(x)`, for x of type real

ii) `selected_real_kind(6, 20)`

iii) `1.7**2`

iv) `1.7**2.0`

v) `(1.7, 2.3)**(-2)`

vi) `(/ (7*i, i=1, 10) /)`

vii) `person("Reid", 25*2.0, 22**2)`

viii) `entry(1.7, 1, null_pointer)`

8. Intrinsic procedures

8.1 Introduction

In a language that has a clear orientation towards scientific applications there is an obvious requirement for the most frequently required mathematical functions to be provided as part of the language itself, rather than expecting each user to code them afresh. When provided with the compiler, they are normally coded to be very efficient and will have been well tested over the complete range of values that they accept. It is difficult to compete with the high standard of code provided by the vendors.

The efficiency of the intrinsic procedures when handling arrays is likely to be particularly marked because a single call may cause a large number of individual operations to be performed, during the execution of which advantage may be taken of the specific nature of the hardware.

Another feature of a substantial number of the intrinsic procedures is that they extend the power of the language by providing access to facilities that are not otherwise available in the language. Examples are inquiry functions for the presence of an optional argument, the parts of a floating-point number, and the length of a character string.

There are over a hundred intrinsic procedures in all, a particularly rich set. They fall into distinct groups, which we describe in turn. A list in alphabetical order, with one-line descriptions, is given in Appendix A. Some processors may offer additional intrinsic procedures. Note that a program containing references to such procedures is portable only to other processors that provide those same procedures. In fact, it does not conform to the standard unless the access is through use association.

8.1.1 Keyword calls

The procedures may be called with keyword actual arguments, using the dummy argument names as keywords. This facility is not very useful for those with a single non-optional argument, but is useful for those with several optional arguments. For example

```
call date_and_time (date=d)
```

returns the date in the scalar character variable d. The rules for positional and keyword argument lists were explained in Section 5.13. In this chapter, the dummy arguments that are optional are indicated with square brackets. We have taken some 'poetic licence' with this notation, which might suggest to the reader that the positional form is permitted following an absent argument (this is not the case).

8.1.2 Categories of intrinsic procedures

There are four categories of intrinsic procedures:

i) *Elemental procedures* are specified for scalar arguments, but may also be applied to conforming array arguments. In order that the rank always be known at compile time, at least one of the array arguments must correspond to a non-optional argument of the elemental procedure. In the case of an elemental function, each element of the result, if any, is as would have been obtained by applying the function to corresponding elements of each of the array arguments. In the case of an elemental subroutine with an array argument, each argument of intent `out` or `inout` must be an array, and each element is as would have resulted from applying the subroutine to corresponding elements of each of the array arguments.

ii) *Inquiry functions* return properties of their principal arguments that do not depend on their values; indeed, for variables, their values may be undefined.

iii) *Transformational functions* are functions that are neither elemental nor inquiry; they usually have array arguments and an array result whose elements depend on many of the elements of the arguments.

iv) *Non-elemental subroutines.*

8.1.3 The intrinsic statement

A name may be specified to be that of an intrinsic procedure in an `intrinsic` statement, which has the general form

```
intrinsic [::] intrinsic-name-list    ! :: in Fortran 95 only
```

where *intrinsic-name-list* is a list of intrinsic procedure names. A name must not appear more than once in the `intrinsic` statements of a scoping unit and must not appear in an `external` statement there (but may appear as a generic name on an interface block if an intrinsic procedure is being extended, see Section 5.18). We believe that it is good programming practice to include such a statement in every scoping unit that contains references to intrinsic procedures, because this makes the use clear to the reader. We particularly recommend it when referencing intrinsic procedures that are not defined by the standard, for then a clear diagnostic message should be produced if the program is ported to a processor that does not support the extra intrinsic procedures.

8.1.4 Argument intents

The functions do not change the values of their arguments. In fact, the non-pointer arguments all have the intent `in`. For the subroutines, the intents vary from case to case (see the descriptions given later in the chapter).

8.2 Inquiry functions for any type

The following are inquiry functions whose arguments may be of any type:

`associated (pointer [,target])`, when `target` is absent, returns the value true if the pointer `pointer` is associated with a target and false otherwise. The pointer association status of `pointer` must not be undefined. If `target` is present, it must have the same type, type parameters, and rank as `pointer`. The value is true if `pointer` is associated with `target`, and false otherwise. In the array case, true is returned only if the shapes are identical and corresponding array elements, in array element order, are associated with each other. If the character length or array size is zero, false is returned. A different bound, as in the case of `associated(p,a)` following the pointer assignment `p => a(:)` when `lbound(a) = 0`, is insufficient to cause false to be returned. The argument `target` may itself be a pointer, in which case its target is compared with the target of `pointer`; the pointer association status of `target` must not be undefined and if either `pointer` or `target` is disassociated, the result is false .

`present (a)` may be called in a subprogram that has an optional dummy argument `a` or accesses such a dummy argument from its host. It returns the value true if the corresponding actual argument is present in the current call to it, and false otherwise. If an absent dummy argument is used as an actual argument in a call of another subprogram, it is regarded as also absent in the called subprogram.

There is an inquiry function whose argument may be of any intrinsic type:

`kind (x)` has type default integer and value equal to the kind type parameter value of `x`.

8.3 Elemental numeric functions

There are 17 elemental functions for performing simple numerical tasks, many of which perform type conversions for some or all permitted types of arguments.

8.3.1 Elemental functions that may convert

If kind is present in the following elemental functions, it must be a scalar integer initialization expression and provide a kind type parameter that is supported on the processor.

abs (a) returns the absolute value of an argument of type integer, real, or complex. The result is of type integer if a is of type integer and otherwise it is real. It has the same kind type parameter as a.

aimag (z) returns the imaginary part of the complex value z. The type is real and the kind type parameter is that of z.

aint (a [,kind]) truncates a real value a towards zero to produce a real that is a whole number. The value of the kind type parameter is the value of the argument kind if it is present, or that of a otherwise.

anint (a [,kind]) returns a real whose value is the nearest whole number to the real value a. The value of the kind type parameter is the value of the argument kind, if it is present, or that of a otherwise.

ceiling (a [,kind]) returns the least integer greater than or equal to its real argument. The optional argument kind is available in **Fortran 95 only**. If kind is present, the value of the kind type parameter of the result is the value of kind, otherwise it is that of the default integer type.

cmplx (x [,y][,kind]) converts x or (x, y) to complex type with the value of the kind type parameter being the value of the argument kind if it is present or that of default complex otherwise. If y is absent, x may be of type integer, real, or complex. If y is present, it must be of type integer or real and x must be of type integer or real.

floor (a [,kind]) returns the greatest integer less than or equal to its real argument. The optional argument kind is available in **Fortran 95 only**. If kind is present, the value of the kind type parameter of the result is the value of kind, otherwise it is that of the default integer type.

int (a [,kind]) converts to integer type with the value of the kind type parameter being the value of the argument kind, if it is present, or that of the default integer otherwise. The argument a may be

- integer, in which case int(a)=a,
- real, in which case the value is truncated towards zero, or
- complex, in which case the real part is truncated towards zero.

nint (a [,kind]) returns the integer value that is nearest to the real a. If kind is present, the value of the kind type parameter of the result is the value of kind, otherwise it is that of the default integer type.

`real (a [,kind])` converts to real type with the value of the kind type parameter being that of `kind` if it is present. If `kind` is absent, the kind type parameter is that of default real when `a` is of type integer or real, and is that of `a` when `a` is type complex. The argument `a` may be of type integer, real, or complex. If it is complex, the imaginary part is ignored.

8.3.2 Elemental functions that do not convert

The following are elemental functions whose result is of type and kind type parameter that are those of the first or only argument. For those having more than one argument, all arguments must have the same type and kind type parameter.

`conjg (z)` returns the conjugate of the complex value `z`.

`dim (x, y)` returns `max(x-y, 0.)` for arguments that are both integer or both real.

`max (a1, a2 [,a3,...])` returns the maximum of two or more integer or real values.

`min (a1, a2 [,a3,...])` returns the minimum of two or more integer or real values.

`mod (a, p)` returns the remainder of `a` modulo `p`, that is `a-int(a/p)*p`. If `p=0`, the result is processor dependent. `a` and `p` must be both integer or both real.

`modulo (a, p)` returns `a` modulo `p` when `a` and `p` are both integer or both real, that is `a-floor(a/p)*p` in the real case, and `a-floor(a÷p)*p` in the integer case, where ÷ represents ordinary mathematical division. If `p=0`, the result is processor dependent.

`sign (a, b)` returns the absolute value of `a` times the sign of `b`. The arguments `a` and `b` must be both integer or both real. If `b` is zero, its sign is taken as positive. **In Fortran 95 only**, however, if `b` is real with the value zero and the processor can distiguish between a negative and a positive real zero, the result has the sign of `b` (see also Section 8.7.1).

8.4 Elemental mathematical functions

The following are elemental functions that evaluate elementary mathematical functions. The type and kind type parameter of the result are those of the first argument, which is usually the only argument.

`acos (x)` returns the arc cosine (inverse cosine) function value for real values x such that $|x| \leq 1$, expressed in radians in the range $0 \leq \text{acos}(x) \leq \pi$.

`asin (x)` returns the arc sine (inverse sine) function value for real values x such that $|x| \leq 1$, expressed in radians in the range $-\frac{\pi}{2} \leq \text{asin}(x) \leq \frac{\pi}{2}$.

`atan (x)` returns the arc tangent (inverse tangent) function value for real x, expressed in radians in the range $-\frac{\pi}{2} \leq \text{atan}(x) \leq \frac{\pi}{2}$.

`atan2 (y, x)` returns the arc tangent (inverse tangent) function value for pairs of reals, x and y, of the same type and type parameter. The result is the principal value of the argument of the complex number (x, y), expressed in radians in the range $-\pi < \text{atan2}(y, x) \leq \pi$. The values of x and y must not both be zero.

`cos (x)` returns the cosine function value for an argument of type real or complex that is treated as a value in radians.

`cosh (x)` returns the hyperbolic cosine function value for a real argument x.

`exp (x)` returns the exponential function value for a real or complex argument x.

`log (x)` returns the natural logarithm function for a real or complex argument x. In the real case, x must be positive. In the complex case, x must not be zero, and the imaginary part w of the result lies in the range $-\pi < w \leq \pi$.

`log10 (x)` returns the common (base 10) logarithm of a real argument whose value must be positive.

`sin (x)` returns the sine function value for a real or complex argument that is treated as a value in radians.

`sinh (x)` returns the hyperbolic sine function value for a real argument.

`sqrt (x)` returns the square root function value for a real or complex argument x. If x is real, its value must be not be negative. In the complex case, the real part of the result is not negative, and when it is zero the imaginary part of the result is not negative.

`tan (x)` returns the tangent function value for a real argument that is treated as a value in radians.

`tanh (x)` returns the hyperbolic tangent function value for a real argument.

8.5 Elemental character and logical functions

8.5.1 Character-integer conversions

The following are elemental functions for conversions from a single character to an integer, and vice-versa.

`achar (i)` is of type default character with length one and returns the character in the position in the ASCII collating sequence that is specified by the integer `i`. `i` must be in the range $0 \leq i \leq 127$, otherwise the result is processor dependent.

`char (i[,kind])` is of type character and length one, with a kind type parameter value that of the value of `kind` if present, or default otherwise. It returns the character in position `i` in the processor collating sequence associated with the relevant kind parameter. The value of `i` must be in the range $0 \leq i \leq n-1$, where n is the number of characters in the processor's collating sequence. If `kind` is present, it must be a scalar integer initialization expression and provide a kind type parameter that is supported on the processor.

`iachar (c)` is of type default integer and returns the position in the ASCII collating sequence of the default character `c`. If `c` is not in the sequence, the result is processor dependent.

`ichar (c)` is of type default integer and returns the position of the character `c` in the processor collating sequence associated with the kind parameter of `c`.

8.5.2 Lexical comparison functions

The following elemental functions accept default character strings as arguments, make a lexical comparison based on the ASCII collating sequence, and return a default logical result. If the strings have different lengths, the shorter one is padded on the right with blanks.

`lge (string_a, string_b)` returns the value true if `string_a` follows `string_b` in the ASCII collating sequence or is equal to it, and the value false otherwise.

`lgt (string_a, string_b)` returns the value true if `string_a` follows `string_b` in the ASCII collating sequence, and the value false otherwise.

`lle (string_a, string_b)` returns the value true if `string_b` follows `string_a` in the ASCII collating sequence or is equal to it, and the value false otherwise.

`llt (string_a, string_b)` returns the value true if `string_b` follows `string_a` in the ASCII collating sequence, and false otherwise.

8.5.3 String-handling elemental functions

The following are elemental functions that manipulate strings. The arguments `string`, `substring`, and `set` are always of type character, and where two are present have the same kind type parameter. The kind type parameter value of the result is that of `string`.

`adjustl (string)` adjusts left to return a string of the same length by removing all leading blanks and inserting the same number of trailing blanks.

`adjustr (string)` adjusts right to return a string of the same length by removing all trailing blanks and inserting the same number of leading blanks.

`index (string, substring [,back])` has type default integer and returns the starting position of `substring` as a substring of `string`, or zero if it does not occur as a substring. If `back` is absent or present with value false, the starting position of the first such substring is returned; the value 1 is returned if `substring` has zero length. If `back` is present with value true, the starting position of the last such substring is returned; the value `len(string)+1` is returned if `substring` has zero length.

`len_trim (string)` returns a default integer whose value is the length of `string` without trailing blank characters.

`scan (string, set [,back])` returns a default integer whose value is the position of a character of `string` that is in `set`, or zero if there is no such character. If the logical `back` is absent or present with value false, the position of the leftmost such character is returned. If `back` is present with value true, the position of the rightmost such character is returned.

`verify (string, set [,back])` returns the default integer value 0 if each character in `string` appears in `set`, or the position of a character of `string` that is not in set. If the logical `back` is absent or present with value false, the position of the left-most such character is returned. If `back` is present with value true, the position of the rightmost such character is returned.

8.5.4 Logical conversion

The following elemental function converts from a logical value with one kind type parameter to another.

`logical (l [,kind])` returns a logical value equal to the value of the logical `l`. The value of the kind type parameter of the result is the value of `kind` if it is present or that of default logical otherwise. If `kind` is present, it must be a scalar integer initialization expression and provide a kind type parameter that is supported on the processor.

8.6 Non-elemental string-handling functions

8.6.1 String-handling inquiry function

`len (string)` is an inquiry function that returns a scalar default integer holding the number of characters in `string` if it is scalar, or in an element of `string` if it is array valued. The value of `string` need not be defined.

8.6.2 String-handling transformational functions

There are two functions that cannot be elemental because the length type parameter of the result depends on the value of an argument.

`repeat (string, ncopies)` forms the string consisting of the concatenation of `ncopies` copies of `string`, where `ncopies` is of type integer and its value must not be negative. Both arguments must be scalar.

`trim (string)` returns `string` with all trailing blanks removed. The argument `string` must be scalar.

8.7 Numeric inquiry and manipulation functions

8.7.1 Models for integer and real data

The numeric inquiry and manipulation functions are defined in terms of a model set of integers and a model set of reals for each kind of integer and real data type implemented. For each kind of integer, it is the set

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

where s is ± 1, q is a positive integer, r is an integer exceeding one (usually 2), and each w_k is an integer in the range $0 \leq w_k < r$. For each kind of real, it is the set

$$x = 0$$

and

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

where s is ± 1, p and b are integers exceeding one, e is an integer in a range $e_{\min} \leq e \leq e_{\max}$, and each f_k is an integer in the range $0 \leq f_k < b$ except that f_1 is also nonzero.

Values of the parameters in these models are chosen for the processor so as best to fit the hardware with the proviso that all model numbers are representable. Note that it is quite likely that there are some machine numbers that lie outside the model. For example, many computers represent the integer $-r^q$, and the IEEE standard for Binary Floating-point Arithmetic (IEEE 754-1985 or IEC 60559:1989) contains reals with $f_1 = 0$ (called denormalized numbers) and register numbers with increased precision and range.

In Section 2.6, we noted that the value of a signed zero is regarded as being the same as that of an unsigned zero. However, many processors distinguish at the hardware level between a negative real zero value and a positive real zero value, and the IEEE standard makes use of this where possible. For example, when the exact result of an operation is nonzero but the rounding produces a zero, the sign is retained.

In Fortran 95, the two zeros are still treated identically in all relational operations, as input arguments to all intrinsic functions (except `sign`), or as the scalar expression in the arithmetic `if`-statement (Appendix C.2.1). However, the function `sign` (Section 8.3.2) has been generalized such that the sign of the second

argument may be taken into account even if its value is zero. On a processor that has IEEE arithmetic, the value of `sign(2.0, -0.0)` is -2.0. Also, a Fortran 95 processor is required to represent all negative numbers on output, including zero, with a minus sign.

8.7.2 Numeric inquiry functions

There are nine inquiry functions that return values from the models associated with their arguments. Each has a single argument that may be scalar or array-valued and each returns a scalar result. The value of the argument need not be defined.

`digits (x)`, for real or integer `x`, returns the default integer whose value is the number of significant digits in the model that includes `x`, that is p or q.

`epsilon (x)`, for real `x`, returns a real result with the same type parameter as `x` that is almost negligible compared with the value one in the model that includes `x`, that is b^{1-p}.

`huge (x)`, for real or integer `x`, returns the largest value in the model that includes `x`. It has the type and type parameter of `x`. The value is

$$(1 - b^{-p})b^{e_{max}}$$

or

$$r^{q-1}$$

`maxexponent (x)`, for real `x`, returns the default integer e_{max}, the maximum exponent in the model that includes `x`.

`minexponent (x)`, for real `x`, returns the default integer e_{min}, the minimum exponent in the model that includes `x`.

`precision (x)`, for real or complex `x`, returns a default integer holding the equivalent decimal precision in the model representing real numbers with the same type parameter value as `x`. The value is

$$\texttt{int}((p-1) * \texttt{log10}(b)) + k,$$

where k is 1 if b is an integral power of 10 and 0 otherwise.

`radix (x)`, for real or integer `x`, returns the default integer that is the base in the model that includes `x`, that is b or r.

`range (x)`, for integer, real, or complex `x`, returns a default integer holding the equivalent decimal exponent range in the models representing integer or real numbers with the same type parameter value as `x`. The value is `int(log10(`*huge*`))` for integers and

`int(min(log10(`*huge*`), -log10(`*tiny*`)))`

for reals, where *huge* and *tiny* are the largest and smallest positive numbers in the models.

`tiny (x)`, for real `x`, returns the smallest positive number

$$b^{e_{min}-1}$$

in the model that includes `x`. It has the type and type parameter of `x`.

8.7.3 Elemental functions to manipulate reals

There are seven elemental functions whose first or only argument is of type real and that return values related to the components of the model values associated with the actual value of the argument.

`exponent (x)` returns the default integer whose value is the exponent part e of `x` when represented as a model number. If `x=0`, the result has value zero.

`fraction (x)` returns a real with the same type parameter as `x` whose value is the fractional part of `x` when represented as a model number, that is `x` b^{-e}.

`nearest (x, s)` returns a real with the same type parameter as `x` whose value is the nearest different machine number in the direction given by the sign of the real `s`. The value of `s` must not be zero.

`rrspacing (x)` returns a real with the same type parameter as `x` whose value is the reciprocal of the relative spacing of model numbers near `x`, that is $|\,x\,b^{-e}|b^{p}$.

`scale (x, i)` returns a real with the same type parameter as `x`, whose value is `x` b^{i}, where b is the base in the model for `x`, and `i` is of type integer.

`set_exponent (x, i)` returns a real with the same type parameter as `x`, whose fractional part is the fractional part of the model representation of `x` and whose exponent part is `i`, that is `x` b^{i-e}.

`spacing (x)` returns a real with the same type parameter as `x` whose value is the absolute spacing of model numbers near `x`. It is b^{e-p} if `x` is nonzero and this result is within range; otherwise, it is `tiny(x)`.

8.7.4 Transformational functions for kind values

There are two functions that return the least kind type parameter value that will meet a given numeric requirement. They have scalar arguments and results, so are classified as transformational.

`selected_int_kind (r)` returns the default integer scalar that is the kind type parameter value for an integer data type able to represent all integer values n in the range $-10^r < n < 10^r$, where `r` is a scalar integer. If more than one is available, a kind with least decimal exponent range is chosen (and least kind value if several have least decimal exponent range). If no corresponding kind is available, the result is -1.

`selected_real_kind ([p][, r])` returns the default integer scalar that is the kind type parameter value for a real data type with decimal precision (as returned by the function `precision`) at least `p`, and decimal exponent range (as returned by the function `range`) at least `r`. If more than one is available, a kind with the least decimal precision is chosen (and least kind value if several have least decimal precision). Both `p` and `r` are scalar integers; at least one of them must be present. If no corresponding kind value is available, the result is -1 if sufficient precision is unavailable, -2 if sufficient exponent range is unavailable, and -3 if both are unavailable.

8.8 Bit manipulation procedures

There are eleven procedures for manipulating bits held within integers. They are based on those in the US Military Standard MIL-STD 1753. They differ only in that here they are elemental, where appropriate, whereas the original procedures accepted only scalar arguments.

These intrinsics are based on a model in which an integer holds s bits w_k, $k = 0, 1, \ldots, s-1$, in a sequence from right to left, based on the non-negative value

$$\sum_{k=0}^{s-1} w_k \times 2^k$$

This model is valid only in the context of these intrinsics. It is identical to the model for integers in Section 8.7.1 when r is an integral power of 2 and $w_{s-1} = 0$, but when $w_{s-1} = 1$ the models do not correspond, and the value expressed as an integer may vary from processor to processor.

8.8.1 Inquiry function

`bit_size (i)` returns the number of bits in the model for bits within an integer of the same type parameter as `i`. The result is a scalar integer having the same type parameter as `i`.

8.8.2 Elemental functions

`btest (i, pos)` returns the default logical value true if bit `pos` of the integer `i` has value 1 and false otherwise. `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

`iand (i, j)` returns the logical and of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`iand(i, j)`	1	0	0	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

`ibclr (i, pos)` returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that bit `pos` is cleared to 0. The argument `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

`ibits (i, pos, len)` returns an integer, with the same type parameter as `i`, and value equal to the `len` bits of `i` starting at bit `pos` right adjusted and all other bits zero. The arguments `pos` and `len` must be integers with non-negative values such that `pos+len` $\leq$ `bit_size(i)`.

`ibset (i, pos)` returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that bit `pos` is set to 1. The argument `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

`ieor (i, j)` returns the logical exclusive or of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`ieor(i, j)`	0	1	1	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

`ior (i, j)` returns the logical inclusive or of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`ior(i, j)`	1	1	1	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

`ishft (i, shift)` returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that the bits are shifted `shift` places to the left (-`shift` places to the right if `shift` is negative). Zeros are shifted in from the other end. The argument `shift` must be an integer with value satisfying the inequality |`shift`| $\leq$ `bit_size(i)`.

`ishftc (i, shift [, size])` returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that the `size` rightmost bits (or all the bits if `size` is absent) are shifted circularly `shift` places to the left (`-shift` places to the right if `shift` is negative). The argument `shift` must be an integer with absolute value not exceeding the value of `size` (or `bit_size(i)` if `size` is absent).

`not (i)` returns the logical complement of all the bits in `i`, according to the truth table

i	0	1
not(i)	1	0

8.8.3 Elemental subroutine

`call mvbits (from, frompos, len, to, topos)` copies the sequence of bits in `from` that starts at position `frompos` and has length `len` to `to`, starting at position `topos`. The other bits of to are not altered. The arguments `from`, `frompos`, `len`, and `topos` are all integers with intent `in`, and they must have values that satisfy the inequalities: `frompos+len` $\leq$ `bit_size(from)`, `len` ≥ 0, `frompos` ≥ 0, `topos+len` $\leq$ `bit_size(to)`, and `topos` ≥ 0. The argument `to` is an integer with intent `inout`; it must have the same kind type parameter as `from`. The same variable may be specified for `from` and `to`.

8.9 Transfer function

The transfer function allows data of one type to be transferred to another without the physical representation being altered. This would be useful, for example, in writing a data storage and retrieval system. The system itself could be written for one type, default integer say, and other types handled by transfers to and from that type, for example:

```
integer          :: store
character(len=4) :: word        ! To be stored and retrieved
:
store = transfer(word, store)   ! Before storage
:
word  = transfer(store, word)   ! After retrieval
:
```

`transfer (source, mold [,size] )` returns a result of type and type parameters those of `mold`. When `size` is absent, the result is scalar if `mold` is scalar, and it is of rank one and size just sufficient to hold all of `source` if `mold` is array-valued. When `size` is present, the result is of rank one and size `size`. If the physical representation of the result is as long as or longer than that

of source, the result contains source as its leading part and the rest is undefined; otherwise the result is the leading part of source. As the rank of the result can depend on whether or not size is specified, the corresponding actual argument must not itself be an optional dummy argument.

8.10 Vector and matrix multiplication functions

There are two transformational functions that perform vector and matrix multiplications. They each have two arguments that are both of numeric type (integer, real, or complex) or both of logical type. The result is of the same type and type parameter as for the multiply or and operation between two such scalars. The functions sum and any, used in the definitions, are defined in Section 8.11.1.

dot_product (vector_a, vector_b) requires two arguments each of rank one and the same size. If vector_a is of type integer or type real, it returns sum(vector_a * vector_b); if vector_a is of type complex, it returns sum(conjg(vector_a) * vector_b); and if vector_a is of type logical, it returns any(vector_a .and. vector_b).

matmul (matrix_a, matrix_b) performs matrix multiplication. For numeric arguments, three cases are possible:

i) matrix_a has shape (n, m) and matrix_b has shape (m, k). The result has shape (n, k) and element (i, j) has the value sum(matrix_a(i, :) * matrix_b(:, j)).

ii) matrix_a has shape (m) and matrix_b has shape (m, k). The result has shape (k) and element (j) has the value sum(matrix_a * matrix_b(:, j)).

iii) matrix_a has shape (n,m) and matrix_b has shape (m). The result has shape (n) and element (i) has the value sum(matrix_a(i, :) * matrix_b).

For logical arguments, the shapes are as for numeric arguments and the values are determined by replacing 'sum' and '*' in the above expressions by 'any' and '.and.'.

8.11 Transformational functions that reduce arrays

There are seven transformational functions that perform operations on arrays such as summing their elements.

8.11.1 Single argument case

In their simplest form, these functions have a single array argument and return a scalar result. All except count have a result of the same type and type parameter

as the argument. The mask array `mask`, used as an argument in `any`, `all`, `count`, and optionally in others, is described also in Section 6.17.

`all (mask)` returns the value true if all elements of the logical array `mask` are true or `mask` has size zero, and otherwise returns the value false.

`any (mask)` returns the value true if any of the elements of the logical array `mask` is true, and returns the value false if no elements are true or if `mask` has size zero.

`count (mask)` returns the default integer value that is the number of elements of the logical array `mask` that have the value true.

`maxval (array)` returns the maximum value of an element of an integer or real array. If `array` has size zero, it returns the negative value of largest magnitude supported by the processor.

`minval (array)` returns the minimum value of an element of an integer or real array. If `array` has size zero, it returns the largest positive value supported by the processor.

`product (array)` returns the product of the elements of an integer, real, or complex array. It returns the value one if `array` has size zero.

`sum (array)` returns the sum of the elements of an integer, real, or complex array. It returns the value zero if `array` has size zero.

8.11.2 Optional argument dim

All these functions have an optional second argument `dim` that is a scalar integer. If this is present, the operation is applied to all rank-one sections that span right through dimension `dim` to produce an array of rank reduced by one and extents equal to the extents in the other dimensions. For example, if `a` is a real array of shape (4,5,6), `sum(a,dim=2)` is a real array of shape (4,6) and element (i, j) has value `sum(a(i,:,j))`.

As the rank of the result depends on whether `dim` is specified, the corresponding actual argument must not itself be an optional dummy argument.

8.11.3 Optional argument mask

The functions `maxval`, `minval`, `product`, and `sum` have a third optional argument, a logical array `mask`. If this is present, it must have the same shape as the first argument and the operation is applied to the elements corresponding to true elements of `mask`; for example, `sum(a, mask = a>0)` sums the positive elements of the array `a`. The argument `mask` affects only the value of the function and does not affect the evaluation of arguments that are array expressions. In **Fortran 95 only**, `mask` is permitted as the second positional argument when `dim` is absent.

8.12 Array inquiry functions

There are five functions for inquiries about the bounds, shape, size and allocation status of an array of any type. Because the result depends only the array properties, the value of the array need not be defined.

8.12.1 Allocation status

`allocated (array)` returns, when the allocatable array `array` is currently allocated, the value true; otherwise it returns the value false . If the allocation status of `array` is undefined, the result is undefined.

8.12.2 Bounds, shape, and size

The following functions enquire about the bounds of an array. In the case of an allocatable array, it must be allocated; and in the case of a pointer, it must be associated with a target. An array section or an array expression is taken to have lower bounds 1 and upper bounds equal to the extents (like an assumed-shape array with no specified lower bounds).

`lbound (array [,dim])`, when `dim` is absent, returns a rank-one default integer array holding the lower bounds. When `dim` is present, it must be a scalar integer and the result is a scalar default integer holding the lower bound in dimension `dim`. As the rank of the result depends on whether `dim` is specified, the corresponding actual argument must not itself be an optional dummy argument.

`shape (source)` returns a rank-one default integer array holding the shape of the array or scalar `source`. In the case of a scalar, the result has size zero.

`size (array [,dim])` returns a scalar default integer that is the size of the array `array` or extent along dimension `dim` if the scalar integer `dim` is present.

`ubound (array [,dim])` is similar to `lbound` except that it returns upper bounds.

8.13 Array construction and manipulation functions

There are eight functions that construct or manipulate arrays of any type.

8.13.1 The merge elemental function

`merge (tsource, fsource, mask)` is an elemental function. The argument `tsource` may have any type and `fsource` must have the same type and type parameters. The argument `mask` must be of type logical. The result is `tsource` if `mask` is true and `fsource` otherwise.

The principal application of merge is when the three arguments are arrays having the same shape, in which case tsource and fsource are merged under the control of mask. Note, however, that tsource or fsource may be scalar in which case the elemental rules effectively broadcast it to an array of the correct shape.

8.13.2 Packing and unpacking arrays

The transformational function pack packs into a rank-one array those elements of an array that are selected by a logical array of conforming shape, and the transformational function unpack performs the reverse operation. The elements are taken in array element order.

pack (array, mask [,vector]), when vector is absent, returns a rank-one array containing the elements of array corresponding to true elements of mask in array element order; mask may be scalar with value true, in which case all elements are selected. If vector is present, it must be a rank-one array of the same type and type parameters as array and size at least equal to the number t of selected elements; the result has size equal to the size n of vector; if $t < n$, elements i of the result for $i > t$ are the corresponding elements of vector.

unpack (vector, mask, field) returns an array of the type and type parameters of vector and shape of mask. The argument mask must be a logical array and vector must be a rank-one array of size at least the number of true elements of mask. field must be of the same type and type parameters as vector and must either be scalar or be of the same shape as mask. The element of the result corresponding to the ith true element of mask, in array element order, is the ith element of vector; all others are equal to the corresponding elements of field if it is an array or to field if it is a scalar.

8.13.3 Reshaping an array

The transformational function reshape allows the shape of an array to be changed, with possible permutation of the subscripts.

reshape (source, shape [,pad][,order]) returns an array with shape given by the rank-one integer array shape, and type and type parameters those of the array source. The size of shape must be constant, and its elements must not be negative. If pad is present, it must be an array of the same type and type parameters as source. If pad is absent or of size zero, the size of the result must not exceed the size of source. If order is absent, the elements of the result, in array element order, are the elements of source in array element order followed by copies of pad in array element order. If order is present, it must be a rank-one integer array with a value that is a permutation of (1,2,...,n); the elements r($s_1, \ldots, s_n$) of the result, taken in subscript order for the array having elements r($s_{\text{order}(1)}, \ldots, s_{\text{order}(n)}$), are

those of source in array element order followed by copies of pad in array element order. For example, if order has the value (/3,1,2/), the elements r(1,1,1), r(1,1,2), ..., r(1,1,k), r(2,1,1), r(2,1,2), ... correspond to the elements of source and pad in array element order.

8.13.4 Transformational function for replication

spread (source, dim, ncopies) returns an array of type and type parameters those of source and of rank increased by one. The argument source may be scalar or array-valued. The arguments dim and ncopies are integer scalars. The result contains max(ncopies, 0) copies of source, and element $(r_1, \ldots, r_{n+1})$ of the result is source$(s_1, \ldots, s_n)$ where $(s_1, \ldots, s_n)$ is $(r_1, \ldots, r_{n+1})$ with subscript dim omitted (or source itself if it is scalar).

