Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

- Day 4:
  - Specification statements.
  - Intrinsic procedures.

Specification statements

Implicit typing

- FORTRAN has as default implicit typing:
  - Variables starting with the letters: i,j,k,l,m,n are INTEGERS.
  - The rest are REAL.
- Changing the defaults:
  - IMPLICIT INTEGER (a-h).
  - IMPLICIT REAL(KIND(1.0D0)) (r,s)
  - IMPLICIT TYPE (mytype) (u,x-z).
- Recommendation: **use strong typing**:
  - IMPLICIT NONE (should appear **after** the USE statements).

“GOD is real
(unless declared integer)”

Implicit typing

Example:

```
PROGRAM main
REAL, DIMENSION(13,13) :: array
DO j=1,SIZE(array,2)
   DO i=1,SIZE(array,1)
      array(I,j) = 0.0
      PRINT*,' ij = ',i,j
   ENDDO
ENDDO
END PROGRAM main
```

This program has a bug.
Using IMPLICIT NONE the compiler will tell us where.
Compiling with '-C' (capital-C) will check array bound violation.

Entities of different shape

- To declare entities of different shape:

  Example:

  INTEGER :: a,b

  INTEGER, DIMENSION(10) :: c,d

  INTEGER, DIMENSION(8,7) :: e
- Can be written:

  INTEGER :: a,b,c(10),d(10),e(8,7)

  INTEGER, DIMENSION(10) :: c,d,e(8,7)

Named constants

- To declare a named constant use the attribute PARAMETER:

  Example:

  INTEGER, PARAMETER :: N = 100

  INTEGER, PARAMETER :: MKS = KIND(1.0E0)

  INTEGER, PARAMETER :: MKD = KIND(1.0D0)

  INTEGER, PARAMETER :: MK = MKS

  REAL(MK), PARAMETER :: Pi = 3.1415926_MK

  REAL(MK), PARAMETER :: Pi = ACOS(-1.0_MK)

  Invalid in ‘9x
  But valid in ‘0x

  CHARACTER(LEN=*), PARAMETER :: string = ‘no need to count!’ ! With the LEN=* construct

Only INTRINSICS OF TYPE INTEGER/CHAR are allowed in ‘9x

Initial value for variables

- A variable is generally undefined (can have any value!) unless it is initialized.
- A variable may be assigned an initial value in a type declaration:

  Example:

  REAL(MK) :: a = 0.0

  REAL, DIMENSION(3) :: b = (/0.0,1.0,2.0/)

The data statement

- An alternative way of specifying the initial value is the DATA statement:

  DATA *obj-list /value-list/* [[,] *obj-list /value-list/*]…

  Example:

  REAL :: a,b,c

  INTEGER :: i,j,k

  DATA a,b,c/1.,2.,3./,i,j,k/1,1,1/

Optional comma

The data statement

Example (cont.):

```
REAL, DIMENSION(5,5) :: a
INTEGER :: i,j,k,p,q
DATA a/25*0.0/ i,j,k/3*1/
DATA a(1,1),a(3,1), a(1,2),a(3,3) /2*1.0,2*2.0/
DATA a(2:5,4) /4*1.0/
DATA (a(i,q),i=1,5,2), q=1,5) /SIZE(a)*0.0/
DATA (a(i,q),i=1,5), q=1,SIZE(a,2)) /SIZE(a)*0.0/ ! Invalid
DATA (a(i,q),i=1,5), q=1,5)) /SIZE(a)*ACOS(-1.)/ ! Invalid
```

Pointer initialization

- The status of *undefined* (**dangling**) pointers may not be checked with any intrinsic procedure (e.g., ASSOCIATED).
- It is the responsibility of the programmer to maintain a correct status of the pointer.
- To prevent dangling pointers use the subroutine NULLIFY or the FUNCTION NULL().

Pointer initialization

Example:

```
PROGRAM main
REAL, DIMENSION(:), POINTER :: p
REAL, DIMENSION(:), ALLOCATABLE, TARGET :: t
ALLOCATE(t(10000))
p => t
DEALLOCATE(t)
NULLIFY(p) !  otherwise p remains falsely ASSOCIATED
IF (ASSOCIATED(p)) THEN
…
ENDIF
```

Default initialization of components

- Components of user defined types may be initialized during the declaration of the type:

  Example:

```
TYPE entry
    REAL :: value = 2.0
    INTEGER :: index
    TYPE (entry), POINTER :: next => NULL()
END TYPE entry
TYPE (entry), DIMENSION(100) :: matrix
```

Default initialization of components

- Alternatively, components of user defined types may be initialized during the declaration of the variable:

  Example:

  ```
  TYPE (entry), DIMENSION(100) :: matrix = &
       entry(2.0,0,NULL())
  ```

PUBLIC and PRIVATE attributes

- Data in modules may be accessed when the module is USEd – the data is default PUBLIC. However, it may be desirable that only the procedures of the module can access certain parts of the module – the data is PRIVATE.
- The entities that may be specified by name in PUBLIC or PRIVATE lists are named *variables*, *procedures*, *derived types*, *name constants*, and *namelist* groups.

PUBLIC and PRIVATE attributes

- The default may be changed by writing PUBLIC or PRIVATE.

  Example:

```
MODULE example
    PRIVATE ! All variables are now private
    REAL, PUBLIC :: public_data
    REAL :: private_data
END MODULE example
```

PUBLIC and PRIVATE attributes

Example:

```
MODULE example
PRIVATE specific_int, specific_real
INTERFACE generic_name
       MODULE PROCEDURE specific_int, specific_real
END INTERFACE
CONTAINS
      SUBROUTINE specific_int(i)
      …
      SUBROUTINE specific_real(r)
      …
END MODULE example
```

Here the specific routines are PRIVATE and ONLY the generic routine can be access from a USE of the module example.

The SAVE attribute

- To save a local variable between calls to the procedure use the SAVE attribute.

  Example:

  ```
  SUBROUTINE sub(x)
  REAL, INTENT(IN) :: x
  INTEGER, SAVE :: counter
  ```

Notice: local variables with a declared initial value are automatically saved.

The SAVE attribute

Example:

```
SUBROUTINE sub(x)
REAL, INTENT(IN) :: x
REAL , SAVE :: a ! a is explicitly saved
LOGICAL :: first = .TRUE. ! The "=.TRUE." implies SAVE
IF (first) THEN
   first = .FALSE.
   a = x
ELSE
   a = a + x
ENDIF
```

The USE statement

- Entities in modules may be renamed to avoid name clash (between different modules):

    USE *module-name*, *rename-list*

  Where each rename has the form:

    local-name => use-name

  Example:

  USE stats_lib, sprod => prod

  USE maths_lib

  ! The prod in stats_lib is now renamed to sprod so we

  ! can use the native prod in maths_lib.

The USE statement

- Name clash is allowed if the variables are not used.
- Name clash is allowed for generic routines – the names are automatically concatenated in the interface block (so it appears as a single interface block).

The USE statement

- For cases where only a subset of the names of a module is needed, the ONLY option is available:

  USE *module-name*, ONLY : [*only-list*]

  where each ONLY has the form:

  access-id

  or

  [local-name =>] use-name

The USE statement

Example:

USE stats_lib, ONLY : sprod => prod, mult

Here we only use ‘mult’ and ‘prod’ from the module and we rename the prod procedure to sprod

- Modules can USE modules.

  Example:

  MODULE one

  INTEGER :: I

  END MODULE one

  MODULE two

  USE one

  …

  END MODULE two

Intrinsic procedures

- More than 100 intrinsic procedures.
- Four categories:
  - Elemental procedures.
  - