Fortran90 for Fortran77 Programmers

Clive Page

2002 November 21

This document may be found at: **http://www.star.le.ac.uk/∼cgp/f90course/f90.html** or **.pdf**

Contents

1 Introduction

1.1 What was wrong with Fortran77?

- There are no dynamic storage facilities at all.

- There are no user-defined data types or data structures (except the COMMON block).

- It was too easy to make mistakes which the compiler could not detect, especially when calling procedures (subroutines or functions). A study of some 4 million lines of professional Fortran showed $\approx 17\%$ of procedure interfaces were defective.

- Programs are less than 100% portable because of a few remaining platform-dependent features.

- Control structures are poor, so it hard to avoid using GOTOs and labels, often leading to spaghetti code.

- Archaic features left over from the punched-card era:

  - fixed-format lines,
  - statements all in upper-case,
  - variable names limited to 6-characters.

- In practice programmers are driven to use extensions to Fortran77 Standard – which reduce portability.

1.2 What's New in Fortran90

- **Free-format source code** and many other simple improvements.

- **Arrays as first-class objects**, whole-array expressions, assignments, and functions.

- **Dynamic memory allocation**; pointers to allow complex dynamic data structures to be constructed.

- **User-defined data types**; existing operators can be overloaded (re-defined) or new ones defined.

- **The MODULE** – a new program unit which can encapsulate data and a related set of procedures (subroutines or functions). Can implement classes and member functions for object-oriented programming.

- **New control structures**: SELECT CASE, CYCLE, and EXIT so labels and explicit jumps are rarely needed.

- Recursive functions, generic names for procedures, optional arguments, calls with keywords, and many other procedure call options.

1.3   Benefits

- Programs often **simpler**, and thus easier to maintain.

- Programs are **safer** and more reliable because the compiler can detect many more mistake (provided sensible safety features are used).

- Programs **more portable**, because very few machine-dependant features remain, and there is little need to use extensions.

- **Parallel processing** supported by whole array operations and other features.

- Fortran77 is a true subset: nothing has been taken out. You can adopt the new features gradually.

1.4   Will old code still work?

Yes – if it used only Standard Fortran77 or extensions which are now part of Fortran90.

But common extensions to Fortran77 which were **not** included in Fortran90 include:

- Tab-formatted source-code lines – change tabs to spaces, maybe use free-format.

- Data type declarations like `INTEGER*2` and `REAL*8` – new syntax better but more complicated.

- Hexadecimal, octal, and binary constants in expressions – allowed only in `DATA` statements.

- VAX data structures – Fortran90 structures have different syntax.

- Functions like `%VAL` in subprogram calls – the new dynamic memory facilities are much better.

- Expressions in formats (e.g. `FORMAT(F<nw>.3)` – can do this indirectly with internal-file I/O.

- Some `OPEN` options such as `ACCESS='APPEND'` – change to `POSITION="APPEND"`.

Function name clashes – Fortran has no reserved words, but problems may arise one of your external function names matches that of an intrinsic function. There are 75 new ones, and names to avoid now include: `ALL`, `ANY`, `COUNT`, `HUGE`, `KIND`, `MAXVAL`, `MERGE`, `MINVAL`, `PACK`, `RANGE`, `SCALE`, `SCAN`, `SIZE`, `SUM`, `TRIM`, and `UNPACK`. This problem can be avoided by declaring your function to be `EXTERNAL` wherever necessary.

Static storage assumption – Fortran77 compilers generally used static storage for all variables, so `SAVE` statements could be omitted with impunity. Most Fortran90 systems use static storage only when required (variables given an initial value, or has an explicit `SAVE` attribute). Otherwise local variables in subprograms will not be preserved after control returns, and missing `SAVE` statements may cause problems.

1.5   Fortran90 Compilers

Wide choice for PC/Windows and most Unix platforms, one or two for PC/Linux, VMS and Macintosh. They tend to be more expensive than for Fortran77, and not all are as efficient or stable.

The following are available free:

- F for Linux is another subset, free for Linux from Imagine1 `http://www.imagine1.com/imagine1`

- A translator from Fortran90 to Fortran77 can be obtained free for Linux from Pacific-Sierra Research **http://www.psrv.com** (but it has many limitations).

The subsets supported by ELF90 and F are slightly different: both support all the modern features of Fortran90 and leave out all the obsolete stuff, so they are not suitable for legacy code.

GNU's free compiler, **g77**, is suitable for legacy code and runs on many platforms, but supports only a few of the new features of Fortran90.

1.6 The Future

Fortran95 was formally adopted as the ISO Standard in November 1997. It adds only a few new features (e.g. a way of specifying initial values for data structures). The first Fortran95 compilers have already started to appear and many more are likely to appear by the end of 1998.

Fortran2000 is currently being defined: major new features are likely to include full support for object-oriented programming, and syntax for inter-operability with C.

2 New Look and Feel

Use of lower-case and long variable names already a common extention to Fortran77; free-format layout is a more radical change.

Examples here will use `UPPER CASE` *for Fortran keywords and intrinsic functions,* `lower case` *for user-chosen names. This is not a recommended convention, just for clarity in these notes.*

2.1 Basic Rules for both free and fixed-format

Lower-case letters may be used, but Fortran is case-insensitive (except within quoted character constants).

Symbolic names can be up to 31 characters long, and names may include underscores as well as digits:

```
temperature_in_fahrenheit = temperature_in_celsius * 1.8 + 32.0
```

Semi-colons separate two or more statements on the same line:

```
sumx = 0.0; sumy = 0.0; sumz = 0.0
```

End-of-line comments start with an exclamation mark (but must not be in column 6 of fixed-format code).

```
nday = mjd(year, month, day)     ! convert to Modified Julian Date
```

Character constants may be enclosed either in a pair of apostrophes or double-quote marks – making it easier to embed the other character in a string:

```
WRITE(*,*) "If it ain't broke don't fix it"
```

Relational operators may be given in old or new forms:

```
old form:    .GE.    .GT.    .EQ.    .NE.    .LE.    .LT.
new form:    >=      >       ==      /=      <=      <
```

2.2 Free-format layout

- Free-format an alternative to column-based fixed-format.

- Must choose either **free** or **fixed** format for each source file (rules differ).

Most compilers assume **free-format** if the source file has an extension of `.f90` and **fixed-format** otherwise (but usually one can over-ride this with command-line switches such as `-free` and `-fixed`).

Free-format layout rules:

1. **Statements** may appear anywhere on a line; lines may be up to 132 characters long.

2. **Comments** start with an exclamation mark "!" (so `C` or `*` in column 1 have to be changed to "!").

3. **To continue a statement** put an ampersand "&" at the end of each incomplete line:

```
CALL predict( mercury, venus, earth, &     ! comment allowed here
              mars, jupiter, saturn, uranus, neptune, pluto)
```

If the line-break splits a name or constant then a comment is not allowed, and the next line must start with another ampersand:

```
WRITE(*,*) "University of Leicester, Department of &
           &Physics & Astronomy"        ! NO comment on preceding line
```

4. **Spaces are significant** in free-format code: embedded spaces are not allowed in variable names or constants, but a space is generally required between two successive words (but they are optional in some two-word Fortran terms including `DOUBLE PRECISION`, `ELSE IF`, `GO TO`, `END DO`, and `END IF`).

```
      MILLION = 1 000 000   ! valid in fixed-layout lines only
```

With care one can write code valid in both formats, which may be useful for INCLUDE files to be used in both old and new code: the secret for continuation lines is to put an ampersand after column 72 and another in column 6 of the next line.