8.13.5 Array shifting functions

cshift (array, shift [,dim]) returns an array of the same type, type parameters, and shape as array. The argument shift is of type integer and must be scalar if array is of rank one. If shift is scalar, the result is obtained by shifting every rank-one section that extends across dimension dim circularly shift times. The argument dim is an integer scalar and, if it is omitted, it is as if it were present with the value 1. The direction of the shift depends on the sign of shift, being to the left for a positive value and to the right for a negative value. Thus, for the case with shift=1 and array of rank one and size m, the element i of the result is array(i+1), where $i = 1, 2, \ldots, m-1$, and element m is array(1). If shift is an array, it must have the same shape as that of array with dimension dim omitted, and it supplies a separate value for each shift. For example, if array is of rank three and shape (k, l, m) and dim has the value 2, shift must be of shape (k, m) and supplies a shift for each of the $k \times m$ rank-one sections in the second dimension of array.

eoshift (array, shift [,boundary][,dim]) is identical to cshift except that an end-off shift is performed and boundary values are inserted into the gaps so created. The argument boundary may be omitted when array has intrinsic type, in which case the value zero is inserted for the integer, real, and complex cases; false in the logical case; and blanks in the character case. If boundary is present, it must have the same type and type parameters as array; it may be scalar and supply all needed values or it may be an array whose shape is that of array with dimension dim omitted and supply a separate value for each shift.

8.13.6 Matrix transpose

The transpose function performs a matrix transpose for any array of rank two.

`transpose (matrix)` returns an array of the same type and type parameters as the rank-two array `matrix`. Element (i, j) of the result is `matrix(`j, i`)`.

8.14 Transformational functions for geometric location

There are two transformational functions that find the locations of the maximum and minimum values of an integer or real array.

`maxloc (array [,mask])` returns a rank-one default integer array of size equal to the rank of `array`. Its value is the sequence of subscripts of an element of maximum value (among those corresponding to true values of the conforming logical array `mask` if it is present), as though all the declared lower bounds of `array` were 1. If there is more than one such element, the first in array element order is taken.

`maxloc (array, dim [,mask])` is available in **Fortran 95 only**. It returns a default integer array of shape equal to the that of `array` with dimension `dim` omitted, where `dim` is a scalar integer with value in the range $1 \leq$ `dim` $\leq$ rank(`array`). The value of each element of the result is the position of the first element of maximum value in the corresponding rank-one section spanning dimension `dim`, among those elements corresponding to true values of the conforming logical array `mask` when it is present.

`minloc (array [,mask])` is identical to `maxloc (array [,mask])` except that the position of an element of minimum value is obtained.

`minloc (array, dim [,mask])` (available in **Fortran 95 only**) is identical to `maxloc (array, dim [,mask])` except that positions of elements of minimum value are obtained.

8.15 Transformational function for pointer disassociation (Fortran 95)

In Fortran 95, the function `null` is available to give the disassociated status to pointer entities.

`null ([mold])` returns a disassociated pointer. The argument `mold` is a pointer of any type and may have any association status, including undefined. The type, type parameter, and rank of the result are those of `mold` if it is present and otherwise are those of the object with which it is associated.

8.16 Non-elemental intrinsic subroutines

There are also in Fortran non-elemental intrinsic subroutines, which were chosen to be subroutines rather than functions because of the need to return information through the arguments.

8.16.1 Real-time clock

There are two subroutines that return information from the real-time clock, the first based on the ISO standard IS 8601 (Representation of dates and times). It is assumed that there is a basic system clock that is incremented by one for each clock count until a maximum `count_max` is reached and on the next count is set to zero. Default values are returned on systems without a clock. All the arguments have intent `out`.

`call date_and_time ([date] [,time] [,zone] [,values])` returns the following (with default values blank or `-huge(0)`, as appropriate, when there is no clock):

`date` is a scalar character variable holding the date in the form *ccyymmdd*, corresponding to century, year, month, and day.

`time` is a scalar character variable holding the time in the form *hhmmss.sss*, corresponding to hours, minutes, seconds, and milliseconds.

`zone` is a scalar character variable that is set to the difference between local time and Coordinated Universal Time (UTC, also known as Greenwich Mean Time) in the form *Shhmm*, corresponding to sign, hours, and minutes. For example, a processor in New York in winter would return the value -0500.

`values` is a rank-one default integer array of size at least 8 holding the sequence of values: the year, the month of the year, the day of the month, the time difference in minutes with respect to UTC, the hour of the day, the minutes of the hour, the seconds of the minute, and the milliseconds of the second.

`call system_clock ([count][,count_rate][,count_max])` returns the following:

`count` is a scalar default integer holding a processor-dependent value based on the current value of the processor clock, or `-huge(0)` if there is no clock. On the first call, the processor may set an initial value that may be zero.

`count_rate` is a scalar default integer holding the number of clock counts per second, or zero if there is no clock.

`count_max` is a scalar default integer holding the maximum value that `count` may take, or zero if there is no clock.

8.16.2 CPU time (Fortran 95 only)

In Fortran 95, there is a non-elemental intrinsic subroutine that returns the processor time.

`call cpu_time (time)` returns the following:

`time` is a scalar real that is assigned a processor-dependent approximation to the processor time in seconds, or a processor-dependent negative value if there is no clock.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same computer or discover which parts of a calculation on a computer are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example

```
real :: t1, t2
:
call cpu_time(t1)     ! Fortran 95
:                     ! Code to be timed.
call cpu_time(t2)
write (*,*) 'Time taken by code was ', t2-t1, ' seconds'
```

8.16.3 Random numbers

A sequence of pseudorandom numbers is generated from a seed that is held as a rank-one array of integers. The subroutine `random_number` returns the pseudo-random numbers and the subroutine `random_seed` allows an inquiry to be made about the size or value of the seed array, and the seed to be reset. The subroutines provide a portable interface to a processor-dependent sequence.

`call random_number (harvest)` returns a pseudorandom number from the uniform distribution over the range $0 \leq x < 1$ or an array of such numbers. `harvest` has intent `out`, may be a scalar or an array, and must be of type real.

`call random_seed ([size] [put] [get])` has the following arguments:

`size` has intent `out` and is a scalar default integer that the processor sets to the size n of the seed array.

`put` has intent `in` and is a default integer array of rank one and size n that is used by the processor to reset the seed. A processor may set the same seed value for more than one value of `put`.

`get` has intent `out` and is a default integer array of rank one and size n that the processor sets to the current value of the seed.

No more than one argument may be specified; if no argument is specified, the seed is set to a processor-dependent value.

8.17 Summary

In this chapter, we introduced the four categories of intrinsic procedures, explained the `intrinsic` statement, and gave detailed descriptions of all the procedures. The procedures of Sections 8.2, 8.6.2, and 8.7 to 8.16 are all new since Fortran 77. Within Sections 8.3 to 8.5 the procedures `ceiling`, `floor`, `modulo`, `achar`, `iachar`, `adjustl`, `adjustr`, `len_trim`, `scan`, `verify`, and `logical` are new; the remaining functions were present in Fortran 77, have been generalized to handle all kind type parameters, have become elemental, and several have been given additional optional arguments. The function `len` has become an inquiry function.

Of the many new intrinsic procedures, some have names that might also be names of external functions in existing Fortran 77 programs. Appendix B15 of the Fortran 77 standard recommended that external procedures be identified as such by using the `external` statement. For any external procedure with a name contained in the following list, it is essential to provide an interface body or to use the `external` statement.

```
achar   all     any     btest   count   cshift
digits  floor   huge    iachar  iand    ibclr
ibits   ibset   ieor    ior     ishft   ishftc
kind    lbound  matmul  maxloc  maxval  merge
minloc  minval  modulo  mvbits  not     null
pack    radix   range   repeat  scale   scan
shape   size    spread  sum     tiny    trim
ubound  unpack  verify
```

Note: `null` is **Fortran 95 only**.

8.18 Exercises

1. Write a program to calculate the real roots or pairs of complex-conjugate roots of the quadratic equation $ax^2 + bx + c = 0$ for any real values of a, b, and c. The program should read these three values and print the results. Use should be made of the appropriate intrinsic functions.

2. Repeat Exercise 1 of Chapter 5, avoiding the use of do constructs.

3. Given the rules explained in Sections 3.12 and 8.2, what are the values printed by the following program?

```
program main
   real, target  :: a(3:10)
   real, pointer :: p1(:), p2(:)
   p1 => a(3:9:2)
   p2 => a(9:3:-2)
   print *, associated(p1, p2)
   print *, associated(p1, p2(4:1:-1))
end program main
```

4. In the following program, two pointer assignments, one to an array the other to an array section, are followed by a subroutine call. Bearing in mind the rules given in Sections 3.12, 6.3, and 8.12.2, what values does the program print?

```
program main
   real, target  :: a(5:10)
   real, pointer :: p1(:), p2(:)
   p1 => a
   p2 => a(:)
   print *, lbound (a), lbound (a(:))
   print *, lbound (p1), lbound (p2)
   call what (a, a(:))
contains
   subroutine what (x, y)
      real, intent (in) :: x(:), y(:)
      print *, lbound (x), lbound (y)
   end subroutine what
end program main
```

9. Data transfer

9.1 Introduction

Fortran has, in comparison with most other high-level programming languages, a particularly rich set of facilities for input/output (I/O). The Fortran 77 standard brought with it important new features including direct-access files, internal files, execution-time format specification, list-directed input/output, file inquiry, and some new edit descriptors. By contrast, the only significant new features in the latest standards are non-advancing I/O, `namelist`, and some new edit descriptors. In addition, there are a number of detailed changes to support new facilities in other areas.

Input/output is an area of Fortran into which not all programmers need to delve very deeply. For most small-scale programs it is sufficient to know how to read a few data records containing input variables, and how to transmit to a terminal or printer the results of a calculation. In large-scale data processing, on the other hand, the programs often have to deal with huge streams of data to and from many disc, tape, and cartridge files; in these cases it is essential that great attention be paid to the way in which the I/O is designed and coded, as otherwise both the execution time and the real time spent in the program can suffer dramatically. The term *file* is used for a collection of data on one of these devices and a file is always organized into a sequence of *records*.

This chapter begins by discussing the various forms of formatted I/O, that is I/O which deals with records that do not use the internal number representation of the computer, but rather a character string which can be displayed for visual inspection by the human eye. It is also the form usually needed for transmitting data between different kinds of computers. The so-called *edit descriptors*, which are used to control the translation between the internal number representation and the external format, are then explained. Finally, the topics of unformatted (or binary) I/O and direct-access files are covered.

9.2 Number conversion

The ways in which numbers are stored internally by a computer are the concern of neither the Fortran standard nor this book. However, if we wish to output values – to display them on a terminal or to print them – then their internal representations

must be converted into a character string which can be read in a normal way. For instance, the contents of a given computer word may be (in hexadecimal) be1d7dbf and correspond to the value -0.000450 . For our particular purpose, we may wish to display this quantity as `-.000450`, or as `-4.5E-04`, or rounded to one significant digit as `-5e-04`. The conversion from the internal representation to the external form is carried out according to the information specified by an edit descriptor contained in a *format specification.* These will both be dealt with fully later in this chapter; for the moment, it is sufficient to give a few examples. For instance, to print an integer value in a field of 10 characters width, we would use the edit descriptor `i10`, where `i` stands for integer conversion, and `10` specifies the width of the output field. To print a real quantity in a field of 10 characters, five of which are reserved for the fractional part of the number, we specify `f10.5`. The edit descriptor `f` stands for floating-point (real) conversion, `10` is the total width of the output field and 5 is the width of the fractional part of the field. If the number given above were to be converted according to this edit descriptor, it would appear as *bb*`-0.00045`, where *b* represents a blank. To print a character variable in a field of 10 characters, we would specify `a10`, where `a` stands for alphanumeric conversion.

A format specification consists of a list of edit descriptors enclosed in parentheses, and can be coded either as a default character expression, for instance

```
'(i10, f10.3, a10)'
```

or as a separate `format` statement, referenced by a statement label, for example

```
10 format(i10, f10.3, a10)
```

To print the scalar variables `j`, `b`, and `c`, of types integer, real, and character respectively, we may then write either

```
print '(i10, f10.3, a10)', j,b,c
```

or

```
   print 10, j,b,c
10 format(i10, f10.3, a10)
```

The first form is normally used when there is only a single reference in a scoping unit to a given format specification, the second when there are several or when the format is complicated. The part of the statement designating the quantities to be printed is known as the *output list* and forms the subject of the following section.

9.3 I/O lists

The quantities to be read or written by a program are specified in an I/O list. For output, they may be expressions but for input must be variables. In both cases, list items may be implied-do lists of quantities. Examples are shown in Figure 9.1, where we note the use of a *repeat count* in front of those edit descriptors that

are required repeatedly. A repeat count must be a positive integer literal constant and not have a kind type parameter. Function references are permitted in an I/O list, provided they do not themselves cause further I/O to occur.

Figure 9.1

```
integer              :: i
real, dimension(10) :: a
character(len=20)    :: word
print '(i10)',      i
print '(10f10.3)', a
print '(3f10.3)',  a(1),a(2),a(3)
print '(a10)',      word(5:14)
print '(5f10.3)', (a(i), i=1,9,2)
print '(2f10.3)',  a(1)*a(2)+i, sqrt(a(3))
```

In all these examples, except the last one, the expressions consist of single variables and would be equally valid in input statements using the read statement, for example

```
read '(i10)', i
```

Such statements may be used to read values which are then assigned to the variables in the input list.

If an array appears as an item, it is treated as if the elements were specified in array element order. For example, the third of the print statements in Figure 9.1 could have been written

```
print '(3f10.3)', a(1:3)
```

However, no element of the array may appear more than once in an input item. Thus, the case in Figure 9.2 is not allowed.

Figure 9.2

```
integer :: j(10), k(3)
:
k = (/ 1, 2, 1 /)
read '(3i10)', j(k)     ! Illegal because j(1) appears twice
```

If an allocatable array appears as an item, it must be currently allocated.

Any pointers in an I/O list must be associated with a target, and transfer takes place between the file and the targets.

An item of derived type is treated as if the components were specified in the same order as in the type declaration. This rule is applied repeatedly for components of derived type, so that it is as if we specified the list of items of

intrinsic type that constitute its ultimate components. For example, if p and t are of the types point and triangle of Figure 2.1, the statement

```
read '(8f10.5)', p, t
```

has the same effect as the statement

```
read '(8f10.5)', p%x, p%y, t%a%x, t%a%y, t%b%x,        &
                           t%b%y, t%c%x, t%c%y
```

Each ultimate component must be accessible (not, for example, be a private component of a public type).

An object in an I/O list is not permitted to be of a derived type that has a pointer component at any level of component selection. One reason for this restriction is because of the problems associated with recursive data structures. For example, supposing chain is a data object of the type entry of Figure 2.3 (in Section 2.13) and is set up to hold a chain of length three, then it has as its ultimate components chain%index, chain%next%index, chain%next%next%index, and chain%next%next%next, the last of which is a disassociated pointer. Another reason is the intention to add defined edit descriptors for data structures to Fortran 2000. Programmers will be able to write procedures that are called as part of the I/O processing. Such a procedure will be much better able to handle structures whose size and composition vary dynamically, the usual case for pointer components.

An I/O list may include an implied-do list, as illustrated by the fifth print statement in Figure 9.1. The general form is

(*do-object-list*, *do-var* = *expr*, *expr* [, *expr*])

where each *do-object* is a variable (for input), an expression (for output), or is itself an implied-do list; *do-var* is a named scalar integer variable; and each *expr* is a scalar integer expression. The loop initialization and execution is the same as for a (possibly nested) set of do constructs (Section 4.5). In an input list, a variable that is an item in a *do-object-list* must not be a *do-var* of any implied-do list in which it is contained, nor be associated[1] with such a *do-var*. In an input or output list, no *do-var* may be a *do-var* of any implied-do list in which it is contained or be associated with such a *do-var*.

Note that a zero-sized array, or an implied-do list with a zero iteration count, may occur as an item in an I/O list. Such an item corresponds to no actual data transfer.

9.4 Format definition

In the print and read statements of the previous section, the format specification was given each time in the form of a character constant immediately following the keyword. In fact, there are three ways in which a format specification may be given. They are:

[1] Such an illegal association could be established by pointer association.

i) As a statement label referring to a format statement containing the relevant specification between parentheses:

```
    print 100, q
    :
100 format(f10.3)
```

The format statement must appear in the same scoping unit, before the contains statement if it has one. It is customary either to place each format statement immediately after the first statement which references it, or to group them all together just before the contains or end statement. It is also customary to have a separate sequence of numbers for the statement labels used for format statements. A given format statement may be used by any number of formatted I/O statements, whether for input or for output.

ii) As a default character expression whose value commences with a format specification in parentheses:

```
print '(f10.3)', q
```

or

```
character(len=*), parameter :: form='(f10.3)'
:
print form, q
```

or

```
character :: carray(7)=(/ '(','f','1','0','.','3',')' /)
:
print carray, q ! Elements of an array expression
                ! are concatenated.
```

or

```
character(4) :: carr1(10)
character(3) :: carr2(10)
integer      :: i, j
:
carr1(10) = '(f10'
carr2(3) = '.3)'
:
i = 10
j = 3
:
print carr1(i)//carr2(j), q
```

From these examples it may be seen that it is possible to program formats in a flexible way, and particularly that it is possible to use arrays, expressions and also substrings in a way which allows a given format to be built up dynamically at execution-time from various components. Any character data which might follow the trailing right parenthesis are ignored and may be undefined. In the case of an array, its elements are concatenated in array element order. However, on input *no* component of the format specification may appear also in the input list, or be associated with it. This is because the standard requires that the whole format specification be established *before* any I/O takes place. Further, no redefinition or undefinition of any characters of the format is permitted during the execution of the I/O statement.

iii) As an asterisk. This is a type of I/O known as *list-directed* I/O, in which the format is defined by the computer system at the moment the statement is executed, depending on both the type and magnitude of the entities involved. This facility is particularly useful for the input and output of small quantities of values, especially in temporary code which is used for test purposes, and which is removed from the final version of the program:

```
print *, 'Square-root of q = ', sqrt(q)
```

This example outputs a character constant describing the expression which is to be output, followed by the value of the expression under investigation. On the terminal screen, this might appear as

```
Square-root of q = 4.392246
```

the exact format being dependent on the computer system used. Character strings in this form of output are normally undelimited, as if an a edit descriptor were in use, but an option in the open statement (Section 10.3) may be used to require that they be delimited by apostrophes or quotation marks. Except for adjacent undelimited strings, values are separated by spaces or commas. Logical variables are represented as T for true and F for false. The processor may represent a sequence of r identical values c by the form $r * c$. Further details of list-directed input/output are deferred until Section 9.9.

Blank characters may precede the left parenthesis of a format specification, and may appear at any point within a format specification with no effect on the interpretation, except within a character string edit descriptor (Section 9.13.4).

9.5 Unit numbers

Input/output operations are used to transfer data between the variables of an executing program, as stored in the computer, and an external medium. There are

many types of external media: the terminal, printer, disc drive, and CD are perhaps the most familiar. Whatever the device, a Fortran program regards each one from which it reads or to which it writes as a *unit*, and each unit, with two exceptions, has associated with it a *unit number*. This number must not be negative and is often in the range 1 to 99. Thus we might associate with a disc drive from which we are reading the unit number 10, and to a hard disc drive to which we are writing the unit number 11. All program units of an executable program that refer to a particular unit number are referencing the same file.

There are two I/O statements, `print` and a variant of `read`, that do not reference any unit number; these are the statements that we have used so far in examples, for the sake of simplicity. A `read` statement without a unit number will normally expect to read from the terminal, unless the program is working in batch (non-interactive) mode in which case there will be a disc file with a reserved name from which it reads. A `print` statement will normally expect to output to the terminal, unless the program is in batch mode in which case another disc file with a reserved name will be used. Such files are usually suitable for subsequent output on a physical output device. The system may implicitly associate unit numbers to these default units.

Apart from these two special cases, all I/O statements must refer explicitly to a unit in order to identify the device to which or from which data are to be transferred. The unit may be given in one of three forms. These are shown in the following examples which use another form of the `read` containing a unit specifier, *u*, and format specifier, *fmt*, in parentheses and separated by a comma, where *fmt* is a format specification as described in the previous section:

`read (`*u*`, `*fmt*`)` *list*

The three forms of *u* are:

i) As a scalar integer expression that gives the unit number:

```
read (4, '(f10.3)') q
read (nunit, '(f10.3)') q
read (4*i+j, 100) a
```

where the value may be any nonnegative integer allowed by the system for this purpose.

ii) As an asterisk:

```
read (*, '(f10.3)') q
```

where the asterisk implies the standard input unit designated by the system, the same as that used for `read` without a unit number.

iii) As a default character variable identifying an *internal file* (see next section).

9.6 Internal files

Internal files allow format conversion between various representations to be carried out by the program in a storage area defined within the program itself. There are two particularly useful applications, one to read data whose format is not properly known in advance, and the other to prepare output lists containing mixed character and numerical data, all of which has to be prepared in character form, perhaps to be displayed as a caption on a graphics display. The character data must be of default kind. The first application will now be described; the second will be dealt with in Section 9.8.

Imagine that we have to read a string of 30 digits, which might correspond to 30 one-digit integers, 15 two-digit integers or 10 three-digit integers. The information as to which type of data is involved is given by the value of an additional digit, which has the value 1, 2, or 3, depending on the number of digits each integer contains. An internal file provides us with a mechanism whereby the 30 digits can be read into a character buffer area. The value of the final digit can be tested separately, and 30, 15, or 10 values read from the internal file, depending on this value. The basic code to achieve this might read as follows (no error recovery or data validation is included, for simplicity):

```
integer      :: ival(30), key, i
character(30):: buffer
character(6) :: form(3) = (/ '(30i1)', '(15i2)', '(10i3)' /)
read (*, '(a30,i1)')      buffer, key
read (buffer, form (key)) (ival(i), i=1,30/key)
```

Here, `ival` is an array which will receive the values, `buffer` a character variable of a length sufficient to contain the 30 input digits, and `form` a character array containing the three possible formats to which the input data might correspond. The first `read` statement reads 30 digits into `buffer` as character data, and a final digit into the integer variable `key`. The second `read` statement reads the data from `buffer` into `ival`, using the appropriate conversion as specified by the edit descriptor selected by `key`. The number of variables read from `buffer` to `ival` is defined by the implied-do loop, whose second specifier is an integer expression depending also on `key`. After execution of this code, `ival` will contain `30/key` values, their number and exact format not having been known in advance.

If an internal file is a scalar, it has a single record whose length is that of the scalar. If it is an array, its elements, in array element order, are treated as successive records of the file and each has length that of an array element. It may not be an array section with a vector subscript.

A record becomes defined when it is written. The number of characters sent must not exceed the length of the record. It may be less, in which case the rest of the record is padded with blanks. For list-directed output (Section 9.4), character constants are not delimited. A record may be read only if it is defined (which need not only be by an output statement). Records are padded with blanks, if necessary.

An internal file is always positioned at the beginning of its first record prior to data transfer (the array section notation may be used to start elsewhere in an array). Of course, if an internal file is an allocatable array or pointer, it must be allocated or associated with a target. Also, no item in the input/output list may be in the file or associated with the file.

An internal file must be of default character type and non-default character items are not permitted in input/output lists. It may be used for list-directed I/O (Section 9.9), but not for `namelist` I/O (Section 9.10).

9.7 Formatted input

In the previous sections we have given complete descriptions of the ways that formats and units may be specified, using simplified forms of the `read` and `print` statements as examples. There are, in fact, two forms of the formatted `read` statement. Without a unit, it has the form

```
read fmt [,list]
```

and with a unit it may take the form

```
read ([unit=]u, [fmt=]fmt [,iostat=ios]                    &
          [, err=error-label] [,end=end-label]) [list]
```

where *u* and *fmt* are the unit and format specifiers described in Sections 9.4 and 9.5; `iostat=`, `err=`, and `end=` are optional specifiers which allow a user to specify how a `read` statement shall recover from various exceptional conditions; and *list* is a list of variables and implied-do lists of variables. The keyword items may be specified in any order, although it is usual to keep the unit number and format specification as the first two. The unit number must be first if it does not have its keyword. If the format does not have its keyword, it must be second, following the unit number without its keyword. Note that this parallels the rules for keyword calls of procedures, except that the positional list is limited to two items.

For simplicity of exposition, we have so far limited ourselves to formats that correspond to a single record in the file, but we will meet later in this chapter cases that lead to the input of a part of a record or of several successive records.

The meanings of the optional specifiers are as follows. If the `iostat=` is specified, then *ios* must be a scalar integer variable of default kind which, after execution of the `read` statement, has a negative value if an end-of-record condition is encountered during non-advancing input (Section 9.12), a different negative value if an endfile condition was detected on the input device (Section 10.2.3), a positive value if an error was detected (for instance a parity error), or the value zero otherwise. The actual values assigned to *ios* in the event of an exception occurring are not defined by the standard, only the signs.

If the `end=` is specified, then *end-label* must be a statement label of a statement in the same scoping unit, to which control will be transferred in the event of the end of the file being reached.

If the err= is specified, then *error-label* is a statement label in the same scoping unit, to which control will be transferred in the event of any other exception occurring. The labels *error-label* and *end-label* may be the same. If they are not specified and an exception occurs, execution will stop, unless iostat is specified. An example of a read statement with its associated error recovery is given in Figure 9.3, in which error and last_file are subroutines to deal with the exceptions. They will normally be system dependent.

Figure 9.3

```
      read (nunit, '(3f10.3)', iostat=ios, err=110, end=120) a,b,c
   !
   !   Successful read - continue execution.
       :
       :
   !
   !   Error condition - take appropriate action.
 110   call error (ios)
       go to 999
   !
   !   End-of-file condition - test whether more
   !   files follow.
 120   call last_file
       :
 999   end
```

If an error or end-of-file condition occurs on input, the statement terminates and all list items and any implied-do variables become undefined. If an end-of-file condition occurs for an external file, the file is positioned following the endfile record (Section 10.2.3); if there is otherwise an error condition, the file position is indeterminate. An end-of-file condition occurs also if an attempt is made to read beyond the end of an internal file.

It is a good practice to include some sort of error recovery in all read statements which are included permanently in a program. On the other hand, input for test purposes is normally sufficiently well handled by the simple form of read without unit number, and without error recovery.

9.8 Formatted output

There are two types of formatted output statements, the print statement which has appeared in many of the examples so far in this chapter, and the write statement whose syntax is similar to that of the read statement:

print *fmt* [,*list*]

and

```
write ([unit=]u, [fmt=]fmt [,iostat=ios]                &
            [,err=error-label] ) [list]
```

where all the components have the same meanings as described for the `read` statement (Section 9.7). An asterisk for *u* specifies the standard output unit, as used by `print`. If an error condition occurs on output, execution of the statement terminates, any implied-do variables become undefined, and the file position becomes indeterminate.

An example of a `write` statement is

```
write (nout, '(10f10.3)', iostat=ios, err=110) a
```

An example using an internal file is given in Figure 9.4, which builds a character string from numeric and character components. The final character string might be passed to another subroutine for output, for instance as a caption on a graphics display.

Figure 9.4

```
integer             :: day
real                :: cash
character(len=50) :: line
:
!   write into line
write (line,'(a, i2, a, f8.2, a)')                    &
    'Takings for day ', day, ' are ', cash, ' dollars'
```

In this example, we declare a character variable that is long enough to contain the text to be transferred to it. (The `write` statement contains a format specification with a edit descriptors without a field width. These assume a field width corresponding to the actual length of the character strings to be converted.) After execution of the `write` statement, `line` might contain the character string

```
Takings for day  3 are  4329.15 dollars
```

and this could be used as a string for further processing.

The number of characters written to `line` must not exceed its length.

9.9 List-directed I/O

In Section 9.4, the list-directed output facility using an asterisk as format specifier was introduced. We assumed that the list was short enough to fit into a single record, but for long lists the processor is free to output several records. Character constants may be split between records, and complex constants that are as long as, or longer than, a record may be split after the comma that separates the two parts. Apart from these cases, a value always lies within a single record. For the sake of carriage control (which is described in Section 9.11), the first character of each

record is blank unless a delimited character constant is being continued. Note that when an undelimited character constant is continued, the first character of the continuation record is blank. The only blanks permitted in a numeric constant are within a split complex constant after the comma.

This facility is equally useful for input, especially of small quantities of test data. On the input record, the various constants may appear in any of their usual forms, just as if they were being read under the usual edit descriptors, as defined in Section 9.13. Exceptions are that complex values must be enclosed in parentheses, character constants may be delimited, a blank must not occur except in a delimited character constant or in a complex constant before or after a numeric field, blanks are never interpreted as zeros, and the optional characters which are allowed in a logical constant (those following `t` or `f`, see Section 9.13.2) must include neither a comma nor a slash.

Character constants that are enclosed in apostrophes or quotation marks may be spread over as many records as necessary to contain them, except that a doubled quotation mark or apostrophe must not be split between records. Delimiters may be omitted for a default character constant if

- it is of nonzero length;
- the constant does not contain a blank, comma, or slash;
- it is contained in one record;
- the first character is neither a quotation mark nor an apostrophe; and
- the leading characters are not numeric followed by an asterisk.

In this case, the constant is terminated when a blank, comma, slash, or end of record is encountered, and apostrophes or quotation marks appearing within the constant must not be doubled.

Whenever a character value has a different length from the corresponding list item, the value is truncated or padded on the right with blanks, as in the character assignment statement.

It is possible to use a repeat count for a given constant, for example `6*10` to specify six occurrences of the integer value 10.

The (optionally repeated) constants are separated in the input by *separators*. A separator is one of the following, appearing other than in a character constant:

- a comma, optionally preceded and optionally followed by one or more contiguous blanks,
- a slash (/), optionally preceded and optionally followed by one or more contiguous blanks, or
- one or more contiguous blanks between two non-blank values or following the last non-blank value.

An end of record not within a character constant is regarded as a blank and, therefore, forms part of a separator. A blank embedded in a complex constant or delimited character constant is not a separator. An input record may be terminated by a slash separator, in which case all the following values in the record are ignored, and the input statement terminates.

If there are no values between two successive separators, or between the beginning of the first record and the first separator, this is taken to represent a *null value* and the corresponding item in the input list is left unchanged, defined or undefined as the case may be. A null value must not be used for the real or imaginary part of a complex constant, but a single null value may be used for the whole complex value. A series of null values may be represented by a repeat count without a constant: `,6*,`. When a slash separator is encountered, null values are given to any remaining list items.

An example of this form of the `read` statement is:

```
integer             :: i
real                :: a
complex             :: field(2)
logical             :: flag
character (len=12) :: title
character (len=4)   :: word
:
read *, i, a, field, flag, title, word
```

If this reads the input record

`10`*b*`6.4`*b*`(1.,0.)`*b*`(2.,0.)`*btb*`test/`

(in which *b* stands for a blank, and blanks are used as separators), then `i`, `a`, `field`, `flag`, and `title` will acquire the values 10, 6.4, (1.,0.) and (2.,0.), `.true.` and `test` respectively, while `word` remains unchanged. For the input records

```
10,.64e1,2*,.true.
'histogramb10'/val1
```

(in which commas are used as separators), the variables `i`, `a`, `flag`, and `title` will acquire the values 10, 6.4, `.true.`, and `histogramb10` respectively. The variable `field` and `word` remain unchanged, and the input string `val1` is ignored as it follows a slash. (Note the apostrophes, which are required as the string contains a blank. Without delimiters, this string would appear to be a string followed by the integer value 10.) Because of this slash, the read statement does not continue with the next record and the list is thus not fully satisfied.

9.10 Namelist I/O

It can be useful, especially for program testing, to input or output an annotated list of values. The values required are specified in a `namelist` group (Section 7.15),

and the I/O is performed by a `read` or `write` statement that does not have an I/O list, and in which either

- the format is replaced by a namelist-group name as the second positional parameter, or
- the `fmt=` specifier is replaced by a `nml=` specifier with that name.

When reading, only those objects which are specified in the input record and which do not have a null value become defined. All other list items remain in their existing state of definition or undefinition. It is possible to define the value of an array element or section without affecting the other portions of the array. When writing, all the items in the group are written to the file specified. This form of I/O is not available for internal files.

The value for a scalar object or list of values for an array is preceded in the records by the name or designator and an equals sign which may optionally be preceded or followed by blanks. The form of the list of values and null values in the input and output records is as that for list-directed I/O (Section 9.9), except that character constants must *always* be delimited in input records and logical constants must not contain an equals sign. A `namelist` input statement terminates on the appearance of a slash in the list outside a character constant. A simple example is

```
integer   ::  no_of_eggs, litres_of_milk, kilos_of_butter
namelist/food/no_of_eggs, litres_of_milk, kilos_of_butter
read (5, nml=food)
```

to read the record

```
&food litres_of_milk=5, no_of_eggs=12 /
```

where we note that the order of the two values given is not the same as their order in the `namelist` group — the orders need not necessarily match. The value of `kilos_of_butter` remains unchanged. The first non-blank item in the record is an ampersand followed without an intervening blank by the group name. The slash is obligatory as a terminator. On output, a similar annotated list of values is produced, starting with the name of the group and ending with a slash. Here the order is that of the `namelist` group. Thus, the statements

```
integer   :: number, list(10)
namelist/out/number, list
write (6, nml=out)
```

might produce the record

```
&OUT NUMBER=1, LIST=14, 9*0 /
```

On output, the names are always in upper case.

Where a subobject designator appears in an input record, all substring expressions, subscripts, and strides must be scalar integer literal constants without

specified kind parameters. All group names, object names, and component names are interpreted without regard to case. Blanks may precede or follow the name or designator, but must not appear within it.

If the object is scalar and of intrinsic type, the equals sign must be followed by one value. If it is of derived type or is an array, the equals sign must be followed by a list of values of intrinsic type corresponding to the replacement of each derived-type value by its ultimate components and each array by its elements in array element order.

The list of values must not be too long, but it may be too short, in which case trailing null values are regarded as having been appended. If an object is of type character, the corresponding item must be of the same kind.

Zero-sized objects must not appear in a `namelist` input record. In any multiple occurrence of an object in a sequence of input records, the final value is taken.

9.10.1 Comments in namelist input (Fortran 95 only)

In Fortran 95, input records for `namelist` input may bear a comment following an object name/value separator other than a slash. This allows programers to document the structure of a `namelist` input file line-by-line. The comment is in the usual format for comments. Taking the input record of Section 9.10, it may be documented thus:

```
&food litres_of_milk=5,   ! Fortran 95
no_of_eggs=12 /
```

A comment line, with `!` as the first non-blank character in an input record, is also permitted, but may not occur in a character context.

9.11 Carriage control

Fortran's formatted output statements were originally designed for line-printers, with their concept of lines and pages of output. On such a device, the first character of each output record must be of default kind. It is not printed but interpreted as a *carriage control character.* If it is a blank, no action is taken, and it is good practice to insert a blank as the first character of each record, either explicitly as `' '` or using the `t2` edit descriptor (described in Section 9.13.5), in order to avoid inadvertent generation of spurious carriage control characters. This can happen when the first character in an output record is non-blank, and might occur, for instance, when printing integer values with the format `'(i5)'`. Here all output values between -999 and 9999 will have a blank in the first position, but all others will generate a character there which may be used mistakenly for carriage control.

The carriage control characters defined by the standard are:

b	to start a new line
+	to remain on the same line (overprint)
0	to skip a line
1	to advance to the beginning of the next page

As a precaution, the first character of each record produced by list-directed and `namelist` output is a blank, unless it is the continuation of a delimited character constant.

In this context, we note that execution of a `print` statement does not imply that any printing will actually occur, and nor does execution of a `write` statement imply that printing will not occur.

9.12 Non-advancing I/O

So far we have considered each `read` or `write` statement to perform the input or output of a complete record. There are, however, many applications, especially in screen management, where this would become an irksome restriction. What is required is the ability to read and write without always advancing the file position to ahead of the next record. This facility is provided by *non-advancing* I/O. To gain access to this facility, the optional `advance=` specifier must appear in the `read` or `write` statement and be associated with a scalar default character expression *advance* which evaluates, after suppression of any trailing blanks and conversion of any upper-case letters to lower case, to the value `no`. The only other allowed value is `yes` which is the default value if the specifier is absent; in this case normal (advancing) I/O occurs.

The following optional specifiers are available for a non-advancing `read` statement:

`eor=`*eor-label*
`size=`*size*

where *eor-label* is a statement label in the same scoping unit and *size* is a default integer scalar variable. The *eor-label* may be the same as an *end-label* or *error-label* of the `read` statement.

An advancing I/O statement always repositions the file after the last record accessed, but a non-advancing I/O statement performs no such repositioning and may therefore leave the file positioned within a record. If a non-advancing input statement attempts to transfer data from beyond the end of the *current* record, an end-of-record condition occurs. The `iostat` variable, if present, will acquire a different negative value to the one indicating an end-of-file condition; and, if the `eor=` specifier is present, control is transferred to the statement specified by its associated *eor-label*. In order to provide a means of controlling this process, the `size=` specifier, when present, sets *size* to the number of characters actually read. A full example is thus

```
    character(len=3) :: key
    integer          :: unit, size
    read (unit, '(a3)', advance='no', size=size, eor=66) key
    :
 ! key is not in one record
66  key(size+1:) = ''
    :
```

As for error and end-of-file conditions, the program terminates when an end-of-record condition occurs if neither `eor=` nor `iostat=` is specified.

If encountering an end-of-record on reading results in the input list not being satisfied, the `pad=` specifier described in Section 10.3 will determine whether any padding with blank characters occurs. Blanks inserted as padding are not included in the `size=` count.

It is possible to perform normal and non-advancing I/O on the same record or file. For instance, a non-advancing read might read the first few characters of a record and a normal read the remainder.

A particular application of this facility is to write a prompt to a terminal screen and to read from the next character position on the screen without an intervening line-feed:

```
write (*, '(a)', advance='no') 'enter next prime number:'
read  (*, '(i10)') prime_number
```

Non-advancing I/O may be performed only on an external file, and may not be used for `namelist` or list-directed I/O. Note that, as for advancing input/output, several records may be processed by a single statement.

9.13 Edit descriptors

In the description of the possible forms of a format specification in Section 9.4, a few examples of the edit descriptors were given. As mentioned there, edit descriptors give a precise specification of how values are to be converted into a character string on an output device or internal file, or converted from a character string on an input device or internal file to internal representations.

With certain exceptions noted in the following text, edit descriptors in a list are separated by commas, and only in the case where an input/output list is empty or specifies only zero-sized arrays may there be no edit descriptor at all in the format specification.

On a processor that supports upper- and lower-case letters, edit descriptors are interpreted without regard to case. This is also true for numerical and logical input fields; an example is 89AB as a hexadecimal input value.

9.13.1 Repeat counts

Edit descriptors fall into three classes: *data, control,* and *character-string*. The data edit descriptors may be preceded by a repeat count (a nonzero unsigned default integer literal constant), as in the example

```
10f12.3
```

Of the remaining edit descriptors, only the slash edit descriptor (Section 9.13.5) may have an associated repeat count. A repeat count may be applied to a group of edit descriptors, enclosed in parentheses:

```
print '(4(i5,f8.2))', (i(j), a(j), j=1,4)
```

(for integer `i` and real `a`). This is equivalent to writing

```
print '(i5,f8.2,i5,f8.2,i5,f8.2,i5,f8.2)', (i(j), a(j), j=1,4)
```

Repeat counts such as this may be nested:

```
print '(2(2i5,2f8.2))', i(1),i(2),a(1),a(2),i(3),i(4),a(3),a(4)
```

If a format specification without components in parentheses is used with an I/O list that contains more elements than the number of edit descriptors, taking account of repeat counts, then a new record will begin, and the format specification repeated. Further records begin in the same way until the list is exhausted. To print an array of 100 integer elements, 10 elements to a line, the following statement might be used:

```
print '(10i8)', (i(j), j=1,100)
```

Similarly, when reading from an input file, new records would be read until the list is satisfied, a new record being taken from the input file each time the specification is repeated *even if the individual records contain more input data than specified by the format specification.* These superfluous data would be ignored. For example, reading the two records (*b* again stands for a blank)

*bbb*10*bbb*15*bbb*20
*bbb*25*bbb*30*bbb*35

under control of the `read` statement

```
read '(2i5)', i,j,k,l
```

would result in the four integer variables `i`, `j`, `k` and `l` acquiring the values 10, 15, 25 and 30, respectively.

If a format contains components in parentheses, as in

```
'(2i5, 3(i2,2(i1,i3)), 2(2f8.2,i2))'
```

whenever the format is exhausted, a new record is taken and format control reverts to the repeat factor preceding the left parenthesis corresponding to the last-but-one right parenthesis, here `2(2f8.2,i2)`, or to the parenthesis itself if it has no repeat factor. This we call *reversion.*

9.13.2 Data edit descriptors

Values of all the intrinsic data types may be converted by the g edit descriptor. However, for reasons of clarity, it is described last.

Integer values may be converted by means of the i edit descriptor. This comes in a basic form, i*w*, where *w* is a nonzero unsigned default integer literal constant that defines the width of the field. The integer value will be read from or written to this field, adjusted to its right-hand side. If we again designate a blank position by *b* then the value -99 printed under control of the edit descriptor i5 will appear as *bb*-99, the sign counting as one position in the field.

For output, an alternative form of this edit descriptor allows the number of digits that are to be printed to be specified exactly, even if some are leading zeros. The form i*w.m* specifies the width of the field, *w*, and that at least *m* digits are to be output, where *m* is an unsigned default integer literal constant. The value 99 printed under control of the edit descriptor i5.3 would appear as *bb*099. The value of *m* is even permitted to be zero, and the field will be then filled with blanks if the value printed is 0. On input, i*w.m* is interpreted in exactly the same way as i*w*.

For the i and all other numeric edit descriptors, if the output field is too narrow to contain the number to be output, it is filled with asterisks.

Integer values may also be converted by the b*w*, b*w.m*, o*w*, o*w.m*, z*w*, and z*w.m* edit descriptors. These are similar to the i form, but are intended for integers represented in the binary, octal, and hexadecimal number systems, respectively (Section 2.6.1). The external form does not contain the leading letter (b, o, or z) or the delimiters.

Real values may be converted by either e, en, es, or f edit descriptors. The f descriptor we have met in earlier examples. Its general form is f*w.d*, where *w* and *d* are unsigned default integer literal constants which define, respectively, the field width and the number of digits to appear after the decimal point in the output field. The decimal point counts as one position in the field. On input, if the input string has a decimal point, the value of *d* is ignored. Reading the input string *b*9.3729*b* with the edit descriptor f8.3 would cause the value 9.3729 to be transferred. All the digits are used, but roundoff may be inevitable because of the actual physical storage reserved for the value on the computer being used.

There are, in addition, two other forms of input string that are acceptable to the f edit descriptor. The first is an optionally signed string of digits without a decimal point. In this case, the *d* rightmost digits will be taken to be the fractional part of the value. Thus *b*-14629 read under control of the edit descriptor f7.2 will transfer the value -146.29. The second form is the standard default real form of literal constant, as defined in Section 2.6.2, and the variant in which the exponent is signed and e is omitted. In this case, the *d* part of the descriptor is again ignored. Thus the value 14.629e-2 (or 14.629-2), under control of the edit descriptor f9.1, will transfer the value 0.14629. The exponent letter may also be written in upper case.

Values are rounded on output following the normal rules of arithmetic. Thus, the value 10.9336, when output under control of the edit descriptor `f8.3`, will appear as *bb*`10.934`, and under the control of `f4.0` as *b*`11`.

The `e` edit descriptor has two forms, e*w.d* and e*w.dee*, and is more appropriate for numbers with a magnitude below about 0.01, or above 1000. The rules for these two forms for input are identical to those for the f*w.d* edit descriptor. For output with the e*w.d* form of the descriptor, a different character string will be transferred, containing a significand with absolute value less than 1 and an exponent field of four characters that consists of either `E` followed by a sign and two digits or of a sign and three digits. Thus, for $1.234 * 10^{23}$ converted by the edit descriptor `e10.4`, the string *b*`.1234E+24` or *b*`.1234+024` will be transferred. The form containing the exponent letter `E` is not used if the magnitude of the exponent exceeds 99. For instance, `e10.4` would cause the value $1.234 * 10^{-150}$ to be transferred as *b*`.1234-149`. Some processors print a zero before the decimal point.

In the second form of the `e` edit descriptor, e*w.dee*, *e* is an unsigned, nonzero default integer literal constant that determines the number of digits to appear in the exponent field. This form is obligatory for exponents whose magnitude is greater than 999. Thus the value $1.234 * 10^{1234}$ with the edit descriptor `e12.4e4` is transferred as the string *b*`.1234E+1235`. An increasing number of computers are able to deal with these very large exponent ranges. It can also be used if only one exponent digit is desired. For example, the value 1.211 with the edit descriptor `e9.3e1` is transferred as the string *b*`0.121E+1`.

The `en` (***engineering***) edit descriptor is identical to the `e` edit descriptor except that on output the decimal exponent is divisible by three, a nonzero significand is greater than or equal to 1 and less than 1000, and the scale factor (Section 9.13.5) has no effect. Thus, the value 0.0217 transferred under an `en9.2` edit descriptor would appear as `21.70E-03` or `21.70-003`.

The `es` (***scientific***) edit descriptor is identical to the `e` edit descriptor, except that on output the absolute value of a nonzero significand is greater than or equal to 1 and less than 10 and the scale factor (Section 9.13.5) has no effect. Thus, the value 0.0217 transferred under an `es9.2` edit descriptor would appear as `2.17E-02` or `2.17-002`.

Complex values may be edited under control of pairs of `f`, `e`, `en`, or `es` edit descriptors. The two descriptors do not need to be identical. The complex value (0.1,100.) converted under control of `f6.1,e8.1` would appear as *bbb*`0.1`*b*`0.1E+03`. The two descriptors may be separated by character string and control edit descriptors (to be described in Sections 9.13.4 and 9.13.5).

Logical values may be edited using the `l`*w* edit descriptor. This defines a field of width *w* which on input consists of optional blanks, optionally followed by a decimal point, followed by `t` or `f` (or `T` or `F`), optionally followed by additional characters. Thus a field defined by `l7` permits the strings `.true.` and `.false.` to be input. The characters `t` or `f` will be transferred as the values true or false respectively. On output, the character `T` or `F` will appear in the right-most position in the output field.

Character values may be edited using the a edit descriptor in one of its two forms, either a or aw. In the first of the two forms, the width of the input or output field is determined by the actual width of the item in the I/O list, measured in number of characters of whatever kind. Thus, a character variable of length 10, containing the value STATEMENTS, when written under control of the a edit descriptor would appear in a field 10 characters wide, and the non-default character variable of length 4 containing the value 国際標準 would appear in a field 4 characters wide. If, however, the first variable were converted under an a11 edit descriptor, it would be printed with a leading blank, *b*STATEMENTS. Under control of a8, the eight left-most characters only would be written: STATEMEN.

Conversely, with the same variable on input, an a11 edit descriptor would cause the 10 right-most characters in the 11 character-wide input field to be transferred: *b*STATEMENTS would be transferred as STATEMENTS. The a8 edit descriptor would cause the eight characters in the field to be transferred to the eight left-most positions in the variable, and the remaining two would be filled with blanks: STATEMEN would be transferred as STATEMEN*bb*.

All characters transferred under the control of an a or aw edit descriptor have the kind of the I/O list item, and we note that this edit descriptor is the *only* one which can be used to transmit non-default characters to or from a record. In the non-default case, the blank padding character is processor dependent.

The g$w.d$ and g$w.d$ee (***general***) edit descriptor may be used for any intrinsic data type. When used for real or complex types, it is identical to the e edit descriptor except that an output value with magnitude n in the range

$$0.1 - 0.5 \times 10^{-d-1} \leq n < 10^d - 0.5$$

or zero when $d = 0$ are converted as if by an f edit descriptor, and followed by the same number of blanks as the e edit descriptor would have used for the exponent part. The equivalent f edit descriptor is f$w'.d'$, where $w' = w - 4$ for g$w.d$ or w-e-2 for g$w.d$ee, and $d' = d - k$ when n lies in the range

$$10^{k-1}(1 - 0.5 \times 10^{-d}) \leq n < 10^k(1 - 0.5 \times 10^{-d})$$

for $k = 0, 1, \ldots, d$ and $d' = d - 1$ when $n = 0$ and $d > 0$. This form is useful for printing values whose magnitudes are not well known in advance, and where an f conversion is preferred where possible, and an e otherwise.

When the g edit descriptor is used for integer, logical, or character types, it follows the rules of the iw, lw, and aw edit descriptors, respectively.

Finally, values of ***derived types*** are edited by the appropriate sequence of edit descriptors corresponding to the intrinsic types of the ultimate components of the derived type. An example is:

```
type string
   integer             :: length
   character(len=20)   :: word
end type string
type(string) :: text
read(*, '(i2, a)') text
```

9.13.3 Minimal field width editing (Fortran 95 only)

In order to allow output records to contain as little unused space as possible, the `i`, `f`, `b`, `o`, and `z` edit descriptors (Section 9.13.2) may specify, in Fortran 95, a field width of zero, as in `i0` or `f0.3`. This does not denote a zero-width field, but a field that is of the minimum width necessary to contain the output value in question. The programmer does not need to worry that a field with too narrow a width will cause output values to overflow and contain only asterisks.

9.13.4 Character string edit descriptor

A *default character* literal constant without a specified kind parameter can be transferred to an output file by embedding it in the format specification itself, as in the example

```
print "(' This is a format statement')"
```

The string will appear each time the statement is executed. In this descriptor, case is significant. Character string edit descriptors must not be used on input.

9.13.5 Control edit descriptors

It is sometimes necessary to give other instructions to an I/O device than just the width of fields and how the contents of these fields are to be interpreted. For instance, it may be that one wishes to position fields at certain columns or to start a new record without issuing a new `write` command. For this type of purpose, the control edit descriptors provide a means of informing the processor which action has to be taken. Some of these edit descriptors contain information that is used as it is processed; others are like switches, which change the conditions under which I/O takes place from the point where they are encountered, until the end of the processing of the I/O statement containing them (including reversions, Section 9.13.1). These latter descriptors we shall deal with first.

Control edit descriptors setting conditions

Embedded blanks in numeric input fields are treated in one of two ways, either as zero, or as null characters that are squeezed out by moving the other characters in the input field to the right, and adding leading blanks to the field (unless the field is totally blank, in which case it is interpreted as zero). The default is given by the `blank=` specifier (Section 10.3) currently in effect for the unit or is null for an internal file. Whatever the default may then be for a file, it may be overridden during a given format conversion by the `bn` (blanks null) and `bz` (blanks zero) edit descriptors. Let us suppose that the mode is that blanks are treated as zeros. The input string *bb*1*b*4 converted by the edit descriptor `i5` would transfer the value 104. The same string converted by `bn,i5` would give 14. A `bn` or `bz` edit descriptor

switches the mode for the rest of that format specification, or until another bn or bz edit descriptor is met. The bn and bz edit descriptors have no effect on output.

Negative numerical values are always written with ***leading signs*** on output. For positive quantities other than exponents, whether the signs are written depends on the processor. The ss (sign suppress) edit descriptor suppresses leading plus signs, that is the value 99 printed by i5 is *bbb*99 and 1.4 is printed by e10.2 as *bb*0.14E+01. To switch on plus sign printing, the sp (sign print) edit descriptors may be used: the same numbers written by sp,i5,e10.2 become *bb*+99 and *b*+0.14E+01. The s edit descriptor restores the option to the processor. An ss, sp, or s will remain in force for the remainder of the format specification, unless another ss, sp, or s edit descriptor is met. These edit descriptors provide complete control over sign printing, and are useful for producing coded outputs which have to be compared automatically, on two different computers.

Scale factors apply to the input of real quantities under the e, f, en, es, and g edit descriptors, and are a means of scaling the input values. Their form is *k*p, where *k* is a default integer literal constant specifying the scale factor. The value is zero at the beginning of execution of the statement. The effect is that any quantity which does not have an exponent field will be reduced by a factor 10^k. Quantities with an exponent are not affected.

The scale factor *k*p also affects output with e, f or g editing, but has no effect with en or es editing. Under control of an f edit descriptor, the quantity will be multiplied by a factor 10^k. Thus, the number 10.39 output by an f6.0 edit descriptor following the scale factor 2p will appear as *b*1039.. With the e edit descriptor, and with g where the e style editing is taken, the quantity is transferred with the exponent reduced by *k*, and the significand multiplied by 10^k. Thus $0.31 * 10^3$, written after a 2p edit descriptor under control of e9.2, will appear as 31.00E+01. This gives a better control over the output style of real quantities which otherwise would have no significant digits before the decimal point.

The comma between a scale factor and an immediately following f, e, en, es, or g edit descriptor may be omitted, but we do not recommend that practice since it suggests that the scale factor applies only to the next edit descriptor, whereas in fact it applies throughout the format until another scale factor is encountered.

Control edit descriptors for immediate processing

Tabulation in an input or output field can be achieved using the edit descriptors t*n*, tr*n* (and *n*x), and tl*n*, where *n* is a positive default integer literal constant. These state, respectively, that the next part of the I/O should begin at position *n* in the current record (where the *left tab limit* is position 1), or at *n* positions to the right of the current position, or at *n* positions to the left of the current position (the left tab limit if the current position is less than or equal to *n*). Let us suppose that, following an advancing read, we read an input record *bb*9876 with the following statement:

```
read (*, '(t3, i4, tl4, i1, i2)') i, j, k
```

The format specification will move a notional pointer firstly to position 3, whence i will be read. The variable i will acquire the value 9876, and the notional pointer is then at position 7. The edit descriptor t14 moves it left four positions, back to position 3. The quantities j and k are then read, and they acquire the values 9 and 87, respectively. These edit descriptors cause replacement on output, or multiple reading of the same items in a record on input. On output, any gaps ahead of the last character actually written are filled with spaces. If any character that is skipped by one of the descriptors is of other than default type, the positioning is processor dependent.

If the current record is the first one processed by the I/O statement and follows non-advancing I/O that left the file positioned within a record, the next character is the left tab limit; otherwise, the first character of the record is the left tab limit.

The *n*x edit descriptor is equivalent to the tr*n* edit descriptor. It is often used to place spaces in an output record. For example, to start an output record with a blank by this method, one writes

```
fmt= '(1x,....)'
```

Spaces such as this can precede a data edit descriptor, but 1x,i5 is not, for instance, exactly equivalent to i6 on output, as any value requiring the full six positions in the field will not have them available in the former case.

The t and x edit descriptors never cause replacement of a character already in an output record, but merely cause a change in the position within the record such that such a replacement might be caused by a subsequent edit descriptor.

New records may be started at any point in a format specification by means of the slash (/) edit descriptor. This edit descriptor, although described here, may in fact have repeat counts; to skip, say, three records one can write either /,/,/ or 3/. On input, a new record will be started each time a / is encountered, even if the contents of the current record have not all been transferred. Reading the two records

*bbb*99*bbb*10
*bb*100*bbb*11

with the statement

```
read '(bz,i5,i3,/,i5,i3,i2)', i, j, k, l, m
```

will cause the values 99, 0, 100, 0 and 11 to be transferred to the five integer variables, respectively. This edit descriptor does not need to be separated by a comma from a preceding edit descriptor, unless it has a repeat count; it does not ever need to be separated by a comma from a succeeding edit descriptor.

The result of writing with a format containing a sequence of, say, four slashes, as represented by

```
print '(i5,4/,i5)', i, j
```

is to separate the two values by three blank records (the last slash starts the record containing j); if i and j have the values 99 and 100, they would appear as

```
bbb99
b
b
b
bb100
```

A slash edit descriptor written to an internal file will cause the following values to be written to the next element of the character array specified for the file. Each such element corresponds to a record, and the number of characters written to a record must not exceed its length.

Colon editing is a means of terminating format control if there are no further items in an I/O list. In particular, it is useful for preventing further output of character strings used for annotation if the output list is exhausted. Consider the following output statement, for an array l(3):

```
print '(" l1 = ", i5, :, " l2 = ", i5, :," l3 = ", i5)', &
      (l(i) ,i=1,n)
```

If n has the value 3, then three values are printed. If n has the value 1 then, without the colons, the following output string would be printed:

```
l1 = 59 l2 =
```

The colon, however, stops the processing of the format, so that the annotation for the absent second value is not printed. This edit descriptor need not be separated from a neighbour by a comma. It has no effect if there are further items in the I/O list.

9.14 Unformatted I/O

The whole of this chapter has so far dealt with formatted I/O. The internal representation of a value may differ from the external form, which is always a character string contained in an input or output record. The use of formatted I/O involves an overhead for the conversion between the two forms, and often a roundoff error too. There is also the disadvantage that the external representation usually occupies more space on a storage medium than the internal representation. These three actual or potential drawbacks are all absent when unformatted I/O is used. In this form, the internal representation of a value is written exactly as it stands to the storage medium, and can be read back directly with neither roundoff nor conversion overhead. Here, a value of derived type is treated as a whole and is not equivalent to a list of its ultimate components. This is another reason for the rule (Section 9.3) that it must not have a pointer component at any level of component selection.

This type of I/O should be used in all cases where the records are generated by a program on one computer, to be read back on the same computer or another computer using the same internal number representations. Only when this is not the case, or when the data have to be visualized in one form or another, should

formatted I/O be used. The records of a file must all be formatted or all be unformatted (apart from the endfile record).

Unformatted I/O has the incidental advantage of being simpler to program since no complicated format specifications are required. The forms of the `read` and `write` statements are the same as for formatted I/O, but without any `fmt=` or `nml=` specifier:

```
read(4) q
write(nout, iostat=ios, err=110) a
```

Non-advancing I/O is not available (in fact, an `advance=` specifier is not allowed).

Each `read` or `write` statement transfers exactly one record. The file must be an external file. The number of values specified by the input list of a `read` statement must not exceed the number of values available in the current record.

On output to a file connected for sequential access, a record of sufficient length is created. On input, the type and type parameters of each entity in the list must agree with those of the value in the record, except that two reals may correspond to one complex when all three have the same kind parameter.

9.15 Direct-access files

The only type of file organization that we have so far dealt with is the sequential file, which has a beginning and an end, and which contains a sequence of records, one after the other. Fortran permits another type of file organization known as *direct access* (or sometimes as random access or indexed). All the records have the same length, each record is identified by an index number, and it is possible to write, read, or re-write any specified record without regard to position. (In a sequential file, only the last record may be rewritten without losing other records; in general, records in sequential files cannot be replaced.) The records are either all formatted or all unformatted.

By default, any file used by a Fortran program is a sequential file, unless declared to be direct access. Tthis declaration has to be made using the `access='direct'` and `recl=`*rl* specifiers of the open statement, which is described in the next chapter, (*rl* is the length of a record in the file). Once this declaration has been made, reading and writing, whether formatted or unformatted, proceeds as described for sequential files, except for the addition of a `rec=`*i* specifier to the `read` and `write` statements, where *i* is a scalar integer expression whose value is the index number of the record concerned. An `end=` specifier is not permitted. Usually, a data transfer statement for a direct-access file accesses a single record, but during formatted I/O any slash edit descriptor increases the record number by one and causes processing to continue at the beginning of this record. A sequence of statements to write, read, and replace a given record is given in Figure 9.5.

The file must be an external file and `namelist` formatting, list-directed formatting, and non-advancing I/O are all unavailable.

Figure 9.5

```
      integer, parameter :: nunit=2, len=100
      integer            :: i, length
      real               :: a(len), b(len+1:2*len)
      :
      inquire (iolength=length) a          ! See Section 10.5
      open (nunit, access='direct', recl=length)
                                           ! See Section 10.3
      :
!     Write array B to direct-access file in record 14
      write (nunit, rec=14) b
      :
!
!     Read the array back into array a
      read (nunit, rec=14) a
      :
      do i = 1, len/2
         a(i) = i
      end do
!
!     Replace modified record
      write (nunit, rec=14) a
```

Direct-access files are particularly useful for applications which involve lots of hopping around inside a file, or where records need to be replaced, for instance in data base applications. A weakness is that the length of all the records must be the same, though on formatted output, the record is padded with blanks if necessary. For unformatted output, if the record is not filled, the remainder is undefined.

This simple and powerful facility allows much clearer control logic to be written than is the case for a sequential file which is repeatedly read, backspaced, or rewound. Only when direct-access files become large may problems of long access times become evident on some computer systems, and this point should always be investigated before heavy investments are made in programming large direct-access file applications.

Some computer systems allow the same file to be regarded as sequential or direct access according to the specification in the open statement or its default. The standard, therefore, regards this as a property of the connection rather than of the file. In this case, the order of records, even for sequential I/O, is that determined by the direct-access record numbering.

9.16 Execution of a data transfer statement

So far, we have used simple illustrations of data transfer statements without dependencies. However, some forms of dependency are permitted and can be very useful. For example, the statement

```
read (*, *) n, a(1:n)                    ! n is an integer
```

allows the length of an array section to be part of the data.

With dependencies in mind, the order in which operations are executed is important. It is as follows:

i) identify the unit;

ii) establish the format (if any);

iii) position the file ready for the transfer (if required);

iv) transfer data between the file and the I/O list or namelist;

v) position the file following the transfer (if required);

vi) cause the `iostat` and `size` variables (if present) to become defined.

The order of transfer of namelist input is that in the input records. Otherwise, the order is that of the I/O list or `namelist`. Each input item is processed in turn, and may affect later subobjects and implied-do indices. All expressions within an I/O list item are determined at the beginning of processing of the item. If an entity is specified more than once during execution of a namelist input statement, the later value overwrites the earlier value. Any zero-sized array or zero-length implied-do list is ignored.

When an input item is an array, no element of the array is permitted to affect the value of an expression within the item. For example, the cases shown in Figure 9.6 are not permitted. This prevents dependencies occurring within the item itself.

Figure 9.6

```
integer :: j(10)
:
read *, j(j)                             ! Not permitted
read *, j(j(1):j(10))                    ! Not permitted
```

In the case of an internal file, an I/O item must not be in the file or associated with it. Nor may an input item contain or be associated with any portion of the established format.

Finally, a function reference must not appear in an expression anywhere in an I/O statement if it causes another I/O statement to be executed.

9.17 Summary

This chapter has begun the description of Fortran's extensive I/O facilities. It has covered the formatted I/O statements, and their associated format specifications, and then turned to unformatted I/O and direct-access files.

The syntax of the `read` and `write` statements has been introduced gradually. The full syntax is

`read` (*control-list*) [*input-list*]

and

`write` (*control-list*) [*output-list*]

where *control-list* contains one or more of the following:

`[unit=]` *u*,
`[fmt=]` *fmt*,
`[nml=]` *nml-name*,
`rec=` *i*,
`iostat=` *ios*,
`err=` *error-label*,
`end=` *end-label*,
`advance=` *advance*,
`size=` *size*,
`eor=` *eor-label*.

A *control-list* must include a unit specifier and must not include any specifier more than once. The `iostat` and `size` variables must not be associated with each other (for instance be identical), nor with any entity being transferred, nor with any *do-var* of an implied-do list of the same statement. If either of these variables is an array element, the subscript value must not be affected by the data transfer, implied-do processing, or the evaluation of any other specifier in the statement.

There are many detailed changes with respect to Fortran 77, often to support new features in other parts of the language, such as derived types. Other new features are `namelist`, non-advancing I/O, the `b`, `o`, `z`, `en` and `es` edit descriptors, and the generalization of the `g` edit descriptor. In Fortran 95, minimal field width editing is new.

9.18 Exercises

1. Write suitable `print` statements to print the name and contents of each of the following arrays:

a) `real :: grid d(10,10)`, 10 elements to a line (assuming the values are between 1.0 and 100.0);

b) `integer :: list(50)`, the odd elements only;

c) `character(len=10) :: titles(20)`, two elements to a line;

d) `real :: power(10)`, five elements to a line in engineering notation;

e) `logical :: flags(10)`, on one line;

f) `complex :: plane(5)`, on one line.

2. Write statements to output the state of a game of tic-tac-toe (noughts-and-crosses) to a unit designated by the variable `unit`.

3. Write a program which reads an input record of up to 132 characters into an internal file and classifies it as a Fortran comment line with no statement, an initial line without a statement label, an initial line with a statement label, a continuation line, or a line containing multiple statements.

4. Write separate list-directed input statements to fill each of the arrays of Exercise 1. For each statement write a sample first input record.

5. Write the function `get_char`, to read single characters from a formatted, sequential file, ignoring any record structure.

10. Operations on external files

10.1 Introduction

So far we have discussed the topic of external files in a rather superficial way. In the examples of the various I/O statements in the previous chapter, an implicit assumption has always been made that the specified file was actually available, and that records could be written to it and read from it. For sequential files, the file control statements described in the next section further assume that it can be positioned. In fact, these assumptions are not necessarily valid. In order to define explicitly and to test the status of external files, three file status statements are provided: `open`, `close`, and `inquire`. Before beginning their description, however, two new definitions are required.

A computer system contains, among other components, a CPU and a storage system. Modern storage systems are usually based on some form of disc, which is used to store files for long or short periods of time. The execution of a computer program is, by comparison, a transient event. A file may exist for years, whereas programs run for only seconds or minutes. In Fortran terminology, a file is said to *exist* not in the sense we have just used, but in the restricted sense that it exists as a file *to which the program might have access.* In other words, if the program is prohibited from using the file because of a password protection system, or because some necessary action has not been taken in the job control language which is controlling the execution of the program, the file 'does not exist'.

A file which exists for a running program may be empty and may or may not be *connected* to that program. The file is connected if it is associated with a unit number known to the program. Such connection is usually made by executing an `open` statement for the file, but many computer systems will *pre-connect* certain files which any program may be expected to use, such as terminal input and output. Thus we see that a file may exist but not be connected. It may also be connected but not exist. This can happen for a pre-connected new file. The file will only come into existence (be *created*) if some other action is taken on the file: executing an `open`, `write`, `print`, or `endfile` statement. A unit must not be connected to more than one file at once, and a file must not be connected to more than one unit at once.

There are a number of other points to note with respect to files:

- The set of allowed names for a file is processor dependent.

- Both sequential and direct access may be available for some files, but normally a file is limited to one or the other.

- A file never contains both formatted and unformatted records.

Finally, we note that no statement described in this chapter applies to internal files.

10.2 File positioning statements

When reading or writing an external file that is connected for sequential access, whether formatted or unformatted, it is sometimes necessary to perform other control functions on the file in addition to input and output. In particular, one may wish to alter the current position, which may be within a record, between records, ahead of the first record (at the *initial point*), or after the last record (at its *terminal point*). The following three statements are provided for these purposes.

10.2.1 The backspace statement

It can happen in a program that a series of records is being written and that, for some reason, the last record written should be replaced by a new one, that is be overwritten. Similarly, when reading records, it may be necessary to reread the last record read, or to check-read a record which has just been written. For this purpose, Fortran provides the backspace statement, which has the syntax

```
backspace u
```

or

```
backspace ([unit=]u [,iostat=ios] [,err=error-label])
```

where *u* is a scalar integer expression whose value is the unit number, and the other optional specifiers have the same meaning as for a read statement. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier.

