united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

SMR/1310 - 3

SPRING COLLEGE ON
NUMERICAL METHODS IN ELECTRONIC STRUCTURE THEORY

(7 - 25 May 2001)

"Advanced programming techniques"

presented by:

C. CAVAZZONI

CINECA Consorzio Interuniversitario
Casalecchio di Reno, Bologna
Italy

Michael Metcalf's Fortran 90 CNL Articles

Fuller details of all these items can be found in Fortran 90/95 Explained, M. Metcalf and J. Reid, (Oxford, 1996), the book upon which these tutorials are based.

Version August 1995

M.G. (October 19th 1995)

http://wwwinfo.cern.ch/asdoc/f90.html

New specifiers

On the OPEN and INQUIRE statements there are new specifiers:

```
POSITION=   'ASIS'        'REWIND'    'APPEND'
ACTION  =   'READ'        'WRITE'     'READWRITE'
DELIM   =   'APOSTROPHE'  'QUOTE'     'NONE'
PAD     =   'YES'         'NO'
```

and on the INQUIRE there are also

```
READ      = }
WRITE     = }'YES'        'NO'        'UNKNOWN'
READWRITE= }
```

Finally, inquiry by I/O list (unformatted only) is possible:

```
INQUIRE (IOLENGTH = length) item1, item2,...
```

and this is useful to set RECL, or to check that a list is not too long. It is in the same processor-dependent units and thus is a portability aid.

Input-output

by Michael Metcalf / CERN CN-AS

Non-advancing I/O

Normally, records of external, formatted files are positioned at their ends after a read or write operation. This can now be overridden with the additional specifiers:

```
          ADVANCE = 'NO'           (default is 'YES')
          EOR = eor_label          (optional, READ only)
          SIZE = size              (optional, READ only)
```

An example shows how to read a record three characters at a time, and to take action if there are fewer than three left in the record:

```
      CHARACTER(3) key
      INTEGER unit, size
      READ (unit, '(A3)', ADVANCE='NO', SIZE=size, EOR=66) key
      :
! key is not in one record
  66 key(size+1:) = ''
      :
```

and how to keep the cursor positioned after a prompt:

```
      WRITE (*, *, ADVANCE='NO') 'Enter next prime number:'
      READ  (*, '(I10)')          prime_number
```

New edit descriptors

The first three new edit descriptors are modelled on the I edit descriptor:

- B - binary,
- O - octal,
- Z - hexadecimal.

There are two new descriptors for real numbers:

- EN - engineering, multiple-of-three exponent:

```
          0.0217        -->    21.70E-03        (EN9.2)
```

- ES - scientific, leading nonzero digit:

```
          0.0217        -->    2.17E-02         (ES9.2)
```

and the G edit descriptor is generalized to all intrinsic types (E/F, I, L, A).

For entities of derived types, the programmer must elaborate a format for the ultimate components:

```
      TYPE string
         INTEGER length
         CHARACTER(LEN=20) word
      END TYPE string
      TYPE(string) :: text
      READ(*, '(I2, A)') text
```

Intrinsic procedures

by Michael Metcalf / CERN CN-AS

We have already met most of the new intrinsic functions before. Here, we deal only with their general classification and with those that have so far been omitted.

All intrinsic procedures have keyword arguments:

```
          CALL DATE_AND_TIME (TIME=t)
```

and many have optional arguments. They are grouped into four categories:

1. elemental - work on scalars or arrays, e.g. ABS(a);
2. inquiry - independent of value of argument (which maybe undefined), e.g. PRECISION(a);
3. transformational - array argument with array result of different shape, e.g. RESHAPE(a, b);
4. subroutines, e.g. SYSTEM_CLOCK.

The procedures not already introduced are::

```
Bit inquiry
      BIT_SIZE            Number of bits in the model

Bit manipulation
      BTEST               Bit testing
      IAND                Logical AND
      IBCLR               Clear bit
      IBITS               Bit extraction
      IBSET               Set bit
      IEOR                Exclusive OR
      IOR                 Inclusive OR
      ISHFT               Logical shift
      ISHFTC              Circular shift
      NOT                 Logical complement

Transfer function, as in
            INTEGER :: i = TRANSFER('abcd', 0)
                              (replaces part of EQUIVALENCE)

Subroutines
      DATE_AND_TIME       Obtain date and/or time
      MVBITS              Copies bits
      RANDOM_NUMBER       Returns pseudorandom numbers
      RANDOM_SEED         Access to seed
      SYSTEM_CLOCK        Access to system clock
```

```
SUBROUTINE s(b, m, c)
   USE mod                                 ! contains a
   REAL, DIMENSION(:, :)              :: b
   REAL, DIMENSION(UBOUND(b, 1) + 5) :: x
   INTEGER                              m
   CHARACTER(LEN=*)                     c
   CHARACTER(LEN= m + LEN(c))           cc
   REAL (SELECTED_REAL_KIND(2*PRECISION(a))) z
```

PUBLIC and PRIVATE

These attributes are used in specifications in modules to limit the scope of entities. The attribute form is

```
REAL, PUBLIC     :: x, y, z            ! default
INTEGER, PRIVATE :: u, v, w
```

and the statement form is

```
PUBLIC  :: x, y, z, OPERATOR(.add.)
PRIVATE :: u, v, w, ASSIGNMENT(=), OPERATOR(*)
```

The statement form has to be used to limit access to operators, and can also be used to change the overall default:

```
PRIVATE                           ! sets default for module
PUBLIC  :: only_this
```

For derived types there are three possibilities: the type and its components are all PUBLIC, the type is PUBLIC and its components PRIVATE (the type only is visible and one can change its details easily), or all of it is PRIVATE (for internal use in the module only):

```
MODULE mine
   PRIVATE
   TYPE, PUBLIC :: list
      REAL x, y
      TYPE(list), POINTER :: next
   END TYPE list
   TYPE(list) :: tree
   :
END MODULE mine
```

USE statement

To gain access to entities in a module, we use the USE statement. It has options to resolve name clashes if an imported name is the same as a local one:

```
USE mine, local_list => list
```

or to restrict the used entities to a specified set:

```
USE mine, ONLY : list
```

These may be combined:

```
USE mine, ONLY : local_list => list
```

M.G. (October 19th 1995)

Specification Statements

by Michael Metcalf / CERN CN-AS

Implicit typing

The imlicit typing rules of FORTRAN 77 still hold. However, it is good practice to explicitly type all variables, and this can be forced by inserting the statement

```
        IMPLICIT NONE
```

at the beginning of each prorgam unit.

PARAMETER attribute

A named constant can be specified directly by adding the PARAMETER attribute and the constant values to a type statement:

```
     REAL, DIMENSION(3), PARAMETER :: field = (/ 0., 1., 2. /)
     TYPE(triplet), PARAMETER :: t =                           &
                                  triplet( 0., (/ 0., 0., 0. /) )
```

DATA statement

The DATA statement can be used also for arrays and variables of derived type. It is also the only way to initialise just parts of such objects, as well as to initialise to binary, octal or hexadecimal values:

```
       TYPE(triplet) :: t1, t2
       DATA t1/triplet( 0., (/ 0., 1., 2. /) )/, t2%u/0./
       DATA array(1:64) / 64*0/
       DATA i, j, k/ B'01010101', O'77', Z'ff'/
```

Characters

There are many variations on the way character arrays may be specified. The shortest and longest are

```
CHARACTER name(4, 5)*20
CHARACTER (KIND = kanji, LEN = 20), DIMENSION (4, 5) :: name
```

Initialization expressions

The values used in DATA and PARAMETER statements, or with these attributes, are constant expressions that may include references to: array and structure constructors, elemental intrinsic functions with integer or character arguments and results, and the six transformational functions REPEAT, SELECTED_INT_KIND, TRIM, SELECTED_REAL_KIND, RESHAPE and TRANSFER:

```
     INTEGER, PARAMETER :: long = SELECTED_REAL_KIND(12),    &
                        array(3) = (/ 1, 2, 3 /)
```

Specification expressions

It is possible to specify details of variables using any non-constant, scalar, integer expression that may also include inquiry function references:

The source code of an extended example of the use of pointers to support a data structure is in appxg.f90 Aug 12th 1994 m.g.

```
rows%next              ! illegal
```

would be such an object, but with an irregular storage pattern. For this reason they are not allowed. However, we can achieve the same effect by defining a derived data type with a pointer as its sole component:

```
TYPE row
   REAL, POINTER :: r(:)
END TYPE
```

and then defining arrays of this data type:

```
TYPE(row) :: s(n), t(n)
```

where the storage for the rows can be allocated by, for instance,

```
DO i = 1, n
   ALLOCATE (t(i)%r(1:i)) ! Allocate row i of length i
END DO
```

The array assignment

```
s = t
```

is then equivalent to the pointer assignments

```
s(i)%r => t(i)%r
```

for all components.

