XMM Survey Science Centre
University of Leicester
Leicester, LE1 7RH, UK

SSC-LUX-TN-0006
version 1.0
1996 September 16

Clive G. Page (cgp@star.le.ac.uk)

Fortran 90 for Fun and Profit

GOD is real (unless declared integer)

— old Fortran joke (origin unknown)

The last good thing written in C was Franz Schubert's Symphony No. 9

— Michael Hodous (Centro Svizzero di Calcolo Scientifico)

Fortran 90 is now the only international standard for Fortran: the release of ISO/IEC 1539:1991 made Fortran 77 obsolete. A radical revision of the language was long overdue, but was probably worth the wait: Fortran 90 is a state-of-the-art procedural language. Programmers will find it not only much more powerful but also easier to use, because so many restrictions are relaxed and so many awkward and antique features can be put into retirement. Fortran 90 also supports more comprehensive compile-time checking, so that a greater variety of programming errors can be caught at an early stage. Despite these substantial changes, Fortran 77 remains a true subset of Fortran 90, so compatibility with existing code is maintained.

Scientific programmers have, on the whole, been rather slow to adopt Fortran 90, although there are some notable exceptions. An early scarcity of compilers may have been partly responsible. I suspect also that those keenest to jump on the latest programming bandwagon have already switched to object-oriented languages like C++, while many of the others remain reluctant to use anything that looks significantly different from Fortran 77. Now that reliable and efficient Fortran 90 compilers are available for all modern computing platforms I think the benefits of adopting Fortran 90 far outweigh the risks.

Existing Fortran programmers should not find it too difficult to adjust to Fortran 90: several of its features are well-known as extensions to many Fortran 77 systems, while the more novel features can if necessary be adopted one-at-a-time as the need arises. Taken together, however, they amount to a substantial rejuvenation of Fortran, indeed they almost turn it into a new language. It will therefore take a little time for everyone to become adept in the rather different style of programming which Fortran 90 makes possible and indeed desirable.

This document highlights some of the most important new features as far as the scientific programmer is concerned. It does not attempt to describe them all in detail, which would take a whole book. Indeed I already know of 18 books on the subject in English and a dozen in other languages. The best book that I have found so far (for anyone with a reasonable background knowledge of Fortran 77) is *Upgrading to Fortran 90* by Cooper Redwine[1].

[1] published 1995 by Springer Verlag, ISBN 0-387-97995-6 (£25 in the UK)

1 Why Use Fortran 90?

The principal aims of those designing the new standard were to increase the expressive power of the language, to make programming simpler, and to increase the reliability, portability, and efficiency of Fortran programs.

1.1 Expressive Power

Fortran now has all the features that one expects in a modern programming language:

- Arrays are first-class objects and can be used in expressions, as arguments to intrinsic functions, and in assignment statements; there is also a concise array-section notation.
- Dynamic storage is fully supported, with both automatic and allocatable arrays; pointers can be used to handle more complex objects such as linked-lists and trees.
- Derived data types (data structures) are fully supported.
- The `MODULE` is a new program unit which fully supports encapsulation of data and procedures, and greatly facilitates code re-use.
- Procedures can be recursive, or generic, they can have optional or keyword arguments, arrays can be passed complete with size/shape information, and functions can return arrays or data structures.
- New operators can be defined, or existing operators can be overloaded for use on objects of derived type.
- Modules and data structures may declare their contents `PRIVATE` to enhance encapsulation and avoid name-space pollution.

1.2 Simplicity

Many changes make it easier to write and to read programs. Many annoying restrictions of the old standard are also removed or relaxed, such as the 6-character limit on symbolic names.

- New control structures make it feasible to program almost without statement labels.
- A free-format source code layout is introduced, as well as end-of-line comments, multi-statement lines, free use of lower-case, and more obvious relational operators such as "`>=`" instead of "`.GE.`", etc.
- A new form of *type* statement allows all attributes of a set of variables to be given in one place (with initial values if needed). This avoids the previous jumble of specification statements such as `DIMENSION`, `PARAMETER`, `SAVE`, and `DATA`.

1.3 Reliability

The increased legibility of Fortran should makes it easier to find mistakes; in addition several new features allow the compiler to provide much more assistance in the production of error-free code:

- The `IMPLICIT NONE` specification requires explicit type statements for all typed objects, so typographical errors in names usually produce error messages.
- Procedures can be given *explicit interfaces*, so that all arguments are checked by the compiler for agreement of data type, array rank, etc. This is a very valuable addition since errors in procedure calls are easy to make and hard to diagnose.
- The `INTENT` of procedure dummy arguments can be specified (`in`, `out`, or `inout`), so compilers can detect errors in argument usage when the interface is explicit.

1.4 Portability

Fortran programs have always been highly portable, but Fortran 90 eliminates almost all the remaining machine-dependent features of the language. It is also worth noting that, since the Standard now provides comprehensive facilities, most compilers provide few extensions and they present little temptation.

- The system of *kind* parameter provides a portable method of selecting data types (e.g. real or double precision) according to the actual precision or exponent range needed.
- File-handling and I/O statements have several new options to ensure machine-independence.

1.5 Efficiency

Compilers for Fortran compare well with those of other languages in the production of highly-optimised code. Some features of Fortran 90 should allow them to do even better, especially on machines with significant parallelism, or multiple processors.

- By using whole-array operations the loops are handled by the compiler and are likely to be faster than those which merely translate `DO`-loops. Array-valued intrinsics functions will also be evaluated using the best machine-code loops.
- Dynamic arrays rely on the memory-management of the operating system but are likely to use physical memory more efficiently than static arrays declared large enough for the worst-case.
- Loops involving pointers cannot be optimised safely (a well-known limitation on the performance of C programs). In Fortran, however, pointers can only point to objects explicitly declared to be *targets*, so accesses to all non-target variables can be fully optimised.

2 Simple Improvements

A number of quite simple changes improve the clarity of Fortran programs, with obvious benefits for ease of use, and for code re-use and maintainability. The example below, though trivial, illustrates many of the new features. **Note:** this, like the other examples which follow, uses `UPPER-CASE` for Fortran keywords and intrinsic functions, but programmer-chosen symbolic names are shown in `lower-case`. *This distinction is made purely to illustrate the syntax, and is* **not** *a recommended style.* Fortran is, of course, case-blind (except within quoted character constants).

```
PROGRAM compute_modified_julian_date                          ! notes 1,2
IMPLICIT NONE                                                 ! note  3
INTEGER, PARAMETER :: offset = 18313                          ! note  4
INTEGER :: year, month, day, status, mjd
DO                                                            ! note  5
  WRITE(*, "(A)", ADVANCE="NO") " Enter day, month, year: "   ! notes 6,7
  READ(*, *, IOSTAT=status) day, month, year
  IF(status /= 0) EXIT                                        ! notes 8,9
  IF( day   < 1    .OR. day   >   31   .OR. &
      month < 1    .OR. month >   12   .OR. &
      year  < 1901 .OR. year  > 2099) THEN                    ! note  10
     WRITE(*,*)"Invalid date, try again" ; CYCLE              ! notes 11,12
  END IF
  mjd = 367*(year-1900) + 275*month/9 + day + offset  &
        - 7*(year+((month+9)/12))/4
  WRITE(*, "(I3,  A,  I2.2,   A,   I4,      A,     I6)") &   ! note  13
            day, "/", month, "/", year, " is MJD", mjd
END DO                                                        ! note  14
END PROGRAM compute_modified_julian_date                      ! note  15
```

Notes:

1. Note the use of free-format layout, so statements can start in column 1, but all comments must start with "`!`".
2. Symbolic names can be up to 31 characters long, and may include underscores.
3. `IMPLICIT NONE` is fully standardised, and strongly recommended.
4. The new declaration syntax (with two colons) allows all the attributes of a set of variables to be given in one place, including initial values or (as here) values of constants.
5. An indefinite `DO`-loop is permitted (but it needs an `EXIT` statement somewhere).
6. Non-advancing I/O allows partial records to be read or written: here it provides a terminal prompt without a new-line, so any reply appears on the same line.
7. Character constants can be enclosed in a pair of double or single quotes.
8. Relational operators may appear as e.g. "`>`" instead of "`.GT.`" and "`/=`" for "`.NE.`".
9. Here, the `EXIT` statement is executed if any I/O exception occurs (including end-of-file) and transfers control to the first statement outside the loop.
10. In free-format code, an incomplete line ends with an "`&`" to show the statement continues on the next line.
11. Multiple-statement lines are permitted, with semi-colon as separators.
12. The `CYCLE` statement continues execution from the top of the loop.
13. The format specification is here embedded in the `WRITE` statement; this avoids the need for a statement label, but allows easy matching of I/O items and format descriptors.

14. The `END DO` statement is (at last) part of the official standard.

15. `END` statements can generally state what it is that they are ending; this is optional but may help the reader, and may also be checked by the compiler.

3 Label-free Programming

Statement labels look untidy whether or not a margin is left for them. More seriously, each labelled statement marks the potential destination for a jump: in order to understand the program properly the origin of each jump must be located. This involves searching not only for `GOTO` and arithmetic-`IF` statements, but also checking all I/O statements (in case they use `END=`*label* etc.) and even `CALL` statements in case they make use of alternate return. The presence of labels, therefore, makes it is harder to check programs for mistakes and they represent a continuing obstacle to maintenance.

Fortunately, new data structures make it feasible to avoid labels nearly all the time. `DO`-loops terminated with an `END DO` can be label-free, especially if good use is made of `EXIT` and `CYCLE` statements. The old computed `GOTO` is superseded by the `SELECT CASE` structure, which needs no labels at all.

This is shown in the program fragment below which selects a suitable ordinal suffix for a day number in the range 1 to 31, e.g. to turn "`3`" into "`3rd`", and so on.

```
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 = '??'
END CASE
WRITE(*, "(I4,A)") day_number, suffix
```

The selection expression in `SELECT CASE` may be of integer or character-string type; the ranges in the case statements must not overlap; a default clause is optional.

4 Arrays and Dynamic Storage

Arrays are likely to remain the principal data structure in scientific computing. In Fortran 90 arrays are first-class objects, which means they can be used almost everywhere just like scalars. One can have an array of constants, and there is an array constructor notation, e.g.

```
INTEGER, PARAMETER :: limits(5) = (/ 12, 31, 24, 60, 60 /)
```

Dynamic storage is fully supported through the use of *automatic* arrays, *allocatable* arrays, and *pointers*. These facilities free the programmer from the need to guess the maximum array size ever likely to be needed, and take advantage of the efficient memory management facilities provided in modern operating systems.

4.1 Whole-array Operations

An important new feature is that whole-array expressions and assignments are permitted: the compiler arranges the necessary looping over all elements. In addition practically all intrinsic functions work element-wise when given an array as an argument. This eliminates the need for many simple DO-loops, for example:

```
  PROGRAM background_subtraction
  REAL, DIMENSION(512,512) :: raw, background, exposure, result, weight
!    (code here defines raw, background, exposure...)
  result = (raw - background) / exposure
  weight = SQRT(0.001 * exposure)
```

In array expressions and assignments the arrays must be *conformable*, i.e. have the same *rank* (number of axes) and the same extent along each axis. A scalar is deemed conformable with any array: conceptually its value is duplicated the required number of times. Thus if you add a constant to an array, every element has that constant added.

In cases where some elements need to be excluded from an array assignment, the `WHERE` block can be convenient. Here it removes the risk of division by zero:

```
  WHERE(exposure > 0.01)
    result = (raw - background) / exposure
  ELSEWHERE
    result = 0.0
  END WHERE
```

4.2 Array Sections

Array sections can be specified by giving the `(first:last)` element of each dimension, or `(first:last:step)` if the step-size is not unity. If the step-size is negative, the elements are accessed in decreasing order. For example, a 2-d array `image(512,512)` can be flipped along the second axis using an assignment statement like this:

```
  image = image(:, 512:1:-1)
```

Note that a colon as a subscript represents the use of all elements along that dimension. Source and destination arrays may have overlapping ranges: the compiler will allocate temporary storage space if necessary.