The action of this statement is to position the file before the current record if it is positioned within a record, or before the preceding record if it is positioned between records. An attempt to backspace when already positioned at the beginning of a file results in no change in the file's position. If the file is positioned after an endfile record (Section 10.2.3), it becomes positioned before that record. It is not possible to backspace a file that does not exist, nor to backspace over a record written by a list-directed or namelist output statement (Sections 9.9 and 9.10). A series of backspace statements will backspace over the corresponding number of records. This statement is often very costly in computer resources and should be used as little as possible.

10.2.2 The rewind statement

In an analogous fashion to rereading, rewriting, or check-reading a record, a similar operation may be carried out on a complete file. For this purpose the `rewind` statement,

```
rewind u
```

or

```
rewind ([unit=]u [,iostat=ios] [,err=error-label])
```

may be used to reposition a file, whose unit number is specified by the scalar integer expression *u*. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier. If the file is already at its beginning, there is no change in its position. The statement is permitted for a file that does not exist, and has no effect.

10.2.3 The endfile statement

The end of a file connected for sequential access is normally marked by a special record which is identified as such by the computer hardware, and computer systems ensure that all files written by a program are correctly terminated by such an *endfile record*. In doubtful situations, or when a subsequent program step will reread the file, it is possible to write an endfile record explicitly using the `endfile` statement:

```
endfile u
```

or

```
endfile ([unit=]u [,iostat=ios] [,err=error-label])
```

where *u*, once again, is a scalar integer expression specifying the unit number. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier. The file is then positioned after the endfile record. This endfile record, if subsequently read by a program, must be handled using the `end=`*end-label* specifier of the `read` statement, otherwise program execution will normally terminate. Prior to data transfer, a file must not be positioned after an endfile record, but it is possible to backspace or rewind across an endfile record, which allows further data transfer to occur. An endfile record is written automatically whenever either a backspace or rewind operation follows a write operation as the next operation on the unit, or the file is closed by execution of a `close` statement (Section 10.4), by an `open` statement for the same unit (Section 10.3), or by normal program termination.

If the file may also be connected for direct access, only the records ahead of the endfile record are considered to have been written and only these may be read during a subsequent direct-access connection.

Note that if a file is connected to a unit but does not exist for the program, it will be made to exist by executing an `endfile` statement on the unit.

10.2.4 Data transfer statements

Execution of a data transfer statement (`read`, `write`, or `print`) also affects the file position. If it is between records, it is moved to the start of the next record. Data transfer then takes place, which usually moves the position. No further movement occurs for non-advancing access. For advancing access, the position finally moves to follow the last record transferred.

10.3 The open statement

The `open` statement is used to connect an external file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain properties of a connection. The syntax is

```
open ([unit=]u [,olist])
```

where *u* is a scalar integer expression specifying the external file unit number, and *olist* is a list of optional specifiers. If the unit is specified with `unit=`, it may appear in *olist*. A specifier must not appear more than once. In the specifiers, all entities are scalar and all characters are of default kind. In character expressions, any trailing blanks are ignored and, except for `file=`, any upper-case letters are converted to lower case. The specifiers are

`iostat=` *ios*, where *ios* is a default integer variable which is set to zero if the statement is correctly executed, and to a positive value otherwise.

`err=` *error-label*, where *error-label* is the label of a statement in the same scoping unit to which control will be transferred in the event of an error occurring during execution of the statement.

`file=` *fln*, where *fln* is a character expression that provides the name of the file. If this specifier is omitted and the unit is not connected to a file, the `status=` specifier must be specified with the value `scratch` and the file connected to the unit will then depend on the computer system. Whether the interpretation is case sensitive varies from system to system.

`status=` *st*, where *st* is a character expression that provides the value `old`, `new`, `replace`, `scratch`, or `unknown`. The `file=` specifier must be present if `new` or `replace` is specified or if `old` is specified and the unit is not connected; the `file=` specifier must not be present if `scratch` is specified. If `old` is specified, the file must already exist; if `new` is specified, the file must not already exist, but will be brought into existence by the action of the `open` statement. The status of the file then becomes `old`. If `replace` is specified and the file does not already exist, the file is created; if the file does exist, the file is deleted, and a new file is created with the same name. In each case the status is changed to `old`. If the value `scratch` is specified, the file is created and becomes connected, but it cannot be kept after completion of

the program or execution of a `close` statement (Section 10.4). If `unknown` is specified, the status of the file is system dependent. This is the default value of the specifier, if it is omitted.

`access=` *acc*, where *acc* is a character expression that provides one of the values `sequential` or `direct`. For a file which already exists, this value must be an allowed value. If the file does not already exist, it will be brought into existence with the appropriate access method. If this specifier is omitted, the value `sequential` will be assumed.

`form=` *fm*, where *fm* is a character expression that provides the value `formatted` or `unformatted`, and determines whether the file is to be connected for formatted or unformatted I/O. For a file which already exists, the value must be an allowed value. If the file does not already exist, it will be brought into existence with an allowed set of forms that includes the specified form. If this specifier is omitted, the default is `formatted` for sequential access and `unformatted` for direct-access connection.

`recl=` *rl*, where *rl* is an integer expression whose value must be positive. For a direct-access file, it specifies the length of the records, and is obligatory. For a sequential file, it specifies the maximum length of a record, and is optional with a default value that is processor dependent. For formatted files, the length is the number of characters for records that contain only default characters; for unformatted files it is system dependent but the `inquire` statement (Section 10.5) may be used to find the length of an I/O list. In either case, for a file which already exists, the value specified must be allowed for that file. If the file does not already exist, the file will be brought into existence with an allowed set of record lengths that includes the specified value.

`blank=` *bl*, where *bl* is a character expression that provides the value `null` or `zero`. This connection must be for formatted I/O. This specifier sets the default for the interpretation of blanks in numeric input fields, as discussed in the description of the `bn` and `bz` edit descriptors (Section 9.13.5). If the value is `null`, such blanks will be ignored (except that a completely blank field is interpreted as zero). If the value is `zero`, such blanks will be interpreted as zeros. If the specifier is omitted, the default is `null`.

`position=` *pos*, where *pos* is a character expression that provides the value `asis`, `rewind`, or `append`. The access method must be sequential, and if the specifier is omitted the default value `asis` will be assumed. A new file is positioned at its initial point. If `asis` is specified and the file exists and is already connected, the file is opened without changing its position; if `rewind` is specified the file is positioned at its initial point; if `append` is specified and the file exists, it is positioned ahead of the endfile record if it has one (and otherwise at its terminal point). For a file which exists but

is not connected, the effect of the asis specifier on the file's position is unspecified.

action= *act*, where *act* is a character expression that provides the value read, write, or readwrite. If read is specified, the write, print and endfile statements must not be used for this connection; if write is specified, the read statement must not be used (and backspace and position='append' may fail on some systems); if readwrite is specified, there is no restriction. If the specifier is omitted, the default value is processor dependent.

delim= *del* where *del* is a character expression that provides the value quote, apostrophe, or none. If apostrophe or quote is specified, the corresponding character will be used to delimit character constants written with list-directed or namelist formatting, and it will be doubled where it appears within such a character constant; also, non-default character values will be preceded by kind values. No delimiting character is used if none is specified, nor does any doubling take place. The default value if the specifier is omitted is none. This specifier may appear only for formatted files.

pad= *pad*, where *pad* is a character expression that provides the value yes or no. If yes is specified, a formatted input record will be regarded as padded out with blanks whenever an input list and the associated format specify more data than appear in the record. (If no is specified, the length of the input record must not be less than that specified by the input list and the associated format, except in the presence of an advance='no' specifier and either an eor= or an iostat= specification.) The default value if the specifier is omitted is yes. For non-default characters, the blank padding character is processor dependent.

An example of an open statement is

```
open (2, iostat=ios, err=99, file='cities',                    &
      status='new', access='direct', recl=100)
```

which brings into existence a new, direct-access, unformatted file named cities, whose records have length 100. The file is connected to unit number 2. Failure to execute the statement correctly will cause control to be passed to the statement labelled 99, where the value of ios may be tested.

The open statements in a program are best collected together in one place, so that any changes which might have to be made to them when transporting the program from one system to another can be carried out without having to search for them. Regardless of where they appear, the connection may be referenced in any program unit of the program.

The purpose of the open statement is to connect a file to a unit. If the unit is, however, already connected to a file then the action may be different. If the file= specifier is omitted, the default is the name of the connected file. If the file in question does not exist, but is pre-connected to the unit, then all the properties

specified by the open statement become part of the connection. If the file is already connected to the unit, then of the existing attributes only the blank=, delim=, pad=, err=, and iostat= specifiers may have values different from those already in effect. If the unit is already connected to another file, the effect of the open statement includes the action of a prior close statement on the unit (without a status= specifier, see next section).

A file already connected to one unit must not be specified for connection to another unit.

In general, by repeated execution of the open statement on the same unit, it is possible to process in sequence an arbitrarily high number of files, whether they exist or not, as long as the restrictions just noted are observed.

10.4 The close statement

The purpose of the close statement is to disconnect a file from a unit. Its form is

```
close ([unit=]u [,iostat=ios] [,err=error-label] [,status=st])
```

where *u, ios,* and *error-label* have the same meanings as described in the previous section for the open statement. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier.

The function of the status= specifier is to determine what will happen to the file once it is disconnected. The value of *st*, which is a scalar default character expression, may be either keep or delete, ignoring any trailing blanks and converting any upper-case letters to lower case. If the value is keep, a file that exists continues to exist after execution of the close statement, and may later be connected again to a unit. If the value is delete, the file no longer exists after execution of the statement. In either case, the unit is free to be connected again to a file. The close statement may appear anywhere in the program, and if executed for a non-existing or unconnected unit, acts as a 'do nothing' statement. The value keep must not be specified for files with the status scratch.

If the status= specifier is omitted, its default value is keep unless the file has status scratch, in which case the default value is delete. On normal termination of execution, all connected units are closed, as if close statements with omitted status= specifiers were executed.

An example of a close statement is

```
close (2, iostat=ios, err=99, status='delete')
```

10.5 The inquire statement

The status of a file can be defined by the operating system prior to execution of the program, or by the program itself during execution, either by an open statement or by some action on a pre-connected file which brings it into existence. At any time during the execution of a program it is possible to inquire about the status and

attributes of a file using the `inquire` statement. Using a variant of this statement, it is similarly possible to determine the status of a unit, for instance whether the unit number exists for that system (that is, whether it is an allowed unit number), whether the unit number has a file connected to it and, if so, which attributes that file has. Another variant permits an inquiry about the length of an output list when used to write an unformatted record.

Some of the attributes that may be determined by use of the `inquire` statement are dependent on others. For instance, if a file is not connected to a unit, it is not meaningful to inquire about the form being used for that file. If this is nevertheless attempted, the relevant specifier is undefined.

The three variants are known as `inquire` by file, `inquire` by unit, and `inquire` by output list. In the description of the `inquire` statement which follows, the first two variants will be described together. Their forms are

```
inquire ([unit=]u, ilist)
```

for `inquire` by unit, where *u* is a scalar integer expression specifying an external unit, and

```
inquire ( file=fln, ilist)
```

for `inquire` by file, where *fln* is a scalar character expression whose value, ignoring any trailing blanks, provides the name of the file concerned. Whether the interpretation is case sensitive is system dependent. If the unit or file is specified by keyword, it may appear in *ilist*. A specifier must not occur more than once in the list of optional specifiers, *ilist*. All assignments occur following the usual rules, and all values of type character, apart from that for the `name=` specifier, are in upper case. The specifiers, in which all variables are scalar and of default kind, are

`iostat=` *ios* and `err=` *error-label*, have the meanings described for them in the `open` statement in Section 10.3. The `iostat=` variable is the only one which is defined if an error condition occurs during the execution of the statement.

`exist=` *ex*, where *ex* is a logical variable. The value true is assigned to *ex* if the file (or unit) exists, and false otherwise.

`opened=` *open*, where *open* is a logical variable. The value true is assigned to *open* if the file (or unit) is connected to a unit (or file), and false otherwise.

`number=` *num*, where *num* is an integer variable that is assigned the value of the unit number connected to the file, or -1 if no unit is connected to the file.

`named=` *nmd* and `name=` *nam*, where *nmd* is a logical variable that is assigned the value true if the file has a name, and false otherwise. If the file has a name, the character variable *nam* will be assigned the name. This value is not necessarily the same as that given in the `file` specifier, if used, but may be qualified in some way. However, in all cases it is a name which is valid

for use in a subsequent open statement, and so the `inquire` can be used to determine the actual name of a file before connecting it. Whether the file name is case sensitive is system dependent.

`access=` *acc*, where *acc* is a character variable that is assigned one of the the values `SEQUENTIAL` or `DIRECT` depending on the access method for a file that is connected, and `UNDEFINED` if there is no connection.

`sequential=` *seq* and `direct=` *dir*, where *seq* and *dir* are character variables that are assigned the value `YES`, `NO`, or `UNKNOWN`, depending on whether the file *may* be opened for sequential or direct access respectively, or whether this cannot be determined.

`form=` *frm*, where *frm* is a character variable that is assigned one of the values `FORMATTED` or `UNFORMATTED`, depending on the form for which the file is actually connected, and `UNDEFINED` if there is no connection.

`formatted=` *fmt* and `unformatted=` *unf*, where *fmt* and *unf* are character variables that are assigned the value `YES`, `NO`, or `UNKNOWN`, depending on whether the file *may* be opened for formatted or unformatted access, respectively, or whether this cannot be determined.

`recl=` *rec*, where *rec* is an integer variable that is assigned the value of the record length of a file connected for direct access, or the maximum record length allowed for a file connected for sequential access. The length is the number of characters for formatted records containing only characters of default type, and system dependent otherwise. If there is no connection, *rec* becomes undefined.

`nextrec=` *nr*, where *nr* is an integer variable that is assigned the value of the number of the last record read or written, plus one. If no record has been yet read or written, it is assigned the value 1. If the file is not connected for direct access or if the position is indeterminate because of a previous error, *nr* becomes undefined.

`blank=` *bl*, where *bl* is a character variable that is assigned the value `NULL` or `ZERO`, depending on whether the blanks in numeric fields are by default to be interpreted as null fields or zeros, respectively, and `UNDEFINED` if there is either no connection, or if the connection is not for formatted I/O.

`position=` *pos*, where *pos* is a character variable that is assigned the value `REWIND`, `APPEND`, or `ASIS`, as specified in the corresponding `open` statement, if the file has not been repositioned since it was opened. If there is no connection, or if the file is connected for direct access, the value is `UNDEFINED`. If the file has been repositioned since the connection was established, the value is processor dependent (but must not be `REWIND` or `APPEND` unless that corresponds to the true position).

`action=` *act*, where *act* is a character variable that is assigned the value `READ`, `WRITE`, or `READWRITE`, according to the connection. If there is no connection, the value assigned is `UNDEFINED`.

`read=` *rd*, where *rd* is a character variable that is assigned the value `YES`, `NO` or `UNKNOWN` according to whether read is allowed, not allowed, or is undetermined for the file.

`write=` *wr*, where *wr* is a character variable that is assigned the value `YES`, `NO` or `UNKNOWN` according to whether write is allowed, not allowed, or is undetermined for the file.

`readwrite=` *rw*, where *rw* is a character variable that is assigned the value `YES`, `NO` or `UNKNOWN` according to whether read/write is allowed, not allowed, or is undetermined for the file.

`delim=` *del*, where *del* is a character variable that is assigned the value `QUOTE`, `APOSTROPHE`, or `NONE`, as specified by the corresponding `open` statement (or by default). If there is no connection, or if the file is not connected for formatted I/O, the value assigned is `UNDEFINED`.

`pad=` *pad*, where *pad* is a character variable that is assigned the value `YES` if so specified by the corresponding `open` statement (or by default), and otherwise `NO`.

A variable that is a specifier in an `inquire` statement or is associated with one must not appear in another specifier in the same statement.

The third variant of the `inquire` statement, `inquire` by I/O list, has the form

```
inquire (iolength=length) olist
```

where *length* is a scalar integer variable of default kind and is used to determine the length of an unformatted output list in processor-dependent units, and might be used to establish whether, for instance, an output list is too long for the record length given in the `recl=` specifier of an `open` statement, or be used as the value of the length to be supplied to a `recl=` specifier, (see Figure 9.5 in Section 9.15).

An example of the `inquire` statement, for the file opened as an example of the `open` statement in Section 10.3, is

```
logical             :: ex, op
character (len=11) :: nam, acc, seq, frm
integer             :: irec, nr
inquire (2, err=99, exist=ex, opened=op, name=nam, access=acc, &
    sequential=seq, form=frm, recl=irec, nextrec=nr)
```

After successful execution of this statement, the variables provided will have been assigned the following values:

```
ex      .true.
op      .true.
nam      citiesbbbbb
acc      DIRECTbbbbb
seq      NObbbbbbbbb
frm      UNFORMATTED
irec     100
nr       1
```

(assuming no intervening read or write operations).

The three I/O status statements just described are perhaps the most indigestible of all Fortran statements. They provide, however, a powerful and portable facility for the dynamic allocation and deallocation of files, completely under program control, which is far in advance of that found in any other programming language suitable for scientific applications.

10.6 Summary

This chapter has completed the description of the input/output features begun in the previous chapter, and together they provide a complete reference to all the facilities available. The features new since Fortran 77 are `inquire` by I/O list and the additional specifiers for the `open` and `inquire` statements: `position`, `action`, `delim`, `pad`, `read`, `write`, and `readwrite`. There have also been detailed changes to accommodate other features of the language.

10.7 Exercises

1. A direct-access file is to contain a list of names and initials, to each of which there corresponds a telephone number. Write a program which opens a sequential file and a direct-access file, and copies the list from the sequential file to the direct-access file, closing it for use in another program.

Write a second program which reads an input record containing either a name or a telephone number (from a terminal if possible), and prints out the corresponding entry (or entries) in the direct-access file if present, and an error message otherwise. Remember that names are as diverse as Wu, O'Hara and Trevington-Smythe, and that it is insulting for a computer program to corrupt or abbreviate people's names. The format of the telephone numbers should correspond to your local numbers, but the actual format used should be readily modifiable to another.

11. Other features

11.1 Introduction

This chapter describes features that are redundant within Fortran 90 and whose use we deprecate. They might become obsolescent in a future revision, but this is a decision that can be made only within the standardization process. We note again that this decision to group certain features into a final chapter and to deprecate their use is ours alone, and does not have the actual or implied approval of either WG5 or J3.

Our deprecated features fall into three groups:

- those linked to storage association;
- those introduced into Fortran 90 because of strong public pressure, but for which there are better ways to achieve the same effects; and
- those which are made redundant by newer features.

Each description mentions how the feature concerned may be effectively replaced by a newer feature or features.

11.2 Storage association

11.2.1 Storage units

Storage units are the fixed units of physical storage allocated to certain data. There is a storage unit called *numeric* for any non-pointer scalar of the default real, default integer, and default logical types, and a storage unit called *character* for any non-pointer scalar of type default character and character length 1. Non-pointer scalars of type default complex or double precision real (Section 11.4.1) occupy two contiguous numeric storage units. Non-pointer scalars of type default character length *len* occupy *len* contiguous character storage units.

As well as numeric and character storage units, there are a large number of *unspecified* storage units. A non-pointer scalar object of type non-default integer, real other than default or double precision, non-default logical, non-default complex, or non-default character of any particular length occupies a single unspecified storage unit that is different for each case. An object with the

`pointer` attribute has an unspecified storage unit, different from that of any non-pointer object and different for each combination of type, type parameters, and rank. The standard makes no statement about the relative sizes of all these storage units and permits storage association to take place only between objects with the same category of storage unit.

An array of intrinsic type occupies a sequence of storage units, one for each element, in array element order.

Objects of derived type have no storage association, each occupying an unspecified storage unit that is different in each case, except where a given type contains a `sequence` statement making it a *sequence type*:

```
type storage
   sequence
   integer i          ! First numeric storage unit;
   real a(0:999)      ! subsequent 1000 numeric storage units.
end type storage
```

Should any other derived types appear in such a definition, they too must be sequence types. A scalar of sequence type occupies a storage sequence that consists of the concatenation of the storage sequences of its components. An array of sequence type occupies a storage sequence that consists of the concatenation of the storage sequences of its elements.

A sequence type whose ultimate components are non-pointers of type default integer, default real, double precision real, default complex and default logical has *numeric storage association*. Similarly, a sequence type whose ultimate components are non-pointers of type default character has *character storage association*.

A derived type with the `sequence` attribute may have private components:

```
type storage
   private
   sequence
   integer i
   real a(0:999)
end type storage
```

The `private` and `sequence` statements may be interchanged but must be the second and third statements of the type definition.

Two type definitions in different scoping units define the same data type if they have the same name[1], both have the `sequence` attribute, and they have components that are not `private` and agree in order, name, and attributes. However, such a practice is prone to error and offers no advantage over having a single definition in a module and accessed by use association.

[1] If one or both types have been accessed by use association and renamed, it is the original names that must agree.

11.2.2 The equivalence statement

The `equivalence` statement specifies that a given storage area may be shared by two or more objects. For instance

```
real aa, angle, alpha, a(3)
equivalence (aa, angle), (alpha, a(1))
```

allows `aa` and `angle` to be used interchangeably in the program text, as both names now refer to the same storage location. similarly, `alpha` and `a(1)` may be used interchangeably.

It is possible to equivalence arrays together. In

```
real a(3,3), b(3,3), col1(3), col2(3), col3(3)
equivalence (col1, a, b), (col2, a(1,2)), (col3, a(1,3))
```

the two arrays `a` and `b` are equivalenced, and the columns of `a` (and hence of `b`) are equivalenced to the arrays `col1`, etc. We note in this example that more than two entities may be equivalenced together, even in a single declaration.

It is possible to equivalence variables of the same intrinsic type and kind type parameter or of the same derived type having the `sequence` attribute. It is also possible to equivalence variables of different types if both have numeric storage association or both have character storage association (see Section 11.2.1). Default character variables need not have the same length, as in

```
character(len=4) a
character(len=3) b(2)
equivalence (a, b(1)(3:))
```

where the character variable `a` is equivalenced to the last four characters of the six characters of the character array `b`. Zero character length is not permitted. An example for different types is

```
integer i(100)
real x(100)
equivalence (i, x)
```

where the arrays `i` and `x` are equivalenced. This might be used, for instance, to save storage space if `i` is used in one part of a program unit and `x` separately in another part. This is a highly dangerous practice, as considerable confusion can arise when one storage area contains variables of two or more data types, and program changes may be made very difficult if the two uses of the one area are to be kept distinct.

Types with default initialization (Fortran 95 only) are permitted, provided each initialized component has the same type, type parameters, and value in any pair of equivalenced objects.

All the various combinations of types that may be equivalenced have been described. No other is allowed. Also, apart from double precision real and the default numeric types, equivalencing objects that have different kind type parameters is not allowed. The general form of the statement is

```
equivalence (object, object-list) [, (object, object-list)]...
```

where each *object* is a variable name, array element, or substring. An object must be a variable and must not be a dummy argument, a function result, a pointer, an object with a pointer component at any level of component selection, an allocatable array, an automatic object, a function, a structure component, or a subobject of such an object. Each array subscript and character substring range must be an initialization expression. The interpretation of an array name is identical to that of its first element. An equivalence object must not have the `target` attribute.

The objects in an equivalence set are said to be *storage associated.* Those of nonzero length share the same first storage unit. Those of zero length are associated with each other and with the first storage unit of those of nonzero length. `equivalence` statements may cause other parts of the objects to be associated, but not such that different subobjects of the same object share storage. For example

```
real a(2), b
equivalence (a(1), b), (a(2), b)  ! Prohibited.
```

is not permitted. Also, objects declared in different scoping units must not be equivalenced. For example

```
use my_module, only : xx
real bb
equivalence(xx, bb)               ! Prohibited.
```

is not permitted.

The various uses to which the `equivalence` was put are replaced by automatic arrays, allocatable arrays, and pointers (reuse of storage, Sections 6.4 and 6.5), pointers as aliases (storage mapping, Section 6.15), and the `transfer` function (mapping of one data type onto another, Section 8.9).

11.2.3 The common block

We have seen in Chapter 5 how two program units are able to communicate by passing variables, or values of expressions between them via argument lists or by using modules. It is also possible to define areas of storage known as `common` blocks. Each has a storage sequence and may be either named or unnamed, as shown by the simplified syntax of the `common` specification statement,

```
common [/[cname]/] vlist
```

in which *cname* is an optional name, and *vlist* is a list of variable names, each optionally followed by an array bounds specification. An unnamed `common` block is known as a *blank* `common` block. Examples of each are

```
common /hands/ nshuff, nplay, nhand, cards(52)
```

and

```
common // buffer(10000)
```

in which the named common block hands defines a data area containing the quantities which might be required by the subroutines of a card playing program, and the blank common defines a large data area which might be used by different routines as a buffer area.

The name of a common block has global scope and must differ from that of any other global entity (external procedure, program unit, or common block). It may, however, be the same as that of a local entity other than a named constant or intrinsic procedure.

No object in a common block may have the parameter attribute or be a dummy argument, an automatic object, an allocatable array, or a function. An array may have its bounds declared either in the common statement or in a type declaration or dimension statement. If it is a non-pointer array, the bounds must be declared explicitly and with constant specification expressions. If it is a pointer array, however, the bounds may not be declared in the common statement itself. An object of derived type must have the sequence attribute and (Fortran 95 only) the type must not have default initialization.

In order for a subroutine to access the variables in the data area, it is sufficient to insert the common definition in each scoping unit which requires access to one or more of the entities in the list. In this fashion, the variables nshuff, nplay, nhand and cards are made available to the those scoping units. No variable may appear more than once in all the common blocks in a scoping unit.

Usually, a common block contains identical variable names in all its appearances, but this is not necessary. In fact, the shared data area may be partitioned in quite different ways in different routines, using different variable names. They are said to be storage associated. It is thus possible for one subroutine to contain a declaration

```
common /coords/ x, y, z, i(10)
```

and another to contain a declaration

```
common /coords/ i, j, a(11)
```

This means that a reference to i(1) in the first routine is equivalent to a reference to a(2) in the second. Through multiple references via use or host association, this can even happen in a single routine. This manner of coding is both untidy and dangerous, and every effort should be made to ensure that all declarations of a given common block declaration are identical in every respect. In particular, the presence or absence of the target attribute is required to be consistent, since otherwise a compiler would have to assume that everything in common has the target attribute in case it has it in another program unit.

A further practice that is permitted but which we do not recommend is to mix different storage units in the same common block. When this is done, each position in the storage sequence must always be occupied by a storage unit of the same category.

The total number of storage units must be the same in each occurrence of a named common block, but blank common is allowed to vary in size and the longest definition will apply for the complete program.

Yet another practice to be avoided is to use the full syntax of the common statement,

common [/[*cname*]/]*vlist* [[,]/[*cname*]/*vlist*]...

which allows several common blocks to be defined in one statement, and a single common block to be declared in parts. A combined example is

```
common /pts/x,y,z /matrix/a(10,10),b(5,5) /pts/i,j,k
```

which is equivalent to

```
common /pts/ x, y, z, i, j, k
common /matrix/ a(10,10), b(5,5)
```

which is certainly a more understandable declaration of two shared data areas. The only need for the piece-wise declaration of one block is when the limit of 39 continuation lines is otherwise too low.

The common statement may be combined with the equivalence statement, as in the example

```
real a(10), b
equivalence (a,b)
common /change/ b
```

In this case, a is regarded as part of the common block, and its length is extended appropriately. Such an equivalence must not cause data in two different common blocks to become storage associated, it must not cause an extension of the common block except at its tail, and two different objects or subobjects in the same common block must not become storage associated. In Fortran 95, it must not cause an object of a type with default initialization to become associated with an object in a common block.

A common block may be declared in a module, and its variables accessed by use association. Variable names in a common block in a module may be declared to have the private attribute, but this does not prevent associated variables being declared elsewhere through other common statements.

An individual variable in a common block may not be given the save attribute, but the whole block may. If a common block has the save attribute in any scoping unit other than the main program, it must have the save attribute in all such scoping units. The general form of the save statement is

save [[::] *saved-entity-list*]

where *saved-entity* is *variable-name* or *common-block-name*. A simple example is

```
save /change/
```

Blank common always has the save attribute.

Data in a common block without the save attribute become undefined on return from a subprogram unless the block is also declared in the main program or in another subprogram that is in execution.

Use of modules (section 5.5) obviates the need for common blocks.

11.2.4 The block data program unit

Non-pointer variables in named common blocks may be initialized in data statements, but such statements must be collected into a special type of program unit, known as a block data program unit. It must have the form

```
block data [block-data-name]
     [specification-stmt]...
end [block data [block-data-name]]
```

where each *specification-stmt* is an implicit, use, type declaration (including double precision), intrinsic, pointer, target, common, dimension, data, equivalence, parameter, or save statement or derived-type definition. A type declaration statement must not specify the allocatable, external, intent, optional, private, or public attributes. An example is

```
block data
   common /axes/ i,j,k
   data i,j,k /1,2,3/
end block data
```

in which the variables in the common block axes are defined for use in any other scoping unit which accesses them.

It is possible to collect many common blocks and their corresponding data statements together in one block data program unit. However, it may be a better practice to have several different block data program units, each containing common blocks which have some logical association with one another. To allow for this eventuality, block data program units may be named in order to be able to distinguish them. A complete program may contain any number of block data program units, but only one of them may be unnamed. A common block must not appear in more than one block data program unit. It is not possible to initialize blank common.

The name of a block data program unit may appear in an external statement. When a processor is loading program units from a library, it may need such a statement in order to load the block data program unit.

Use of modules (Section 5.5) obviates the need for block data.

11.2.5 Shape and character length disagreement

In Fortran 77, it was often convenient, when passing an array, not to have to specify the size of the dummy array. For this case, the *assumed-size* array declaration is available, where the last *bounds* in the *bounds-list* is

[*lower-bound*:] *

and the other bounds (if any) must be declared explicitly. Such an array must not be a function result.

Since an assumed-size array has no bounds in its last dimension, it does not have a shape and, therefore, must not be used as a whole array in an executable statement, except as an argument to a procedure that does not require its shape. However, if an array section is formed with an explicit upper bound in the last dimension, this has a shape and may be used as a whole array.

An object of one size or rank may be passed to an explicit-shape or assumed-size dummy argument array that is of another size or rank, except when the dummy argument has the `target` attribute and the actual argument is a target other than an array section with a vector subscript. If an array element is passed to an array, the actual argument is regarded as an array with elements that are formed from the parent array from the given array element onwards, in array element order. Figure 11.1 illustrates this. Here only the last 49 elements of `a` are available to `sub`, as the first array element of `a` which is passed to `sub` is `a(52)`. Within `sub`, this element is referenced as `b(1)`.

Figure 11.1

```
real a(100)
:
call sub (a(52), 49)
:
subroutine sub(b,n)
:
real b(n)
:
```

In the same example, it would also be perfectly legitimate for the declaration of `b` to be written as

```
real b(7, 7)
```

and for the last 49 elements of `a` to be addressed as though they were ordered as a 7×7 array. The converse is also true. An array dimensioned 10×10 in a calling subroutine may be dimensioned as a singly-dimensioned array of size 100 in the called subroutine. Within `sub`, it is illegal to address `b(50)` in any way, as that would be beyond the declared length of `a` in the calling routine. In all cases, the association is by storage sequence, in array element order.

In the case of default character type, agreement of character length is not required. For a scalar dummy argument of character length *len*, the actual argument may have a greater character length and its leftmost *len* characters are associated with the dummy argument. For example, if chasub has a single dummy argument of character length 1,

```
call chasub(word(3:4))
```

is a valid `call` statement. For an array dummy argument, the restriction is on the total number of characters in the array. An array element or array element substring is regarded as a sequence of characters from its first character to the last character of the array. For an assumed-size array, the size is the number of characters in the sequence divided by the character length of the dummy argument.

Shape or character length disagreement of course cannot occur when the dummy argument is assumed-shape (by definition the shape is assumed from the actual argument). It can occur for explicit-shape and assumed-size arrays. Implementations are likely to receive explicit-shape and assumed-size arrays in contiguous storage, but permit any uniform spacing of the elements of an assumed-shape array. They will need to make a copy of any array argument that is not stored contiguously (for example, the section `a(1:10:2)`), unless the dummy argument is assumed shape. To avoid unnecessary copies of this kind, a scalar actual argument is permitted to be associated with an array only if the actual argument is an element of an array that is not an assumed-shaped array, an array pointer, a dummy argument with the `target` attribute, or is a subobject of such an element.

When a procedure is invoked through a generic name, as a defined operation, or as a defined assignment, rank agreement between the actual and the dummy arguments is required. Note also that only a scalar dummy argument may be associated with a scalar actual argument.

Assumed-shape arrays (Section 6.3) supplant this feature.

11.2.6 The entry statement

A subprogram usually defines a single procedure, and the first statement to be executed is the first executable statement after the header statement. In some cases it is useful to be able to define several procedures in one subprogram, particularly when wishing to share access to some saved local variables or to a section of code. This is possible for external and module subprograms (but not for internal subprograms) by means of the `entry` statement. This is a statement that has the form

`entry` *entry-name* `[([`*dummy-argument-list*`]) [result(`*result-name*`)]]`

and may appear anywhere between the header line and `contains` (or `end` if it has no `contains`) statement of a subprogram, except within a construct. The `entry`

statement provides a procedure with an associated dummy argument list, exactly as does the `subroutine` or `function` statement, and these arguments may be different from those given on the `subroutine` or `function` statement. Execution commences with the first executable statement following the `entry` statement.

In the case of a function, each `entry` defines another function, whose characteristics (that is, shape, type, type parameters, and whether a pointer) are given by specifications for the *result-name* (or *entry-name* if there is no `result` clause). If the characteristics are the same as for the main entry, a single variable is used for both results; otherwise, they must not be pointers, must be scalar, and must both be one of the default integer, default real, double precision real (Section 11.4.1), or default complex types, and they are treated as equivalenced. The `result` clause plays exactly the same rôle as for the main entry.

Each entry is regarded as defining another procedure, with its own name. The name of an entry has the same scope as the name of the subprogram. It must not be the name of a dummy argument of any of the procedures defined by the subprogram. An `entry` statement is not permitted in an interface block; there must be another body for each entry whose interface is wanted, using a `subroutine` or `function` statement, rather than an `entry` statement.

An `entry` is called in exactly the same manner as a subroutine or function, depending on whether it appears in a subroutine subprogram or a function subprogram. An example is given in Figure 11.2 which shows a search function with two entry points. We note that `looku` and `looks` are synonymous within the function, so that it is immaterial which value is set before the return.

None of the procedures defined by a subprogram is permitted to reference itself, unless the keyword `recursive` is present on the `subroutine` or `function` statement. For a function, such a reference must be indirect unless there is a `result` clause on the `function` or `entry` statement. If a procedure may be referenced directly in the subprogram that defines it, the interface is explicit in the subprogram.

The name of an `entry` dummy argument that appears in an executable statement preceding the `entry` statement in the subprogram must also appear in a `function`, `subroutine`, or `entry` statement that precedes the executable statement. Also, if a dummy argument is used to define the array size or character length of an object, the object must not be referenced unless the argument is present in the procedure reference that is active.

During the execution of one of the procedures defined by a subprogram, a reference to a dummy argument is permitted only if it is a dummy argument of the procedure referenced.

The `entry` statement is made unnecessary by the use of modules (Section 5.5), with each procedure defined by an entry becoming a module procedure. Its presence has substantially complicated the standard because the reader has to remember that a subprogram may define several procedures.

Figure 11.2

```
      function looku(list, member)
      integer looku, list(:), member, looks
!
!      To locate member in an array list.
!      If list is unsorted, entry looku is used;
!      if list is sorted, entry looks is used.
!
!      list is unsorted
      do looku = 1, size(list)
         if list(looku) .eq. member) go to 9
      end do
      go to 3
!
!      entry for sorted list
      entry looks(list, member)
!
      do looku = 1, size(list)
         if (list(looku) .ge. member) go to 2
      end do
      go to 3
!
!      is member at position looku?
   2  if (list(looku) .eq. member) go to 9
!
!      member not in list
   3  looku = 0
!
   9  end function looku
```

11.3 New redundant features

11.3.1 The include line

It is sometimes useful to be able to include source text from somewhere else into the source stream presented to the compiler. This facility is possible using an `include` line:

`include` *char-literal-constant*

where *char-literal-constant* must not have a kind parameter that is a named constant. This line is not a Fortran statement and must appear as a single source line where a statement may occur. It will be replaced by material in a processor-dependent way determined by the character string *char-literal-constant*. The included text may itself contain `include` lines, which are similarly replaced. An

include line must not reference itself, directly or indirectly. When an include line is resolved, the first included line must not be a continuation line and the last line must not be continued. An include line may have a trailing comment, but may not be labelled nor, when expanded, may it contain incomplete statements.

The include line was available as an extension to many Fortran 77 systems and was often used to ensure that every occurrence of global data in common was identical. In Fortran 90, the same effect is better achieved by placing global data in a module (Section 5.5). This cannot lead to accidental declarations of local variables in each procedure.

11.3.2 The do while form of loop control

In Section 4.5, a form of the do construct was described that may be written as

```
do
   if (scalar-logical-expr) exit
   :
end do
```

An alternative, but redundant, form of this is its representation using a do while statement:

```
do [label] [,] while (.not.scalar-logical-expr)
```

We prefer the form that uses the exit statement because this can be placed anywhere in the loop, whereas the do while statement always performs its test at the loop start. If the *scalar-logical-expr* becomes false in the middle of the loop, the rest of the loop is still executed. Potential optimization penalties that the use of the do while entails are fully described in Chapter 10 of *Optimizing Supercompilers for Supercomputers*, M. Wolfe (Pitman, 1989).

11.4 Old redundant features

11.4.1 Double precision real

Another *type* that may be used in a type declaration, function, implicit, or component declaration statement is double precision which specifies double precision real. The precision is greater than that of default real.

Literal constants written with the exponent letter d (or D) are of type double precision real by default; no kind parameter may be specified if this exponent letter is used. Thus, 1d0 is of type double precision real. If dp is an integer named constant with the value kind(1d0), double precision is synonymous with real(kind=dp).

There is a d (or D) edit descriptor that was originally intended for double precision quantities but, in Fortran 90, it is identical to the e edit descriptor except that the output form may have a D instead of an E as its exponent letter. A

double precision real literal constant, with exponent letter d, is acceptable on input whenever any other real literal constant is acceptable.

There are two elemental intrinsic functions which were not described in Chapter 8 because they have result of type double precision real:

`dble (a)` for a of type integer, real, or complex returns the double precision real value `real(a, kind(0d0))`.

`dprod (x, y)` returns the product `x*y` for x and y of type default real as a double precision real result.

The double precision real data type has been replaced by the real type of kind `kind(0.d0)`.

11.4.2 The dimension and parameter statements

To declare entities, we normally use type specifications. However, if all the entities involved are arrays, they may be declared *without* type specifications in a `dimension` statement:

```
dimension i(10), b(50,50), c(n,m)    ! n and m are dummy scalar
                                     ! integer arguments or
                                     ! named constants.
```

The general form is

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

Here, the type may either be specified in a type declaration statement such as

```
integer i
```

that does not specify the dimension information, or be specified implicitly. Our view is that neither of these is sound practice: the type declaration statement looks like a declaration of a scalar and we explained in Section 7.2 that we regard implicit typing as dangerous. Therefore, the use of the `dimension` statement is not recommended.

An alternative way to specify a named constant is by the `parameter` statement. It has the general form

`parameter (` *named-constant-definition-list* `)`

where each *named-constant-definition* is

constant-name `=` *initialization-expr*

Each constant named must either have been typed in a previous type declaration statement in the scoping unit, or take its type from the first letter of its name according to the implicit typing rule of the scoping unit. In the case of implicit typing, the appearance of the named constant in a subsequent type declaration

statement in the scoping unit must confirm the type and type parameters, and there must not be an `implicit` statement for the letter subsequently in the scoping unit. Similarly, the shape must have been specified previously or be scalar. Each named constant in the list is defined with the value of the corresponding expression according to the rules of intrinsic assignment.

An example using implicit typing and a constant expression including a named constant that is defined in the same statement is

```
implicit integer (a, p)
parameter (apple = 3, pear = apple**2)
```

For the same reasons as for `dimension`, we recommend avoiding the `parameter` statement.

11.4.3 Specific names of intrinsic procedures

There are a number of intrinsic functions that may have arguments that are all of one type and type parameters or all of another. For instance, we may write

```
a = sqrt(b)
```

and the appropriate square-root function will be invoked, depending on whether the variable b is real or complex and on its type parameters. In this case, the name `sqrt` is known as a *generic name*, meaning that the appropriate function is supplied, depending on the type and type parameters of the actual arguments of the function, and that a single name may be used for what are, probably, different specific functions.

Some of the intrinsic functions have *specific names* and are specified by the standard. They are listed in Tables 11.1 and 11.2. In the Tables, 'Character' stands for default character, 'Integer' stands for default integer, 'Real' stands for default real, 'Double' stands for double precision real, and 'Complex' stands for default complex. Those in Tables 11.2 may be passed as actual arguments to a subprogram, provided they are specified in an `intrinsic` statement (Section 8.1.3).

All the procedures that we described in Chapter 8 are regarded as generic, even where there is only one version.

Table 11.1. Specific intrinsic functions not available as actual arguments

Description	Generic Form	Specific Name	Argument Type	Function Type
Conversion to integer	`int(a)`	`int`	Real	Integer
		`ifix`	Real	Integer
		`idint`	Double	Integer
Conversion to real	`real(a)`	`real`	Integer	Real
		`float`	Integer	Real
		`sngl`	Double	Real
max(a1,a2,...)	`max(a1,a2,...)`	`max0`	Integer	Integer
		`amax1`	Real	Real
		`dmax1`	Double	Double
		`amax0`	Integer	Real
		`max1`	Real	Integer
min(a1,a2,...)	`min(a1,a2,...)`	`min0`	Integer	Integer
		`amin1`	Real	Real
		`dmin1`	Double	Double
		`amin0`	Integer	Real
		`min1`	Real	Integer

Table 11.2. Specific intrinsic functions available as actual arguments

Description	Generic Form	Specific Name	Argument Type	Function Type
absolute value of a	`sign(a,b)`	`isign`	Integer	Integer
times sign of b		`sign`	Real	Real
		`dsign`	Double	Double
max(x-y,0)	`dim(x,y)`	`idim`	Integer	Integer
		`dim`	Real	Real
		`ddim`	Double	Double
x*y		`dprod(x,y)`	Real	Double
truncation	`aint(a)`	`aint`	Real	Real
		`dint`	Double	Double
nearest whole	`anint(a)`	`anint`	Real	Real
number		`dnint`	Double	Double
nearest integer	`nint(a)`	`nint`	Real	Integer
		`idnint`	Double	Integer
absolute value	`abs(a)`	`iabs`	Integer	Integer
		`abs`	Real	Real
		`dabs`	Double	Double
		`cabs`	Complex	Real
remainder	`mod(a,p)`	`mod`	Integer	Integer
modulo p		`amod`	Real	Real
		`dmod`	Double	Double
square root	`sqrt(x)`	`sqrt`	Real	Real
		`dsqrt`	Double	Double
		`csqrt`	Complex	Complex
exponential	`exp(x)`	`exp`	Real	Real
		`dexp`	Double	Double
		`cexp`	Complex	Complex
Natural logarithm	`log(x)`	`alog`	Real	Real
		`dlog`	Double	Double
		`clog`	Complex	Complex
Common logarithm	`log10(x)`	`alog10`	Real	Real
		`dlog10`	Double	Double
Sine	`sin(x)`	`sin`	Real	Real
		`dsin`	Double	Double
		`csin`	Complex	Complex

Cosine	`cos(x)`	`cos`	Real	Real
		`dcos`	Double	Double
		`ccos`	Complex	Complex
Tangent	`tan(x)`	`tan`	Real	Real
		`dtan`	Double	Double
Arcsine	`asin(x)`	`asin`	Real	Real
		`dasin`	Double	Double
Arccosine	`acos(x)`	`acos`	Real	Real
		`dacos`	Double	Double
Arctangent	`atan(x)`	`atan`	Real	Real
		`datan`	Double	Double
	`atan2(y,x)`	`atan2`	Real	Real
		`datan2`	Double	Double
Hyperbolic sine	`sinh(x)`	`sinh`	Real	Real
		`dsinh`	Double	Double
Hyperbolic cosine	`cosh(x)`	`cosh`	Real	Real
		`dcosh`	Double	Double
Hyperbolic tangent	`tanh(x)`	`tanh`	Real	Real
		`dtanh`	Double	Double
Imaginary part	`aimag(z)`	`aimag`	Complex	Real
Complex conjugate	`conjg(z)`	`conjg`	Complex	Complex
Character length	`len(s)`	`len`	Character	Integer
Starting position	`index(s,t)`	`index`	Character	Integer

12. Floating-point exception handling

12.1 Introduction

Exception handling is required for the development of robust and efficient numerical software, a principal application of Fortran. Indeed, the existence of such a facility makes it possible to develop more efficient software than than would otherwise be possible. The clear need for exception handling, something that had been left out of the standards so far, led to a facility being developed on a 'fast track' as a Technical Report[1], suitable for immediate implementation as an extension to existing Fortran compilers (Section 1.5).

The subject of this chapter is this extension of Fortran 95. WG5 has promised that it will be included in the next revision of the Fortran standard, apart from correcting any defects found in the field. The intention is that this promise will encourage vendors to implement the feature in their compilers, confident that their efforts will have a secure future.

Similarly, programmers are encouraged to use the feature if it is available to them. Their code may have limited portability initially, but it will eventually become fully portable. This is why we feel that a book on Fortran 95 needs to include its description. This chapter is written with exactly the same aims as the rest of the book: to provide a complete description of all the facilities and provide some explanation of the choices the committees made in their design.

Most computers nowadays have hardware based on the IEEE standard for binary floating-point arithmetic[2], which later became an ISO Standard[3]. Therefore, the Fortran exception handling features are based on the ability to test and set the five flags for floating-point exceptions that the IEEE standard specifies. However, non-IEEE computers have not been ignored; they may provide support for some of the features and the programmer is able to find out what is supported or state that certain features are essential.

---

[1] Technical Report ISO/IEC 15580: 1998(E).

[2] IEEE 754-1985, Standard for binary floating-point arithmetic.

[3] IEC 559:1989, Binary floating-point arithmetic for microprocessor systems.

Few (if any) computers support every detail of the IEEE standard. This is because considerable economies in construction and increases in execution performance are available by omitting support for features deemed to be necessary to few programmers. It was therefore decided to include inquiry facilities for the extent of support of the standard, and for the programmer to be able to state which features are essential.

The mechanism finally chosen by the committees is based on a set of procedures for setting and testing the flags and inquiring about the features, collected in an intrinsic module called `ieee_exceptions`. An alternative that was seriously considered is described in the next subsection.

Given that procedures were being provided for the IEEE flags, it seemed sensible to provide procedures for other aspects of the IEEE standard. These are collected in a separate intrinsic module, `ieee_arithmetic`, which contains a `use` statement for `ieee_exceptions`.

To provide control over which features are essential, there is a third intrinsic module, `ieee_features` containing named constants corresponding to the features. If a named constant is accessible in a scoping unit, the corresponding feature must be available there.

12.1.1 Abandoned alternative

An alternative approach to providing exception handling that was seriously considered [4] [5] was based on a construct of the form

```
enable [(exceptions)]
   [enable block]
[handle [(exceptions)]
    handle block]
end enable
```

The idea was that certain exceptions are 'enabled' during execution of the enable block and, if any signals, control is transferred to the handle block. The enable block would contain fast code that is usually successful, and slower but more reliable alternative code in the handle block would be executed only if needed. The transfer is imprecise, so the compiler can use all its optimization features within the enable block. If the programmer needed precision, this could be obtained by using short enable blocks, but that may have inhibited optimization. If an exception led to slow execution, the programmer could limit its use to places where it is really needed.

Everyone was attracted to the enable construct, but there were serious difficulties associated with scoping. If a procedure is called in an enable block, it cannot

[4] Reid, J. K. (1995). Exception handling in Fortran. ACM Fortran Forum, 14, 9-15.

[5] Reid, J. K. (1997). Two approaches to exception handling in Fortran 90. In *The quality of numerical software: assessment and enhancement*, Ed R. F. Boisvert, Chapman and Hall (1997), 210-223.

be assumed that the enabled conditions are enabled within the procedure. If a condition signals but there is no local handler, is the condition handled somewhere else in the call chain? If so, what happens in an intermediate scope with no condition enabling? Maintaining good optimization properties when conditions do not signal, while defining precisely what happens when they do, proved to be difficult and confusing. For these reasons the procedures approach, whose effect is easier for everyone to understand, was adopted.

12.2 Intrinsic modules

Fortran 95 does not have the concept of an intrinsic module. This has been introduced as part of this feature. New syntax on the `use` statement provides control over whether it is intended to access an intrinsic or a non-intrinsic module:

```
use, intrinsic     :: ieee_arithmetic
use, non_intrinsic :: my_ieee_arithmetic
```

For the old form of the syntax:

```
use ieee_arithmetic
```

the processor looks first for a non-intrinsic module. The double colon is obligatory when one of the new keywords is present and is permitted when neither is there:

```
use :: ieee_arithmetic
```

All other aspects of the `use` statement, including renaming and the `only` option (see Section 7.10) are unchanged. The `intrinsic` statement itself is not extended.

12.3 The IEEE standard

In this section, we explain those aspects of the IEEE standard that the reader needs to know in order to understand the features of this extension of Fortran 95. We do not attempt to give a complete description of the standard.

Two floating-point data formats are specified, one for real and one for double precision arithmetic. They are supersets of the Fortran model, repeated here (see Section 8.7.1),

$$x = 0$$

and

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

where s is ± 1, p and b are integers exceeding one, e is an integer in a range $e_{\min} \leq e \leq e_{\max}$, and each f_k is an integer in the range $0 \leq f_k < b$ except that f_1 is also nonzero. Both IEEE formats are binary, with $b = 2$. The precisions

are $p = 24$ and $p = 53$, and the exponent ranges are $-125 \leq e \leq 128$ and $-1021 \leq e \leq 1024$, for real and double precision, respectively.

In addition, there are numbers with $e = e_{\min}$ and $f_1 = 0$, which are known as *denormalized* numbers; note that they all have absolute values less than that returned by the intrinsic `tiny` since it considers only numbers within the Fortran model. Also, zero has a sign and both 0 and -0 have inverses, ∞ and $-\infty$. Within Fortran 95, -0 is treated as the same as a zero in all intrinsic operations and comparisons, but it can be detected by the `sign` function and is respected on formatted output.

The IEEE standard also specifies that some of the binary patterns that do not fit the model be used for the results of exceptional operations, such as 0/0. Such a number is known as a *NaN* (Not a Number). A NaN may be *signaling* or *quiet*. Whenever a signaling NaN appears as an operand, the invalid exception signals. Quiet NaNs propagate through almost every arithmetic operation without signaling an exception.

The standard specifies four rounding modes:

nearest rounds the exact result to the nearest representable value.

to-zero rounds the exact result towards zero to the next representable value.

up rounds the exact result towards $+\infty$ to the next representable value.

down rounds the exact result towards $-\infty$ to the next representable value.

Some computers perform division by inverting the denominator and then multiplying by the numerator. The additional roundoff that this involves means that such an implementation does not conform with the IEEE standard. The IEEE standard also species that `sqrt` properly rounds the exact result and returns -0 for $\sqrt{-0}$. The Fortran facilities include inquiry functions for IEEE division and `sqrt`.

The presence of -0, ∞, $-\infty$, and the NaNs allows IEEE arithmetic to be closed, that is, every operation has a result. This is very helpful for optimization on modern hardware since several operations, none needing the result of any of the others, may actually be progressing in parallel. If an exception occurs, execution continues with the corresponding flag signaling, and the flag remains signaling until explicitly set quiet by the program. The flags are therefore called *sticky*.

There are five flags:

overflow occurs if the exact result of an operation with two normal values is too large for the data format. The stored result is ∞, `huge(x)`, `-huge(x)`, or $-\infty$, according to the rounding mode in operation, always with the correct sign.

divide_by_zero occurs if a finite nonzero value is divided by zero. The stored result is ∞ or $-\infty$ with the correct sign.

invalid occurs if the operation is invalid, for example, $\infty \times 0$, 0/0, or when an operand is a signaling NaN.

underflow occurs if the result of an operation with two finite nonzero values cannot be represented exactly and is too small to represent with full precision. The stored result is the best available, depending on the rounding mode in operation.

inexact occurs if the exact result of an operation cannot be represented in the data format without rounding.

The standard specifies the possibility of exceptions being trapped by user-written handlers, but this inhibits optimization and is not supported by the Fortran feature. Instead it supports the possibility of halting program execution after an exception signals. For the sake of optimization, such halting need not occur immediately.

The standard specifies several functions which are implemented in this Fortran extension as `ieee_copy_sign`, `ieee_logb`, `ieee_next_after`, `ieee_rem`, `ieee_rint`, `ieee_scalb`, and `ieee_unordered`, and which are described in Section 12.9.3.

12.4 Access to the features

To access the features of this chapter, we recommend that the user employ `use` statements for one or more of the intrinsic modules `ieee_exceptions`, `ieee_arithmetic` (which contains a `use` statement for `ieee_exceptions`), and `ieee_features`. If the processor does not support a module accessed in a `use` statement, the compilation must, of course, fail.

If a scoping unit does not access `ieee_exceptions` or `ieee_arithmetic`, the level of support is processor dependent, and need not include support for any exceptions. If a flag is signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

The module `ieee_features` contains the derived type:

```
ieee_features_type
```

for identifying a particular feature. The only possible values objects of this type may take are those of named constants defined in the module, each corresponding to an IEEE feature. If a scoping unit has access to one of these constants, the compiler must support the feature in the scoping unit or reject the program. For example, some hardware is much faster if denormalized numbers are not supported and instead all underflowed values are flushed to zero. In such a case, the statement

```
use, intrinsic :: ieee_features, only: ieee_denormal
```

will ensure that the scoping unit is compiled with (slower) code supporting denormalized numbers.

The module is unusual in all that all a code ever does is to access it with use statements, which affect the way the code is compiled in the scoping units with access to one or more of the module's constants. There is no purpose in declaring data of type ieee_features_type, though it is permitted; the components of the type are private, no operation is defined for it, and only intrinsic assignment is available for it. In a scoping unit containing a use statement, the effect is that of a compiler directive, but the other properties of use make the feature more powerful than would be possible with a directive.

The complete set of named constants in the module and the effect of their accessibility is:

ieee_datatype The scoping unit must provide IEEE arithmetic for at least one kind of real.

ieee_denormal The scoping unit must support denormalized numbers for at least one kind of real.

ieee_divide The scoping unit must support IEEE divide for at least one kind of real.

ieee_halting The scoping unit must support control of halting for each flag supported.

ieee_inexact_flag The scoping unit must support the inexact exception for at least one kind of real.

ieee_inf The scoping unit must support ∞ and $-\infty$ for at least one kind of real.

ieee_invalid_flag The scoping unit must support the invalid exception for at least one kind of real.

ieee_nan The scoping unit must support NaNs for at least one kind of real.

ieee_rounding The scoping unit must support control of the rounding mode for all four rounding modes on at least one kind of real.

ieee_sqrt The scoping unit must support IEEE square root for at least one kind of real.

ieee_underflow_flag The scoping unit must support the underflow exception for at least one kind of real.

Execution may be slowed on some processors by the support of some features. If ieee_exceptions is accessed but ieee_features is not accessed, the vendor is free to choose which subset to support. The processor's fullest support is provided when all of ieee_features is accessed:

```
use, intrinsic :: ieee_arithmetic
use, intrinsic :: ieee_features
```

but execution may then be slowed by the presence of a feature that is not needed. In all cases, the extent of support may be determined by the inquiry functions of Sections 12.8.2 and 12.9.2.

12.5 The Fortran flags

There are five Fortran exception flags, corresponding to the five IEEE flags. Each has a value that is either quiet or signaling. The value may be determined by the function `ieee_get_flag` (Section 12.8.3). Its initial value is quiet and it signals when the associated exception occurs in a real or complex operation. Its status may also be changed by the subroutine `ieee_set_flag` (Section 12.8.3) or the subroutine `ieee_set_status` (Section 12.8.4). Once signaling, it remains signaling unless set quiet by an invocation of the subroutine `ieee_set_flag` or the subroutine `ieee_set_status`. If any exception is signaling when the program terminates, the processor issues a warning on the unit identified by * in a `write` statement, indicating which conditions are signaling.

If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return. This allows exception handling within the procedure to be independent of the state of the flags on entry, while retaining their 'sticky' properties: within a scoping unit, a signaling flag remains signaling until explicitly set quiet.

If a scoping unit has access to `ieee_exceptions` and references an intrinsic procedure that executes normally, the values of the overflow, divide-by-zero and invalid flags are as on entry to the intrinsic procedure, even if one or more signals during the calculation. If a real or complex result is too large for the intrinsic procedure to handle, overflow may signal. If a real or complex result is a NaN because of an invalid operation (for example, `log(-1.0)`), invalid may signal. Similar rules apply to the evaluation of specification expressions on entry to a procedure, to format processing, and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set signaling because of an intermediate calculation that does not affect the result.

An implementation may provide alternative versions of an intrinsic procedure; for example, one might be rather slow but be suitable for a call from a scoping unit with access to `ieee_exceptions`, while an alternative faster one might be suitable for other cases.

If it is known that an intrinsic procedure will never need to signal an exception, there is no requirement for it to be handled – after all, there is no way that the programmer will be able to tell the difference. The same principle applies to a sequence of in-line code with no invocations of `ieee_get_flag`, `ieee_set_flag`, `ieee_get_status`, `ieee_set_status`, or `ieee_set_halting`. If the code, as written, includes an operation that would signal a flag, but after execution of the sequence no value of a variable depends on that operation, whether the exception signals is processor dependent. Thus, an implementation is permitted to optimize such an operation away. For example, when y has the value zero, whether the code

```
x = 1.0/y
x = 3.0
```

signals divide-by-zero is processor dependent. Another example is:

```
real, parameter :: x=0.0, y=6.0
:
if (1.0/x == y) print *,'Hello world'
```

where the processor is permitted to discard the `if` statement since the logical expression can never be true and no value of a variable depends on it.

An exception does not signal if this could arise only during execution of code not required or permitted by the standard. For example, the statement

```
if (f(x) > 0.0) y = 1.0/z
```

must not signal divide-by-zero when both `f(x)` and `z` are zero and the statement

```
where(a > 0.0) a = 1.0/a
```

must not signal divide-by-zero. On the other hand, when `x` has the value 1.0 and `y` has the value 0.0, the expression

```
x > 0.00001 .or. x/y > 0.00001
```

is permitted to cause the signaling of divide-by-zero.

The processor need not support the invalid, underflow, and inexact exceptions. If an exception is not supported, its flag is always quiet. The function `ieee_support_flag` (Section 12.8.2) may be used to inquire whether a particular flag is supported. If invalid is supported, it signals in the case of conversion to an integer (by assignment or an intrinsic procedure) if the result is too large to be representable.

12.6 Halting

Some processors allow control during program execution of whether to abort or continue execution after an exception has occurred. Such control is exercised by invocation of the subroutine `ieee_set_halting_mode` (Section 12.8.3). Halting is not precise and may occur any time after the exception has occurred. The function `ieee_support_halting` (Section 12.8.2) may be used to inquire whether this facility is available. The initial halting mode is processor dependent.

In a procedure other than `ieee_set_halting_mode`, the processor does not change the halting mode on entry, and on return ensures that the halting mode is the same as it was on entry.

12.7 The rounding modes

Some processors support alteration of the rounding mode during execution. In this case, the subroutine `ieee_set_rounding_mode` (Section 12.9.4) may be used

to alter it. The function ieee_support_rounding (Section 12.9.2) may be used to inquire whether this facility is available for a particular mode.

In a procedure other than ieee_set_rounding_mode, the processor does not change the rounding mode on entry, and on return ensures that the rounding mode is the same as it was on entry.

Note that the value of a literal constant is not affected by the rounding mode.

12.8 The module ieee_exceptions

When the module ieee_exceptions is accessible, the overflow and divide-by-zero flags are supported in the scoping unit for all available kinds of real and complex data. This minimal level of support has been designed to be possible also on a non-IEEE computer. Which other exceptions are supported may be determined by the function ieee_support_flag, see Section 12.8.2. Whether control of halting is supported may be determined by the function ieee_support_halting, see Section 12.8.2. The extent of support of the other exceptions may be influenced by the accessibility of the named constants ieee_inexact_flag, ieee_invalid_flag, and ieee_underflow_flag of the module ieee_features, see Section 12.4.

The module contains two derived types (Section 12.8.1), named constants of these types (Section 12.8.1), and a collection of generic procedures (Sections 12.8.2, 12.8.3, and 12.8.4). None of the procedures is permitted as an actual argument.

12.8.1 Derived types

The module ieee_exceptions contains two derived types:

ieee_flag_type for identifying a particular exception flag. The only possible values that can be taken by objects of this type are those of named constants defined in the module:

```
ieee_overflow   ieee_divide_by_zero  ieee_invalid
ieee_underflow  ieee_inexact
```

and these are used in the module to define the named array constants

```
type(ieee_flag_type), parameter ::                          &
ieee_usual(3) =                                             &
    (/ieee_overflow, ieee_divide_by_zero, ieee_invalid/), &
ieee_all(5) = (/ieee_usual, ieee_underflow, ieee_inexact/)
```

These array constants are convenient for inquiring about the state of several flags at once by using elemental procedures. Besides convenience, such elemental calls may be more efficient than a sequence of calls for single flags.

`ieee_status_type` for saving the current floating-point status, which includes the values of all the flags supported. It also includes the current rounding mode if dynamic control of rounding is supported and the halting mode if dynamic control of halting is supported.

The components of both types are private. No operation is defined for them and only intrinsic assignment is available for them.

12.8.2 Inquiry functions for IEEE exceptions

The module `ieee_exceptions` contains two inquiry functions. Their argument `flag` must be of type `type(ieee_flag_type)` with one of the values `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, and `ieee_inexact`. The inquiries are in terms of reals, but the same level of support is provided for the corresponding kinds of complex type.

`ieee_support_flag (flag [,x])` returns `.true.` if the processor supports the exception `flag` for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_halting (flag)` returns `.true.` if the processor supports the ability to control during program execution whether to abort or continue execution after the exception `flag`. Otherwise, it returns `.false.`.

12.8.3 Elemental subroutines

The module `ieee_exceptions` contains the following elemental subroutines:

`call ieee_get_flag (flag, flag_value)` where:

`flag` is of type `type(ieee_flag_type)` and has intent `in`. It specifies a flag.

`flag_value` is of type default logical and has intent `out`. If the value of `flag` is `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`, `flag_value` is given the value true if the corresponding exception flag is signaling and false otherwise.

`call ieee_get_halting_mode (flag, halting)` where:

`flag` is of type `type(ieee_flag_type)` and has intent `in`. It must have one of the values `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`.

`halting` is of type default logical has intent `out`. If the exception specified by `flag` will cause halting, `halting` is given the value true; otherwise, it is given the value false.

`call ieee_set_flag (flag, flag_value)` where:

`flag` is of type `type(ieee_flag_type)` and has intent `in`. It specifies a flag.

`flag_value` is of type default logical and has intent `in`. If the value of `flag` is `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`, the corresponding flag is set to be signaling if `flag_value` has the value true, and to be quiet if `flag_value` has the value false.

`call ieee_set_halting_mode (flag, halting)` where:

`flag` is of type `type(ieee_flag_type)` and has intent `in`. It must have one of the values `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`.

`halting` is of type default logical has intent `in`. If the value is true, the exception specified by `flag` will cause halting. Otherwise, execution will continue after this exception. The processor must either already be treating this exception in this way or be capable of changing the mode so that it does.

12.8.4 Non-elemental subroutines

The module `ieee_exceptions` contains the following non-elemental subroutines:

`call ieee_get_status (status_value)` where:

`status_value` is scalar and of type `type(ieee_status_type)` and has intent `out`. It returns the floating-point status, including all the exception flags, the rounding mode, and the halting mode.

`call ieee_set_status (status_value)` where:

`status_value` is scalar and of type `type(ieee_status_type)` and has intent `in`. Its value must have been set in a previous invocation of `ieee_get_status`. The floating-point status, including all the exception flags, the rounding mode, and the halting mode, is reset to as it was then.

These subroutines have been included for convenience and efficiency when a subsidiary calculation is to be performed, and one wishes to resume the main calculation with exactly the same environment, as shown in Figure 12.1. There are no facilities for finding directly the value held within such a variable of a particular flag, of the rounding mode, or of the halting mode.

Figure 12.1

```
use, intrinsic         :: ieee_arithmetic
type(ieee_status_type) :: status_value
   :
call ieee_get_status(status_value)   ! Get the flags
call ieee_set_flag(ieee_all,.false.) ! Set the flags quiet.
  :  ! Calculation involving exception handling
call ieee_set_status(status_value)   ! Restore the flags
```

12.9 The module ieee_arithmetic

The module ieee_arithmetic contains a use statement for the the module ieee_exceptions, so all the features of ieee_exceptions are accessed when ieee_arithmetic is accessed.

The module contains two derived types (Section 12.9.1), named constants of these types (Section 12.9.1), and a collection of generic procedures (Sections 12.9.2, 12.9.3, 12.9.4 and 12.9.5). None of the procedures is permitted as an actual argument.

12.9.1 Derived types

The module ieee_arithmetic contains two derived types:

ieee_class_type for identifying a class of floating-point values. The only possible values objects of this type may take are those of the named constants defined in the module:

```
ieee_signaling_nan     ieee_quiet_nan
ieee_negative_inf      ieee_negative_normal
ieee_negative_denormal ieee_negative_zero
ieee_positive_zero     ieee_positive_denormal
ieee_positive_normal   ieee_positive_inf
```

ieee_round_type for identifying a particular rounding mode. The only possible values objects of this type may take are those of the named constants defined in the module:

```
ieee_nearest ieee_to_zero
ieee_up      ieee_down
```

for the IEEE modes and

```
ieee_other
```

for any other mode.

The components of both types are private. No operation is defined for them and only intrinsic assignment is available for them.

12.9.2 Inquiry functions for IEEE arithmetic

The module `ieee_arithmetic` contains the following inquiry functions. The inquiries are in terms of reals, but the same level of support is provided for the corresponding kinds of complex type.

`ieee_support_datatype ([x])` returns `.true.` if the processor supports IEEE arithmetic for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`. Complete conformance with the IEEE standard is not required for `.true.` to be returned, but the normalized numbers must be exactly those of IEEE single or IEEE double; the arithmetic operators `+`, `-`, and `*` must be implemented with at least one of the IEEE rounding modes; and the functions `ieee_copy_sign`, `ieee_scalb`, `ieee_logb`, `ieee_next_after`, `ieee_rem`, and `ieee_unordered` must implement the corresponding IEEE functions.

`ieee_support_denormal ([x])` returns `.true.` if the processor supports the IEEE denormalized numbers for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_divide ([x])` returns `.true.` if the processor supports divide with the accuracy specified by the IEEE standard for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_inf ([x])` returns `.true.` if the processor supports the IEEE infinity facility for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_nan ([x])` returns `.true.` if the processor supports the IEEE Not-A-Number facility for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_rounding (round_value [,x])` for a `round_value` of the type `ieee_round_type` returns `.true.` if the processor supports that rounding mode for all reals (`x` absent) or for reals of the same kind type parameter as the argument `x`. Otherwise, it returns `.false.`. Here, support includes the ability to change the mode by the invocation

```
call ieee_set_rounding_mode (round_value)
```

`ieee_support_sqrt ([x])` returns `.true.` if `sqrt` implements IEEE square root for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

`ieee_support_standard ([x])` returns `.true.` if the processor supports all the IEEE facilities defined in this chapter for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. Otherwise, it returns `.false.`.

12.9.3 Elemental functions

The module `ieee_arithmetic` contains the following elemental functions for the reals `x` and `y` for which the values of `ieee_support_datatype(x)` and `ieee_support_datatype(y)` are true:

`ieee_class (x)` is of type `type(ieee_class_type)` and returns the IEEE class of the real argument `x`. The possible values are explained in Section 12.9.1.

`ieee_copy_sign (x, y)` returns a real with the same type parameter as `x`, holding the value of `x` with the sign of `y`. This is true even for the IEEE special values, such as NaN and ∞ (on processors supporting such values).

`ieee_is_finite (x)` returns the value `.true.` if `ieee_class (x)` has one of the values

```
ieee_negative_normal    ieee_negative_denormal
ieee_negative_zero      ieee_positive_zero
ieee_positive_denormal  ieee_positive_normal
```

and `.false.` otherwise.

`ieee_is_nan (x)` returns the value `.true.` if the value of `x` is an IEEE NaN and `.false.` otherwise.

`ieee_is_negative (x)` returns the value `.true.` if `ieee_class (x)` has one of the values

```
ieee_negative_normal ieee_negative_denormal
ieee_negative_zero   ieee_negative_inf
```

and `.false.` otherwise.

`ieee_is_normal (x)` returns the value `.true.` if `ieee_class (x)` has one of the values

```
ieee_negative_normal  ieee_negative_zero
ieee_positive_zero    ieee_positive_normal
```

and `.false.` otherwise.

`ieee_logb (x)` returns a real with the same type parameter as `x`. If `x` is neither zero, infinity, nor NaN, the value of the result is the unbiased exponent of `x`, that is, `exponent(x)-1`. Otherwise,

i) If `x==0`, `ieee_divide_by_zero` signals and the result is $-\infty$ if `ieee_support_inf(x)` is true and `-huge(x)` otherwise.

ii) If `x` has an infinite value, the result is $+\infty$.

iii) If `x` has a quiet NaN value, the result is the same NaN.

iv) If `x` has a signaling NaN value, the result is a quiet NaN.

`ieee_next_after (x, y)` returns a real with the same type parameter as `x`. If `x==y`, the result is `x`, without an exception ever signaling. Otherwise, the result is the neighbour of `x` in the direction of `y`. The neighbours of zero (of either sign) are both nonzero. If either `x` or `y` is a NaN, the result is one of the input NaNs. Overflow is signaled when `x` is finite but `ieee_next_after (x, y)` is infinite; underflow is signaled when `ieee_next_after (x, y)` is denormalized; in both cases, `ieee_inexact` signals.

`ieee_rem (x, y)` returns a real with the same type parameter as `x` and value exactly `x-y*`n, where n is the integer nearest to the exact value `x/y`; whenever $|n - \texttt{x/y}| = 1/2$, n is even. If the result value is zero, the sign is that of `x`.

`ieee_rint (x, y)` returns a real with the same type parameter as `x` whose value is that of `x` rounded to an integer value according to the current rounding mode.

`ieee_scalb (x, i)` returns a real with the same type parameter as `x` whose value is $2^{\texttt{i}}\texttt{x}$ if this is within the range of normalized numbers. If $2^{\texttt{i}}\texttt{x}$ is too large, `ieee_overflow` signals; if `ieee_support_inf(x)` is true, the result value is infinity with the sign of `x`; otherwise, it is `sign(huge(x),x)`. If $2^{\texttt{i}}\texttt{x}$ is too small, `ieee_underflow` signals; the result is the nearest representable number with the sign of `x`.

`ieee_unordered (x, y)` returns `.true.` if `x` or `y` is a NaN or both are, and `.false.` otherwise.

`ieee_value (x, class)` is of type `type(ieee_class_type)` with a value specified by `class`. The argument `class` may have value

`ieee_signaling_nan` or `ieee_quiet_nan` if `ieee_support_nan(x)` is true,

`ieee_negative_inf` or `ieee_positive_inf` if `ieee_support_inf(x)` is true,

`ieee_negative_denormal` or `ieee_positive_denormal` if the value of `ieee_support_denormal(x)` is true, or

`ieee_negative_normal`, `ieee_negative_zero`, `ieee_positive_zero`, or `ieee_positive_normal`.

Although in most cases the value is processor dependent, it does not vary between invocations for any particular kind type parameter of `x` and value of `class`.

12.9.4 Non-elemental subroutines

The module `ieee_arithmetic` contains the following non-elemental subroutines:

`call ieee_get_rounding_mode (round_value)` where:

`round_value` is scalar and of type `type(ieee_round_type)` and has intent `out`. It returns the floating-point rounding mode, with value `ieee_nearest`, `ieee_to_zero`, `ieee_up`, or `ieee_down` if one of the IEEE modes is in operation, and `ieee_other` otherwise.

`call ieee_set_rounding_mode (round_value)` where:

`round_value` is scalar, of type `type(ieee_round_type)`, and has intent `in`. It specifies the mode to be set. The value of `ieee_support_rounding (round_value, x)` must be true for any `x` such that the value of `ieee_support_datatype(x)` is true.

The example in Figure 12.2 shows the use of these subroutines to store all the exception flags, perform a calculation involving exception handling, and restore them later.

Figure 12.2

```
use, intrinsic :: ieee_arithmetic
type(ieee_round_type) round_value
   :
call ieee_get_rounding_mode(round_value) ! Store the rounding
                                         ! mode
call ieee_set_rounding_mode(ieee_nearest)
  :  ! Calculation with round to nearest
call ieee_set_rounding_mode(round_value) ! Restore the
                                         ! rounding mode
```

12.9.5 Transformational function for kind value

The module `ieee_arithmetic` contains the following transformational function:

`ieee_selected_real_kind ([p][, r])` is just like `selected_real_kind` (Section 8.7.4) except that the result is the kind value of a real `x` for which `ieee_support_datatype(x)` is true.

12.10 Examples

12.10.1 Dot product

Our first example, Figure 12.3, is of a module for the dot product of two real arrays of rank 1. It contains a logical scalar `dot_error`, which acts as an error flag. If the sizes of the arrays are different, an immediate return occurs with `dot_error` true. If overflow occurs during the actual calculation, the overflow flag will signal and `dot_error` is set true. If all is well, its value is unchanged.

Figure 12.3

```
module dot
 ! Module for dot product of two real arrays of rank 1.
 ! The caller must ensure that exceptions do not cause halting.
   use, intrinsic :: ieee_exceptions
   logical        :: dot_error = .false.
   interface operator(.dot.)
      module procedure mult
   end interface
contains
   real function mult(a,b)
      real, intent(in) :: a(:), b(:)
      integer          :: i
      logical          :: overflow
      if (size(a)/=size(b)) then
         dot_error = .true.
         return
      end if
! The processor ensures that ieee_overflow is quiet
      mult = 0.0
      do i = 1, size(a)
         mult = mult + a(i)*b(i)
      end do
      call ieee_get_flag(ieee_overflow,overflow)
      if (overflow) dot_error = .true.
   end function mult
end module dot
```

12.10.2 Calling alternative procedures

Suppose the function fast_inv is a code for matrix inversion that 'lives dangerously' and may cause a condition to signal. The alternative function slow_inv is far less likely to cause a condition to signal, but is much slower. The following code, Figure 12.4, tries fast_inv and, if necessary, makes another try with slow_inv. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

Figure 12.4

```
use, intrinsic :: ieee_exceptions
use, intrinsic :: ieee_features, only: ieee_invalid_flag
! The other exceptions of ieee_usual (ieee_overflow and
! ieee_divide_by_zero) are always available with ieee_exceptions
type(ieee_status_type) :: status_value
logical, dimension(3)  :: flag_value
  :
call ieee_get_status(status_value)
call ieee_set_halting_mode(ieee_usual,.false.) ! Needed in case the
!            default on the processor is to halt on exceptions.
call ieee_set_flag(ieee_usual,.false.)         ! Elemental
! First try the "fast" algorithm for inverting a matrix:
matrix1 = fast_inv(matrix) ! This must not alter matrix.
call ieee_get_flag(ieee_usual,flag_value)      ! Elemental
if (any(flag_value)) then
! "Fast" algorithm failed; try "slow" one:
   call ieee_set_flag(ieee_usual,.false.)
   matrix1 = slow_inv(matrix)
   call ieee_get_flag(ieee_usual,flag_value)
   if (any(flag_value)) then
      write (*, *) 'Cannot invert matrix'
      stop
   end if
end if
call ieee_set_status(status_value)
```

12.10.3 Calling alternative in-line code

This example, Figure 12.5, is similar to the inner part of the previous one, but here the code for matrix inversion is in line, we know that only overflow can signal, and the transfer is made more precise by adding extra tests of the flag.

Figure 12.5

```
use, intrinsic :: ieee_exceptions
logical        :: flag_value
   :
call ieee_set_halting_mode(ieee_overflow,.false.)
call ieee_set_flag(ieee_overflow,.false.)
! First try a fast algorithm for inverting a matrix.
do k = 1, n
   :
   call ieee_get_flag(ieee_overflow,flag_value)
   if (flag_value) exit
end do
if (flag_value) then
! Alternative code which knows that k-1 steps have
! executed normally.
:
end if
```

12.10.4 Reliable hypotenuse function

The most important use of a floating-point exception handling facility is to make possible the development of much more efficient software than is otherwise possible. The code in Figure 12.6 for the 'hypotenuse' function, $\sqrt{x^2 + y^2}$, illustrates the use of the facility in developing efficient software.

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation may involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if x and y are both near the overflow limit). If the overflow or underflow flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost.

Figure 12.6

```
  real function hypot(x, y)
! In rare circumstances this may lead to the signaling of
! ieee_overflow.
! The caller must ensure that exceptions do not cause halting.
   use, intrinsic :: ieee_arithmetic
   use, intrinsic :: ieee_features, only: ieee_underflow_flag
! ieee_overflow is always available with ieee_arithmetic
   real                  :: x, y
   real                  :: scaled_x, scaled_y, scaled_result
   logical, dimension(2) :: flags
   type(ieee_flag_type), parameter, dimension(2) ::        &
         out_of_range = (/ ieee_overflow, ieee_underflow /)
   intrinsic :: sqrt, abs, exponent, max, digits, scale
! The processor clears the flags on entry
! Try a fast algorithm first
   hypot = sqrt( x**2 + y**2 )
   call ieee_get_flag(out_of_range, flags)
   if ( any(flags) ) then
     call ieee_set_flag(out_of_range, .false.)
     if ( x==0.0 .or. y==0.0 ) then
       hypot = abs(x) + abs(y)
     else if ( 2*abs(exponent(x)-exponent(y)) > digits(x)+1 ) then
       hypot = max( abs(x), abs(y) )! We can ignore one of x and y
     else     ! Scale so that abs(x) is near 1
       scaled_x = scale( x, -exponent(x) )
       scaled_y = scale( y, -exponent(x) )
       scaled_result = sqrt( scaled_x**2 + scaled_y**2 )
       hypot = scale(scaled_result, exponent(x)) ! May cause
     end if                                      ! overflow
   end if
! The processor resets any flag that was signaling on entry
end function hypot
```

13. Allocatable array extensions

13.1 Introduction

The subject of this chapter is another extension[1] of Fortran 95. It involves the use of allocatable arrays as dummy arguments, function results, and components of structures. Pointer arrays may be used instead in Fortran 90/95, but there are significant advantages for memory management and execution speed in using allocatable arrays when the added functionality of pointers is not needed.

This is another extension that WG5 has promised will be included in the next revision of the Fortran standard, apart from correcting any defects found in the field. Again, the intention is that this promise will encourage vendors to implement the feature in their compilers, confident that their efforts will have a secure future (Section 1.5).

Why is this extension needed? Firstly, code for a pointer array is likely to be less efficient because allowance has to be made for strides other than unity. For example, its target might be the section `vector(1:n:2)` or the section `matrix(i,1:n)` with non-unit strides, whereas most computers hold allocatable arrays in contiguous memory.

Secondly, if a defined operation involves a temporary variable of a derived type with a pointer component, the compiler will probably be unable to deallocate its target when storage for the variable is freed. Consider, for example, the statement

```
a = b + c*d        ! a, b, c, and d are of the same derived type
```

This will create a temporary for `c*d`, which is not needed once `b + c*d` has been calculated. The compiler is unlikely to be sure that no other pointer has the component or part of it as a target, so is unlikely to deallocate it.

Thirdly, intrinsic assignment is often unsuitable for a derived type with a pointer component because the assignment

```
a = b
```

will leave `a` and `b` sharing the same target for their pointer component. Therefore, a defined assignment that allocates a fresh target and copies the data will be used instead. However, this is very wasteful if the right-hand side is a temporary such as that of the assignment of the previous paragraph.

[1] Technical Report ISO/IEC 15581: 1998(E).

Fourthly, similar considerations apply to a function invocation within an expression. The compiler will be unlikely to be able to deallocate the pointer after the expression has been calculated.

Although the Fortran standard does not mention descriptors, it is very helpful to think of an allocatable array as being held as a descriptor that records whether it is allocated and, if so, its address and its bounds in each dimension. This is like a descriptor for a pointer, but no strides need be held since these are always unity. As for pointers, the expectation is that the array itself is held separately.

13.2 Allocatable dummy arguments

A dummy argument is permitted to have the allocatable attribute. In this case, the corresponding actual argument must be an allocatable array of the same type, kind parameters, and rank; also, the interface must be explicit. The dummy argument always receives the allocation status (descriptor) of the actual argument on entry and the actual argument receives that of the dummy argument on return. In both cases, this may be 'not currently allocated'.

Our expectatation is that some compilers will perform copy in / copy out of the descriptor. Rule i) of Section 5.7.2 is applicable and is designed to permit compilers to do this. In particular, this means that no reference to the actual argument (for example, through it being a module variable) is permitted from the invoked procedure if the dummy array is allocated or deallocated there.

For the array itself, we see no need for the compiler ever to perform copy in / copy out since the actual argument has to be allocatable too. However, it is permitted even if both arrays have the `target` attribute[2]. No reliance should be placed on pointer associations with the actual argument on entry applying also to the dummy argument, or pointer associations with the dummy argument on return applying also to the actual argument.

An allocatable dummy argument is permitted to have intent and this applies not to the allocation status (the descriptor) but to the array itself. If the intent is `in`, the array is not permitted to be allocated or deallocated since this would, of necessity, alter the value. An example of the application of an allocatable dummy argument to reading arrays of variable bounds is shown in Figure 13.1.

13.3 Allocatable functions

A function result is permitted to have the allocatable attribute. The allocation status on each entry to the function is 'not currently allocated'. The result may be allocated and deallocated any number of times during execution of the procedure, but it must be allocated and have a defined value on return.

---

[2]The Technical Report does not specify what happens when both arrays have the `target` attribute. Our belief is that an interpretation is likely to specify that pointer associations will be valid in this case, that is, copy in / copy out will not be permitted. For the moment, this cannot be relied upon.

Figure 13.1

```
subroutine load(array, unit)
   real, allocatable, intent(out), dimension(:,:,:) :: array
   integer, intent(in)          :: unit
   integer                      :: n1, n2, n3

   read(unit) n1, n2, n3
   if (allocated(array)) deallocate(array)
   allocate(array(n1, n2, n3))
   read(unit) array
end subroutine load
```

The interface must be explicit in any scoping unit in which the function is referenced. The result array is automatically deallocated after execution of the statement in which the reference occurs, even if it has the target attribute. We can thus recast the example in Section 5.10, and safely use the function result in an expression as shown in Figure 13.2, without fearing a memory leak.

Figure 13.2

```
program no_leak
   real, dimension(100) :: x, y
   :
   y(:size(compact(x)) = compact(x)**2
   :

contains

   function compact(x) ! A procedure to remove duplicates from
                       ! the array x
      real, allocatable, dimension(:) :: compact
      real, dimension(:), intent(in)  :: x
      integer                         :: n
      :                ! Find the number of distinct values, n
      allocate(compact(n))
      :                ! Copy the distinct values into compact
   end function compact
end program no_leak
```

13.4 Allocatable components

Components of derived type are permitted to have the `allocatable` attribute. For example, given the type declaration

```
type stack
   integer :: index
   integer, allocatable :: content(:)
end type stack
```

for each scalar variable of type `type(stack)`, the bounds of component `content` are determined by an `allocate` statement, by assignment, or by argument association.

In Section 9.3, we used the term *ultimate component* when a sequence of component selections involves no pointer components of derived type and ends with an intrinsic type. It is convenient to extend the term to include the case that ends with a component of derived type that is allocatable or is a pointer. For example, if the components called `alloc` are allocatable and the components called `point` are pointers, the components `obj%point` and `obj%alloc` are ultimate components, but `obj%point%comp` and `obj%alloc%comp` are not. The parent object (`obj` in our example) may be allocatable or a pointer.

Just as for an ordinary allocatable array, the initial state of an allocable component is always 'not currently allocated'. This is also true for an ultimate allocatable component of an object created by an allocate statement. Hence, there is no need for default initialization of allocatable components. In fact, initialization in a derived type definition (Section 7.11) of an allocatable component is not permitted.

In a structure constructor (Section 3.8), an expression corresponding to an allocatable component must be an array or `null()`. If it is an allocatable array, the component takes the same allocation status and, if allocated, the same bounds and value. If it is an array, but not an allocatable array, the component is allocated with the same bounds and is assigned the same value. If it is `null()`, the component receives the allocation status of 'not currently allocated'.

Allocatable components are illustrated in Figure 13.3, where code to manipulate polynomials with variable numbers of terms is shown.

Just as an allocatable array is not permitted to have the `parameter` attribute (be a constant), so an object with an ultimate allocatable component is not permitted to have the `parameter` attribute. An array constructor of such a type cannot be a constant and cannot participate in an initialization expression (Section 7.4). To correspond with this, a variable of such a type is not permitted to be initialized.

In an array subobject (Section 6.13), a *part-ref* to the right of a *part-ref* with nonzero rank must not be an element of an allocatable component, for example,

Figure 13.3

```
module real_polynomial_module
   type real_polynomial
     real, allocatable, dimension(:) :: coeff
   end type real_polynomial
   interface operator(+)
      module procedure rp_add_rp
   end interface operator(+)
contains
   function rp_add_rp(p1, p2)
      type(real_polynomial)             :: rp_add_rp
      type(real_polynomial), intent(in) :: p1, p2
      integer                           :: m, m1, m2
      m1 = ubound(p1%coeff,1)
      m2 = ubound(p2%coeff,1)
      allocate(rp_add_rp%coeff(0:max(m1,m2)))
      m = min(m1,m2)
      rp_add_rp%coeff(:m) = p1%coeff(:m) +p2%coeff(:m)
      if (m1 > m) rp_add_rp%coeff(m+1:) = p1%coeff(m+1:)
      if (m2 > m) rp_add_rp%coeff(m+1:) = p2%coeff(m+1:)
    end function rp_add_rp
end module real_polynomial_module
program example
   use real_polynomial_module
   type(real_polynomial) :: p, q, r
   p = real_polynomial((/4.0, 2.0, 1.0/))  ! Set p to 4+2x+x**2
   q = real_polynomial((/-1.0, 1.0/))
   r = p + q
   print *, 'Coefficients are: ', r%coeff
end program example
```

```
type(stack) :: a(10)  ! The type was declared at the
  :                   ! start of this section.
... a(:)%content(1)   ! Not permitted.
```

This is because such an object would not be an ordinary array – its elements are likely to be stored without any regular pattern, each having been separately given storage by an `allocate` statement.

When a variable of derived type is deallocated, any ultimate allocatable component that is currently allocated is also deallocated, as if by a deallocate statement. The variable may be a pointer or an allocatable array, and the rule applies recursively, so that all allocated allocatable components at all levels (apart from any lying beyond pointer components) are deallocated. Note the convenience to the programmer of this feature; to avoid memory leakage with pointer components,

the programmer would need to deallocate each one explicitly and be careful to order the deallocations correctly.

Intrinsic assignment

variable = *expr*

for a type with an ultimate allocatable component consists of the following steps for each such component:

i) If the component of *variable* is currently allocated, it is deallocated.

ii) If the component of *expr* is currently allocated, the component of *variable* is allocated with the same bounds and the value is then transferred using intrinsic assignment.

If the allocatable component of *expr* is 'not currently allocated', nothing happens in step ii), so the component of *variable* is left 'not currently allocated'. Note that if the component of *variable* is already allocated with the same shape, the compiler may chose to avoid the overheads of deallocation and reallocation. Note also that if the compiler can tell that there will be no subsequent reference to *expr*, because it is a function reference or a temporary variable holding the result of expression evaluation, no allocation or assignment is needed – all that has to happen is the deallocation of any allocated ultimate allocatable components of *variable* followed by copying of the descriptor.

If an actual argument and the corresponding dummy argument have an ultimate allocatable component, rule i) of Section 5.7.2 is applicable and requires all allocations and deallocations of the component to be performed through the dummy argument, in case copy in / copy out is in effect.

If a statement contains a reference to a function whose result is of a type with an ultimate allocatable component, any allocated ultimate allocatable components of the function result are deallocated after execution of the statement. This parallels the rule for allocatable function results (Section 13.3).

Just as for pointer components (see Section 9.3), a structure with an ultimate allocatable component is not permitted in an I/O list. The reason is the intention to add defined edit descriptors for data structures to Fortran 2000. Programmers will be able to write procedures that are called as part of the I/O processing. Such a procedure will be much better able to handle structures whose size and composition vary dynamically, the usual case with allocatable components.

Similarly, such a structure is not permitted in a namelist group (Section 7.15). This is entirely consistent with not permitting allocatable arrays or structures with pointer components here.

A sequence type (Section 11.2.1) is permitted to have an allocatable component, which permits independent declarations of the same type in different scopes, but such a type has an unspecified storage unit, that is, does not have numeric or character storage association (as for a type with a pointer component).

This dynamic feature is ill adapted to the fixed storage model implied by storage association, and so, just as for pointer components (see Section 11.2.2),

a structure with an ultimate allocatable component is not permitted as an `object` in an `equivalence` statement. Just as for allocatable arrays (see Section 11.2.3), a structure with an ultimate allocatable component, is not permitted in a `common` block. Thus, such a structure is not permitted at all in a storage association context.

A. Intrinsic procedures

Name	Section	Description
`abs (a)`	8.3.1	Absolute value.
`achar (i)`	8.5.1	Character in position `i` of ASCII collating sequence.
`acos (x)`	8.4	Arc cosine (inverse cosine) function.
`adjustl (string)`	8.5.3	Adjust left, removing leading blanks and inserting trailing blanks.
`adjustr (string)`	8.5.3	Adjust right, removing trailing blanks and inserting leading blanks.
`aimag (z)`	8.3.1	Imaginary part of complex number.
`aint (a [,kind])`	8.3.1	Truncate to a whole number.
`all (mask [,dim])`	8.11	True if all elements are true.
`allocated (array)`	8.12.1	True if the array is allocated.
`anint (a [,kind])`	8.3.1	Nearest whole number.
`any (mask [,dim])`	8.11	True if any element is true.
`asin (x)`	8.4	Arcsine (inverse sine) function.
`associated (pointer [,target])`	8.2	True if pointer is associated with target.
`atan (x)`	8.4	Arctangent (inverse tangent) function.
`atan2 (y, x)`	8.4	Argument of complex number `(x, y)`.
`bit_size (i)`	8.8.1	Maximum number of bits that may be held in an integer.
`btest (i, pos)`	8.8.2	True if bit `pos` of integer `i` has value 1.

`ceiling (a [, kind])`	8.3.1	Least integer greater than or equal to its argument (`kind` permitted only in Fortran 95).
`char (i [,kind])`	8.5.1	Character in position `i` of the processor collating sequence.
`cmplx (x [,y] [,kind])`	8.3.1	Convert to complex type.
`conjg (z)`	8.3.2	Conjugate of a complex number.
`cos (x)`	8.4	Cosine function.
`cosh (x)`	8.4	Hyperbolic cosine function.
`count (mask [,dim])`	8.11	Number of true elements.
`cpu_time (time)`	8.16.2	Processor time (Fortran 95 only)
`cshift (array, shift [,dim])`	8.13.5	Perform circular shift.
`call date_and_time ([date] [,time] [,zone] [,values])`	8.16.1	Real-time clock reading date and time.
`dble (a)`	11.4.1	Convert to double precision real.
`digits (x)`	8.7.2	Number of significant digits in the model for `x`.
`dim (x, y)`	8.3.2	`max(x-y, 0)`.
`dot_product (vector_a, vector_b)`	8.10	Dotproduct.
`dprod (x, y)`	11.4.1	Double precision real product of two default real scalars.
`eoshift (array, shift [,boundary][,dim])`	8.13.5	Perform end-off shift.
`epsilon (x)`	8.7.2	Number that is almost negligible compared with one in the model for numbers like `x`.
`exp (x)`	8.4	Exponential function.
`exponent (x)`	8.7.3	Exponent part of the model for `x`.
`floor (a [, kind])`	8.3.1	Greatest integer less than or equal to its argument (`kind` permitted only in Fortran 95).
`fraction (x)`	8.7.3	Fractional part of the model for `x`.
`huge (x)`	8.7.2	Largest number in the model for numbers like `x`.
`iachar (c)`	8.5.1	Position of character `c` in ASCII collating sequence.

`iand (i, j)`	8.8.2	Logical and on the bits.
`ibclr (i, pos)`	8.8.2	Clear bit pos to zero.
`ibits (i, pos, len)`	8.8.2	Extract a sequence of bits.
`ibset (i, pos)`	8.8.2	Set bit pos to one.
`ichar (c)`	8.5.1	Position of character c in the processor collating sequence.
`ieor (i, j)`	8.8.2	Exclusive or on the bits.
`index (string, substring [,back])`	8.5.3	Starting position of substring within string.
`int (a [,kind])`	8.3.1	Convert to integer type.
`ior (i, j)`	8.8.2	Inclusive or on the bits.
`ishft (i, shift)`	8.8.2	Logical shift on the bits.
`ishftc (i, shift [,size])`	8.8.2	Logical circular shift on a set of bits on the right.
`kind (x)`	8.2	Kind type parameter value.
`lbound (array [,dim])`	8.12.2	Array lower bounds.
`len (string)`	8.6.1	Character length.
`len_trim (string)`	8.5.3	Length of string without trailing blanks.
`lge (string_a, string_b)`	8.5.2	True if string_a equals or follows string_b in ASCII collating sequence.
`lgt (string_a, string_b)`	8.5.2	True if string_a follows string_b in ASCII collating sequence.
`lle (string_a, string_b)`	8.5.2	True if string_a equals or precedes string_b in ASCII collating sequence.
`llt (string_a, string_b)`	8.5.2	True if string_a precedes string_b in ASCII collating sequence.
`log (x)`	8.4	Natural (base *e*) logarithm function.
`logical (l, [,kind])`	8.5.4	Convert between kinds of logicals.
`log10 (x)`	8.4	Common (base 10) logarithm function.
`matmul (matrix_a, matrix_b)`	8.10	Matrix multiplication.
`max (a1, a2 [,a3,...])`	8.3.2	Maximum value.

`maxexponent (x)`	8.7.2	Maximum exponent in the model for numbers like `x`.
`maxloc (array [,mask])` or `maxloc (array, dim [,mask])`	8.14	Location of maximum array element (`dim` in Fortran 95 only).
`maxval (array [,mask])` or `maxval (array, dim [,mask])`	8.11	Value of maximum array element (`mask` as second positional argument in Fortran 95 only).
`merge (tsource, fsource, mask)`	8.13.1	`tsource` when `mask` is true and `fsource` otherwise.
`min (a1, a2 [,a3,...])`	8.3.2	Minimum value.
`minexponent (x)`	8.7.2	minimum exponent in the model for numbers like `x`.
`minloc (array [,mask])` or `minloc (array, dim [,mask])`	8.14	Location of minimum array element (`dim` in Fortran 95 only).
`minval (array [,mask])` or `minval (array, dim [,mask])`	8.11	Value of minimum array element (`mask` as second positional argument in Fortran 95 only).
`mod (a, p)`	8.3.2	Remainder modulo `p`, that is `a-int(a/p)*p`.
`modulo (a, p)`	8.3.2	`a` modulo `p`.
`call mvbits (from, frompos, len, to, topos)`	8.8.3	Copy bits.
`nearest (x, s)`	8.7.3	Nearest different machine number in the direction given by the sign of `s`.
`nint (a [,kind])`	8.3.1	Nearest integer.
`not (i)`	8.8.2	Logical complement of the bits.
`null([mold])`	8.15	Disassociated pointer (Fortran 95 only).
`pack (array, mask [,vector])`	8.13.2	Pack elements corresponding to true elements of `mask` into rank-one result.
`precision (x)`	8.7.2	Decimal precision in the model for `x`.
`present (a)`	8.2	True if optional argument is present.

`product (array [,mask])` or `product (array, dim [,mask])`	8.11	Product of array elements (`mask` as second positional argument in Fortran 95 only).
`radix (x)`	8.7.2	Base of the model for numbers like `x`.
`call random_number (harvest)`	8.16.3	Random numbers in range $0 \leq x < 1$.
`call random_seed ([size] [put] [get])`	8.16.3	Initialize or restart random number generator.
`range (x)`	8.7.2	Decimal exponent range in the model for `x`.
`real (a [,kind])`	8.3.1	Convert to real type.
`repeat (string, ncopies)`	8.6.2	Concatenates `ncopies` of `string`.
`reshape (source, shape [,pad] [,order])`	8.13.3	Reshape `source` to shape `shape`.
`rrspacing (x)`	8.7.3	Reciprocal of the relative spacing of model numbers near `x`.
`scale (x, i)`	8.7.3	$x \times b^i$, where b=`radix(x)`.
`scan (string, set [,back])`	8.5.3	Index of left-most (right-most if `back` is true) character of `string` that belongs to `set`; zero if none belong.
`selected_int_kind (r)`	8.7.4	Kind of type parameter for specified exponent range.
`selected_real_kind ([p] [,r])`	8.7.4	Kind of type parameter for specified precision and exponent range.
`set_exponent (x, i)`	8.7.3	Model number whose sign and fractional part are those of `x` and whose exponent part is `i`.
`shape (source)`	8.12.2	Array (or scalar) shape.
`sign (a, b)`	8.3.2	Absolute value of `a` times sign of `b`.
`sin (x)`	8.4	Sine function.
`sinh (x)`	8.4	Hyperbolic sine function.
`size (array [,dim])`	8.12.2	Array size.
`spacing (x)`	8.7.3	Absolute spacing of model numbers near `x`.
`spread (source, dim, ncopies)`	8.13.4	`ncopies` copies of `source` forming an array of rank one greater.

`sqrt (x)`	8.4	Square root function.
`sum (array [,mask])` or `sum(array, dim [,mask])`	8.11	Sum of array elements (`mask` as second positional argument in Fortran 95 only).
`call system_clock ([count] [,count_rate] [,count_max])`	8.16.1	Integer data from real-time clock.
`tan (x)`	8.4	Tangent function.
`tanh (x)`	8.4	Hyperbolic tangent function.
`tiny (x)`	8.7.2	Smallest positive number in the model for numbers like `x`.
`transfer (source, mold [,size])`	8.9	Same physical representation as `source`, but type of `mold`.
`transpose (matrix)`	8.13.6	Matrix transpose.
`trim (string)`	8.6.2	Remove trailing blanks from a single string.
`ubound (array [,dim])`	8.12.2	Array upper bounds.
`unpack (vector, mask, field)`	8.13.2	Unpack elements of `vector` corresponding to true elements of `mask`.
`verify (string, set [,back])`	8.5.3	Zero if all characters of `string` belong to `set` or index of left-most (right-most if `back` true) that does not.

B. Fortran 90/95 statements

Notes:

- Obsolescent features (see Appendix C) have not been included in this list.
- Where no optional blank is indicated between two adjacent keywords, the blank is mandatory.

Statement	Section
NON-EXECUTABLE STATEMENTS	
Program Units and Subprograms	
`program` *program-name*	5.2
`module` *module-name*	5.5
`end[ ][module` [*module-name*]]	5.5
`use` *module-name* [,*rename-list*]	7.10
`use` *module-name*, `only:` [*only-list*]	7.10
`private` [[::] *access-id-list*]	7.6 & 7.11
`public` [[::] *access-id-list*]	7.6
`external` *external-name-list*	5.11
`intrinsic` [::] *intrinsic-name-list* (:: in Fortran 95 only)	8.1.3
[*prefix*] `subroutine` *subroutine-name* [([*dummy-argument-list*])]	5.20
where *prefix* is `recursive` or, for Fortran 95, pure or `elemental`	
[*prefix*] `function` *function-name* ([*dummy-argument-list*]) [`result`(*result-name*)]	5.20
where *prefix* is *type* [`recursive`] or `recursive` [*type*] or also, for Fortran 95, pure or `elemental`	
`entry` *entry-name* [([*dummy-argument-list*]) [`result`(*result-name*)]]	11.2.6
`intent` (*inout*) [::] *dummy-argument-name-list*	7.8
where *inout* is `in`, `out`, or `in[ ]out`	
`optional` [::] *dummy-argument-name-list*	7.8
`save` [[::] *saved-entity-list*]	7.9

where *saved-entity* is *variable-name* or */common-block-name/*	
`contains`	5.2
`interface` [*generic-spec*]	5.18
where *generic-spec* is *generic-name*, `operator(`*defined-operator*`)`, or `assignment(=)`	
`end[ ]interface` [*generic-spec*] (*generic-spec* in Fortran 95 only)	5.18
`module procedure` *procedure-name-list*	5.18

Data Specification

type [[, *attribute*]... ::] *entity-list*	7.12
where *type* is `integer[([kind=]`*kind-value*`)]`, `real[([kind=]`*kind-value*`)]`, `logical[([kind=]`*kind-value*`)]`, `complex[([kind=]`*kind-value*`)]`, `character[`*actual-parameter-list*`]`, `double[ ]precision`, or `type(`*type-name*`)`	7.13
and *attribute* is `parameter`, `public`, `private`, `pointer`, `target`, `allocatable`, `dimension(`*bounds-list*`)`, `intent(`*inout*`)`, `external`, `intrinsic`, `optional` or `save`	7.12
`implicit none`	7.2
`implicit` *type*`(`*letter-spec-list*`)` [,*type*`(`*letter-spec-list*`)`]...	7.2
`type` [[, *access*]::] *type-name*	7.11
where *access* is `public` or `private`	7.6
type[[,*component-attr*]... ::] *component-decl-list*	7.11
where *component-attr* is `pointer` or `dimension(`*bounds-list*`)` and *component-decl* is *component-name*`[(`*bounds-list*`)]` [**char-len*] [*comp-init*] and *comp-init* (Fortran 95 only) is = *expr* or => `null()`	
`end[ ]type` [*type-name*]	7.11
`sequence`	11.2.1
`data` *object-list/value-list/* [[,] *object-list/value-list/*]...	7.5.2
`block[ ]data` [*block-data-name*]	11.2.4
`end[[ ]block[ ]data` [*block-data-name*]]	11.2.4
`parameter` (*named-constant-definition-list*)	11.4.2
`namelist` */namelist-group-name/ variable-name-list* [[,]*/namelist-group-name/ variable-name-list*]...	7.15
`dimension` [::] *array-name*`(`*array-spec*`)` [, *array-name*`(`*array-spec*`)`]...	11.4.2
`allocatable` [::] *array-name* [`(`*array-spec*`)`] [,*array-name* [`(`*array-spec*`)`]]...	7.7

`pointer` [::] *object-name* [(*array-spec*)] [,*object-name* [(*array-spec*)]]...	7.7
`target` [::] *object-name* [(*array-spec*)] [,*object-name* [(*array-spec*)]]...	7.7
`equivalence` (*object, object-list*) [, (*object, object-list*)]...	11.2.2
`common` [/[*cname*]/]*vlist* [[,]/[*cname*]/ *vlist*]...	11.2.3

EXECUTABLE STATEMENTS

Assignment

variable = *expr*	
where *variable* may be an array and may be a subobject	Chap. 3
pointer => *target*	3.12
`if` (*scalar-logical-expr*) *action-stmt*	4.3.1
where (*logical-array-expr*) *array-variable* = *expr*	6.8
`forall`(*index* = *lower: upper* [:*stride*] [, *index* = *lower: upper* [:*stride*]]... [,*scalar-logical-expr*]) *assignment* (Fortran 95)	6.9

Program Units and Subprograms

`call` *subroutine-name* [([*actual-argument-list*])]	5.13
`return`	5.8
end[ ][*unit* [*unit-name*]]	Chap. 5
where *unit* is `program`, `subroutine`, or `function`.	

Dynamic Storage Allocation

`allocate` (*allocation-list* [, `stat`=*stat*])	6.5.2
`deallocate` (*allocate-object-list* [, `stat`=*stat*])	6.5.3
`nullify` (*pointer-object-list*)	6.5.4

Control Constructs

[*do-name*:] `do` [*label*] [[,] *do-variable* = *scalar-integer-expr*, *scalar-integer-expr* [,*scalar-integer-expr*]]	4.5
[*do-name*:] `do` [*label*] [,] `while`(*scalar-logical-expr*)	11.3.2
`cycle` [*do-name*]	4.5
`exit` [*do-name*]	4.5
`continue`	4.5
end[ ]do [*do-name*]	4.5
[*if-name*:] `if` (*scalar-logical-expr*) `then`	4.3.2
`else`[[ ]`if` (*scalar-logical-expr*) `then`] [*if-name*]	4.3.2

end[ ]if [*if-name*]	4.3.2
[*select-name*:] select[ ]case (*scalar-expr*)	4.4
case (*case-value-list*) [*select-name*]	4.4
case default [*select-name*]	4.4
end[ ]select [*select-name*]	4.4
go[ ]to *label*	4.2
stop [*access-code*]	5.3
[*where-name*:] where (*logical-array-expr*)	6.8
elsewhere [*where-name*]	6.8
elsewhere (*logical-array-expr*) [*where-name*] (Fortran 95)	6.8.1
end[ ]where [*where-name*]	6.8
[*forall-name*:] forall(*index = lower: upper* [:*stride*] [, *index = lower: upper* [:*stride*]]... [,*scalar-logical-expr*]) (Fortran 95)	6.9
end[ ]forall [*forall-name*] (Fortran 95)	6.9

Input-Output

read (*control-list*) [*input-list*]	9.17
read *format* [, *input-list*]	9.7
write (*control-list*) [*output-list*]	9.17
print *format* [, *output-list*]	9.8
rewind *external-file-unit*	10.2.2
rewind (*position-list*)	10.2.2
end[ ]file *external-file-unit*	10.2.3
end[ ]file (*position-list*)	10.2.3
backspace *external-file-unit*	10.2.1
backspace (*position-list*)	10.2.1
open (*connect-list*)	10.3
close (*close-list*)	10.4
inquire (*inquire-list*)	10.5
inquire (iolength = *length*) *olist*	10.5
format ([*format-list*]) (this statement is actually non-executable).	9.4

C. Obsolescent features

C.1 Obsolescent in Fortran 95 only

The features of this section were of first-class status in Fortran 90. However, the Fortran 95 standard defines them to be obsolescent, and they are thus, in a Fortran 95 context, part of this Appendix. Their replacements are described in the relevant subsections.

C.1.1 Fixed source form

The fixed source form has been replaced by the free source form (Section 2.4). In the old form, each statement consists of one or more *lines* exactly 72 characters long,[1] and each line is divided into three *fields*. The first field consists of positions 1 to 5 and may contain a *statement label*. A Fortran statement may be written in the second fields of up to 20 consecutive lines. The first line of a multi-line statement is known as the *initial line* and the succeeding lines as *continuation lines*.

A non-comment line is an initial line or a continuation line depending on whether there is a character, other than zero or blank, in position 6 of the line, which is the second field. The first field of a continuation line must be blank. The ampersand is not used for continuation.

The third field, from positions 7 to 72, is reserved for the Fortran statements themselves. Note that if a construct is named, the name must be placed here and not in the label field.

Except in a character context, blanks are insignificant.

The presence of an asterisk (*) or a character c in position 1 of a line indicates that the whole line is commentary. An exclamation mark indicates the start of commentary, except in position 6, where it indicates continuation.

Several statements separated by a semi-colon (;) may appear on one line. The semi-colon may not, in this case, be in column 6, where it would indicate continuation. Only the first of the statements on a line may be labelled. A semi-colon that is the last non-blank character of a line, or the last non-blank character ahead of commentary, is ignored.

[1] This limit is processor dependent if the line contains characters other than those of the default type.

A program unit end statement must not be continued, and any other statement with an initial line that appears to be a program unit end statement must not be continued.

A processor may restrict the appearance of its defined control characters, if any, in the fixed source form.

In applications where a high degree of compatibility between the old and the new source forms is required, for instance in code to be included into several programs which might exist in different forms, observance of the following rules can be of great help:

- confine statement labels to positions 1 to 5 and statements to positions 7 to 72;
- treat blanks as being significant;
- use only ! to indicate a comment (but not in position 6);
- for continued statements, place an ampersand in both position 73 of a continued line and position 6 of a continuing line.

Also, a tool to facilitate the conversion of Fortran 77 code to the free source form can be obtained by anonymous ftp to *ftp.numerical.rl.ac.uk*; the directory is */pub/MandR* and the file name is *convert.f90*. An alternative ftp address is available at *asisftp.cern.ch*, in the directory *dist*.

C.1.2 Computed go to

A form of branch statement is the computed `go to`, which enables one path among many to be selected, depending on the value of a scalar integer expression. The general form is

`go to` (*sl1, sl2, sl3*`,...`) [`,`] *intexpr*

where *sl1, sl2, sl3* etc. are labels of statements in the same scoping unit, and *intexpr* is any scalar integer expression. The same statement label may appear more than once. An example is

```
go to (6,10,20) i(k)**2+j
```

which references three statement labels. When the statement is executed, if the value of the integer expression is 1, the first branch will be taken, and control is transferred to the statement labelled 6. If the value is 2, the second branch will be taken, and so on. If the value is less than 1, or greater than 3, no branch will be taken, and the next statement following the `go to` will be executed.

This statement is replaced by the `case` construct (Section 4.4).

C.1.3 Character length specification character*

Alternatives to

`character([len=]`*len-value*`)`

as a *type* in a type declaration, `function`, `implicit`, or component definition statement are

```
character*(len-value)[,]
```

and

```
character*len[,]
```

where *len* is an integer literal constant without a specified kind value and the optional comma is permitted only in a type declaration statement and only when `::` is absent:

```
character*20 word, letter*1
```

Note that these alternative forms are provided only for default characters.

C.1.4 Data statements among executables

The `data` statement may be placed among the executable statements, but such placement offers no extra advantage and is rarely used and not recommended, since data initialization properly belongs with the specification statements.

C.1.5 Statement functions

It may happen that within a single program unit there are repeated occurrences of a computation which can be represented as a single statement. For instance, to calculate the parabolic function represented by

$$y = a + bx + cx^2$$

for different values of x, but with the same coefficients, there may be references to

```
y1 = 1. + x1*(2. + 3.*x1)
:
y2 = 1. + x2*(2. + 3.*x2)
:
```

etc. In Fortran 77, it was more convenient to invoke a so-called *statement function* (now better coded as an internal subroutine, Section 5.6), which must appear after any `implicit` and other relevant specification statements and before the executable statements. The example above would become

```
parab(x) = 1. + x*(2. + 3.*x)
:
y1 = parab(x1)
:
y2 = parab(x2)
```

Here, x is a dummy argument, which is used in the definition of the statement function. The variables x1 and x2 are actual arguments to the function.

The general form is

function-name([*dummy-argument-list*]) = *scalar-expr*

where the *function-name* and each *dummy-argument* must be specified, explicitly or implicitly, to be scalar data objects. To make it clear that this is a statement function and not an assignment to a host array element, we recommend declaring the type by placing the *function-name* in a type declaration statement; this is *required* whenever a host entity has the same name. The *scalar-expr* must be composed of constants, references to scalar variables, references to functions, and intrinsic operations. If there is a reference to a function, the function must not be a transformational intrinsic nor require an explicit interface, the result must be scalar, and any array argument must be a named array. A reference to a non-intrinsic function must not require an explicit interface. A named constant that is referenced or an array of which an element is referenced must be declared earlier in the scoping unit or be accessed by use or host association. A scalar variable referenced may be a dummy argument of the statement function or a variable that is accessible in the scoping unit. A dummy argument of the host procedure must not be referenced unless it is a dummy argument of the main entry or of an `entry` that precedes the statement function. If any entity is implicitly typed, a subsequent type declaration must confirm the type and type parameters. The dummy arguments are scalar and have a scope of the statement function statement only.

A statement function always has an implicit interface and may not be supplied as a procedure argument. It may appear within an internal procedure, and may reference other statement functions appearing before it in the same scoping unit, but not itself nor any appearing after. A function reference in the expression must not redefine a dummy argument. A statement function is pure (Section 6.10) if it references only pure functions.

A statement function statement is not permitted in an interface block.

Note that statement functions are irregular in that use and host association are not available.

C.1.6 Assumed character length of function results

A non-recursive external function whose result is scalar, character, and non-pointer may have assumed character length as in Figure C.1. Such a function is not permitted to specify a defined operation. In a scoping unit that invokes such

a function, the interface must be implicit and there must be a declaration of the length, as in Figure C.2, or such a declaration must be accessible by use or host association.

Figure C.1

```
function copy(word)
   character(len=*) copy, word
   copy = word
end function copy
```

Figure C.2

```
program main
   external copy              ! Interface block not allowed.
   character(len=10) copy
   write(*, *) copy('This message will be truncated')
end program main
```

This facility is included only for compatibility with Fortran 77 and is completely at variance with the philosophy of Fortran 90/95 that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association.

This facility may be replaced by use of a subroutine whose arguments correspond to the function result and the function arguments.

C.2 Obsolescent in Fortran 90 and 95

The features of this section are obsolescent in both Fortran 90 and Fortran 95. Their replacements are described in the relevant subsections.

C.2.1 Arithmetic if statement

The arithmetic `if` provides a three-way branching mechanism, depending on whether an arithmetic expression has a value which is less than, equal to, or greater than zero. It is replaced by the `if` statement and construct (Section 4.3). Its general form is

`if` (*expr*) *sl1, sl2, sl3*

where *expr* is any scalar expression of type integer or real, and *sl1*, *sl2*, and *sl3* are the labels of statements in the same scoping unit. If the result obtained by evaluating *expr* is negative then the branch to *sl1* is taken, if the result is zero the branch to *sl2*, and if the result is greater than zero the branch to *sl3*. An example is

```
      if (p-q) 1,2,3
1     p = 0.
      go to 4
2     p = 1.
      q = 1.
      go to 4
3     q = 0.
4     ...
```

in which a branch to 1, 2 or 3 is taken depending on the value of p-q. The arithmetic `if` may be used as a two-way branch when two of the labels are identical:

```
      if (x-y) 1,2,1
```

C.2.2 Shared do loop termination

A do-loop may be terminated on a labelled statement other than an `end do` or `continue`. Such a statement must be an executable statement other than `go to`, `return` or an `end` statement of a subprogram, `stop` or an `end` statement of a main program, `exit`, `cycle`, arithmetic-`if`, or assigned `go to` statement. Nested do-loops may share the same labelled terminal statement, in which case all the usual rules for nested blocks hold, but a branch to the label must be from within the innermost loop. Thus we may write a matrix multiplication as

```
      a(1:n, 1:n) = 0.
      do 1 i = 1,n
         do 1 j = 1,n
            do 1 l = 1,n
1              a(i,j) = a(i,j)+b(i,l)*c(l,j)
```

Execution of a `cycle` statement restarts the loop without execution of the terminal statement. This form of do-loop offers no additional functionality but considerable scope for unexpected mistakes.

C.2.3 Alternate return

When calling certain types of subroutines, it is possible that specific exceptional conditions will arise, which should cause a break in the normal control flow. It is possible to anticipate such conditions, and to code different flow paths following a subroutine call, depending on whether the called subroutine has terminated normally, or has detected an exceptional or abnormal condition. This is achieved using the alternate `return` facility which uses the argument list in the following manner. Let us suppose that a subroutine `deal` receives in an argument list the number of cards in a shuffled deck, the number of players and the number of cards to be dealt to each hand. In the interests of generality, it would be a reasonable

precaution for the first executable statement of deal to be a check that there is at least one player and that there are, in fact, enough cards to satisfy each player's requirement. If there are no players or insufficient cards, it can signal this to the main program which should then take the appropriate action. This may be written in outline as

```
      call deal(nshuff, nplay, nhand, cards, *2, *3)
      call play
      :
2     ........   ! Handle no-player case.
      :
3     ........   ! Handle insufficient-cards case.
      :
```

If the cards can be dealt, normal control is returned, and the call to play executed. If an exception occurs, control is passed to the statement labelled 2 or 3, at which point some action must be taken — to stop the game or shuffle more cards. The relevant statement label is defined by placing the statement label preceded by an asterisk as an actual argument in the argument list. It must be a label of an executable statement of the same scoping unit. Any number of such alternate returns may be specified, and they may appear in any position in the argument list. Since, however, they are normally used to handle exceptions, they are best placed at the end of the list.

In the called subroutine, the corresponding dummy arguments are asterisks and the alternate return is taken by executing a statement of the form

```
return intexpr
```

where *intexpr* is any scalar integer expression. The value of this expression at execution time defines an index to the alternate return to be taken, according to its position in the argument list. If *intexpr* evaluates to 2, the second alternate return will be taken. If *intexpr* evaluates to a value which is less than 1, or greater than the number of alternate returns in the argument list, a normal return will be taken. Thus, in deal, we may write simply

```
subroutine deal(nshuff, nplay, nhand, cards, *, *)
:
if (nplay.le.0) return 1
if (nshuff .lt. nplay*nhand) return 2
```

This feature is also available for subroutines defined by entry statements. It is not available for functions or elemental subroutines.

This feature is replaced by use of an integer argument holding a return code used in a computed go to following the call statement. A following case construct is now an even better alternative.

C.3 Obsolescent in Fortran 90, deleted in Fortran 95

The features of this section are obsolescent in Fortran 90 and are deleted from the Fortran 95 language entirely. They should thus be regarded, in a Fortran 95 context, as being absent from this book. Although it can be expected that compilers will continue to support these features for some period, their use should be completely avoided to ensure very long-term portability and to avoid unnecessary compiler warning messages. Their replacements are described in the relevant subsections.

C.3.1 Non-integer do indices

The do variable and the expressions that specify the limits and stride of a do-construct or an implied-do in an I/O statement may be of type default real or double precision real. We may therefore write a loop such as

```
do 1 a = 1, 15.7, 2.1
```

in which the real variable a will assume the initial value of 1.0 (note the conversion), and will subsequently have the values 3.1, 5.2, etc. up to 15.7.

There are, however, serious problems associated with do-loops with real indices, and in order to understand them we recall how the number of iterations of a do-loop is actually determined:

```
max(int((expr2-expr1+expr3)/expr3),0)
```

A consequence of this formula is rather insidious, and results from the application of the int function. Consider the statement

```
do 1 a = -0.3, -2.1, -0.3
```

which we would normally expect to result in seven iterations of the loop it controls. The number of iterations is obtained from the result of a computation whose intermediate value may not be 7.00000.. but 6.99999.., due to rounding errors. After applying the int function we then have the integer 6 as the number of iterations. Whether or not this rounding error will occur for a given loop on a given computer is difficult to foresee, and this is the reason for avoiding the use of real do-loop parameters. A similar problem can arise with a long loop: the repeated addition of the one to the other can lead to an unexpected loss of precision.

C.3.2 Assigned go to and assigned formats

Another form of branch statement is actually written in two parts, an assign statement and an assigned go to statement. The form is

```
assign sl1 to intvar
:
```

```
assign sl2 to intvar
:
go to intvar [[,](sl1,sl2,... )]
```

where *sl1*, *sl2* etc. are labels of statements in the same scoping unit, and *intvar* is a named scalar default integer variable (so cannot be an array element or structure component). When an `assign` statement is executed, *intvar* acquires a representation of a statement label. Different labels may be assigned in different parts of the scoping unit. When the assigned `go to` is executed, then depending on the value of *intvar*, the appropriate path is taken. If the optional statement label list in the `go to` statement is present, it must contain the label in *intvar*, and permits a check that *intvar* has acquired an expected value during program execution. A label may appear more than once in the list.

The assigned `go to`'s main purpose is to control logic flow in a scoping unit having a number of paths which come together at one point at which some common code is executed, and from which a new branch is taken depending on the path taken before. This is shown in Figure C.3, where we see three paths joining before the `go to`, and three after. For this type of application, an internal subprogram (Section 5.6) is more appropriate.

Figure C.3

```
      :
      x = y+1.
      assign 4 to jump
      go to 3
4     :
1     x = y+2.
      assign 5 to jump
      go to 3
5     :
2     x = y+3.
      assign 6 to jump
6     :
3     z = x**2
      :
      go to jump (4,5,6)
      :
```

A default integer variable to which a statement label has been assigned in an `assign` statement may also be used to specify a `format` statement:

```
      assign 10 to key
      :
      print key, q
10    format(f10.3)
```

The label must be that of a format statement in the same scoping unit as the I/O statement.

This use of the assign statement is replaced by character expressions to define format specifiers (Section 9.4). This feature is not part of Fortran 95.

C.3.3 Branching to an end if statement

It is permissible to branch to an end if statement from outside the construct that it terminates. A branch to the following statement is a replacement for this practice, which is not part of Fortran 95.

C.3.4 The pause statement

At certain points in the execution of a program it might be useful to pause, in order to allow some possible external intervention in the running conditions to be made, for instance for an operator to activate a peripheral device required by the program. This can be achieved by executing a pause statement which may also contain a default character constant, or a string of up to five digits, for example

```
pause 'Please mount the next disc pack'
pause 1234
```

Execution is resumed by some form of external command, for instance one given by an operator. The effect may be achieved with a read statement that awaits data. This feature is not part of Fortran 95.

C.3.5 H edit descriptor

The H (or h) edit descriptor provides an alternative character string edit descriptor: the output string is preceded by an *n*H edit descriptor, where *n* is the number of default characters in the following string (blanks being significant):

```
100  format(23HI must count characters)
```

The value *n* must be an integer literal constant without a kind parameter. If the Hollerith string occurs within a character constant delimited by apostrophes and contains an apostrophe, the apostrophe must be represented by two apostrophes, but counts as only one in the *n*H character count, as in the example

```
print '(7H Don''t, a)', caution
```

and similarly for quotes.

The H edit descriptor provides the same functionality as the character string edit descriptor but is prone to error as it is easy to miscount the number of characters in the string. The feature is not part of Fortran 95.

D. Pointer example

A recurring problem in computing is the need to manipulate a linked data structure. This might be a simple linked list like the one encountered in Section 2.13, but often a more general tree structure is required.

The example in this Appendix consists of a module that establishes and navigates one or more such trees, organized as a 'forest', and a short test program for it. Here, each node is identified by a name and has any number of children, any number of siblings, and (optionally) some associated real data. Each root node is regarded as having a common parent, the 'forest root' node, whose name is 'forest_root'. Thus, every node has a parent. The module provides facilities for adding a named node to a specified parent, for enquiring about all the nodes that are offspring of a specified node, for removing a tree or subtree, and for performing I/O operations on a tree or subtree.

The user-callable interfaces are:

start: must be called to initialize a forest.

add_node: stores the data provided at the node whose parent is specified and sets up pointers to the parent and siblings (if any).

remove_node: deallocates all the storage occupied by a complete tree or subtree.

retrieve: retrieves the data stored at a specified node and the names of the parent and children.

dump_tree: writes a complete tree or subtree.

restore_tree: reads a complete tree or subtree.

finish: deallocates all the storage occupied by all the trees of the forest.

The source code can be obtained by anonymous ftp to *ftp.numerical.rl.ac.uk.* When prompted for a userid, reply with

```
anonymous
```

and give your name as password. The directory is */pub/MandR* and the file name is *pointer.f90*. An alternative address is *asisftp.cern.ch*, in the directory *dist.*

```
module directory
!
! Strong typing imposed
   implicit none
!
! Only subroutine interfaces, the length of the character
! component, and the I/O unit number are public
   public  :: start, add_node, remove_node, retrieve,              &
              dump_tree, restore_tree, finish
   private :: find, remove
!
! Module constants
   character(len=*), private, parameter :: eot = "End-of-Tree....."
   integer, parameter, public :: unit = 4,   & ! I/O unit number
                                 max_char = 16 ! length of character
                                               ! component
!
! Define the basic tree type
   type, private :: node
       character(len=max_char)     :: name           ! name of node
       real, pointer, dimension(:) :: y              ! stored real data
       type(node), pointer         :: parent         ! parent node
       type(node), pointer         :: sibling        ! next sibling node
       type(node), pointer         :: child          ! first child node
   end type node

!
! Module variables
   type(node), pointer, private   :: current     ! current node
   type(node), pointer, private   :: forest_root ! the root of the forest
   integer,private                :: max_data    ! max size of data array
   character(len=max_char), private, allocatable, target, dimension(:)  &
                                                          :: names
                                       ! for returning list of names
! The module procedures

contains

    subroutine start ()
! Initialize the tree.
      allocate (forest_root)
      current => forest_root
      forest_root%name = "forest_root"
      nullify(forest_root%parent, forest_root%sibling, forest_root%child)
      allocate(forest_root%y(0))
      max_data = 0
      allocate (names(0))
   end subroutine start
```

```
   subroutine find(name)
      character(len=*), intent(in) :: name
! Make the module variable current point to the node with given name,
! or be null if the name is not there.
      type(node), pointer    :: root
! For efficiency, we search the tree rooted at current, and if this
! fails try its parent and so on until the forest root is reached.
      if (associated(current)) then
         root => current
         nullify (current)
      else
         root => forest_root
      end if
      do
         call look(root)
         if (associated(current)) then
            return
         end if
         root => root%parent
         if (.not.associated(root)) then
            exit
         end if
      end do
   contains
      recursive subroutine look(root)
         type(node), pointer    :: root
! (type(node), intent(in), target :: root   is standard conforming too)
! Look for name in the tree rooted at root. If found, make the
! module variable current point to the node
         type(node), pointer    :: child
!
         if (root%name == name) then
            current => root
         else
            child => root%child
            do
               if (.not.associated(child)) then
                 exit
               end if
               call look(child)
               if (associated(current)) then
                 return
               end if
               child => child%sibling
            end do
         end if
      end subroutine look
   end subroutine find
```

```
   subroutine add_node(name, name_of_parent, data)
      character(len=*), intent(in)             :: name, name_of_parent
! For a root, name_of_parent = ""
      real, intent(in), optional, dimension(:) :: data
! Allocate a new tree node of type node, store the given name and
! data there, set pointers to the parent and to its next sibling
! (if any). If the parent is not found, the new node is treated as
! a root. It is assumed that the node is not already present in the
! forest.
      type(node), pointer :: new_node
!
      allocate (new_node)
      new_node%name = name
      if (present(data)) then
         allocate(new_node%y(size(data)))
         new_node%y = data
         max_data = max(max_data, size(data))
      else
         allocate(new_node%y(0))
      end if
!
! If name of parent is not null, search for it.
! If not found, print message.
      if (name_of_parent == "") then
         current => forest_root
      else
         call find (name_of_parent)
         if (.not.associated(current)) then
            print *, "no parent ", name_of_parent, " found for ", name
            current => forest_root
         end if
      end if
      new_node%parent => current
      new_node%sibling => current%child
      current%child => new_node
      nullify(new_node%child)
   end subroutine add_node

   subroutine remove_node(name)
      character(len=*), intent(in) :: name
! Remove node and the subtree rooted on it (if any),
! deallocating associated pointer targets.
      type(node), pointer :: parent, child, sibling
!
      call find (name)
      if (associated(current)) then
         parent =>  current%parent
         child => parent%child
         if (.not.associated(child, current)) then
```

```
! Make it the first child, looping through the siblings to find it
! and resetting the links
            parent%child => current
            sibling => child
            do
              if (associated (sibling%sibling, current)) then
                exit
              end if
              sibling => sibling%sibling
            end do
            sibling%sibling => current%sibling
            current%sibling => child
          end if
          call remove(current)
        end if
    end subroutine remove_node

    recursive subroutine remove (old_node)
! Remove a first child node and the subtree rooted on it (if any),
! deallocating associated pointer targets.
        type(node), pointer :: old_node
        type(node), pointer :: child, sibling
!
        child => old_node%child
        do
          if (.not.associated(child)) then
            exit
          end if
          sibling => child%sibling
          call remove(child)
          child => sibling
        end do
! remove leaf node
        if (associated(old_node%parent)) then
          old_node%parent%child => old_node%sibling
        end if
        deallocate (old_node%y)
        deallocate (old_node)
    end subroutine remove

    subroutine retrieve(name, data, parent, children)
        character(len=*), intent(in)            :: name
        real, pointer, dimension(:)             :: data
        character(len=*), intent(out)           :: parent
        character(len=*), pointer, dimension(:) :: children
! Returns a pointer to the data at the node, the name of the
! parent, and a pointer to the names of the children.
        integer :: counter, i
        type(node), pointer :: child
```

```
!
      call find (name)
      if (associated(current)) then
         data => current%y
         parent = current%parent%name
! Count the number of children
         counter = 0
         child => current%child
         do
           if (.not.associated(child)) then
             exit
           end if
           counter = counter + 1
           child => child%sibling
         end do
         deallocate (names)
         allocate (names(counter))
! and store their names
         children => names
         child => current%child
         do i = 1, counter
            children(i) = child%name
            child => child%sibling
         end do
      else
         nullify(data)
         parent = ""
         nullify(children)
      end if
   end subroutine retrieve

   subroutine dump_tree(root)
      character(len=*), intent(in) :: root
! Write out a complete tree followed by an end-of-tree record
! unformatted on the file unit.
      call find (root)
      if (associated(current)) then
         call tree_out(current)
      end if
      write(unit = unit) eot, 0, eot
   contains
      recursive subroutine tree_out (root)
! Traverse a complete tree or subtree, writing out its contents
        type(node), intent(in) :: root     ! root node of tree
! Local variable
        type(node), pointer    :: child
!
        write(unit = unit) root%name, size(root%y), root%y, &
                           root%parent%name
```

```
         child => root%child
         do
            if (.not.associated(child)) then
              exit
            end if
            call tree_out (child)
            child => child%sibling
         end do
      end subroutine tree_out
   end subroutine dump_tree

   subroutine restore_tree ()
! Reads a subtree unformatted from the file unit.
      character(len=max_char)         :: name
      integer :: length_y
      real, allocatable, dimension(:) :: y
      character(len=max_char)         :: name_of_parent
!
      allocate(y(max_data))
      do
         read (unit= unit) name, length_y, y(:length_y), name_of_parent
         if (name == eot) then
           exit
         end if
         call add_node( name, name_of_parent, y(:length_y) )
      end do
      deallocate(y)
   end subroutine restore_tree

   subroutine finish ()
! Deallocate all allocated targets.
      call remove (forest_root)
      deallocate(names)
   end subroutine finish

end module directory

module tree_print

   use  directory
   implicit none
   private
   public :: print_tree

contains

   recursive subroutine print_tree(name)
! To print the data contained in a subtree
      character(len=*), intent(in)                    :: name
```

```
      integer                                                :: i
      real, pointer, dimension(:)                            :: data
      character(len=max_char)                                :: parent, self
      character(len=max_char), pointer, dimension(:)         :: children
      character(len=max_char), allocatable, dimension(:) :: siblings
!
      call retrieve(name, data, parent, children)
      if (.not.associated(data)) then
        return
      end if
      self = name
      write(unit=*,fmt=*) self, data
      write(unit=*,fmt=*) "   parent:   ", parent
      if (size(children) > 0 ) then
         write(unit=*,fmt=*)  "   children: ", children
      end if
      allocate(siblings(size(children)))
      siblings = children
      do i = 1, size(children)
         call print_tree(siblings(i))
      end do
      deallocate(siblings)
   end subroutine print_tree
end module tree_print

program test
   use directory
   use tree_print
   implicit none
!
! Initialize a tree
   call start ()
! Fill it with some data
   call add_node("ernest","",(/1.0,2.0/))
   call add_node("helen","ernest",(/3.0,4.0,5.0/))
   call add_node("douglas","ernest",(/6.0,7.0/))
   call add_node("john","helen",(/8.0/))
   call add_node("betty","helen",(/9.0,10.0/))
   call add_node("nigel","betty",(/11.0/))
   call add_node("peter","betty",(/12.0/))
   call add_node("ruth","betty")
! Manipulate subtrees
   open(unit=unit, form="unformatted", status="scratch",    &
        action="readwrite", position="rewind")
   call dump_tree("betty")
   call remove_node("betty")
   write(unit=*,fmt=*)
   call print_tree("ernest")
   rewind (unit=unit)
```

```
   call restore_tree ()
   rewind (unit=unit)
   write(unit=*,fmt=*)
   call print_tree("ernest")
   call dump_tree("john")
   call remove_node("john")
   write(unit=*,fmt=*)
   call print_tree("ernest")
   rewind (unit=unit)
   call restore_tree ()
   write(unit=*,fmt=*)
   call print_tree("ernest")
! Return storage
   call finish ()

end program test
```

E. Fortran terms

The following is a list of the principal technical terms used in this book and their definitions. To facilitate reference to the standard, we have kept closely to the meanings used there. Where the definition uses a term that is itself defined in this glossary, the first occurrence of the term is printed in italics. Some terms used in Fortran 77 have a different meaning here and we draw the reader's attention to each such term by marking it with a bold asterisk *. We make no reference to obsolescent or deleted features (Appendix C) in this Appendix.

Actual argument An *expression*, a *variable*, or a *procedure* that is specified in a procedure *reference*.

Allocatable array A *named array* having the `allocatable` *attribute*. Only when it has space allocated for it does it have a *shape* and may it be *referenced* or *defined*.

Argument An *actual argument* or a *dummy argument*.

Argument association The relationship between an *actual argument* and a *dummy argument* during the execution of a *procedure reference*.

Argument keyword A *dummy argument name*. It may used in a *procedure reference* ahead of the equals symbol provided the procedure has an *explicit interface*.

Array * A set of *scalar data*, all of the same *type* and *type parameters*, whose individual elements are arranged in a rectangular pattern. It may be a *named* array, an *array section*, a *structure component*, a *function* value, or an *expression*. Its *rank* is at least one. [In Fortran 77, arrays were always named and never constants.]

Array element One of the *scalar data* that make up an *array* that is either *named* or is a *structure component*.

Array pointer A *pointer* that is an *array*.

Array section A *subobject* whose *designator* contains a *subscript triplet*, a *vector subscript*, or an *array component selector* that is followed by one or more further component selectors.

Array-valued Having the property of being an *array*.

Assignment statement A *statement* of the form '*variable* = *expression*'.

Assignment token The *lexical token* = used in an *assignment statement*.

Association *Name association*, *pointer association*, or *storage association*.

Assumed-size array A *dummy array* whose *size* is assumed from the associated *actual argument*. Its last upper bound is specified by an asterisk.

Attribute A property of a *data object* that may be specified in a *type declaration statement*.

Belong If an `exit` or a `cycle` *statement* contains a *construct name*, the statement **belongs** to the `do` construct using that name. Otherwise, it **belongs** to the innermost `do` construct in which it appears.

Block A sequence of *executable constructs* embedded in another executable construct, bounded by *statements* that are particular to the construct, and treated as an integral unit.

Block data program unit A *program unit* that provides initial values for *data objects* in *named common blocks*.

Bounds For a *named array*, the limits within which the values of the *subscripts* of its *array elements* must lie.

Character A letter, digit, or other symbol.

Character storage unit The unit of storage for holding a *scalar* of *type* default character and character length one that is not a *pointer*.

Character string A sequence of *characters* numbered from left to right 1, 2, 3,...

Characteristics

i) Of a *procedure*, its classification as a *function* or *subroutine*, the characteristics of its *dummy arguments*, and the characteristics of its *function result* if it is a function.

ii) Of a *dummy argument*, whether it is a *data object*, is a *procedure*, or has the `optional` *attribute*.

iii) Of a *data object*, its *type*, *type parameters*, *shape*, the exact dependence of an array bound or the character length on other entities, *intent*, whether it is optional, whether it is a *pointer* or a *target*, and whether the *shape*, *size*, or *character length* is assumed.

iv) Of a *dummy procedure*, whether the interface is explicit, its characteristics as a procedure if the interface is explicit, and whether it is optional.

v) Of a *function result*, its type, type parameters, whether it is a pointer, rank if it is a pointer, shape if it is not a pointer, the exact dependence of an array bound or the character length on other entities, and whether the character length is assumed.

Collating sequence An ordering of all the different *characters* of a particular *kind type parameter.*

Common block A block of physical storage that may be accessed by any of the *scoping units* in an *executable program.*

Component A constituent of a *derived type.*

Conformable Two *arrays* are said to be **conformable** if they have the same *shape.* A *scalar* is **conformable** with any array.

Conformance An *executable program* conforms to the standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the standard. A *program unit* conforms to the standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming. A *processor* conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

Connected

i) For an *external unit*, the property of referring to an *external file.*

ii) For an *external file*, the property of having an *external unit* that refers to it.

Constant * A *data object* whose value must not change during execution of an *executable program.* It may be a *named constant* or a *literal constant.*

Constant expression An *expression* satisfying rules that ensure that its value does not vary during program execution.

Construct A sequence of *statements* starting with a `select case`, `do`, `if`, or `where` statement and ending with the corresponding terminal statement.

Data Plural of *datum.*

Data entity An *entity* that has or may have a data value. It may be a *constant*, a *variable*, an *expression*, or a *function result.*

Data object A *datum* of *intrinsic* or *derived type* or an *array* of such *data.* It may be a *literal constant*, a *named* data object, a *target* of a *pointer*, or it may be a *subobject.*

Data type A *named* category of *data* that is characterized by a set of values, together with a way to denote these values and a collection of *operations* that interpret and manipulate the values. For an *intrinsic* data type, the set of data values depends on the values of the *type parameters.*

Datum A single quantity that may have any of the set of values specified for its *data type.*

Definable A *variable* is **definable** if its value may be changed by the appearance of its *name* or *designator* on the left of an *assignment statement.* A *allocatable array* that has not been allocated is an example of a *data object* that is not definable. An example of a *subobject* that is not definable is `c(i)` when `c` is an *array* that is a *constant* and `i` is an integer variable.

Defined For a *data object,* the property of having or being given a valid value.

Defined assignment statement An *assignment statement* that is not an *intrinsic* assignment statement and is defined by a *subroutine subprogram* and an *interface block.*

Defined operation An *operation* that is not an *intrinsic* operation and is defined by a *function subprogram* and an *interface block.*

Deleted feature A feature in Fortran 77 that has been deleted from Fortran 95. No features in Fortran 77 have been deleted from Fortran 90. Note that a feature designated as an *obsolescent feature* may become a deleted feature in a future revision.

Derived type A *type* whose *data* have *components* each of which is either of *intrinsic* type or of another derived type.

Designator See *subobject designator.*

Disassociated A *pointer* is **disassociated** following execution of a `deallocate` or `nullify` *statement.*

Dummy argument An *entity* whose *name* appears in the parenthesized list following the *procedure* name in a `function` *statement,* a `subroutine` statement, an `entry` statement, or a *statement function* statement.

Dummy array A *dummy argument* that is an *array.*

Dummy pointer A *dummy argument* that is a *pointer.*

Dummy procedure A *dummy argument* that is specified or *referenced* as a *procedure.*

Elemental An adjective applied to an *intrinsic operation, procedure,* or *assignment statement* that is applied independently to the elements of an *array* or corresponding elements of a set of *conformable* arrays and *scalars.*

Entity The term used for any of the following: a *program unit*, a *procedure*, an *operator*, an *interface block*, a *common block*, an *external unit*, a *statement function*, a *type*, a *data entity*, a *statement label*, a *construct*, or a namelist group.

Executable construct A `case`, `do`, `if`, or `where` *construct.*

Executable program A set of *program units* that includes exactly one *main program.*

Executable statement An instruction to perform or control one or more computational actions.

Explicit interface For a *procedure referenced* in a *scoping unit*, the property of being an *internal procedure*, a *module procedure*, an *intrinsic procedure*, an *external procedure* that has an *interface block* or is defined by the scoping unit and is recursive, or a *dummy procedure* that has an interface block.

Explicit-shape array A *named array* that is declared with *explicit bounds.*

Expression A sequence of *operands*, *operators*, and parentheses. It may be a *variable*, a *constant*, a *function reference*, or may represent a computation.

Extent The size of one dimension of an *array.*

External file A sequence of *records* that exists in a medium external to the *executable program.*

External procedure A *procedure* that is defined by an *external subprogram* or by a means other than Fortran.

External subprogram * A *subprogram* that is not contained in a *main program*, *module*, or another subprogram. [In Fortran 77, a *block data program unit* is called a subprogram.]

External unit A mechanism that is used to refer to an *external file.* It is identified by a nonnegative integer.

File An *internal file* or an *external file.*

Function A *procedure* that is invoked in an *expression.*

Function result The *data entity* that returns the value of a *function.*

Function subprogram A sequence of *statements* beginning with a `function` statement that is not in an *interface block* and ending with the corresponding `end` statement.

Generic identifier A *name*, *operator*, or *assignment token* specified in an `interface` *statement* to provide an alternative means of invoking any of the *procedures* in the *interface block.*

Global entity An *entity* identified by a *lexical token* whose *scope* is an *executable program*. It may be a *program unit*, a *common block*, or an *external procedure*.

Host A *main program* or *subprogram* that contains an *internal subprogram* is called the **host** of the internal subprogram. A *module* that contains a *module subprogram* is called the **host** of the module subprogram.

Host association The process by which a *subprogram* or a *derived type* definition accesses *entities* of its *host*.

Implicit interface A *procedure referenced* in a *scoping unit* is said to have an **implicit interface** if the procedure does not have an *explicit interface* there.

Inquiry function An *intrinsic function* whose result depends on properties of the principal *argument* other than the value of the argument.

Instance of a subprogram The copy of a *subprogram* that is created when a *procedure* defined by the subprogram is *invoked*.

Intent Of a *dummy argument* that is a neither a *procedure* nor a *pointer*, whether it is intended to transfer data into the procedure, out of the procedure, or both.

Interface block A sequence of *statements* beginning with an `interface` statement and ending with the corresponding `end interface` statement.

Interface body A sequence of *statements* in an *interface block* beginning with a `function` or `subroutine` statement and ending with the corresponding `end` statement.

Interface of a procedure See *procedure interface*.

Internal file A character *variable* that is used to transfer and convert *data* from internal storage to internal storage.

Internal procedure A *procedure* that is defined by an *internal subprogram*.

Internal subprogram A *subprogram* contained in a *main program* or another subprogram.

Intrinsic An adjective applied to *types*, *operations*, *assignment statements*, and *procedures* that are defined in the standard and may be used in any *scoping unit* without further definition or specification.

Invoke

i) To call a *subroutine* by a `call` *statement* or by a *defined assignment statement*.

ii) To call a *function* by a *reference* to it by *name* or *operator* during the evaluation of an *expression.*

Keyword *Statement keyword* or *argument keyword.*

Kind type parameter A parameter whose values label the available kinds of an *intrinsic type.*

Label See *statement label.*

Length of a character string The number of *characters* in the *character string.*

Lexical token A sequence of one or more characters with an indivisible interpretation.

Line A source-form *record* containing from 0 to 132 *characters.*

Literal constant A *constant* without a *name.* [In Fortran 77, this was called a constant.]

Local entity An *entity* identified by a *lexical token* whose *scope* is a *scoping unit.*

Main program A *program unit* that is not a *module*, *external subprogram*, or *block data program unit.*

Many-one array section An *array section* with a *vector subscript* having two or more elements with the same value.

Module A *program unit* that contains or accesses definitions to be accessed by other program units.

Module procedure A *procedure* that is defined by a *module subprogram.*

Module subprogram A *subprogram* that is contained in a *module* but is not an *internal subprogram.*

Name * A *lexical token* consisting of a letter followed by up to 30 alphanumeric characters (letters, digits, and underscores). [In Fortran 77, this was called a symbolic name.]

Name association *Argument association*, *use association*, or *host association.*

Named Having a *name.*

Named constant * A *constant* that has a *name.* [In Fortran 77, this was called a symbolic constant.]

Numeric storage unit The unit of storage for holding a *scalar* of *type* default real, default integer, or default logical that is not a *pointer.*

Numeric type Integer, real, or complex *type.*

Object *Data object.*

Obsolescent feature A feature that is considered to be redundant but that is still in frequent use. It may be deleted in a future revision of the standard.

Operand An *expression* that precedes or succeeds an *operator.*

Operation A computation involving one or two *operands.*

Operator A *lexical token* that specifies an *operation.*

Pointer A *data object* that has the `pointer` *attribute.* It may not be *referenced* or *defined* unless it is *pointer associated* with a *target.* If it is an *array,* it does not have a *shape* unless it is pointer associated.

Pointer assignment The *pointer association* of a *pointer* with a *target* by the execution of a *pointer assignment statement* or the execution of an *assignment statement* for a *data object* of *derived type* having the pointer as a *subobject.*

Pointer assignment statement A *statement* of the form '*pointer* => *target*'.

Pointer associated The relationship between a *pointer* and a *target* following a *pointer assignment* or a valid execution of an `allocate` *statement.*

Pointer association The process by which a *pointer* becomes *pointer associated* with a *target.*

Present A *dummy argument* is **present** in an *instance* of a *subprogram* if it is *associated* with an *actual argument* and the actual argument is a dummy argument that is present in the invoking *procedure* or is not a dummy argument of the invoking procedure.

Procedure A computation that may be *invoked* during program execution. It may be a *function* or a *subroutine.* It may be an *intrinsic procedure,* an *internal procedure,* an *external procedure,* a *module procedure,* a *dummy procedure,* or a *statement function.* A *subprogram* may define more than one procedure if it contains `entry` *statements.*

Procedure interface The *characteristics* of a *procedure,* the *name* of the procedure, the name of each *dummy argument,* and the *generic identifiers* (if any) by which it may be *referenced.*

Processor The combination of a computing system and the mechanism by which *executable programs* are transformed for use on that computing system.

Program See *executable program* and *main program.*

Program unit The fundamental component of an *executable program.* A sequence of *statements* and comment lines. It may be a *main program,* a *module,* an *external subprogram,* or a *block data program unit.*

Rank The number of dimensions of an *array*. Zero for a *scalar*.

Record A sequence of values that is treated as a whole within a *file* .

Reference The appearance of a *data object name* or *subobject designator* in a context requiring the value at that point during execution, or the appearance of a *procedure* name, its *operator* symbol, or a *defined assignment statement* in a context requiring execution of the procedure at that point. Note that neither the act of defining a *variable* nor the appearance of the name of a procedure as an *actual argument* is regarded as a reference.

Scalar

i) A single *datum* that is not an *array*.

ii) Not having the property of being an *array*.

Scope That part of an *executable program* within which a *lexical token* has a single interpretation. It may be an *executable program*, a *scoping unit*, a single *statement*, or a part of a statement.

Scoping unit One of the following:

i) A *derived-type* definition,

ii) An *interface body*, excluding any derived-type definitions and interface bodies contained within it, or

iii) A *program unit* or *subprogram*, excluding derived-type definitions, interface bodies, and subprograms contained within it.

Section subscript A *subscript*, *subscript triplet*, or *vector subscript* in an *array section selector*.

Selector A syntactic mechanism for designating

i) Part of a *data object*. It may designate a *substring*, an *array element*, an *array section*, or a *structure component*.

ii) The set of values for which a `case` *block* is executed.

Shape For an *array*, the *rank* and *extents*. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

Size For an *array*, the total number of elements.

Standard module A *module* standardized as a separate collateral standard.

Statement A sequence of *lexical tokens*. It usually consists of a single line, but the ampersand symbol may be used to continue a statement from one line to another and the semicolon symbol may be used to separate statements within a line.

Statement entity An *entity* identified by a *lexical token* whose *scope* is a single *statement* or part of a statement.

Statement function A *procedure* specified by a single *statement* that is similar in form to an *assignment statement.*

Statement keyword A word that is part of the syntax of a *statement* and that may be used to identify the statement.

Statement label A *lexical token* consisting of up to five digits that precedes a *statement* and may be used to refer to the statement.

Storage association The relationship between two *storage sequences* if a storage unit of one is the same as a storage unit of the other.

Storage sequence A sequence of contiguous *storage units.*

Storage unit A *character storage unit*, a *numeric storage unit*, or an *unspecified storage unit.*

Stride The increment specified in a *subscript triplet.*

Structure A *scalar data object* of *derived type.*

Structure component The part of an *object* of *derived-type* corresponding to a *component* of its type.

Subobject Of a *named data object* or *target* of a *pointer*, a portion that may be *referenced* or *defined* independently of other portions. It may be an *array element*, an *array section*, a *structure component*, or a *substring.*

Subobject designator A *name*, followed by one or more *component selectors*, *array section* selectors, *array element* selectors, and *substring* selectors.

Subprogram * A *function subprogram* or a *subroutine subprogram.* [In Fortran 77, a *block data program unit* was called a subprogram.]

Subroutine A *procedure* that is *invoked* by a `call` *statement* or by a *defined assignment statement.*

Subroutine subprogram A sequence of *statements* from a `subroutine` statement that is not in an *interface block* to the corresponding `end` statement.

Subscript * One of the list of *scalar* integer *expressions* in an *array element selector.* [In Fortran 77, the whole list was called the subscript.]

Subscript triplet An item in the list of an *array section selector* that contains a colon and specifies a regular sequence of integer values.

Substring A contiguous portion of a *scalar character string*. Note that an *array section* can include a *substring selector*; the result is called an array section and not a substring.

Target A *named data object* specified in a *type declaration statement* containing the `target` *attribute*, a data object created by an `allocate` statement for a *pointer*, or a *subobject* of such an object.

Transformational function An *intrinsic function* that is neither an *elemental* function nor an *inquiry function*. It usually has *array arguments* and an array result whose elements have values that depend on the values of many of the elements of the arguments.

Type *Data type*.

Type declaration statement An `integer`, `real`, `double precision`, `complex`, `character`, `logical`, or `type`(*type-name*) *statement*.

Type parameter A parameter of an *intrinsic data type*.

Type parameter values The values of the *type parameters* of a *data entity* of an *intrinsic data type*.

Ultimate component For a *derived type* or a *structure*, a *component* that is of *intrinsic type*, has the *allocatable* or *pointer attribute*, or is an *ultimate component* of a component that is of derived type and does not have the allocatable or pointer attribute.

Undefined For a *data object*, the property of not having a determinate value.

Unspecified storage unit A unit of storage for holding a *pointer* or a *scalar object* of non-default *intrinsic type* that is not a pointer.

Use association The relationship specified by a `use` *statement* between two *names* in different *scoping units*.

Variable * A *data object* whose value can be *defined* and redefined during the execution of an *executable program*. It may be a *named* data object, an *array element*, an *array section*, a *structure component*, or a *substring*. [In Fortran 77, a variable was always *scalar* and named.]

Vector subscript A *section subscript* that is an integer *expression* of *rank* one.

F. Solutions to exercises

Note: A few exercises have been left to the reader.

Chapter 2

1.

```
B is less than M          true
8 is less than 2          false
* is greater than T       not determined
$ is less than /          not determined
blank is greater than A   false
blank is less than 6      true
```

2.

```
   x = y                          correct
3  a = b+c ! add                  correct, with commentary
   word = 'string'                correct
   a = 1.0; b = 2.0               correct
   a = 15. ! initialize a; b = 22. ! and b
                                  incorrect (embedded commentary)
   song = "Life is just&          correct, initial line
      & a bowl of cherries"       correct, continuation
   chide = 'Waste not,            incorrect, trailing & missing
      want not!'                  incorrect, leading & missing
0  c(3:4) = 'up"                  incorrect (invalid statement
                                      label; invalid form of
                                      character constant)
```

3.

```
-43            integer        'word'             character
4.39           real           1.9-4              not legal
0.0001e+20     real           'stuff & nonsense' character
4 9            not legal      (0.,1.)            complex
(1.e3,2)       complex        'I can''t'         character
'(4.3e9, 6.2)' character      .true._1           legal logical
                                    provided kind=1 available
e5             not legal      'shouldn' 't'      not legal
```

```
1_2            legal integer     "O.K."            character
   provided kind=2 available
z10            not legal         z'10'             hexadecimal
```

4.

name	legal	name32	legal
quotient	legal	123	not legal
a182c3	legal	no-go	not legal
stop!	not legal	burn_	legal
no_go	legal	long__name	legal

5.

```
real, dimension(11)      :: a    a(1), a(10), a(11), a(11)
real, dimension(0:11)    :: b    b(0), b(9), b(10), b(11)
real, dimension(-11:0)   :: c    c(-11), c(-2), c(-1), c(0)
real, dimension(10,10)   :: d    d(1,1), d(10,1), d(1,2), d(10,10)
real, dimension(5,9)     :: e    e(1,1), e(5,2), e(1,3), e(5,9)
real, dimension(5,0:1,4) :: f    f(1,0,1), f(5,1,1), f(1,0,2),
                                                     f(5,1,4)
```

Array constructor: (/ (i, i = 1,11) /)

6.

c(2,3)	legal	c(4:3)(2,1)	not legal
c(6,2)	not legal	c(5,3)(9:9)	legal
c(0,3)	legal	c(2,1)(4:8)	legal
c(4,3)(:)	legal	c(3,2)(0:9)	not legal
c(5)(2:3)	not legal	c(5:6)	not legal
c(5,3)(9)	not legal	c(,)	not legal

7.

a)
```
type vehicle_registration
   character(len=3) :: letters
   integer          :: digits
end type vehicle_registration
```

b)
```
type circle
   real               ::   radius
   real, dimension(2) ::   centre
end type circle
```

c)
```
type book
   character(len=20)                :: title
   character(len=20), dimension(2)  :: author
   integer                          :: no_of_pages
end type book
```

Derived type constants:

```
vehicle_registration('PQR', 123)
circle(15.1, (/ 0., 0. /))
book("Pilgrim's Progress", (/ 'John  ', 'Bunyan' /), 250 )
```

8.

t	array	t(4)%vertex(1)	scalar
t(10)	scalar	t(5:6)	array
t(1)%vertex	array	t(5:5)	array (size 1)

9.

```
a)  integer, parameter ::  twenty = selected_int_kind(20)
    integer (kind = twenty) :: counter
b)  integer, parameter :: high = selected_real_kind(12,100)
    real(kind = high)  :: big
c)  character(kind=2)  :: sign
```

Chapter 3

1.

a+b	valid	-c	valid
a+-c	invalid	d+(-f)	valid
(a+c)**(p+q)	valid	(a+c)(p+q)	invalid
-(x+y)**i	valid	4.((a-d)-(a+4.*x)+1)	invalid

2.

```
c+(4.*f)
((4.*g)-a)+(d/2.)
a**(e**(c**d))
((a*e)-((c**d)/a))+e
(i .and. j) .or. k
((.not. l) .or. ((.not. i) .and. m)) .neqv. n
((b(3).and.b(1)).or.b(6)).or.(.not.b(2))
```

3.

```
3+4/2   = 5      6/4/2    = 0
3.*4**2 = 48.    3.**3/2  = 13.5
-1.**2  = -1.    (-1.)**3 = -1.
```

4.

```
ABCDEFGH
ABCD0123
ABCDEFGu            u = unchanged
ABCDbbuu            b = blank
```

5.

```
.not.b(1).and.b(2) valid   .or.b(1)                  invalid
b(1).or..not.b(4)  valid   b(2)(.and.b(3).or.b(4))   invalid
```

6.

```
d .le. c          valid      p .lt. t > 0          invalid
x-1 /= y          valid      x+y < 3 .or. > 4.     invalid
d.lt.c.and.3.0    invalid    q.eq.r .and. s>t      valid
```

7.

a) 4*l
b) b*h/2.
c) 4./3.*pi*r**3

(assuming pi has value π)

8.

```
integer :: n, one, five, ten, twenty_five
twenty_five = (100-n)/25
ten         = (100-n-25*twenty_five)/10
five        = (100-n-25*twenty_five-10*ten)/5
one         = 100-n-25*twenty_five-10*ten-5*five
```

9.

```
a = b + c     valid
c = b + 1.0   valid
d = b + 1     invalid
r = b + c     valid
a = r + 2     valid
```

10.

```
a = b       valid      c = a(:,2) + b(5,:5)    valid
a = c+1.0   invalid    c = a(2,:) + b(:,5)     invalid
a(:,3) = c  valid      b(2:,3) = c + b(:5,3)   invalid
```

Chapter 4

1.

```
integer                  :: i, j, k, temp
integer, dimension(100) :: reverse
do i = 1,100
   reverse(i) = i
end do
read *, i, j
do k= i, i+(j-i-1)/2
   temp = reverse(k)
   reverse(k) = reverse(j-k+i)
```

```
      reverse(j-k+i) = temp
   end do
   end
```

Note: A simpler method for performing this operation will become apparent in Section 6.13.

2.

```
   integer :: limit, f1, f2, f3
   read *, limit
   f1 = 1
   if (limit.ge.1) print *, f1
   f2 = 1
   if (limit.ge.2) print *, f2
   do i = 3, limit
      f3 = f1+f2
      print *, f3
      f1 = f2
      f2 = f3
   end do
   end
```

6.

```
   real x
   do
      read *, x
      if (x.eq.-1.) then
         print *, 'input value -1. invalid'
      else
         print *, x/(1.+x)
         exit
      end if
   end do
   end
```

7.

```
type(entry), pointer :: first, current, previous
current => first
if (current%index == 10) then
   first => first%next
else
   do
      previous => current
      current => current%next
      if (current%index == 10) exit
   end do
   previous%next => current%next
end if
```

Chapter 5

1.

```
subroutine calculate(x, n, mean, variance, ok)
   integer, intent(in)            :: n
   real, dimension(n), intent(in) :: x
   real, intent(out)              :: mean, variance
   logical :: ok
   integer :: i
   mean = 0.
   variance = 0.
   ok = n > 1
   if (ok) then
      do i = 1, n
         mean = mean + x(i)
      end do
      mean = mean/n
      do i = 1, n
         variance = variance + (x(i) - mean)**2
      end do
      variance = variance/(n-1)
   end if
end subroutine calculate
```

Note: A simpler method will become apparent in Chapter 8. 2.

```
subroutine matrix_mult(a, b, c, i, j, k)
   integer, intent(in)               :: i, j, k, l, m, n
   real, dimension(i,j), intent(in)  :: a
   real, dimension(j,k), intent(in)  :: b
   real, dimension(i,k), intent(out) :: c
   c(1:i, 1:k) = 0.
   do n = 1, k
      do l = 1, j
         do m = 1, i
            c(m, n) = c(m, n) + a(m,l)*b(l, n)
         end do
      end do
   end do
end subroutine matrix_mult
```

3.

```
subroutine shuffle(cards)
   integer, dimension(52), intent(in) :: cards
   integer                            :: left, choice, i, temp
   real r
   cards = (/ (i, i=1,52) /)      ! Initialize deck.
   do left = 52,1,-1              ! Loop over number of cards left.
      call random_number(r)       ! Draw a card
```

```
      choice = r*left + 1          !    from remaining possibilities
      temp = cards(left)           !    and swap with last
      cards(left) = cards(choice)!      one left.
      cards(choice) = temp
    end do
  end subroutine shuffle
```

4.

```
  character function earliest(string)
    character(len=*), intent(in) :: string
    integer j, length
    length = len(string)
    if (length <= 0) then
      earliest = ''
    else
      earliest = string(1:1)
      do j = 2, length
        if (string(j:j) < earliest) earliest = string(j:j)
      end do
    end if
  end function earliest
```

5.

```
    subroutine sample
      real :: r, l, v, pi
      pi = acos(-1.)
      :
      r = 3.
      l = 4.
      v = volume(r, l)
      :
    contains
      function volume(radius, length)
        real, intent(in) :: radius, length
        real             :: volume
        volume = pi*radius**2*length
      end function volume
    end subroutine sample
```

7.

```
    module string_type
      type string
        integer :: length
        character(len=80)   :: string_data
      end type string
      interface assignment(=)
        module procedure c_to_s_assign, s_to_c_assign
      end interface (=)
```

```
    interface len
      module procedure string_len
    end interface
    interface operator(//)
      module procedure string_concat
    end interface (//)
  contains
    subroutine c_to_s_assign(s, c)
      type (string), intent(out)   :: s
      character(len=*), intent(in) :: c
      s%string_data = c
      s%length = len(c)
      if (s%length > 80) s%length = 80
    end subroutine c_to_s_assign
    subroutine s_to_c_assign(c, s)
      type (string), intent(in)     :: s
      character(len=*), intent(out) :: c
      c = s%string_data(1:s%length)
    end subroutine s_to_c_assign
    function string_len(s)
      integer      :: string_len
      type(string) :: s
      string_len = s%length
    end function string_len
    function string_concat(s1, s2)
      type (string), intent(in) :: s1, s2
      type (string)             :: string_concat
      string_concat%string_data =                    &
                  s1%string_data(1:s1%length) // &
                  s2%string_data(1:s2%length)
      string_concat%length = s1%length + s2%length
      if (string_concat%length > 80)                 &
                  string_concat%length = 80
    end function string_concat
  end module string_type
```

Note: The intrinsic `len` function, used in subroutine `c_to_s_assign`, is first described in Section 8.6.

Chapter 6

1.

i) `a(1, :)`

ii) `a(:, 20)`

iii) `a(2:50:2, 2:20:2)`

iv) `a(50:2:-2, 20:2:-2)`

v) `a(1:0, 1)`

2.

```
where (z.gt.0) z = 2*z
```

3.

```
integer, dimension(16) :: j
```

4.

```
w      explicit-shaped
a, b   assumed-shape
d      pointer
```

5.

```
real, pointer :: x(:, :, :)
x => tar(2:10:2, 2:20:2, 2:30:2)%du(3)
```

6.

```
ll = ll + ll
ll = mm + nn + n(j:k+1, j:k+1)
```

7.

```
program backwards
   integer                  :: i, j
   integer, dimension(100) :: reverse
   reverse = (/ (i, i=1, 100) /)
   read *, i, j
   reverse(i:j) = reverse(j:i:-1)
end program backwards
```

Chapter 7

1.

i) `integer, dimension(100) :: bin`

ii) `real(selected_real_kind(6, 4)), dimension(0:20, 0:20) :: &`
`iron_temperature`

iii) `logical, dimension(20) :: switches`

iv) `character(len=70), dimension(44) :: page`

2.

Value of i is 3.1, but may be changed;
value of i is 3.1, but may not be changed.

3.

i) `integer, dimension(100) :: i=(/ (0, k=1, 100) /)`

ii) `integer, dimension(100) :: i=(/ (0, 1, k=1, 50) /)`

iii) `real, dimension(10, 10) :: x=reshape((/ (1.0, k=1, 100) /), &`
`(/10, 10/) )`

iv) `character(len=10) :: string = '0123456789'`

Note: the reshape function will be met in Section 8.13.3. **4.**

	mod	outer	inner	fun
a-b	character(10,2)	-	-	-
c,d,e	real	-	-	-
f	real	-	-	real
g,h	real	-	-	-
i-n	integer	-	-	-
o-w	real	-	-	-
x	real	-	-	real
y	real	-	-	-
z	real	-	complex	-

5.

i) `type(person) boss = person('Smith', 48.7, 22)`

ii) (a) This is impossible because a pointer component cannot be a constant.
(b)
```
type(entry) current
data current%value, current%index /1.0, 1/
```

6.
The following are not:
iv) because of the real exponent, and
viii) because of the pointer component.

Chapter 8

1.

```
   program qroots        ! Solution of quadratic equation.
!
      real :: a, b, c, d, x1, x2
!
      read(*, *) a, b, c
      write( *, *) ' a = ', a, 'b = ', b, 'c = ', c
      if (a == 0.) then
         if (b /= 0.) then
            write(*, *) ' Linear: x = ', -c/b
         else
            write(*, *) ' No roots!'
         endif
      else
         d = b**2 - 4.*a*c
         if (d < 0.) then
            write (*, *) ' Complex', -b/(2.*a), '+-',      &
                         sqrt(-d)/(2.*a)
         else
            x1 = -(b + sign(sqrt(d), b))/(2.*a)
```

```
            x2 = c/(x1*a)
            write(*, *) ' Real roots', x1, x2
         endif
      endif
   end program qroots
```

Historical note: A similar problem was set in one of the first books on Fortran programming — *A FORTRAN Primer* by E. Organick (Addison-Wesley, 1963). It is interesting to compare Organick's solution, written in FORTRAN II, on p. 122 of that book, with the one above. (It is reproduced in the *Encyclopedia of Physical Science & Technology* (Academic Press, 1987), vol. 5, p. 538.)

2.

```
   subroutine calculate(x, mean, variance, ok)
      real, intent(in)  :: x(:)
      real, intent(out) :: mean, variance
      logical ok
      ok = size(x) > 1
      if (ok) then
         mean = sum(x)/size(x)
         variance = sum((x-mean)**2)/(size(x)-1)
      end if
   end subroutine calculate
```

3.

```
F     p1 and p2 are associated with the same array elements,
      but in reverse order
T     p1 and p2(4:1:-1) are associated with exactly the
      same array elements, a(3), a(5), a(7), a(9)
```

4.

```
5   1     a has bounds 5:10 and a(:) has bounds 1:6
5   1     p1 has bounds 5:10 and p2 has bounds 1:6
1   1     x and y both have bounds 1:6
```

Chapter 9

1.

```
a) print '(a/ (t1, 10f6.1))', ' grid', grid
b) print '(a, " ", 25i5)', ' list', (list(i), i = 1, 49, 2)
or                              list(1:49:2)
c) print '(a/ (" ", 2a12))', ' titles', titles
d) print '(a/ (t1, 5en15.6))', ' power', power
e) print '(a, 1012)', ' flags', flags
f) print '(a, 5(" (", 2f6.1, ")"))', ' plane', plane
```

2.

```
character, dimension(3,3) :: tic_tac_toe
integer                   :: unit
:
write(unit, '(t1, 3a2)' ) tic_tac_toe
```

4.

```
(a) read(*, *) grid
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0

(b) read(*, *) list(1:49:2)
25*1

(c) read(*, *) titles
data transfer

(d) read(*, *) power
1.0 1.e-03

(e) read(*, *) flags
t f t f t f f t f t

(f) read(*, *) plane
(0.0, 1.0),(2.3, 4)
```

5.

```
  character function get_char(unit)
     integer :: unit
10   read(unit, '(a1)', advance='no', eor=10) get_char
     return
  end function get_char
```

Index