2.3   New Forms for Specification Statements

`IMPLICIT NONE` is now standard (and recommended so the compiler flags more mistakes).

The `DOUBLE PRECISION` data type is now just a special case of `REAL` so all facilities are identical; this means that double-precision complex is fully standardised.

`INCLUDE` statements are also standard (but the `MODULE` now provides better facilities).

Type statements – new form with double-colon allows all attributes of variables to be specified at once:

```
  INTEGER, DIMENSION(800,640) :: screen, copy, buffer
```

Define constants without separate `PARAMETER` statement:

```
  REAL, PARAMETER :: pi = 3.14159, rtod = 180.0/pi
```

Initialise variables too:

```
  CHARACTER(LEN=50) :: infile = "default.dat"
  INTEGER :: file_number = 1, error_count = 0
```

`DATA` statement almost redundant – still useful to initialise just part of an array, use a repeat-count, or a hexadecimal constant:

```
  INTEGER :: dozen(12), forty_two, sixty_three, max_byte
  DATA dozen / 6*0, 6*1 /, forty_two / B'101010' /,      &
       sixty_three / O'77' /, max_byte / Z'FF'/
```

The `SAVE` attribute is applied automatically to any variable given an initial value, whether in a `DATA` or type statement.

`INTENT` may be specified for procedure arguments: useful aid to documentation, and allows the compiler to check usage more carefully:

```
  SUBROUTINE readfile(iounit, array, status)
  IMPLICIT NONE                     ! not essential but good practice
  INTEGER, INTENT(IN)    :: iounit ! unit number to read from
  REAL, INTENT(OUT)      :: array  ! data array returned
  INTEGER, INTENT(INOUT) :: status ! error-code (must be zero on entry)
```

3   New Control Structures

3.1   The `SELECT CASE` Structure

`SELECT CASE` replaces the computed `GO TO` which required a plethora of statement labels. The new structure is often easier to use and more efficient than a set of `ELSE IF` clauses and is label-free. The

example below, given a day-number between 1 and 31, selects a suitable suffix, e.g. to turn "3" into "3rd", etc.:

```
SELECT CASE(day_number)
CASE(1, 21, 31)
    suffix = 'st'
CASE(2, 22)
    suffix = 'nd'
CASE(3, 23)
    suffix = 'rd'
CASE(4:20, 24:30)
    suffix = 'th'
CASE DEFAULT
    suffix = '??'
    WRITE(*,*)'invalid date: ', day_number
END SELECT
WRITE(*, "(I4,A2)") day_number, suffix
```

The selection expression may be of integer or character type; the ranges in each `CASE` statement must not overlap. The default clause is optional.

3.2   DO, EXIT and CYCLE statements

The `END DO` statement is at last part of the Standard, so a label is no longer needed in each `DO` statement. In addition `CYCLE` will cause the next iteration to start at once, while `EXIT` exits the loop structure prematurely.

This example scans the headers of a FITS file:

```
CHARACTER(LEN=80) :: header
DO line = 1,36
  READ(unit, "(a80)") header
  IF( header(1:8) == "COMMENT") THEN   ! ignore comments - loop again
    CYCLE
  ELSE IF( header(1:8) == "END") THEN  ! need READ no more lines
    EXIT                               ! so exit from the loop
  ELSE
! process this header...
  END DO
```

An **indefinite** `DO` also exists – here an `EXIT` from the loop is essential:

```
sum  = 0.0
DO
  READ(*, IOSTAT=status) value
  IF(status /= 0) EXIT
  sum = sum + value                 ! or whatever
END DO
```

3.3   DO WHILE statement

`DO WHILE` is supported, but an indefinite DO with an `EXIT` does much the same:

```
DO WHILE( ABS(x - xmin) > 1.0e-5)
    CALL iterate(x, xmin)
END DO
```

3.4 Structure Names

Names may be given to `DO`-loops, `IF`-blocks, or `CASE`-structures – helps readability when they are deeply nested, and required to `EXIT` from (or `CYCLE` around) anything other than the innermost loop.

```
        sum = 0.0
outer:  DO j = 1,ny              ! sum values until zero encountered
inner:     DO i = 1,nx
               IF(array(i,j) == 0.0) EXIT outer
               sum = sum + array(i,j)
           END DO inner
        END DO outer
```

Note that structure names like `inner` do not have the drawbacks of statement labels because it is not possible to jump to them using a `GO TO` statement.

3.5 Label-free programming

Statement labels should be avoided because each one marks the site of a jump from elsewhere, and thus makes it harder to see the execution sequence. Label-free programming is now feasible in many cases:

1. `DO`-loops with `END DO` no longer need labels, and error-handling can mostly use `EXIT` or `RETURN`.

2. Computed-`GO TO` should be replaced by `SELECT CASE`.

3. `FORMAT` statements can be replaced by a format string in the `READ` or `WRITE` statement itself, e.g.:

   ```
   WRITE(unit, "(A,F10.3,A)") "flux =", source_flux, " Jansky"
   ```

3.6 Internal Procedures

Generalisation of statement functions – no longer limited to one line:

```
   SUBROUTINE polygon_area(vertices)    ! an external procedure
   IMPLICIT NONE                        ! applies throughout
!...
      area1 = triangle_area(a, b, x)
!...
      area2 = triangle_area(x, c, d)
!...
   CONTAINS                    ! internal procedures follow...
      REAL FUNCTION triangle_area(a, b, c)  ! internal procedure
      REAL, INTENT(IN) :: a, b, c
      REAL :: s                ! local variable in the function
         s = 0.5 * (a + b + c)
         triangle_area = sqrt(s * (s-a) * (s-b) * (s-c))
      END FUNCTION triangle_area
   END SUBROUTINE polygon_area
```

Rules for internal procedures:

- May be subroutines or functions

- Placed within **host** program unit after `CONTAINS` statement (which ends executable code of host).

- A host may have any number of internal procedures, which may call one another (but may not be nested).

- Need `END SUBROUTINE` or `END FUNCTION` at the end – appending the procedure name is optional.

Host association: internal procedures may access to all the host's variables (unless they declare local variables with the same name) but not vice-versa.

Host association has its risks: e.g. using a variable `x` in the internal procedure (above) without declaring it would inadvertently use the host's `x`.