Pointers as dynamic aliases

Given an array

```
REAL, TARGET :: table(100,100)
```

that is frequently referenced with the fixed subscripts

```
table(m:n, p:q)
```

these references may be replaced by

```
REAL, DIMENSION(:, :), POINTER :: window
   :
window => table(m:n, p:q)
```

The subscripts of window are 1:n-m+1, 1:q-p+1. Similarly, for

```
      tar%u
```

(as defined in Part 7), we can use, say,

```
      taru => tar%u
```

to point at all the u components of tar, and subscript it as

```
      taru(1, 2)
```

The subscripts are as those of tar itself. (This replaces yet more of EQUIVALENCE.)

causes current to overwrite first and is equivalent to

```
    first%value = current%value
    first%index = current%index
    first%next => current%next
```

Pointer arguments

If an actual argument is a pointer then, if the dummy argument is also a pointer,

- it must have same rank,
- it receives its association status from the actual argument,
- it returns its final association status to the actual argument (note: the target may be undefined!),
- it may not have the INTENT attribute (it would be ambiguous),
- it requires an interface block.

If the dummy argument is not a pointer, it becomes associated with the target of the actual argument:

```
   REAL, POINTER :: a (:,:)
      :
   ALLOCATE (a(80, 80))
      :
   CALL sub(a)
      :
SUBROUTINE sub(c)
   REAL c(:, :)
```

Pointer functions

Function results may also have the POINTER attribute; this is useful if the result size depends on calculations performed in the function, as in

```
   USE data_handler
   REAL x(100)
   REAL, POINTER :: y(:)
   :
   y => compact(x)
```

where the module data_handler contains

```
   FUNCTION compact(x)
      REAL, POINTER :: compact(:)
      REAL x(:)
! A procedure to remove duplicates from the array x
      INTEGER n
      :              ! Find the number of distinct values, n
      ALLOCATE(compact(n))
      :              ! Copy the distinct values into compact
   END FUNCTION compact
```

The result can be used in an expression (but must be associated with a defined target).

Arrays of pointers

These do not exist as such: given

```
   TYPE(entry) :: rows(n)
```

then

```
chain%index            chain%next%index
chain%next             chain%next%next
```

but we would normally define additional pointers to point at, for instance, the first and current entries in the list.

Association

A pointer's association status is one of

- undefined (initial state);
- associated (after allocation or a pointer assignment);
- disassociated:

```
DEALLOCATE (p, q)  ! for returning storage
NULLIFY (p, q)     ! for setting to 'null'
```

Some care has to be taken not to leave a pointer 'dangling' by use of DEALLOCATE on its target without NULLIFYing any other pointer referring to it.

The intrinsic function ASSOCIATED can test the association status of a defined pointer:

```
IF (ASSOCIATED(pointer)) THEN
```

or between a defined pointer and a defined target (which may, itself, be a pointer):

```
IF (ASSOCIATED(pointer, target)) THEN
```

Pointers in expressions and assignments

For intrinsic types we can 'sweep' pointers over different sets of target data using the same code without any data movement. Given the matrix manipulation y = B C z, we can write the following code (although, in this case, the same result could be achieved more simply by other means):

```
REAL, TARGET  :: b(10,10), c(10,10), r(10), s(10, z(10)
REAL, POINTER :: a(:,:), x(:), y(:)
INTEGER mult
:
DO mult = 1, 2
   IF (mult == 1) THEN
      y => r              ! no data movement
      a => c
      x => z
   ELSE
      y => s              ! no data movement
      a => b
      x => r
   END IF
   y = MATMUL(a, x)       ! common calculation
END DO
```

For objects of derived type we have to distinguish between pointer and normal assignment. In

```
TYPE(entry), POINTER :: first, current
:
first => current
```

the assignment causes first to point at current, whereas

```
first =  current
```

Pointers

by Michael Metcalf / CERN CN-AS

Basics

Pointers are variables with the POINTER attribute; they are not a distinct data type (and so no 'pointer arithmetic' is possible).

```
REAL, POINTER :: var
```

They are conceptually a descriptor listing the attributes of the objects (targets) that the pointer may point to, and the address, if any, of a target. They have no associated storage until it is allocated or otherwise associated (by pointer assignment, see below):

```
ALLOCATE (var)
```

and they are dereferenced automatically, so no special symbol required. In

```
var = var + 2.3
```

the value of the target of var is used and modified. Pointers cannot be transferred via I/O. The statement

```
WRITE *, var
```

writes the value of the target of var and not the pointer descriptor itself.

A pointer can point to other pointers, and hence to their targets, or to a static object that has the TARGET attribute:

```
REAL, POINTER :: object
REAL, TARGET  :: target_obj
var => object                   ! pointer assignment
var => target_obj
```

but they are strongly typed:

```
INTEGER, POINTER :: int_var
var => int_var                  ! illegal - types must match
```

and, similarly, for arrays the ranks as well as the type must agree.

A pointer can be a component of a derived type:

```
TYPE entry                         ! type for sparse matrix
   REAL value
   INTEGER index
   TYPE(entry), POINTER :: next    ! note recursion
END TYPE entry
```

and we can define the beginning of a linked chain of such entries:

```
TYPE(entry), POINTER :: chain
```

After suitable allocations and definitions, the first two entries could be addressed as

```
chain%value          chain%next%value
```

Array reshape

RESHAPE	Reshape an array

Array manipulation

CSHIFT	Circular shift
EOSHIFT	End-off shift
TRANSPOSE	Transpose of an array of rank two

Array location

MAXLOC	Location of first maximum value in an array
MINLOC	Location of first minimum value in an array

M.G. (October 19th 1995)

```
REAL a(10, 10)
a(i, 1:n)                ! part of one row
a(1:m, j)                ! part of one column
a(i, : )                 ! whole row
a(i, 1:n:3)              ! every third element of row
a(i, 10:1:-1)            ! row in reverse order
a( (/ 1, 7, 3, 2 /), 1)  ! vector subscript
a(1, 2:11:2)             ! 11 is legal as not referenced
a(:, 1:7)                ! rank two section
```

Note that a vector subscript with duplicate values cannot appear on the left-hand side of an assignment as it would be ambiguous. Thus,

```
b( (/ 1, 7, 3, 7 /) ) = (/ 1, 2, 3, 4 /)
```

is illegal. Also, a section with a vector subscript must not be supplied as an actual argument to an OUT or INOUT dummy argument. Arrays of arrays are not allowed:

```
tar%du             ! illegal
```

We note that a given value in an array can be referenced both as an element and as a section:

```
a(1, 1)            !  scalar (rank zero)
a(1:1, 1)          !  array section (rank one)
```

depending on the circumstances or requirements. By qualifying objects of derived type, we obtain elements or sections depending on the rule stated earlier:

```
tar%u              !  array section (structure component)
tar(1, 1)%u        !  component of an array element
```

Arrays intrinsic functions

Vector and matrix multiply

DOT_PRODUCT	Dot product of 2 rank-one arrays
MATMUL	Matrix multiplication

Array reduction

ALL	True if all values are true
ANY	True if any value is true. Example: IF (ANY(a > b)) THEN
COUNT	Number of true elements in array
MAXVAL	Maximum value in an array
MINVAL	Minimum value in an array
PRODUCT	Product of array elements
SUM	Sum of array elements

Array inquiry

ALLOCATED	Array allocation status
LBOUND	Lower dimension bounds of an array
SHAPE	Shape of an array (or scalar)
SIZE	Total number of elements in an array
UBOUND	Upper dimension bounds of an array

Array construction

MERGE	Merge under mask
PACK	Pack an array into an array of rank
SPREAD	Replicate array by adding a dimension
UNPACK	Unpack an array of rank one into an array under mask

Often, we need to mask an assignment. This we can do using the WHERE, either as a statement:

```
WHERE (a /= 0.0) a = 1.0/a  ! avoid division by 0
```

(note: test is element-by-element, not on whole array), or as a construct:

```
WHERE (a /= 0.0)
   a = 1.0/a
   b = a             ! all arrays same shape
END WHERE
```

or

```
WHERE (a /= 0.0)
   a = 1.0/a
ELSEWHERE
   a = HUGE(a)
END WHERE
```

Array elements

Simple case: given

```
    REAL, DIMENSION(100, 100) :: a
```

we can reference a single element as, for instance, a(1, 1). For a derived-data type like

```
TYPE triplet
   REAL                 u
   REAL, DIMENSION(3) :: du
END TYPE triplet
```

we can declare an array of that type:

```
TYPE(triplet), DIMENSION(10, 20) :: tar
```

and a reference like

```
                    tar(n, 2)
```

is an element (a scalar!) of type triplet, but

```
                    tar(n, 2)%du
```

is an array of type real, and

```
                    tar(n, 2)%du(2)
```

is an element of it. The basic rule to remember is that an array element always has a subscript or subscripts qualifying at least the last name.