Another new feature is that subscripts can be vectors, e.g. `image( (/15, 7, 31/), 123)` is a section with three elements. If such a section is used on the left-hand side of an assignment the elements of the vector must all be different.

4.3 Automatic Arrays

The automatic array is a local array in a procedure which has a size which depends on the arguments of the procedure when it is called. The array vanishes each time control returns to the calling routine.

```
  SUBROUTINE my_process(npoints, array)
  INTEGER, INTENT(IN) :: npoints
  REAL, INTENT(INOUT) :: array(npoints)        ! argument array
  DOUBLE PRECISION    :: workspace(2*npoints)  ! automatic array twice as big
```

4.4 Allocatable arrays

These provide a more general mechanism: only the rank (number of dimensions) has to be declared in advance, the actual dimension bounds are specified later in an `ALLOCATE` statement.

```
  INTEGER :: nx, ny
  DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:,:) :: image
!... compute suitable values for nx,ny
  ALLOCATE(image(nx,ny))    !allocate space for rank 2 array
```

The allocatable array can be used just like any other array, but when no longer needed the space should be released using:

```
  DEALLOCATE(image)
```

Once an array has been allocated, its bounds can only be changed by deallocating and re-allocating it, which loses any previous contents. An allocatable array in a procedure may also be given the `SAVE` attribute so that its contents are preserved from one invocation to another. It is important to have properly matched `ALLOCATE` and `DEALLOCATE` statements, of course, and simplest if they appear in the same procedure. `ALLOCATE` statements have an optional `STATUS` argument which returns an error-code if there is not enough dynamic memory left for successful allocation.

4.5 Pointers

The pointer provides a more general way of using dynamic storage with even greater flexibility, e.g. to handle a collection of objects all of different sizes. In C and C++ pointers are used extensively, but they account for a large proportion of programming errors, since it is very easy to access invalid memory areas inadvertently. Fortran has not entirely eliminated this risk, but has controlled it by requiring that each pointer can only point to items of a specified data type which have also been explicitly declared to be a *target*. Despite these restrictions, Fortran pointers can be used to implement all forms of dynamic data structure such as linked-lists, B-trees, queues, etc.

5 Derived data types

Fortran's support for derived data types (sometimes known as user-defined data types or data structures) is now superior to that found in most other high-level languages. First one defines the structure of a derived type in a block enclosed in `TYPE` and `END TYPE` statements, for example:

```
  TYPE celestial_position
    REAL             :: ra
    REAL             :: dec
    CHARACTER(LEN=5) :: equinox
  END TYPE celestial_position
```

Then one can declare variables (including arrays) using further `TYPE` statements of this form:

```
  TYPE(celestial_position) :: target, obs_list(10)
```

Fortran uses a percent sign "`%`" between components of compound names where most other languages use a dot to avoid a syntax ambiguity (operators like `.AND.` are to blame). This notation looks ugly, but one gets used to it. Thus `target%ra` is a *real* variable, and `obs_list(5)%dec` is an element of a *real* array; both can be used just like simple variables.

```
  call convert(vector, target%ra, target%dec)
```

```
obs_list(1)%equinox = "J2000"
```

Fortran extends the syntax further than most other languages, so that, for example, `obs_list%ra` is an array of 10 real elements. These elements are (probably) not located in contiguous memory locations, but this is the compiler's problem not yours. Note that simple symbolic names such as `ra` and `dec` can be used concurrently without ambiguity, since names of structure components always contain at least one percent sign.

The first line below shows how to use a *structure constructor* to set values for all components in one operation. It is followed by another assignment statement, which simply copies all the components to the new location. Variables of derived type can also be used in I/O statements, but in formatted transfers one has to provide a list of format descriptors corresponding to the list of components.

```
obs_list(1)  = celestial_position(12.34, -45.67, "B1950")
obs_list(42) = obs_list(1)
write(*, "(2F8.2,A)" )  obs_list(1)
```

It is easy to define one derived type in terms of another; references to these components use an obvious extension to the notation:

```
TYPE star_type
   CHARACTER(LEN=20)        :: name
   TYPE(celestial_position) :: position
   REAL                     :: magnitude
END TYPE observed
TYPE (star_type) :: mystar, catalogue(10000)
mystar%name         = "HD12345"
mystar%position%ra  = 65.4321
mystar%position%dec = 12.345
mystar%magnitude    =  9.5
```

Variables of derived type can be used in expressions only if all the operators involved have their actions defined in advance for the data types involved: this is described in section 8 below. If existing operators are re-defined this is called *operator overloading*.

The main point of using derived types is to group related data together in a single named object, this can simplifies procedure calls which would otherwise involve passing a list of separate arguments, one for each component. In order to pass a derived type object to a procedure the same structure definition must be available in both the calling and called program. The best way to do this is to put the derived type definition in a `MODULE`, and `USE` it in both places.

6 Modules

The module is an entirely new type of program unit which, though not executed directly, allows other program units to share items such as constants, arrays, data structure definitions, and procedures. It is likely to have a more revolutionary effect on the way that Fortran programs are constructed than anything since the invention of the subroutine around 1960. The module makes the `COMMON` block, the `INCLUDE` statement, and the `BLOCK DATA` program unit almost redundant.

The very simplest use for a module is to define some constants so that exactly the same values are available in a number of other program units. Such a module can be constructed just like this:

```
MODULE basic_constants
  DOUBLE PRECISION, PARAMETER :: pi = 3.1415926535890, &
                                 dtor = pi/180.0d0, rtod = 180.0d0/pi
END MODULE basic_constants
```

Then in each program unit which makes use of these constants, all one needs is a `USE` statement at the top, like this:

```
USE basic_constants
```

This could, of course, have been handled with an `INCLUDE` statement, but the advantage of using a module is that its text is parsed and pre-compiled when first encountered, so there is some gain in compilation speed for larger modules.

The `USE` statement can also contain an `ONLY` clause which controls which names are to be accessed, and there is also a rename facility if name-clashes are otherwise unavoidable.

More important uses for modules are for them to contain:

- Derived-type definitions; needed to pass a derived type object to a procedure.
- Global data: a module can contain a list of variables and arrays which then become accessible in all the program units which use the module. This provides a replacement for the `COMMON` block, and reduces the risk of having of inconsistent definition in different places.
- Both a data area and a set of procedures: these have full access to the data. This allows the construction of a modular package or library. For example, a set of graphics routines would probably require a global data area to hold the attributes of the current graphics device, scaling factors, etc.

Fortran supports *encapsulation* and data hiding by allowing the programmer to choose whether each item in a module is to be `PUBLIC`, i.e. accessible in the program unit which uses it, or to be `PRIVATE`, i.e. accessible within the module code alone. A more general structure for a module defining a package is then something like this:

```
MODULE module_name
  IMPLICIT NONE
  PRIVATE                         ! set default status for all names
  PUBLIC sub1, sub5, ...          ! allow public use of these items
  <data structure definitions>
  <global data storage area>
CONTAINS
  SUBROUTINE sub1
    <code which may access the global data area>
  END SUBROUTINE sub1
  <any number of further module procedures>
END MODULE module_name
```

7 Procedures

Perhaps the greatest defect of Fortran 77 was the lack of checking of procedure calls. Even in professional code it has been estimated that around 20% of procedure calls are defective in some way. This is now remedied by what Fortran calls the *explicit interface*.

7.1 Explicit Interfaces

The term is not very informative: an interface is said to be explicit if the compiler has access at the same time to both the dummy arguments of a procedure and the actual arguments of the call. In such cases it can perform many valuable consistency checks, for example that the number of arguments is the same, that each has the same data type, and that arrays have the same shape and size. If the dummy arguments also have their `INTENT` specified, this can also be checked (so that output arguments which correspond to a constant or expression will raise an error). In this way a great many programming mistakes can be identified at compile-time. These advantages are so great that many Fortran experts now think that all programmers should use explicit interfaces as a matter of course.

There are three different ways of making procedure interfaces explicit:

1. The simplest is to put the procedures in a module: as described above this automatically makes their interfaces explicit to each other and to any unit which uses the module.

2. Interfaces are also automatically explicit for all *internal procedures*. The internal procedure is a useful generalisation of the statement function: any number of them of any length may follow a `CONTAINS` statement at the end of any other type of procedure.

3. For external procedures, it is possible provide the an interface in a separate interface block. These are rather like function prototypes in the C language, and require similar care to ensure that what is declared in the interface block matches the actual procedure interfaces.

When a procedure has an explicit interface several other useful facilities become available:

- Assumed-shape arrays: these have their shape and size transmitted to the procedure automatically, thus simplifying its interface.

- Optional arguments: likely to be useful in a wide range of applications. Each optional argument needs to be tested using the `PRESENT` intrinsic function; typically some default action would be taken when no argument is provided.

- Keyword arguments: calling arguments by keyword may save effort when there is a long list of them. In addition, if optional arguments are omitted from other than the end of an argument list, the remainder of the arguments must be called by keyword.

- Arguments may be pointers.

- Functions may be array-valued, or return derived-type objects.

7.2 Recursive Procedures

Recursive functions (and subroutines) will be useful not only in defining certain mathematical functions (the factorial is the obvious example) but also whenever it is necessary to handle self-similar data structures, for example recursive descent of a file directory, or of a B-tree data structure. If two recursive procedures are to call each other they must both be placed in the same module. Recursive functions will sometimes need a separate results variable to avoid ambiguity: this automatically has the same data type as the function name, for example:

```
RECURSIVE INTEGER FUNCTION factorial(n) RESULT(n_fact)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
```

```
IF(N > 0) THEN
   n_fact = n * factorial(n-1)
ELSE
   n_fact = 1
END IF
END FUNCTION factorial
```

7.3 Generic Procedures

With the aid of an interface block (which may be put in the same module as the procedures) it is now possible to define generic names for a group of procedures which carry out similar operations but on a range of different data types. This is likely to be useful in many low-level packages, e.g. when implementing a set of data access routines. Although generic interfaces do not in themselves result in any simplification of the code, they do simplify interface for the user of the package, and the corresponding documentation.

7.4 Intrinsic Procedures

Fortran 77 already had a better collection of built-in functions than any other common language, but Fortran 90 provides another 75 intrinsic functions and subroutines. There is only space here to mention a few of them.

Mathematical functions now include `MODULO` (which is like `MOD` except for negative arguments), while `CEILING` and `FLOOR` round to integer upwards and downwards respectively.

Array-handling functions include those for taking `DOT_PRODUCT` of vectors, and `MATMUL` and `TRANSPOSE` for matrices. For arrays of any rank one can now find minimum/maximum values and their locations using `MINVAL`, `MAXVAL`, `MINLOC` and `MAXLOC`, or simply `COUNT` the elements or find their `SUM` or `PRODUCT` or logical arrays whether `ALL` or `ANY` of them are true. There are several more complex routines such as those to `PACK` and `UNPACK` rank-one arrays, and `CSHIFT` to do a circular shift of elements.

Character-handling is much easier with function such as `LEN_TRIM` to find string-length ignoring trailing spaces, `TRIM` to trim them off, and `ADJUSTL` and `ADJUSTR` to justify strings while preserving length. There is also `SCAN` and `VERIFY` to check for presence or absence of sets of characters, while `ACHAR` and `IACHAR` convert single characters to/from integer with conversion guaranteed to use the ASCII collating sequence.

Bit-wise operations often have to be performed on integers when dealing with raw data from instruments. The full set of procedures originally defined in MIL-STD-1753 is supported in Fortran 90, including `IAND`, `IOR`, `ISHFT` and `MVBITS`.

Numerical enquiry functions appear for the first time, such as `TINY` and `HUGE` to find the smallest and largest numbers of any given type; one can also find the `BIT_SIZE`, `PRECISION`, `MAXEXPONENT` etc. for the floating-point types.

Miscellaneous intrinsics include functions to find the `SIZE` and `SHAPE` of an argument array, whether an optional argument is `PRESENT`, or an allocatable array actually `ALLOCATED`. The `RESHAPE` function can do clever things with multi-dimensional arrays, while `TRANSFER` allows data to be moved to another data type just by copying its bit-pattern. New subroutines include those to get the current `DATE_AND_TIME` in several formats, and to read

the `SYSTEM_CLOCK`. And a single call to `RANDOM_NUMBER` can generate an array of pseudo-random numbers.

Note that all intrinsic procedures may also be called by keyword, and some of were designed with this in mind. For example one can now get the current date-and-time in several different formats:

```
CHARACTER(8) :: mydate
INTEGER      :: iarray(8)
CALL DATE_AND_TIME(DATE=mydate)   ! returns date as "yyyymmdd"
CALL DATE_AND_TIME(VALUES=iarray) ! returns date/time in integer array
```

8 Defined and Overloaded Operators

When a new data type is defined, it will often be desirable for objects of the derived type to be used in expressions. Note how much easier it is to write: `a * b + c * d` than e.g.: `add(mult(a,b),mult(c,d))`. Before using operators on operands of non-intrinsic data type it is necessary to define what operations they perform in each case. The example below defines a new data type, `fuzzy`, which contains a real value and its standard-error. When two imprecise values are added together the errors add quadratically (assuming they are uncorrelated). So it is with *fuzzy* values, given this overloading of 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
    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

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
```

The result is, as you would expect: `27.5 5.0`

This module defines only what happens when two *fuzzy* values are added. Clearly, to be of practical value it would be necessary to provide further definitions for subtraction, multiplication, etc. One might also want to overload intrinsic functions such as `SQRT`, or to provide a function to multiply a fuzzy value by a *real*. All of these are easy to do, but to show them all here would

take up too much space.

Once a suitable set of overloadings has been defined for the mathematical operators and perhaps some intrinsic functions, then one can make use of them in a variety of ways. For example, a Fourier transform routine could be converted to use *fuzzy* arrays instead of *real* arrays by simply replacing all `REAL` declarations for those of `TYPE (fuzzy)`. The errors would then be propagated through the processing and the transform would contain a standard error at each frequency.

Such encapsulation greatly simplifies software maintenance. Suppose you decide later on that this representation of a *fuzzy* requires one to take too many square-roots, and that it would be better to store instead the square of the error, then it is only necessary to alter one or two lines or two in each procedure. Of course, the module and all the code which uses it needs to be recompiled, but in the higher level software no code are needed at all.

It is sensible to overload an existing operator only if the meaning stays essentially unchanged: in other cases it is better to invent a new operator name. For example, if you decide to implement an operator to compare two character string while ignoring case differences, none of the existing operator symbols seems an especially good choice, but it would be sensible to invent name such as `.like.` or `.similar.` for the purpose.

9 Input/Output enhancements

Non-advancing I/O is a new facility which allows a formatted `READ` or `WRITE` statement to transfer less than a complete record, and is particularly valuable in providing a standard way of writing a terminal prompt without a terminating new-line, as shown in section 2.

The `OPEN` statement has several new keywords e.g. `POSITION="APPEND"` to append output to an existing file, `ACTION="READ"` to specify that read-only access is required, and `STATUS="REPLACE"` which creates a new file in general, but replaces an old file of the same name if one already exists.

When opening an unformatted direct-access file the record-length has to be specified, but the units are system-dependent (often bytes, sometimes longwords). This minor portability problem has been solved in Fortran 90 by adding to the `INQUIRE` statement an option to determine the length of a specimen record in local units.

There are also several new format descriptors. Integers can now be read or written in other number bases, using `Bw.m` for binary, `Ow.m` for octal, and `Zw.m` for hexadecimal. If floating-point numbers are written using `ESw.d` they appear in *scientific format*, which is like `Ew.d` but there is always one non-zero digit before the decimal point. There is also *engineering format* using `ENw.d`, in which the exponent is always a multiple of three. In addition `G` format is extended in scope to cope with input and output of all data types, not just the numerical ones, which may be useful in generic code.

There are several of minor improvements, for example internal files can be handled with list-directed formatting.

10 Fortran 90 in Practice

Fortran 90 compilers are now available for almost all computing platforms from super-computers downwards, with more than half-a-dozen compilers on the market for Unix systems and a similar number for PCs, some of these products come from reputable suppliers of Fortran 77 compilers and are now quite mature and stable. The Macintosh has been somewhat neglected, but two compilers are due for release in 1996.

There are, however, a few potential snags in using Fortran 90 rather than Fortran 77, especially when using code which already exists and was designed for the older standard.

- Fortran 90 introduces some 70 new intrinsic functions (and a handful of intrinsic subroutines). If existing programs happen to use one of these names for an external function (or subroutine) the compiler may get confused. The solution is simple, to specify the name in an `EXTERNAL` statement in each calling program unit; of course the intrinsic of the same name cannot be used in that unit. This is just about the only area in which there is less than 100% compatibility with Fortran 77 code.

- Few Fortran 77 programs conformed strictly to the ISO Standard; many extensions in regular use were incorporated into the Fortran 90 Standard but others were not (although the functionality is nearly always there). Examples include the `%val` construct, VAX data structures, and type statements of the form `INTEGER*2`. Many vendors do, in fact, still support such features in their Fortran 90 compilers, but continued use of them is inadvisable.

- Fortran 90 compilers are relatively new products and more complex, so some problems are to be expected. Although there were many reports of bugs in new compilers soon after their release, several of them have now been on the market for a few years and they seem to be stable and reliable.

- In principle Fortran 90 code, able to take advantage of whole-array operations etc., should run faster, but in practice most compilers produce code which executes barely faster than before, at least on single-processor systems. Vendors are likely to produce better optimisation in due course.

- Fortran 90 compilers are mostly more expensive than those for Fortran 77, and there are as yet no free ones (but see section 11.1 below). Academic users of Digital Equipment systems with a DEC-campus licence can, of course, use their Fortran 90 compiler without extra charge. GNU's g77 compiler includes many Fortran 90 features, but by no means all.

- It will be natural to write new software using the free-format coding style, but the choice is less clear-cut for those making minor changes to existing programs, since the whole program unit has to be in either fixed or free format. Programs to convert from fixed to free-format style are freely available on the Internet, but they make only the essential changes.

- Some systems will allow existing object libraries, compiled using Fortran 77, to be linked with Fortran 90 code, but others require a complete re-compilation of the code.

- In Fortran 90 each module needs to be compiled before any other program units which use it. This imposes additional constraints on code management, and slows down compilation at least initially. If a modification is made to a module it usually require the recompilation of that module and all other modules and program units which use it.

None of these problems, however, seems sufficiently serious as to constitute a serious obstacle to the widespread use of Fortran 90.

11 Fortran Evolution

11.1 Free Fortran 90 subsets

F and ELF90 are two different products from well-known stables, each designed as a subset of Fortran 90 with all the obsolete features removed. These are designed for teaching Fortran, or for compiling new code, but will not cope with most old-style Fortran 77 code. ELF90 comes from Lahey Computer Systems Inc. A slightly different subset called `F` is marketed by Imagine1 of Albuquerque and is based on technology from Salford Software and NAG Ltd. A free version Lahey's ELF90 compiler is available for MS-DOS/Windows, while F is free for use on PCs running Linux. Textbooks based on these are already on the market.

11.2 Fortran 95

Fortran continues to evolve, and Fortran 95 is now defined and likely to become an approved standard within a few months. It clears up a few minor errors and ambiguities in the definition of Fortran 90, but adds only few new features, mainly those to support High Performance Fortran (HPF): a set of extensions to Fortran 90 designed for highly parallel architectures. The most important of these are:

- Data structure definitions can now include default initial values.
- The `FORALL` statement (and block construct) supports parallel execution of loops.
- `PURE` procedures (with no side-effects) can be defined to help optimisation.
- `ELEMENTAL` procedures work element-wise on arrays.
- Non-global allocatable arrays are automatically deallocated on procedure exit.

Fortran 95 also removes from the official Standard a few obsolete features including computed `GOTO`, `DO` statements with control variables of type *real* or *double precision*; `PAUSE`, `ASSIGN`, and assigned `GO TO` statements; the `nH` format descriptor; and branching to an `END IF` statement from outside the block (allowed by mistake in earlier standards). In practice Fortran 95 compilers are likely to keep these, perhaps just issuing a warning messages. Because of the modest nature of these improvements Fortran 90 compilers are likely to appear very soon. Meanwhile work on Fortran2000 is already well under way!