May use scoping rules to set up set of procedures with a few global variables, e.g.

```
    SUBROUTINE main(args)
    REAL :: args              !accessible to internal procedures
    REAL :: global_variables  ! likewise
    CALL internal
    CONTAINS
      SUBROUTINE internal
      !...
      END SUBROUTINE internal
      SUBROUTINE lower_level
      !...
      END SUBROUTINE lower_level
    END SUBROUTINE main
```

4 Arrays

4.1 Declaring and Initialising Arrays

New form of type statement with double colon can declare arrays and simple scalars:

```
    REAL :: array(3,4,5), scalar, vector(12345)
```

Dimension attribute useful if several arrays have the same shape:

```
    INTEGER, DIMENSION(1024,768) :: screen, window, new_window
```

An **Array constant** is a list of elements enclosed in `(/` and `/)` and may be used to give an initial value to a variable or to define an array constant.

```
    INTEGER :: options(3) = (/ 5, 10, 20 /)            ! initial values
    CHARACTER(LEN=3), PARAMETER :: day(0:6) = &
        (/'Sun','Mon','Tue','Wed','Thu','Fri','Sat'/)  ! array constant
```

Array terminology: An array declared like this:

```
    REAL :: X(2,5,-1:8)
```

has a **rank** of 3, **extents** of 2, 5, and 10, a **shape** of `(/ 2, 5, 10 /)`, and a **size** of 100.

4.2 Array-valued expressions and assignments

Arrays are now first-class objects, and array-valued expressions are evaluated element-wise, which saves writing many simple loops:

```
  REAL, DIMENSION(512,1024) :: raw, background, exposure, result, std_err
!...
  result  = (raw - background) / exposure
```

Similarly all appropriate intrinsic functions operate element-wise if given an array as their argument:

```
  std_err = SQRT(raw) / exposure
```

Array expressions may also include **scalar** constants and variables: these are effectively replicated (or expanded) to the required number of elements:

```
  std_err    = 0.0                        ! every element set to zero
  background = 0.1 * exposure + 0.125
```

All the elements in an array-valued expression must be **conformable**, that is they are either either scalars or arrays all of which have the same **shape**, i.e. the same number of elements along each axis (the actual lower and upper-bounds may be different).

4.3   Array Constructors

An **array constructor**, which is generalisation of the array constant, may appear in any array expression, and and may contain a list of scalars, arrays, or loops:

```
array = (/ 1.51, x, 2.58, y, 3.53 /)
ramp  = (/ (REAL(i), i = 1,10) /)
```

The array constructor only works for 1-dimensional arrays. For arrays of higher rank the `RESHAPE` function is useful: its second argument specifies the **shape** of the output array:

```
INTEGER :: list(2,3) = RESHAPE( (/ 11, 12, 21, 22, 31, 32 /), (/2,3/))
```

4.4   Array Sections

An **array section** or slice is specified with a colon separating the lower and upper bounds. Thus `ramp(7:9)` is a 3-element slice of array `ramp`. Similarly `raw(2:101,301:500)` is a slice of the array called `raw` of shape 100×200 elements. Note that a slice does not have to occupy contiguous storage locations – Fortran takes care of this. It also allows assignments statements involving overlapping slices:

```
a(2:10) = a(1:9)     ! shift up one element
b(1:9)  = b(3:11)    ! shift down two elements
```

In such cases the compiler must generate code to work through the elements in the right order (or copy to some temporary space) to avoid overwriting.

Array triplet notation allows sparse sub-arrays to be selected; the **stride** (third item in the triplet) must not of course be zero:

```
b(1:10:2)    ! selects five elements: 1, 3, 5, 7, 9
b(90:80:-3)  ! selects four elements 90, 87, 84, 81 in that order
```

Zero-sized arrays may be referenced, just as if a DO-loop had been used which specified no iterations. Thus `b(k:n)` has no elements if `k` is greater than `n`.

Vector subscripts may also be used:

```
INTEGER :: mysub(4)
REAL    :: vector(100)
mysub = (/ 32, 16, 17, 18 /)
WRITE(*,*) vector(mysub)
```

This outputs only elements 32, 16, 17, and 18 of the vector in that order.

Note that **vector subscripts** may only be used on the left-hand side of an assignment if there are no repeated values in the list of subscripts (otherwise one element would have to be set to two different values).

4.5   Array Intrinsic Functions

Arguments in *italics* are optional.

Array reduction functions

```
        l = ALL(mask, dim)              .true. if all elements are true
        l = ANY(mask, dim)              .true. if any elements are true
        i = COUNT(mask, dim)            Number of true elements
        x = SUM(array, dim, mask)       Sum of elements
        x = PRODUCT(array, dim, mask)   Product of elements
        x = MAXVAL(array, dim, mask)    Maximum value in array
        x = MINVAL(array, dim, mask)    Minimum value in array
        x = DOT_PRODUCT(va, vb)         Dot product of two vectors
```

Example: if **REAL :: myarray(2,3)** contains   myarray

$$\begin{matrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{matrix}$$

```
SUM(myarray) returns 21
SUM(myarray, DIM=1) returns  (/ 9, 12 /)  SUM(myarray, DIM=2) returns  (/ 3, 7, 11/)
```

Other array manipulation functions

```
    a = MATMUL(mata, matb)              Matrix multiplication (or matrix X vector)
    a = TRANSPOSE(matrix)              Transpose of 2-d array
    a = CSHIFT(array, shift, dim)      Circular shift of elements
    a = EOSHIFT(array, shift, dim)     End-off shift of elements
    a = PACK(array, mask, pad)         Pack values of array which pass the mask
    a = MERGE(tsource, fsource, mask)  Use tsource if mask is true, else fsource elements.
    a = MAXLOC(array, mask)            Location of maximum element
    a = MINLOC(array, mask)            Location of minimum element
```

Note: `MAXLOC` and `MINLOC` used on a 1-d array return an array of one element, which is not the same as a scalar.

An example of their use is to find the mean and variance of an array but with values of zero to be ignored:

```
  mean = SUM(x /= 0.0) / COUNT(x /= 0.0)
  variance = SUM((x-mean)**2, MASK= x /= 0.0) / COUNT(x /= 0.0)
```

4.6   WHERE structure

When some elements of an array expression have to be treated specially, the `WHERE` structure may be useful:

```
  WHERE(x /= 0.0)
    inverse = 1.0 / x
  ELSEWHERE
    inverse = 0.0
  END WHERE
```

There is also a single statement form of it:

```
  WHERE(array > 100.0) array = 0.0
```

5   Dynamic Storage

There are three forms of dynamic array: automatic, allocatable, and pointer array.

5.1   Automatic Arrays

An **automatic array** is a local array in a procedure which has its size set when the procedure is called:

```
  SUBROUTINE smooth(npts, spectrum)
  IMPLICIT NONE
```

```
  INTEGER, INTENT(IN) :: npts
  REAL, INTENT(INOUT) :: spectrum
  REAL :: space(npts), bigger(2*npts)  ! automatic arrays
```
The dimension bounds may be integer expressions involving any variables accessible at that point: normally this means other arguments of the routine. Within the procedure an automatic array is just like any other; it may be passed to lower-level routines, but it becomes undefined as soon as control returns to above the level at which it is defined. An automatic array cannot be defined initially or be used to save values from one call to another.