Array subobjects (sections)

The general form of subscript for an array section is

```
  [lower] : [upper] [:stride]
```

as in

```
SUBROUTINE swap(a, b)
   REAL, DIMENSION(:)       :: a, b
   REAL, DIMENSION(SIZE(a)) :: work
   work = a
   a = b
   b = work
END SUBROUTINE swap
```

The actual storage is maintained on a stack.

ALLOCATABLE and ALLOCATE

Fortran 90 provides dynamic allocation of storage; it relies on a heap storage mechanism (and replaces another use of EQUIVALENCE). An example, for establishing a work array for a whole program, is

```
MODULE work_array
   INTEGER n
   REAL, DIMENSION(:,:,:), ALLOCATABLE :: work
END MODULE
PROGRAM main
   USE work_array
   READ (input, *) n
   ALLOCATE(work(n, 2*n, 3*n), STAT=status)
   :
   DEALLOCATE (work)
```

The work array can be propagated through the whole program via a USE statement in each program unit. We may specify an explicit lower bound and allocate several entities in one statement. To free dead storage we write, for instance,

```
DEALLOCATE(a, b)
```

We will meet this later, in the context of pointers.

Elemental operations and assignments

We have already met whole array assignments and operations:

```
REAL, DIMENSION(10) :: a, b
a = 0.          ! scalar broadcast; elemental assignment
b = sqrt(a)     ! intrinsic function result as array object
```

In the second assignment, an intrinsic function returns an array-valued result for an array-valued argument. We can write array-valued functions ourselves (they require an explicit interface):

```
PROGRAM test
   REAL, DIMENSION(3) :: a = (/ 1., 2., 3./),       &
                         b = (/ 2., 2., 2. /),  r
   r = f(a, b)
   PRINT *, r
CONTAINS
   FUNCTION f(c, d)
   REAL, DIMENSION(:) :: c, d
   REAL, DIMENSION(SIZE(c)) :: f
   f = c*d        ! (or some more useful functio of c and d)
   END FUNCTION f
END PROGRAM test
```

WHERE

Array Handling

by Michael Metcalf / CERN CN-AS

Array handling is included in Fortran 90 for two main reasons:

- the notational convenience it provides, bringing he code closer to the underlying mathematical form;
- for the additional optimization opportunities it gives compilers (although there are plenty of opportunities for degrading optimization too!).

At the same time, major extensions of the functionality in this area have been added. We have already met whole arrays in Parts 2 and 3 of this series - here we develop the theme.

Zero-sized arrays

A zero-sized array is handled by Fortran 90 as a legitimate object, without special coding by the programmer. Thus, in

```
DO i = 1,n
   x(i) = b(i) / a(i, i)
   b(i+1:n) = b(i+1:n) - a(i+1:n, i) * x(i)
END DO
```

no special code is required for the final iteration where i = n. We note that a zero-sized array is regarded as being defined; however, an array of shape (0,2) is not conformable with one of shape (0,3), whereas

```
x(1:0) = 3
```

is a valid 'do nothing' statement.

Assumed-shape arrays

These are an extension and replacement for assumed-size arrays. Given an actual argument like:

```
REAL, DIMENSION(0:10, 0:20) :: a
   :
CALL sub(a)
```

the corresponding dummy argument specification defines only the type and rank of the array, not its size. This information has to be made available by an explicit interface, often using an interface block (see *Arguments, interface blocks and recursion*). Thus we write just

```
SUBROUTINE sub(da)
   REAL, DIMENSION(:, :) :: da
```

and this is as if da were dimensioned (11,21). However, we can specify any lower bound and the array maps accordingly. The shape, not bounds, is passed, where the default lower bound is 1 and the default upper bound is the corresponding extent.

Automatic arrays

A partial replacement for the uses to which EQUIVALENCE is put is provided by this facility, useful for local, temporary arrays, as in

```
FUNCTION fy(y)
   USE func           ! module func contains function f
   REAL fy, y
   yval = y
   fy = integrate(f, xbounds)
END
```

Direct recursion is when a procedure calls itself, as in

```
RECURSIVE FUNCTION factorial(n) RESULT(res)
   INTEGER res, n
   IF(n.EQ.1) THEN
      res = 1
   ELSE
      res = n*factorial(n-1)
   END IF
END
```

Here, we note the RESULT clause and termination test.

M.G. (October 19th 1995)

```
   :
   f = func(x)    ! invocation of the user function.
   :
END FUNCTION minimum
```

An explicit interface is obligatory for:

- optional and keyword arguments;
- POINTER and TARGET arguments (see later article);
- POINTER function result (later);
- new-style array arguments and array functions (later).

It allows full checks at compile time between actual and dummy arguments.

Overloading and generic interfaces

Interface blocks provide the mechanism by which we are able to define generic names for specific procedures:

```
INTERFACE gamma                        ! generic name
   FUNCTION sgamma(X)                  ! specific name
      REAL (SELECTED_REAL_KIND( 6)) sgamma, x
   END
   FUNCTION dgamma(X)                  ! specific name
      REAL (SELECTED_REAL_KIND(12)) dgamma, x
   END
END INTERFACE
```

where a given set of specific names corresponding to a generic name must all be of functions or all of subroutines. If this interface is within a module, then it is simply

```
INTERFACE gamma
   MODULE PROCEDURE sgamma, dgamma
END INTERFACE
```

We can use existing names, e.g. SIN, and the compiler sorts out the correct association.

We have already seen the use of interface blocks for defined operators and assignment (see Part 3).

Recursion

Indirect recursion is useful for multi-dimensional integration. For

```
volume = integrate(fy, ybounds)
```

We might have

```
RECURSIVE FUNCTION integrate(f, bounds)
   ! Integrate f(x) from bounds(1) to bounds(2)
   REAL integrate
   INTERFACE
      FUNCTION f(x)
         REAL f, x
      END FUNCTION f
   END INTERFACE
   REAL, DIMENSION(2), INTENT(IN) :: bounds
   :
END FUNCTION integrate
```

and to integrate f(x, y) over a rectangle:

```
     :
END MODULE interval_arithmetic
```

and the simple statement

```
USE interval_arithmetic
```

provides use association to all the module's entities. Module subprograms may, in turn, contain internal subprograms.

Arguments

We may specify the intent of dummy arguments:

```
SUBROUTINE shuffle (ncards, cards)
   INTEGER, INTENT(IN)   :: ncards
   INTEGER, INTENT(OUT), DIMENSION(ncards) :: cards
```

Also, INOUT is possible: here the actual argument must be a variable (unlike the default case where it may be a constant).

Arguments may be optional:

```
SUBROUTINE mincon(n, f, x, upper, lower, equalities,      &
           inequalities, convex, xstart)
   REAL, OPTIONAL, DIMENSION :: upper, lower
   :
```

allows us to call mincon by

```
   CALL mincon (n, f, x, upper)
   :
   IF (PRESENT(lower)) THEN   ! test for presence of actual argument
   :
```

Arguments may be keyword rather than positional (which come first):

```
   CALL mincon(n, f, x, equalities=0, xstart=x0)
```

Optional and keyword arguments are handled by explicit interfaces, that is with internal or module procedures or with interface blocks.