Most systems store automatic arrays on the stack; some Unix systems do not allocate much stack space by default. The following command may be used to increase it:

```
>  limit stack unlimited
```

5.2   Allocatable Arrays

Allocatable arrays are more generally useful as their size may be set at any point. Only the rank has to be declared in advance, with a colon marking the each dimension:

```
  REAL, ALLOCATABLE :: vector(:), matrix(:,:), three_d(:,:,:)
```
The actual dimension bounds may then be set anywhere in the executable code (the lower bound is 1 by default):

```
  ALLOCATE(vector(12345), matrix(0:511,0:255))
```
Allocatable arrays may be passed to lower-level routines in the usual way. But they need to be explicitly deallocated before the procedure which declares them exits, otherwise a **memory leak** may occur.

```
  DEALLOCATE(matrix, vector)
```
Once a its size has been allocated, it cannot be altered, except by deallocating the array and then allocating it again. If you want to preserve the contents they need to be copied somewhere else temporarily.

Most systems use heap storage for allocatable arrays. With very large arrays one might use up all the space available, so a status variable can be used to check. It normally returns zero, but is set non-zero if the allocation fails:

```
  ALLOCATE(huge_array(1:npts), STAT=ierror)
  IF(ierror /= 0) THEN
     WRITE(*,*)"Error trying to allocate huge_array"
     STOP
  END IF
```
In such cases there may be another less memory-intensive algorithm available, otherwise the program should exit gracefully.

It is important to ensure that you do not attempt to allocate the same array twice; the `ALLOCATED` intrinsic function helps here:

```
  IF(ALLOCATED(myarray)) THEN
      DEALLOCATE(myarray)
  END IF
  ALLOCATE(myarray(1:newsize))
```
An allocatable array can also be declared to have the `SAVE` attribute, in which case it will survive after the exit of the procedure which creates it. This is especially useful in connection with modules (as explained later).

5.3   Pointer arrays

An allocatable array cannot be passed to a procedure when in an un-allocated state. But this can be done with a pointer array:

```
  PROGRAM pdemo
```

```
    IMPLICIT NONE
REAL, POINTER :: parray(:)
  OPEN(UNIT=9, FILE='mydata', STATUS='old')
  CALL readin(9, parray)
  WRITE(*,*)'array of ', SIZE(array), ' points:'
  WRITE(*,*) parray
  DEALLOCATE(parray)
  STOP                          ! STOP is optional
CONTAINS
  SUBROUTINE readin(iounit, z)
  INTEGER, INTENT(IN) :: iounit
  REAL, POINTER       :: z(:)    ! cannot use INTENT for pointer
    INTEGER :: npoints
    READ(iounit) npoints          ! find how many points to read
    ALLOCATE(z(1:npoints))        ! allocate the space
    READ(iounit) z                ! read the entire array
  END SUBROUTINE readin
END PROGRAM pdemo
```

This example is especially simple because an internal procedure is used, so that the compiler knows all the details of the interface when it compiles the subroutine call: a so-called **explicit interface**, which is required when passing a pointer to a procedure.

6 Modules and Interfaces

The there are now four types of program unit in Fortran:

1. Main program – should start with a main `PROGRAM` statement.

2. External procedures (subprograms) – start with `SUBROUTINE` or `FUNCTION` statement.

3. Block data subprograms (now superseded along with common blocks) – starts with `BLOCK DATA` statement.

4. Module – starts with `MODULE` statement.

The **module** may contain any combination of:

- definitions of constants

- definitions of derived types (data structures)

- data storage declarations

- procedures (subroutines and functions)

The module may be accessed with a `USE` statement in any other program unit (including another module).

6.1 Example defining constants

```
MODULE trig_consts
IMPLICIT NONE
  DOUBLE PRECISION, PARAMETER :: pi = 3.141592653589d0, &
      rtod = 180.0d0/pi, dtor = pi/180.0d0
END MODULE trig_consts
```

```
  PROGRAM calculate
  USE trig_consts
  IMPLICIT NONE
      WRITE(*,*) SIN(30.0*dtor)
  END PROGRAM calculate
```
Note that:

- `USE` statements always precede all other types of specification statement, even `IMPLICIT NONE`.

- The module must be compiled **before** all other program units which use it; it may be in the same file or a separate file. Most compilers support separate compilation, and leave a `.mod` file, or something similar, containing the module information for the use of subsequently encountered `USE` statements.

These simple uses of the module barely distinguish it from an `INCLUDE` file (now part of the Fortran Standard), but the module is actually a much more powerful facility, because of module procedures.

6.2 Module Procedures

The general structure of a module:

1. starts with a data section

2. then has a `CONTAINS` statement (if any procedures follow)

3. any number of module procedures follow.

Module procedures have direct access to all the definitions and data storage in the data section via **host association**.

Allows encapsulation of data and a set of procedures which operate on the data or use the storage area for inter-communication.

This (slightly shortened) module handles output to a VT terminal or X-term window:

```
MODULE vt_mod
  IMPLICIT NONE                    ! applies to whole module
  CHARACTER(1), PARAMETER :: escape = achar(27)
  INTEGER, SAVE :: screen_width = 80, screen_height = 24
CONTAINS
  SUBROUTINE clear  ! Clears screen, moves cursor to top left
    CALL vt_write(escape // "[H" // escape // "[2J")
  END SUBROUTINE clear

  SUBROUTINE set_width(width)    ! sets new screen width
  INTEGER, INTENT(IN) :: width      ! preferred width (80/132)
    IF(WIDTH > 80) THEN             ! switch to 132-column mode
      CALL vt_write( escape // "[?3h" )
      screen_width = 132
    ELSE                           ! switch to 80-column mode
      CALL vt_write( escape // "[?3l" )
      screen_width = 80
    END IF
  END SUBROUTINE set_width

  SUBROUTINE get_width(width)   ! returns screen width (80/132)
  INTEGER, INTENT(OUT) :: width
    width = screen_width
```

```
   END SUBROUTINE get_width

   SUBROUTINE vt_write(string)       ! for internal use only
   INTEGER, INTENT(IN) :: string
      WRITE(*, "(1X,A)", ADVANCE="NO") string
   END SUBROUTINE vt_write
END MODULE vt_mod
```

To use this module one just needs at the top:
```
   USE vt_mod
```

6.3 Public and Private accessibility

By default all module variables are available by all program units which `USE` the module. This may not always be desirable: if the module procedures provide all the access functions necessary, it is safer if package users cannot interfere with its internal workings. By default all names in a module are `PUBLIC` but this can be changed using the the `PRIVATE` statement:
```
   MODULE vt_mod
      IMPLICIT NONE
      PRIVATE           ! change default so all items private
      PUBLIC  :: clear_screen, set_width, get_width
```
Now a program unit which uses the module will not be able to access the subroutine `vt_write` nor variables such as `screen_width`.

6.4 Avoiding name clashes

Even with the precautions suggested above, sometimes a module will contain a procedure (or variable) name which clashes with one that the user has already chosen. There are two easy solutions. If the name is one that is not actually used but merely made available by the module, then the `USE ONLY` facility is sufficient:
```
   USE vt_mod, ONLY: clear_screen
```
But supposing that one needs access to two procedures both called `get_width`, the one accessed in the `vt_mod` module can be renamed:
```
   USE vt_mod, gwidth => get_width
```
so it acquires the temporary alias of `gwidth`.

6.5 Pros and Cons of Modules

- Modules can be used to encapsulate a data structure and the set of procedures which manipulate it – or a class and its methods in OO-speak.

- When a module procedure is called it is said to have an **explicit interface** - this means that the compiler can check actual and dummy arguments for consistency. This is a very valuable feature, since in Fortran77 such interfaces cannot usually be checked and errors are common.

- The module supersedes common blocks, `BLOCK DATA` and `ENTRY` statements.

- Modules provide an additional structural level in program design:

  1. Program
  2. Modules
  3. Procedures

4. Statements.

- The explicit interfaces which come automatically with module procedures allow a range of advanced features to be used, including: assumed-shape arrays, pointer arrays, optional arguments, functions returning arrays or pointers, user-defined operators, generic names. These will be described in detail later.

But there are a few potential drawbacks:

- Each module has to be compiled before any other program unit which uses it: this needs care and means that the main program has to be last rather than first, as is traditional.

- Changes to a module mean re-compilation of all units which use it: this needs to be coded into `make` files, and can lead to slow compiles.

- Usually a module compiles into a single object module, so that reference to any procedure in a module causes all the code in the module to be present into the executable program.

6.6   Explicit Interfaces

An explicit interface is one where the dummy arguments of the procedure are visible to the compiler when compiling the procedure call. Explicit interfaces are needed for a variety of advanced features. An interface is explicit:

- When a module procedure is called from a program unit which uses it, or from any other procedure of the same module.

- When an internal procedure is called from its host or from any other internal procedure of the same host.

- A recursive procedure calls itself directly or indirectly.

- Any intrinsic function is called.

- An explicit `INTERFACE` block is provided.

Here is an example of an interface block:

```
INTERFACE
   DOUBLE PRECISION FUNCTION sla_dat (utc)
      IMPLICIT NONE
      DOUBLE PRECISION :: utc
   END FUNCTION sla_dat

   SUBROUTINE sla_cr2tf (ndp, angle, sign, ihmsf)
      IMPLICIT NONE
      INTEGER :: ndp
      REAL    :: angle
      CHARACTER (LEN=*) :: sign
      INTEGER, DIMENSION (4) :: ihmsf
   END SUBROUTINE sla_cr2tf
END INTERFACE
```

Note that an `IMPLICIT NONE` is needed in each procedure definition, since an interface block inherits nothing from the enclosing module.

An interface block may, of course, be put in a module to facilitate use. When using an existing (Fortran77) library, it may be worth-while to create a module containing all the procedure interfaces – may be done automatically using Metcalf's `convert` program.

7 Procedures and Arguments

7.1 Assumed shape arrays

The assumed-shape array is strongly recommended for all arrays passed to procedures: the rank has to be specified, but the bounds are just marked with colons. This means the actual shape is taken each time it is called from that of the corresponding actual argument.

```
MODULE demo
IMPLICIT NONE
CONTAINS
   SUBROUTINE showsize(array)
   IMPLICIT NONE
   REAL, INTENT(IN) :: array(:,:)  ! 2-dimensional.
   WRITE(*,*) "array size", SIZE(array,1), " X ", SIZE(array,2)
   END SUBROUTINE showsize
END MODULE demo

PROGRAM asize
USE demo
IMPLICIT NONE
REAL :: first(3,5), second(123,456)
   CALL showsize(first)
   CALL showsize(second)
END PROGRAM asize
```

The lower bound is one by default, it does not have to be the same as that of the actual argument, as only the **shape** (extent along each axis) is passed over, so that intrinsic functions such as `LBOUND` and `UBOUND` provide no additional information.

7.2 Keyword calls and optional arguments

Any procedure which has an explicit interface, may be called using keyword notation in the call, as an alternative to the positional notation. All intrinsic functions may also be called by keyword, which is handy when one wants to omit optional arguments:

```
   INTEGER :: intarray(8)
   CALL DATE_AND_TIME(VALUES=intarray)
```

After any argument uses keyword notation, all subsequent ones in same call must do so.

Arguments of user-written procedures may also be made optional, these should be tested to see whether they are `PRESENT` before use:

```
   SUBROUTINE write_text(string, nskip)
   CHARACTER(*), INTENT(IN) :: string      ! line of text
   INTEGER, INTENT(IN), OPTIONAL :: nskip  ! no of lines to skip
! local storage
   INTEGER :: localskip
   IF(PRESENT(nskip)) then
      localskip = nskip
   ELSE
      localskip = 0           ! default value
   END IF
! rest of code to skip lines etc.
```

Optional arguments at the end of the list may simply be omitted in the procedure call, but if you omit earlier ones you cannot simply use two adjacent commas (as in some extensions to Fortran77). After an optional argument has been omitted, all subsequent arguments must use the keyword notation.

7.3 Generic names

Intrinsic functions often have generic names, thus `ABS` does something different depending on whether its argument is real, integer, or complex. User-written functions may now be given a generic name in a similar way.

Suppose you have a module containing several similar data sorting routines, for example `sort_int` to sort an array of integers, `sort_real` to sort reals, etc. A generic name such as `sort` may be declared in the head of the module like this:

```
INTERFACE sort
    MODULE PROCEDURE sort_int, sort_real, sort_string
END INTERFACE
```

The rules for resolving generic names are complicated but it is sufficient to ensure that each procedure differs from all others with the same generic name in the **data type**, or **rank** of at least one non-optional argument.

7.4 Recursive procedures

Procedures may now call themselves directly or indirectly if declared to be `RECURSIVE`. Typical uses will be when handling self-similar data structures such as directory trees, B-trees, quad-trees, etc. The classical example is that of computing a factorial:

```
RECURSIVE FUNCTION factorial(n) RESULT(nfact)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
INTEGER :: nfact
IF(n > 0) THEN
    nfact = n * factorial(n-1)
ELSE
    nfact = 1
END IF
END FUNCTION factorial
```

But it is easy to see how to do this just as easily using a DO-loop.

The use of a `RESULT` variable is optional here, but required when the syntax would otherwise be ambiguous, e.g. when the function returns an array so an array element reference cannot be distinguished from a function call.

8 Derived Data Types (structures)

The terms **user-defined type**, and **data structure** and **derived type** all mean the same thing. A simple example is shown here, designed to keep handle a list of celestial objects in an observing proposal. The first step is to define the structure:

```
TYPE :: target_type
    CHARACTER(15) :: name     ! name of object
    REAL ::          ra, dec ! celestial coordinates, degrees
    INTEGER ::       time    ! exposure time requested, secs
END TYPE target_type
```

Note that one can mix character and non-character items freely (unlike in common blocks). The compiler arranges the physical layout for efficient access.

This only specifies the structure: to create actual variables with this user-defined data type the `TYPE` statement is used in a different form:

```
TYPE(target_type) :: old_target, new_list(30)
```

This has created a structured variable, and an array of 30 elements, each of which has the four specified components.

8.1 Accessing Components

Components of a structure are accessed using per-cent signs (unfortunately not dots as in many other languages, because of syntax ambiguities).

Thus `old_target%name` is a character variable, while `new_list(13)%ra` is a real variable. Such structure components can be used exactly like simple variables of the same data type:

```
    new_list(1)%name = "Cen X-3"
    new_list(1)%ra   = 169.758
    new_list(1)%dec  = -60.349
    new_list(1)%time = 15000
! .....
    new_list(2) = old_target      ! copy all components
    new_list(2)%time = 2 * new_list(2)%time
```

A space is optional either side of the per-cent sign. Note also that component names are local to the structure, so that there is no problem if the same program unit also uses simple variables with names like `name`, `ra`, `dec` etc.

8.2 Structure constructors

These allow all the components of a structure to be set at once, the type-name is used as if it were a conversion function, with a list of the component values as arguments:

```
    new_list(3) = target_type("AM Her", 273.744, 49.849, 25000)
```

If you have an array of some structured type, each component may be treated as if it were an array: thus `new_list%dec` is an array of 30 real values. The elements may not be in adjacent locations in memory, but the compiler takes care of this:

```
    total_time = SUM(new_list%time)
```

Besides their use in assignment statements, structured variables can be used in input/output statements. With unformatted or list-directed I/O this is straight-forward, but with formatted I/O one has to provide an appropriate list of format descriptors:

```
    WRITE(*,*) old_target         ! list-directed format easy
    READ(file, "(A,2F8.3,I6)") new_list(4)
```

8.3 Nested Structures

Two or more structure definitions may be **nested**:

```
    TYPE :: point
       REAL :: x, y   ! coordinates
    END TYPE point

    TYPE :: line
       TYPE(point) :: end(2)   ! coordinates of ends
       INTEGER     :: width    ! line-width in pixels
    END TYPE line

    TYPE(line) :: v
    REAL       :: length
    v = line( (/ point(1.2,2.4), point(3.5,7.9) /), 2)
    length = SQRT((v%end(1)%x - v%end(2)%x)**2          &
```

```
               +  (v%end(1)%y - v%end(1)%y)**2)
```

8.4 Pointers as Components

One limitation of Fortran structures is that array components must have their length fixed in advance: an an allocatable array cannot be a component of a structure. Fortunately pointer components are permitted:

```
   TYPE :: document_type
      CHARACTER(80), POINTER :: line(:)
   END TYPE document_type
!
   TYPE(document_type) :: mydoc  ! declare a structured variable
   ALLOCATE(mydoc%line(1200))    ! space for 1200-lines of text
```

To make the structure even more flexible one might allocate an array of `CHARACTER(LEN=1)` variables to hold each line of text, although this would not be as easy to use.

In order to pass a structured variable to a procedure it is necessary for the same structure definition to be provided on both sides of the interface. The easiest way to do this is to use a module.

There are, however, two limitations on the use of derived type variables containing pointer components:

1. They may not be used in the I/O lists of `READ` or `WRITE` statements.

2. If an assignment statement copies one derived type variable to another, any pointer component merely **clones** the pointer, the new pointer still points to the same area of storage.

8.5 Defined and overloaded operators

When a new data type is defined, it would often be nice if objects of that type could be used in expressions, because it is much easier to write, say
`a * b + c * d` than
`add(mult(a,b),mult(c,d))`.

Each operator you want to use has to be defined, or **overloaded**, for each derived data type.

This example defines a new data type, `fuzzy`, which contains a real value and its standard-error. When two *fuzzy* values are added the errors add quadratically. Here we define or **overload** the "+" operator:

```
  MODULE fuzzy_maths
    IMPLICIT NONE
    TYPE fuzzy
      REAL :: value, error
    END TYPE fuzzy
    INTERFACE OPERATOR (+)
      MODULE PROCEDURE fuzzy_plus_fuzzy
    END INTERFACE
  CONTAINS
    FUNCTION fuzzy_plus_fuzzy(first, second) RESULT (sum)
      TYPE(fuzzy), INTENT(IN) :: first, second  ! INTENT(IN) required
      TYPE(fuzzy)             :: sum
      sum%value = first%value + second%value
      sum%error = SQRT(first%error**2 + second%error**2)
    END FUNCTION fuzzy_plus_fuzzy
  END MODULE fuzzy_maths

PROGRAM test_fuzzy
IMPLICIT NONE
```

```
USE fuzzy_maths
TYPE(fuzzy) a, b, c
  a = fuzzy(15.0, 4.0) ;  b = fuzzy(12.5, 3.0)
  c = a + b
  PRINT *, c
END PROGRAM test_fuzzy
```

The result is, as you would expect: 27.5   5.0

In a similar way, the assignment operator, = can also be overloaded for derived data types, but in this case one uses a subroutine with one argument INTENT(IN) and the other INTENT(OUT).

Further definitions might cover:

- Other operators: **-** **/** ***** ****** etc.

- Combinations of *real* and *fuzzy* operands.

- Intrinsic functions like SQRT.

When a new data type has been defined in this way:

- It can be retro-fitted to existing software with a USE statement and a few changes from REAL to TYPE(FUZZY).

- It is possible to change the internal representation without affecting the software which uses the fuzzy type if the interfaces are preserved.

Overloading an existing operator is sensible only if the meaning is unchanged. Otherwise it is best to invent a new one. For example, **.like.** to compare to character-strings, or **.union.** for a set-operator.

The precedence of an existing operator is unchanged by overloading; new unary operators have a higher precedence, and new binary operators have a lower precedence than all intrinsic operators.

9   Input Output

9.1   New OPEN and INQUIRE options

- ACTION="READ" – to ensure read-only access (other actions are   "WRITE" or "READWRITE").

- POSITION="APPEND" – to append to an existing sequential file.

- POSITION="REWIND" – to ensure that existing file opened at beginning (more portable).

- STATUS="REPLACE" – to overwrite any old file or create a new one.

- RECL=length – can also be used when creating a **sequential** file such as a text file to specify the (maximum) record length required (units are *characters* for formatted access, otherwise system-dependent).

The INQUIRE statement has additional keywords to return information on these aspects of an open unit.

The **record-length units** of an unformatted (binary) direct-access are system-dependent: there is now a portable solution using a new form of the INQUIRE statement. You supply a specimen I/O list and it returns the length to use in the OPEN statement.

```
  INQUIRE(IOLENGTH=length) specimen, list, of, items
  OPEN(UNIT=unit, FILE=fname, STATUS="new", ACCESS="direct", RECL=length)
```

9.2 Internal File I/O

List-directed (free-format) reads and writes can now be used with internal files:

```
CHARACTER(LEN=10) :: string
string = "   3.14   "
READ(string, *) somereal
```

9.3 Formatted I/O

New/improved descriptors for formatted read and write:

```
REAL :: x = 0.125
WRITE(*, "(E12.3, ES12.3, EN12.3)") x, x, x
```

produces:

```
  0.125E+00   1.250E-01 125.000E-03
```

`ESw.d` produces **scientific format** with the decimal after the first digit, while `ENw.d` produces **engineering format** with an exponent which is always a multiple of 3. For input they are all exactly equivalent.

Integers can be read/written using hexadecimal, octal, or binary conversions:

```
INTEGER :: n = 125
WRITE(*, "(I10, Z10, O10, B10)") n, n, n, n
```

Produces:

```
       125         7D        175    1111101
```

All these may specify a **minimum number of digits** to be output: e.g. `Z10.6`

The **generic descriptor** `Gw.d` may be used for all data types, including logical and character.

9.4 Non-advancing I/O

This is a new facility, not quite stream-I/O, but nearly. Normal (advancing) `READ` and `WRITE` statements always process at least one whole record. Non-advancing ones only move a notional pointer as far as needed. A non-advancing write allows user input on the same line as a screen-prompt:

```
WRITE(*, "(A)", ADVANCE="no") "Enter the number of iterations: "
READ(*, *) nloops
```

A non-advancing read can measure the actual length of an input line using the new `SIZE` keyword.

```
CHARACTER(LEN=80) :: text
INTEGER :: nchars, code
READ(unit, "(A)", ADVANCE="no", SIZE=nchars, IOSTAT=code) text
```

If the line entered is too short then the `IOSTAT` return-code will be negative (and different from the value signalling end-of-file).

10 Character Handling

Much simpler, especially because of the many new or improved **intrinsic functions**:

```
c = ACHAR(I)                        Char in Ith position in ASCII table
i = IACHAR(C)                       Position of Char in ASCII table
i = LEN_TRIM(STRING)                Length ignoring trailing spaces
s = TRIM(STRING)                    String with trailing spaces removed
s = ADJUSTL(STRING)                 Adjust left by removing leading spaces
s = ADJUSTR(STRING)                 Adjust right by removing trailing spaces
s = REPEAT(STRING, NCOPIES)         Repeated concatenation
i = INDEX(STRING, SUBSTRING, back)  reverse search if back .true.
i = SCAN(STRING, SET, back)         Scan for 1st of any of set of chars
i = VERIFY(STRING, SET, back)       Scan for 1st char not in set
```

Other changes:

Overlapping substrings in assignments are permitted:

```
text(1:5) = test(3:7)   ! now ok, invalid in Fortran77
```

The **concatenation operator** // may be used without restriction on procedure arguments of passed-length.

Character functions may return a string with a length which depends on the function arguments, e.g.

```
FUNCTION concat(s1, s2)
IMPLICIT NONE
CHARACTER(LEN=LEN_TRIM(s1)+LEN_TRIM(s2)) :: concat  ! function name
CHARACTER(LEN=*), INTENT(IN) :: s1, s2
concat = TRIM(s1) // TRIM(s2)
END FUNCTION concat
```

Zero-length strings are permitted, e.g. a sub-string reference like `string(k:n)` where $k > n$, or a constant like `""`.

Sub-strings of constants are permitted, e.g. to convert an integer, **k**, in the range 0 to 9 into the corresponding character:

```
achar = "0123456789"(k:k)   ! note: error if k < 0 or k > 9.
```

11   Pointers

Many programming languages support pointers, as they make it easier to implement dynamic data structures such as linked lists, stacks, and trees. Programs in C are heavily dependent on pointers because an array passed to a function instantly turns into a pointer. **But:**

- Pointers may force the programmer to do low-level accounting better left to the the compiler.

- Excessive use of pointers leads to obscure and unmaintainable code.

- It is easy to make mistakes detectable only at run-time: a high proportion of bugs in C arise from accidental misuse of pointers.

- Pointers inhibit compiler optimisation (because two apparently distinct objects may be just pointers to the same memory location).

The Java language is, to a large extent, a pointer-free dialect of C++. Clearly pointers must to be used with care. Fortunately Fortran pointers are relatively tame.

11.1   Pointer Rules

A pointer can only point to another pointer or to a variable explicitly declared to be a valid `TARGET`.

Unfortunately a pointer starts life in limbo, neither associated nor disassociated (fixed in Fortran95). The best practice is to nullify each pointer at the start of execution, like this:

```
      NULLIFY(parray)
```

and then a test of `ASSOCIATED(parray)` would be valid, and would return `.false.` until it had been pointed at some actual storage.

When a pointer array is passed as an argument to a procedure which also declares it to be a pointer, the lower-bounds of the argument are passed across as well as the upper-bounds. In all other cases, the lower bounds need to be specifically declared in the procedure, and default to one unless otherwise specified.

11.2 Array of arrays

Fortran does not allow an array of pointers, but it does allow an array of derived-type objects which have pointers as components.

```
   TYPE :: ptr_to_array
      REAL, DIMENSION(:), POINTER :: arr
   END TYPE ptr_to_array
   TYPE(ptr_to_array), ALLOCATABLE :: x(:)
   !...
   ALLOCATE(x(nx))
   DO i = 1,nx
       ALLOCATE(x(i)%arr(m))
   END DO
```

11.3 Pointer as alias

Pointers are valuable as short-hand notation for array sections, e.g.

```
   REAL, TARGET :: image(1000,1000)
   REAL, DIMENSION(:,:), POINTER :: alpha, beta
   alpha => image(1:500, 501:1000)
   beta  => image(1:1000:2, 1000:1,-2)   ! axis flipped
```

Note that pointer assignment uses the symbol `=>` to distinguish the operation from actual assigment of a value.

11.4 Function may return a pointer

A case in which it is useful for a function to return a pointer to an array is illustrated by the reallocate function below.

```
MODULE realloc_mod
CONTAINS
  FUNCTION reallocate(p, n)                   ! reallocate REAL
    REAL, POINTER, DIMENSION(:) :: p, reallocate
    INTEGER, intent(in) :: n
    INTEGER :: nold, ierr
    ALLOCATE(reallocate(1:n), STAT=ierr)
    IF(ierr /= 0) STOP "allocate error"
    IF(.NOT. ASSOCIATED(p)) RETURN
    nold = MIN(SIZE(p), n)
    reallocate(1:nold) = p(1:nold)
    DEALLOCATE(p)
  END FUNCTION REALLOCATE
END MODULE realloc_mod

PROGRAM realloc_test
USE realloc_mod
```

```
IMPLICIT NONE
REAL, POINTER, DIMENSION(:) :: p
INTEGER :: j, nels = 2
  ALLOCATE(p(1:nels))
  p(1) = 12345
  p => reallocate(p, 10000)    ! note pointer assignment
  WRITE(*,*) "allocated ", nels, size(p), " elements"
  WRITE(*,*) "p(1)=", p(1)
END PROGRAM realloc_test
```

Note that pointer assignment uses the symbol `=>` since it needs to be distinguished from simple assignment of a value.

11.5   Dynamic Data Structures

Pointers can be used to construct complex dynamic data structures of all types, such as singly and doubly-linked-lists, binary-trees, etc. This is possible because a variable of derived type may contain a pointer which points to itself or to another object of the same type.

Pointers may only point to objects which have been declared with the `TARGET` attribute, to other pointers, or to arrays allocated to a pointer.

12   Portable Precision

It was a common extension to Fortran77 to allow declarations of the form `LOGICAL*1`, `INTEGER*2`, or `REAL*8`. But this simple scheme was **not** adopted for Fortran90.

Instead Fortran90 defines 5 distinct intrinsic data types (character, logical, integer, real, complex) but allows for different **kinds** of them to exist. Two kinds of **real** and **complex** are required (the second kind of real has the alias of `DOUBLE PRECISION`. Systems may support additional kinds of any of the 5 intrinsic data types.

The **kind** is specified with an integer, e.g. `INTEGER(2)` instead of `INTEGER*2` but the Standard does not define what the integer means. To make software portable, two intrinsic functions are provided: `SELECTED_INT_KIND` selects an integer kind value for the minimum number of decimal digits you want, and `SELECTED_REAL_KIND` does the same for reals given the minimum significant decimal digits and exponent range. Thus:

```
INTEGER, PARAMETER :: &
  short = SELECTED_INT_KIND(4), &   ! >= 4-digit integers
  long  = SELECTED_INT_KIND(9), &   ! >= 9-digit integers
  dble  = SELECTED_REAL_KIND(15, 200)  ! 15-digit reals to 10**200

INTEGER(short) :: myimage(1024,1024)
INTEGER(long)  :: counter
REAL(double) :: processed_data(2000,2000)
```

It may be that on some systems the same kind of integer would be used for those declared `short` and `long`, but this should not affect portability.

The best practice is to include definitions of kind parameters (like those above) in a **module** which is used throughtout the program.

Constants may have their kind parameter appended, where kind matching is required (e.g. in procedure arguments):

```
CALL somesub( 3.14159265358_dble, 12345_long, 42_short)
```

Another intrinsic function, `KIND` returns the kind parameter of any variable.

```
WRITE(*,*) " Double precision kind is ', KIND(0d0)
```

In principle the kind system may be extended to characters – Fortran systems are free to support 16-bit character-sets such as Unicode.

13   Other features

13.1   Bit-wise operations on integers

All the MIL-STD intrinsics for bit-manipulation are now standardized. Bit are numbered from 0 on the right, i.e. the least-significant end.

`i = BTEST(i, ipos)`	Bit testing
`i = IAND(i, j)`	Logical AND
`i = IBCLR(i, ipos)`	Clear bit
`i = IBITS(i, ipos, len)`	Bit extraction
`i = IBSET(i, ipos)`	Set bit
`i = IEOR(i, j)`	Exclusive OR
`i = IOR(i, j)`	Inclusive OR
`i = ISHFT(i, j)`	Logical shift left (right if `j` -ve)
`i = ISHFTC(i, j)`	Circular shift left (right if `j` -ve)
`i = NOT(i)`	Logical complement
`i = BIT_SIZE(i)`	Number of bits in variables of type `i`

`CALL MVBITS(from, frompos, len, to, topos)` is an intrinsic subroutine which copies bits from one integer to another.

Binary, octal, and **hex** values may be read and written using new format descriptors `Bw.d`, `Ow.d`, `Zw.d`, and that `DATA` statements may contain binary, octal, and hex constants.

13.2   Other intrinsic functions

`FLOOR` and `MODULO` work like `AINT` and `MOD` but do sensible things on negative numbers, and `CEILING` which rounds up to the next whole number.

`TRANSFER` may be used to copy the bits from one data type to another – a type-safe alternative to tricks formerly played with `EQUIVALENCE` statements.

    LOGICAL, PARAMETER :: bigend = IACHAR(TRANSFER(1,"a")) == 0

This sets `bigend` to `.TRUE.` on a big-endian hardware platform, and `.FALSE.` otherwise.

Numerical enquiry functions include `BIT_SIZE`, `DIGITS EPSILON`, `MAXEXPONENT`, `MINEXPONENT`, `PRECISION`, `RADIX`, and `RANGE`. Perhaps the most useful of these are `TINY` which returns the smallest non-zero real (of whatever kind), and `HUGE` which returns the largest representable number (integer or real).

System access intrinsics include:
`DATE_AND_TIME`, an intrinsic subroutine, which returns the current date and time as a string or an array of integers,
`RANDOM_NUMBER` which returns a whole array of pseudo-random numbers,
`RANDOM_SEED` which can randomise the seed.
`SYSTEM_CLOCK` useful in timing tests.
In Fortran95 a true `CPU_TIME` routine is introduced.

14   Resources

Best WWW resources:
The Fortran market: `http://www.fortran.com/fortran`

FAQ at `http://www.ifremer.fr/ditigo/molagnon/fortran90/engfaq.html`

These have links to tools such as style-converters and interface block generators, free software, and commercial products.

The Usenet news group `comp.lang.fortran` now has almost as many postings on Fortran90 as on Fortran77.

The mailing list `comp-fortran-90` has on-line archives at `http://www.jiscmail.ac.uk` which also contains joining instructions.

The best **book** on Fortran90 for existing Fortran users is, in my opinion, **Upgrading to Fortran 90** by Cooper Redwine, published by Springer, 1995, ISBN 0-387-97995-6.

An alternative is *Fortran90/95 Explained* by Michael Metcalf & John Reid, published by Oxford University Press, ISBN 0 19 851888 9. This is comprehensive, but in my opinion, sometimes a bit too concise.

See also **Numerical Recipes in Fortran90** by Press et. al. published by CUP, ISBN 0-521-57439-0.

15 Language Progression

15.1 Deprecated features

The following features of antique Fortran are officially termed **deprecated** and some of them have been officially removed from Fortran95 (but most compilers just issue warnings if you use them):

- `DO` with a control variable of type `REAL` or `DOUBLE PRECISION`.

- A `DO` loop ending on a statement other than `CONTINUE` or `END DO`.

- Two or more `DO` loops ending on the same statement.

- The **arithmetic** `IF` statement (three-way branch).

- Hollerith `FORMAT` descriptor nH*string*.

- A branch to `END IF` from outside the IF-block (allowed in Fortran77 by mistake).

- The `PAUSE` statement.

- The `ASSIGN` statement together with **assigned** `GO TO` and **assigned** `FORMAT` statements.

- The alternate `RETURN` facility.

If you are unfamiliar with these, then you never need to know about them.

15.2 Superseded features

Some other features, still commonly used in Fortran77, are essentially redundant and should be avoided in newly-written code. For example: Fixed source form, implicit data typing, COMMON blocks, assumed-size arrays, **EQUIVALENCE**, **ENTRY**, **INCLUDE**, **BLOCK DATA** program units. Specific names of intrinsics.

15.3 Main New Features of Fortran95

The major features include

- `FORALL` statement and construct, e.g.
  `FORALL(i=1:20, j=1,20) x(i,j) = 3*i + j**2`

- `PURE` and `ELEMENTAL` user-defined subprograms

- initial association status for pointers using `=> NULL()`

- implicit initialisation of derived type objects

- new intrinsic function **CPU_TIME**

- automatic deallocation of allocatable arrays

- Format width zero (e.g. `i0`) produces minimum number of digits required.

A good on-line coverage of these new features is given in http://www.nsc.liu.se/ boein/f77to90/f95.html

15.4   Possible New Features of Fortran2000

- High Performance, Scientific and Engineering Computing:

  - Asynchronous I/O
  - Floating point exception handling
  - Interval arithmetic

- Data Abstraction / User Extensibility:

  - Allocatable components
  - Derived type I/O

- Object-oriented Fortran:

  - Constructors/destructors
  - Inheritance
  - Polymorphism

- Parameterised derived types

- Procedure pointers

- Internationalization

- Inter-operability with C

The name is not yet decided, may be Fortran 2002!