Interface blocks

Any reference to an internal or module subprogram is through an interface that is 'explicit' (that is, the compiler can see all the details). A reference to an external (or dummy) procedure is usually 'implicit' (the compiler assumes the details). However, we can provide an explicit interface in this case too. It is a copy of the header, specifications and END statement of the procedure concerned, either placed in a module or inserted directly:

```
     REAL FUNCTION minimum(a, b, func)
! returns the minimum value of the function func(x)
! in the interval (a,b)
        REAL, INTENT(in) :: a, b
        INTERFACE
           REAL FUNCTION func(x)
              REAL, INTENT(IN) :: x
           END FUNCTION func
        END INTERFACE
        REAL f,x
```

Program units and procedures

by Michael Metcalf / CERN CN-ASD

In order to discuss this topic we need some definitions. In logical terms, an executable program consists of one main program and zero or more subprograms (or procedures) - these do something. Subprograms are either functions or subroutines, which are either external, internal or module subroutines. (External subroutines are what we know from FORTRAN 77.)

From an organizational point of view, however, a complete program consists of program units. These are either main programs, external subprograms or modules and can be separately compiled.

An internal subprogram is one contained in another (at a maximum of one level of nesting) and provides a replacement for the statement function:

```
SUBROUTINE outer
   REAL x, y
   :
CONTAINS
   SUBROUTINE inner
      REAL y
      y = x + 1.
      :
   END SUBROUTINE inner      ! SUBROUTINE mandatory
END SUBROUTINE outer
```

We say that outer is the host of inner, and that inner obtains access to entities in outer by host association (e.g. to x), whereas y is a local variable to inner.

The scope of a named entity is a scoping unit, here outer less inner, and inner.

The names of program units and external procedures are global, and the names of implied-DO variables have a scope of the statement that contains them.

Modules are used to package

- global data (replaces COMMON and BLOCK DATA);
- type definitions (themselves a scoping unit);
- subprograms (which among other things replaces the use of ENTRY);
- interface blocks (another scoping unit, see next article);
- namelist groups (later in the series).

An example of a module containing a type defition, interface block and function subprogram is:

```
MODULE interval_arithmetic
   TYPE interval
      REAL lower, upper
   END TYPE interval
   INTERFACE OPERATOR(+)
      MODULE PROCEDURE add_intervals
   END INTERFACE
   :
CONTAINS
   FUNCTION add_intervals(a,b)
      TYPE(interval), INTENT(IN) :: a, b
      TYPE(interval) add_intervals
      add_intervals%lower = a%lower + b%lower
      add_intervals%upper = a%upper + b%upper
   END FUNCTION add_intervals             ! FUNCTION mandatory
```

Control statements

by Michael Metcalf / CERN CN-AS

The CASE construct is a replcement for the computed GOTO, but is better structured and does not require the use of statement labels:

```
SELECT CASE (number)       ! NUMBER of type integer
CASE (:-1)                 ! all values below 0
   n_sign = -1
CASE (0)                   ! only 0
   n_sign = 0
CASE (1:)                  ! all values above 0
   n_sign = 1
END SELECT
```

Each CASE selector list may contain a list and/or range of integers, character or logical constants, whose values may not overlap within or between selectors:

```
CASE (1, 2, 7, 10:17, 23)
```

A default is available:

```
CASE DEFAULT
```

There is only one evaluation, and only one match.

A simplified but sufficient form of the DO construct is illustrated by

```
outer: DO
inner:    DO i = j, k, l      ! only integers
             :
             IF (...) CYCLE
             :
             IF (...) EXIT outer
          END DO inner
       END DO outer
```

where we note that loops may be named so that the EXIT and CYCLE statements may specify which loop is meant.

Many, but not all, simple loops can be replaced by array expressions and assignments, or by new intrinsic functions. For instance

```
tot = 0.
DO i = m, n
   tot = tot + a(i)
END DO
```

becomes simply

```
tot = SUM( a(m:n) )
```

vector machines. Of course, any operators for arrays of derived type must be defined.

There are some new real intrinsic functions that are useful for numeric computations:

```
        CEILING         FLOOR         MODULO (also integer)
        EXPONENT        FRACTION
        NEAREST         RRSPACING     SPACING
        SCALE           SET_EXPONENT
```

Like all FORTRAN 77 functions (SIN, ABS, etc.), except LEN, these are array valued for array arguments (i.e. are elemental).

M.G. (October 19th 1995)

association by an interface block, also in the module (we shall return to this later). For the moment, here is an example of an interface for string concatenation

```
INTERFACE OPERATOR(//)
   MODULE PROCEDURE string_concat
END INTERFACE
```

and an example of part of a module containing the definitions of character-to-string and string-to-character assignment. The string concatenation function was shown already in *Language Elements*.

```
MODULE string_type
   TYPE string
      INTEGER length
      CHARACTER(LEN=80)   :: string_data
   END TYPE string
   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE c_to_s_assign, s_to_c_assign
   END INTERFACE
   INTERFACE OPERATOR(//)
      MODULE PROCEDURE string_concat
   END INTERFACE
CONTAINS
   SUBROUTINE c_to_s_assign(s, c)
      TYPE (string), INTENT(OUT)    :: s
      CHARACTER(LEN=*), INTENT(IN)  :: c
      s%string_data = c
      s%length = LEN(c)
   END SUBROUTINE c_to_s_assign
   SUBROUTINE s_to_c_assign(c, s)
      TYPE (string), INTENT(IN)     :: s
      CHARACTER(LEN=*), INTENT(OUT) :: c
      c = s%string_data(1:s%length)
   END SUBROUTINE s_to_c_assign
   FUNCTION string_concat(s1, s2)
      :
   END FUNCTION string_concat
END MODULE string_type
```

Defined operators such as these are required for the expressions that are allowed too in structure constructors (see Part 1):

```
str1 = string(2, char1//char2)  ! structure constructor
```

So far we have discussed scalar variables. In the case of arrays then, as long as they are of the same shape (conformable), operations and assignments are extended in an obvious way, on an element-by-element basis. For

```
REAL, DIMENSION(10, 20) :: a, b, c
REAL, DIMENSION(5)      :: v, w
LOGICAL                    flag(10, 20)
```

can write

```
a = b                      ! whole array assignment
c = a/b                    ! whole array division and assignment
c = 0.                     ! whole array assignment of scalar value
w = v + 1.                 ! whole array addition to scalar value
w = 5/v + a(1:5, 5)        ! array division, and addition to section
flag = a==b                ! whole array relational test and assignment
c(1:8, 5:10) = a(2:9, 5:10) + b(1:8, 15:20)
                           ! array section addition and assignment
v(2:5) = v(1:4)            ! overlapping section assignment
```

The order of expression evaluation is not specified in order to allow for optimization on parallel and

Expressions and assignments

by Michael Metcalf / CERN CN-AS

The rules for scalar numeric expresions and assignments, as known from FORTRAN 77, are extended to accommodate the non-default kinds we encountered in Part 1. Thus, the mixed-mode numeric expression and assignment rules incorporate different kind type parameters in an expected way:

```
real2 = integer + real1
```

converts integer to a real value of the same kind as real1; the result is of same kind, and is converted to the kind of real2 for assignment.

For scalar relational operations, there is a set of new, alternative operators:

```
<   <=   ==   /=   >   >=
```

so we can write expressions such as

```
IF (a < b .AND. i /= j) THEN ! for numeric variables
flag = a == b                  ! for logical variable flags
```

In the case of scalar characters, two old restrictions are lifted. Given

```
CHARACTER(8) result
```

it is now legal to write

```
result(3:5) = result(1:3)    ! overlap allowed
result(3:3) = result(3:2)    ! no assignment of null string
```

For an operation between derived-data types, or between a derived type and an intrinsic type, we must define the meaning of the operator. (Between intrinsic types, there are intrinsic operations only.) Given

```
TYPE string
   INTEGER       length
   CHARACTER(80) value
END TYPE string
CHARACTER    char1, char2, char3
TYPE(string) str1,  str2,  str3
```

we can write

```
str3  = str1//str2         ! must define operation
str3  = str1.concat.str2 ! must dedine operation
char3 = char2//char3       ! intrinsic operator only
str3  = char1              ! must define assignment
```

For the first two cases, assignment applies on a component-by-component basis (but can be overridden), and they also require us to define the exact meaning of the // symbol. We see here the use of an intrinsic symbol and of a named operator, `.concat.` . A difference is that, for an intrinsic operator token, the usual precedence rules apply, whereas for named operators their precedence is the highest as a unary operator or the lowest as a binary one. In

```
vector3 = matrix    *    vector1  + vector2
vector3 =(matrix .times. vector1) + vector2
```

the two expressions are equivalent only if appropriate parentheses are added as shown. In each case, we have to provide, in a module, procedures defining the operator and assignment, and make the making use of the implied-DO loop notation familiar from I/O lists. A derived data type may, of course, contain array components:

```
TYPE triplet
   REAL, DIMENSION(3) :: vertex
END TYPE triplet
TYPE(triplet), DIMENSION(4) :: t
```

so that

```
t(2)           is a scalar (a structure)
t(2)%vertex    is an array component of a scalar
```

There are some other interesting character extensions. Just as a substring as in

```
CHARACTER(80), DIMENSION(60) :: page
... = page(j)(i:i)          ! substring
```

was already possible, so now are the substrings

```
'0123456789'(i:i)
you%name(1:2)
```

Also, zero-length strings are allowed:

```
page(j)(i:i-1)       ! zero-length string
```

Finally, there are some new intrinsic character functions:

```
ACHAR                IACHAR  (for ASCII set)
ADJUSTL              ADJUSTR
LEN_TRIM             INDEX(s1, s2, BACK=.TRUE.)
REPEAT               SCAN  (for one of a set)
TRIM                 VERIFY(for all of a set)
```

M.G. (October 19th 1995)

and then create structures of that type:

```
TYPE(person) you, me
```

To select components of a derived type, we use the % qualifier:

```
you%age
```

and the form of a literal constant of a derived type is shown by:

```
you = person('Smith', 23.5)
```

which is known as a structure constructor. Definitions may refer to a previously defined type:

```
TYPE point
   REAL x, y
END TYPE point
TYPE triangle
   TYPE(point) a, b, c
END TYPE triangle
```

and for a variable of type triangle, as in

```
TYPE(triangle) t
```

we have components of type point:

```
t%a   t%b   t%c
```

which, in turn, have ultimate components of type real:

```
t%a%x   t%a%y   t%b%x   etc.
```

We note that the % qualifier was chosen rather than . because of ambiguity difficulties.

Arrays are considered to be variables in their own right. Given

```
REAL a(10)
INTEGER, DIMENSION(0:100, -50:50) :: map
```

(the latter an example of the syntax that allows grouping of attributes to the left of :: and of variables sharing the attributes to the right), we have two arrays whose elements are in array element order (column major), but not necessarily in contiguous storage. Elements are, for example,

```
a(1)                a(i*j)
```

and are scalars. The subscripts may be any scalar integer expression. Sections are

```
a(i:j)                ! rank one
map(i:j, k:l:m)       ! rank two
a(map(i, k:l))        ! vector subscript
a(3:2)                ! zero length
```

Whole arrays and array sections are array-valued objects. Array-valued constants (constructors) are available:

```
(/ 1, 2, 3, 4, 5 /)
(/ (i, i = 1, 9, 2) /)
(/ ( (/ 1, 2, 3 /), i = 1, 10) /)
(/ (0, i = 1, 100) /)
(/ (0.1*i, i = 1, 10) /)
```

CHARACTER

```
'A string'   "Another"   'A "quote"'   ''
```

(the last being a null string). Other kinds are allowed, especially for support of non-European languages:

```
2_'   '
```

and again the kind value is given by the KIND function:

```
KIND('ASCII')
```

LOGICAL

Here, there may also be different kinds (to allow for packing into bits):

```
.FALSE.   .true._one_bit
```

and the KIND function operates as expected:

```
KIND(.TRUE.)
```

The numeric types are based on model numbers with associated inquiry functions (whose values are independent of the values of their arguments):

`DIGITS(X)`	Number of significant digits
`EPSILON(X)`	Almost negligible compared to one (real)
`HUGE(X)`	Largest number
`MAXEXPONENT(X)`	Maximum model exponent (real)
`MINEXPONENT(X)`	Minimum model exponent (real)
`PRECISION(X)`	Decimal precision (real, complex)
`RADIX(X)`	Base of the model
`RANGE(X)`	Decimal exponent range
`TINY(X)`	Smallest postive number (real)

These functions are important for portable numerical software.

We can specify scalar variables corresponding to the five intrinsic types:

```
INTEGER(KIND=2) i
REAL(KIND=long) a
COMPLEX         current
LOGICAL         Pravda
CHARACTER(LEN=20) word
CHARACTER(LEN=2, KIND=Kanji) kanji_word
```

where the optional KIND parameter specifies a non-default kind, and the LEN= specifier replaces the *len form. The explicit KIND and LEN specifiers are optional:

```
CHARACTER(2, Kanji) kanji_word
```

works just as well.

For derived-data types we must first define the form of the type:

```
TYPE person
   CHARACTER(10) name
   REAL          age
END TYPE person
```

INTEGER

```
1    0    -999    32767    +10
```

for the default kind; but we may also define, for instance for a desired range of -10**4 to +10**4, a named constant, say two_bytes:

```
INTEGER, PARAMETER :: two_bytes = SELECTED_INT_KIND(4)
```

that allows us to define constants of the form

```
-1234_two_bytes    +1_two_bytes
```

Here, two_bytes is the kind type parameter; it can also be a default integer literal constant, like

```
-1234_2
```

but use of an explicit literal constant would be non-portable.

The KIND function supplies the value of a kind type parameter:

```
KIND(1)             KIND(1_two_bytes)
```

and the RANGE function supplies the actual decimal range (so the user must make the actual mapping to bytes):

```
RANGE(1_two_bytes)
```

Also, in DATA statements, binary, octal and hexcadecimal constants may be used:

```
B'01010101'    O'01234567'    Z'10fa'
```

REAL

There are at least two real kinds - the default, and one with greater precision (this replaces DOUBLE PRECISION). We might specify

```
INTEGER, PARAMETER :: long = SELECTED_REAL_KIND(9, 99)
```

for at least 9 decimal digits of precision and a range of 10*(-99) to 10**99, allowing

```
1.7_long
```

Also, we have the intrinsic functions

```
KIND(1.7_long)    PRECISION(1.7_long)    RANGE(1.7_long)
```

that give in turn the kind type value, the actual precision (here at least 9), and the actual range (here at least 99).

COMPLEX

This data type is built of two integer or real components:

```
(1, 3.7_long)
```

The forms of literal constants for the two non-numeric data types are:

Language elements

by Michael Metcalf / CERN CN-ASD

The basic components of the Fortran language are its character set. The members are:

- the letters A ... Z and a ... z (which are equivalent outside a character context);
- the numerals 0 ... 9;
- the underscore _; and
- the special characters

```
= : + blank - * / ( ) , . $ ' (old)
! " % & ; < > ?                (new)
```

From these components, we build the tokens that have a syntactic meaning to the compiler. There are six classes of token:

```
Label:        123                 Constant: 123.456789_long

Keyword:      ALLOCATABLE         Operator: .add.

Name:         solve_equation (up to 31 characters, including _)

Separator:  /   (   )   (/   /)   ,   =   =>   :   ::   ;   %
```

From the tokens, we can build statements. These can be coded using the new free source form which does not require positioning in a rigid column structure:

```
FUNCTION string_concat(s1, s2)                ! This is a comment
   TYPE (string), INTENT(IN) :: s1, s2
   TYPE (string) string_concat
   string_concat%string_data = s1%string_data(1:s1%length) // &
      s2%string_data(1:s2%length)             ! This is a continuation
   string_concat%length = s1%length + s2%length
END FUNCTION string_concat
```

Note the trailing comments and the trailing continuation mark. There may be 39 continuation lines, and 132 characters per line. Blanks are significant. Where a token or character constant is split across two lines:

```
               ...        start_of&
        &_name
               ...   'a very long &
        &string'
```

a leading & on the continued line is also required.

Automatic conversion of source form for existing programs can be carried out by convert.f90. Its options are:

- significant blank handling;
- indentation;
- CONTINUE replaced by END DO;
- name added to subprogram END statement; and
- INTEGER*2 etc. syntax converted.

Fortran has five intrinsic data types. For each there is a corresponding form of literal constant. For the three numeric intrinsic types they are:

USEFUL LINKS

WHERE YOU CAN FIND SOME ON-LINE DOCUMENTATION ON FORTRAN 90

http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html

http://www.cineca.it/manuali/f77to90/

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

APPENDIX

```
FUNCTION IRAND()
  INTEGER MSEED, KONST, RND_MAX
  COMMON/MYRAND/RND_MAX, MSEED, KONST
  MSEED = MSEED * KONST
  MSEED = MSEED - RND_MAX * ( MSEED / RND_MAX )
  IRAND = MSEED
  RETURN
END FUNCTION

SUBROUTINE INITRAND(M)
  INTEGER M, MSEED, KONST, RND_MAX
  COMMON/MYRAND/RND_MAX, MSEED, KONST
  KONST   = 125
  RND_MAX = 2796203
  MSEED   = M
  RETURN
END SUBROUTINE INITRAND

FUNCTION RRAND()
  REAL*8 RRAND
  INTEGER MSEED, KONST, RND_MAX, IRAND
  COMMON/MYRAND/RND_MAX, MSEED, KONST
  RRAND = DBLE(IRAND()) / DBLE(RND_MAX)
  RETURN
END FUNCTION

FUNCTION RANDEXP(A)
  REAL*8 RANDEXP
  REAL*8 A, X
  INTEGER MSEED, KONST, RND_MAX
  COMMON/MYRAND/RND_MAX, MSEED, KONST
  INTEGER IRAND
  X = DBLE(RND_MAX - IRAND()) / DBLE(RND_MAX)
  RANDEXP = - LOG( X ) / A
  RETURN
END FUNCTION

PROGRAM MYRAND_MAIN
  REAL*8 RANDEXP, A, X
  INTEGER I
  CALL INITRAND(100001)
  A = 2.0D0
  DO I=1,100
    X = RANDEXP(A)
    PRINT *,X
  END DO
END PROGRAM
```

```
MODULE MYRAND

  IMPLICIT NONE
  PRIVATE
  SAVE
  PUBLIC :: INITRAND, RANDEXP, IRAND, RRAND
  INTEGER :: MSEED
  INTEGER, PARAMETER :: KONST   = 125
  INTEGER, PARAMETER :: RND_MAX = 2796203

CONTAINS

  FUNCTION IRAND()
    INTEGER :: IRAND
      MSEED = MSEED * KONST
      MSEED = MSEED - RND_MAX * ( MSEED / RND_MAX )
      IRAND = MSEED
    RETURN
  END FUNCTION

  SUBROUTINE INITRAND(M)
    INTEGER, INTENT(IN) :: M
      MSEED = M
    RETURN
  END SUBROUTINE INITRAND

  FUNCTION RRAND()
    REAL*8 :: RRAND
      RRAND = DBLE(IRAND()) / DBLE(RND_MAX)
    RETURN
  END FUNCTION

  FUNCTION RANDEXP(A)
    REAL*8, INTENT(IN) :: A
    REAL*8 :: X, RANDEXP
      X = DBLE(RND_MAX - IRAND()) / DBLE(RND_MAX)
      RANDEXP = - LOG( X ) / A
    RETURN
  END FUNCTION

END MODULE MYRAND

PROGRAM MYRAND_MAIN
  USE MYRAND, ONLY: INITRAND, RANDEXP
  REAL*8 A, X
  INTEGER I
  CALL INITRAND(100001)
  A = 2.0D0
  DO I = 1, 100
    X = RANDEXP(A)
    PRINT *,X
  END DO
END PROGRAM
```

```
program main
  real pi, one, two, half, sqrt2, sqrt3
  common /constants/ pi, one, two, half, sqrt2, sqrt3
  call setconst()
  call shwconst()
end

subroutine setconst
  real pi, one, two, half, sqrt2, sqrt3
  common /constants/ pi, one, two, half, sqrt2, sqrt3
  pi    = 3.1415
  one   = 1.0
  two   = 2.0
  half  = 0.5
  sqrt2 = sqrt( 2.0 )
  sqrt3 = sqrt( 3.0 )
end

subroutine shwconst
  real pi, one, two, half, sqrt2, sqrt3
  common /constants/ pi, one, two, half, sqrt2, sqrt3
  write(6,*) '* defined constants *'
  write(6,*) '  pi    = ', pi
  write(6,*) '  one   = ', one
  write(6,*) '  two   = ', two
  write(6,*) '  half  = ', half
  write(6,*) '  sqrt2 = ', sqrt2
  write(6,*) '  sqrt3 = ', sqrt3
end
```

```
module constants
  implicit none
  save
    real, parameter :: pi   = 3.1415
    real, parameter :: one  = 1.0
    real, parameter :: two  = 2.0
    real, parameter :: half = 0.5
    real            :: sqrt2
    real            :: sqrt3
 contains
  subroutine set_constants
    sqrt2 = sqrt( 2.0 )
    sqrt3 = sqrt( 3.0 )
  end subroutine set_constants
  subroutine show_constants
    write(6,*) '* defined constants *'
    write(6,*) '  pi    = ', pi
    write(6,*) '  one   = ', one
    write(6,*) '  two   = ', two
    write(6,*) '  half  = ', half
    write(6,*) '  sqrt2 = ', sqrt2
    write(6,*) '  sqrt3 = ', sqrt3
  end subroutine
end module constants

program main
  use constants, only: set_constants, show_constants
  call set_constants()
  call show_constants()
end program main
```

```
MODULE BLACK_BODY
  REAL*8, PRIVATE :: INTERNAL_TEMPERATURE = 0.0D0
CONTAINS
  SUBROUTINE HEATUP(D)
    REAL*8 D
    INTERNAL_TEMPERATURE = INTERNAL_TEMPERATURE + D
  END SUBROUTINE HEATUP
  SUBROUTINE COOLDOWN(D)
    REAL*8 D
    INTERNAL_TEMPERATURE = INTERNAL_TEMPERATURE - D
  END SUBROUTINE COOLDOWN
  REAL*8 FUNCTION TEMPERATURE()
    TEMPERATURE = INTERNAL_TEMPERATURE
  END FUNCTION TEMPERATURE
END MODULE BLACK_BODY

PROGRAM EXEMPLE1
  USE BLACK_BODY
  REAL*8 DEGREE
  DEGREE = 100.0D0
  CALL HEATUP(DEGREE)
  WRITE(6,FMT="(' BLACK BODY TEMPERATURE = ',F8.3)") TEMPERATURE()
  DEGREE = 50.0D0
  CALL COOLDOWN(DEGREE)
  WRITE(6,10) TEMPERATURE()
END PROGRAM EXEMPLE1
```

```
MODULE VECTOR_SPACE
  INTERFACE NORMA2
    MODULE PROCEDURE NORMA2_I, NORMA2_R
  END INTERFACE
CONTAINS
  INTEGER FUNCTION NORMA2_I(V)
    INTEGER, INTENT(IN) :: V(:)
    INTEGER I
    NORMA2_I = 0
    DO I=1,SIZE(V)
      NORMA2_I = NORMA2_I + V(I)**2
    END DO
  END FUNCTION NORMA2_I
  REAL(KIND=4) FUNCTION NORMA2_R(V)
    REAL(KIND=4), INTENT(IN) :: V(:)
    INTEGER I
    NORMA2_R = 0.0D0
    DO I=1,SIZE(V)
      NORMA2_R = NORMA2_R + V(I)**2
    END DO
  END FUNCTION NORMA2_R
END MODULE VECTOR_SPACE

PROGRAM EXEMPLE2
     USE VECTOR_SPACE
     INTEGER :: IV(3) = (/ 1, 2, 3 /)
     REAL(KIND=4)  :: RV(3) = (/ 1.0D0, 2.0D0, 3.0D0 /)
     INTEGER IN
     REAL(KIND=4)  RN
     IN = NORMA2(IV)
     RN = NORMA2(RV)
     WRITE(6,10) IN
     WRITE(6,11) RN
10   FORMAT('NORM of the INTEGER vector = ',I8)
11   FORMAT('NORM of the REAL    vector = ',F8.3)
END PROGRAM EXEMPLE2
module data_types
 type vect
   real x,y,z
 end type vect
end module data_types

module data_functions
   interface operator( + )
     module procedure my_sum
   end interface
 contains
   function my_sum (vect1,vect2) result ( new )
     use data_types
     type (vect), intent(IN) :: vect1
     type (vect), intent(IN) :: vect2
     type (vect), new
     new%x = vect1%x + vect2%x
```

```
     new%y = vect1%y + vect2%y
     new%z = vect1%z + vect2%z
    end function my_sum
end module data_functions
module algebra
  use data_types
  use data_functions
end module algebra

program exemple3
  use algebra
  implicit NONE
  type (vect) vect1,vect2,vect3
  vect1 = vect(1.0d0,2.0d0,3.0d0)
  vect2 = vect(3.0d0,2.0d0,1.0d0)
  vect3 = vect1 + vect2
  print *,'main :',vect3%x, vect3%y, vect3%z
end program
```

```
! In F90 there are different kind of intrinsic data types
! (integer, real, logical, complex). These are useful
! to control the representation of data on differet
! architecturs.

! The value of kind specifier itself is default integer

program kinds
  implicit none
  real    :: a
  integer :: i
  write(6,*) 'the default kind of real    is: ',kind(a)
  write(6,*) 'the default kind of integer is: ',kind(i)
  write(6,*) 'the kind of long integer    is: ', &
    selected_int_kind(9)
    ! integer number in the range -10^9 and 10^9
  write(6,*) 'the kind of double precision real is: ', &
    selected_real_kind(14,300)
    ! at leas 14 digids and exponent range -300 300
  write(6,*) 'the kind of double precision real is: ', &
    kind(1.0d0)
  write(6,*) 'the smallest double precision number is : ', &
    tiny(1.0d0)
  write(6,*) 'the largest  double precision number is : ', &
    huge(1.0d0)
  write(6,*) 'the number of significant digit      is : ', &
    digits(1.0d0)
  write(6,*) 'the maximum exponent                 is : ', &
    maxexponent(1.0d0)
  write(6,*) 'the minimum exponent                 is : ', &
    minexponent(1.0d0)

end program
```

```
! DEFINE YOUR OWN KIND

! The programmer is allowed to define its own standard
! kind for the intrinsic type


program kinds
  implicit none
  integer, parameter :: dp  = selected_real_kind(14,300)
  integer, parameter :: i4b = selected_int_kind(9)
  real(dp)      :: a
  integer(i4b)  :: i
  write(6,*) 'programmer defined kind for real    is: ',dp
  write(6,*) 'programmer defined kind for integer is: ',i4b
  write(6,*) 'a real constant of programmer defined kind :', &
    1.0_dp
end program
```

```
! F90 allows user defined data types


program types

  implicit none
  integer, parameter :: dp   = selected_real_kind(14,300)
  type atom
    real(dp) :: tau(3)
    integer  :: z
  end type

  type (atom) :: litium(3)
  type (atom) :: nitrogen

  real(dp)    :: r(3),q(3)
  integer     :: i

  r = (/ 1.0_dp, 0.0_dp, 0.0_dp /)
  q = (/ 0.0_dp, 0.0_dp, 0.0_dp /)

  litium(1) = atom ( r, 3)

  litium(2)%tau = (/ 0.0_dp, 1.0_dp, 0.0_dp /)
  litium(2)%z      = 3

  litium(3)%tau    = q
  litium(3)%tau(3) = 1.0_dp
  litium(3)%z      = 3

  nitrogen = atom (q, 5)

  write(6,*) ' Atomic positions '
  write(6,fmt="('litium  ',I2,' =',3F8.3)") &
   ((i, litium(i)%tau),i=1,3)
  write(6,fmt="('nitrogen',I2,' =',3F8.3)") i, nitrogen%tau
  write(6,fmt="('distance between litium and nitrogen&
    & = ',3F8.3)") distance(litium(1),nitrogen)

contains

  function distance( a, b ) result ( d )
    type (atom), intent(in) :: a
    type (atom), intent(in) :: b
    real(dp)                :: d
    d = sqrt( (a%tau(1)-b%tau(1))**2 + &
              (a%tau(2)-b%tau(2))**2 + &
              (a%tau(3)-b%tau(3))**2     )
    return
  end function

end program
```

```
! in F90 a new program unit (MODULE) has been introduced.
! the modules substitutes the common block to share data
! between program units.
! They could contain procedure and functions together with
! the data.

module my_kinds
  implicit none  !
  save           ! all defined variables are now global
  integer, parameter :: dp  = selected_real_kind(14,300)
end module my_kinds

  module my_types
    use my_kinds, only: dp
    implicit none
    save
    type atom
      real(dp) :: tau(3)
      integer  :: z
    end type
  end module my_types
```

```
program types
  use my_kinds, only: dp
  use my_types, only: atom
  implicit none
  type (atom) :: litium(3)
  type (atom) :: nitrogen
  real(dp)    :: r(3),q(3)
  integer     :: i
  interface
    function distance( a, b ) result ( d )
      use my_kinds, only: dp
      use my_types, only: atom
      implicit none
      type (atom), intent(in) :: a
      type (atom), intent(in) :: b
      real(dp)                :: d
    end function
  end interface

  r = (/ 1.0_dp, 0.0_dp, 0.0_dp /)
  q = (/ 0.0_dp, 0.0_dp, 0.0_dp /)
  litium(1) = atom ( r, 3)
  litium(2)%tau = (/ 0.0_dp, 1.0_dp, 0.0_dp /)
  litium(2)%z      = 3
  litium(3)%tau = (/ 0.0_dp, 0.0_dp, 1.0_dp /)
  litium(3)%z      = 3
  nitrogen = atom (q, 7)

  write(6,*) ' Atomic positions '
  write(6,fmt="('litium  ',I2,' =',3F8.3)") &
    ((i, litium(i)%tau),i=1,3)
  write(6,fmt="('nitrogen',I2,' =',3F8.3)") &
    i, nitrogen%tau
  write(6,fmt="('distance between litium and &
    & nitrogen = ',3F8.3)") &
    distance(litium(1),nitrogen)

end program

  function distance( a, b ) result ( d )
    use my_kinds, only: dp
    use my_types, only: atom
    implicit none
    type (atom), intent(in)  :: a
    type (atom), intent(in)  :: b
    real(dp)                 :: d
    d = sqrt( (a%tau(1)-b%tau(1))**2 + &
              (a%tau(2)-b%tau(2))**2 + &
              (a%tau(3)-b%tau(3))**2     )
    return
  end function
```

```
! F90 INTERFACE allows to implement subprogram prototiping
! and subprogram and operator overloading


subroutine swap_real(a, b)
  use my_kinds, only: dp
  real(dp), intent(inout) :: a, b
  real(dp) :: t
  t = a
  a = b
  b = t
  return
end subroutine

subroutine swap_integer(i, j)
  use my_kinds, only: dp
  integer, intent(inout) :: i, j
  integer :: t
  t = i
  i = j
  j = t
  return
end subroutine

subroutine swap_atom(a, b)
  use my_kinds, only: dp
  use my_types, only: atom
  type (atom), intent(inout) :: a, b
  real(dp) :: pos(3)
  pos   = a%tau
  a%tau = b%tau
  b%tau = pos
  return
end subroutine
```

```
program interfaces
  use my_kinds, only: dp
  use my_types, only: atom
  !
  real(dp) :: s = 1.0_dp
  real(dp) :: r = 2.0_dp
  real(dp) :: p(3) = (/ 0.0_dp, 0.0_dp, 0.0_dp /)
  real(dp) :: q(3) = (/ 1.0_dp, 1.0_dp, 1.0_dp /)
  integer  :: k = 10, l = 20
  type (atom) :: nitrogen, litium
  !
  interface swap
    subroutine swap_real(a, b)
      use my_kinds, only: dp
      real(dp), intent(inout) :: a, b
      real(dp) :: t
    end subroutine
    subroutine swap_integer(i, j)
      use my_kinds, only: dp
      integer, intent(inout) :: i, j
      integer :: t
    end subroutine
    subroutine swap_atom(a, b)
      use my_kinds, only: dp
      use my_types, only: atom
      type (atom), intent(inout) :: a, b
      real(dp) :: pos(3)
    end subroutine
  end interface
  !
  nitrogen = atom (p, 7)
  litium   = atom (q, 3)
  write(6,fmt="(' litium   atom position: ',3F8.3)") litium%tau
  write(6,fmt="(' nitrogen atom position: ',3F8.3)") nitrogen%tau
  write(6,fmt="(' swapping ... ')")
  call swap(litium, nitrogen)
  write(6,fmt="(' litium   atom position: ',3F8.3)") litium%tau
  write(6,fmt="(' nitrogen atom position: ',3F8.3)") nitrogen%tau
  !

end program
```

```
! F90 extend the control constructs of F77
! find the abscissa xw whose sine is yw
program loops
  integer :: iter
  integer, parameter :: itermax = 200
  integer :: control = 1
  real :: xu, xd, xw
  real :: y, yh, xh, yw = 0.5
  real, parameter :: tol = 1.0e-6
  xd = 0.0; xu = 1.0
  iterative_loop: do iter = 1, itermax
    yd = sin(xd)
    yu = sin(xu)
    if ( .not. ( yd <= yw .and. yw <= yu ) ) then
      control = 1
      exit iterative_loop
    end if
    if ( yd == yw ) then
      xw = xd;  control = 2
      exit iterative_loop
    else if ( yu == yw ) then
      xw = xu;  control = 3
      exit iterative_loop
    else if ( abs(yw-yd) < tol ) then
      xw = xd;  control = 4
      exit iterative_loop
    else if ( abs(yu-yw) < tol ) then
      xw = xu;  control = 5
      exit iterative_loop
    end if
    xh = (xu+xd)/2.0;  yh = sin(xh)
    if(yh >= yw ) then
      xu = xh
    else
      xd = xh
    end if
    control = control + 1
  end do iterative_loop
  select case ( control )
    case (1)
      write(*,fmt="('desired value not in the interval ',F9.6,' :',F9.6)") &
        yd, yu
    case (2:3)
      write(*,fmt="('wanted x (within machine accurancy) ',F9.6)") xw
    case (4:5)
      write(*,fmt="('wanted x (within tolerance) ',F9.6)") xw
    case default
      write(*,fmt="('maximum number of iteration exceeded')")
  end select
end program
```

```
program loops
  integer :: ic = 1, control = 1, i = 1, j = 1
  character(len=80) :: str
  character(len=40) :: first_name, family_name
  write(6,fmt="('insert your first name and family name')")
  read (5,fmt="(A80)") str
  first_name (1:40) = ' '
  family_name(1:40) = ' '
  str = adjustl(str);
  parse: do while ( control /= 4 )
    select case (control)
      case (1)
        if( str(ic:ic) /= ' ') then
          if( i > 40 ) then
            write(6,fmt="(' first name too long ')")
            exit parse
          end if
          first_name(i:i) = str(ic:ic);  i = i + 1
        else
          control = 2
          cycle parse
        end if
      case(2)
        if( str(ic:ic) == ' ' ) then
          if( ic >= 80 ) then
            write(6,fmt="(' family name not found ')")
            exit parse
          end if
        else
          control = 3
          cycle parse
        end if
      case default
        if( str(ic:ic) /= ' ') then
          if( j > 40 ) then
            write(6,fmt="(' family name too long ')")
            exit parse
          end if
          family_name(j:j) = str(ic:ic);  j = j + 1
        else
          control = 4
          cycle parse
        end if
    end select
    ic = ic + 1
  end do parse
  if( control == 4 ) then
    write(6,fmt="(' first  name : ',A40)") first_name
    write(6,fmt="(' family name : ',A40)") family_name
  end if
end program
```

```
! optional argument allow the implementation of
! more flexible and safer codes

module my_tools
  use my_kinds
  implicit none
  save

  contains

  subroutine copy(x, y, nel, incx, incy)
    real(dp), intent(in)  :: y(:)
    real(dp), intent(out) :: x(:)
    integer, optional, intent(in)   :: nel      ! number of
elements
    integer, optional, intent(in) :: incx, incy ! increments
    integer :: i, ix = 1, iy = 1, inx = 1, iny = 1
    integer :: nm
    if( .not. present(incx) .and. .not. present(incy) ) then
      nm = min( size(x), size(y) )
      if( present(nel) ) then
        check_size_1: if(nel > nm ) then
          write(6,fmt="(' *** copy: nel is too large')")
          stop
        end if check_size_1
        x(1:nel) = y(1:nel)
      else
        x(1:nm ) = y(1:nm )
      end if
    else
      if(present(incx)) inx = incx
      if(present(incy)) iny = incy
      check_size_2: if(present(nel )) then
        if( ( 1 + (nel-1)*inx ) > size(x) ) then
          write(6,fmt="(' *** copy: nel is too large')")
          stop
        end if
        if( ( 1 + (nel-1)*iny ) > size(y) ) then
          write(6,fmt="(' *** copy: nel is too large')")
          stop
        end if
      end if check_size_2
      do i = 1, nel
        x( ix ) = y( iy ); ix = ix + inx; iy = iy + iny
      end do
    end if
  end subroutine
end module my_tools
```

```
program tools

  use my_kinds
  use my_tools, only: copy

  integer, parameter :: nmax = 1000
  integer  :: i
  real(dp) :: s
  real(dp) :: a(nmax) = 0.0_dp
  real(dp) :: b(nmax) = 0.0_dp

  a = (/ ( real(i,dp), i=1, nmax ) /)

  call copy(b, a, nel = 10, incx = 2, incy = 3 )

  write(6,fmt="(10F7.3)") b(1:50)

end program
```

```
! F90 INTERFACE allows to implement subprogram prototiping
! and subprogram and operator overloading
! the MODULE program unit help in managing interfaces
! In this exaple see also the use of PRIVATE and PUBLIC specifier
!
module my_utilities
  implicit none
  save
  interface swap
    module procedure swap_real, swap_integer, swap_atom
  end interface
  private ::  swap_real, swap_integer, swap_atom
  public  ::  swap, distance
  contains
    subroutine swap_real(a, b)
      use my_kinds, only: dp
      real(dp), intent(inout) :: a, b
      real(dp) :: t
      t = a; a = b;  b = t
      return
    end subroutine
    subroutine swap_integer(i, j)
      use my_kinds, only: dp
      integer, intent(inout) :: i, j
      integer :: t
      t = i; i = j; j = t
      return
    end subroutine
    subroutine swap_atom(a, b)
      use my_kinds, only: dp
      use my_types, only: atom
      type (atom), intent(inout) :: a, b
      real(dp) :: pos(3)
      pos   = a%tau;  a%tau = b%tau;  b%tau = pos
      return
    end subroutine
    function distance( a, b ) result ( d )
      use my_kinds, only: dp
      use my_types, only: atom
      implicit none
      type (atom), intent(in) :: a
      type (atom), intent(in) :: b
      real(dp)                :: d
      d = sqrt(  (a%tau(1)-b%tau(1))**2 + &
                 (a%tau(2)-b%tau(2))**2 + &
                 (a%tau(3)-b%tau(3))**2      )
      return
    end function
end module my_utilities
```

```
program interfaces
  use my_kinds, only: dp
  use my_types, only: atom
  use my_utilities
  !
  real(dp) :: s = 1.0_dp
  real(dp) :: r = 2.0_dp
  real(dp) :: p(3) = (/ 0.0_dp, 0.0_dp, 0.0_dp /)
  real(dp) :: q(3) = (/ 1.0_dp, 1.0_dp, 1.0_dp /)
  integer  :: k = 10, l = 20
  type (atom) :: nitrogen, litium
  !
  nitrogen = atom (p, 7)
  litium   = atom (q, 3)
  write(6,fmt="(' litium   atom position: ',3F8.3)") litium%tau
  write(6,fmt="(' nitrogen atom position: ',3F8.3)") nitrogen%tau
  write(6,fmt="(' swapping ... ')")
  call swap(litium, nitrogen)
  write(6,fmt="(' litium   atom position: ',3F8.3)") litium%tau
  write(6,fmt="(' nitrogen atom position: ',3F8.3)") nitrogen%tau
  !
  write(6,fmt="(' distance between nitrogen and litium: ',F8.3)") &
    distance(nitrogen, litium)
  !
  write(6,fmt="(' value of s: ',F8.3)") s
  write(6,fmt="(' value of r: ',F8.3)") r
  write(6,fmt="(' swapping ... ')")
  call swap(s, r)
  !
  write(6,fmt="(' value of s: ',F8.3)") s
  write(6,fmt="(' value of r: ',F8.3)") r


end program
```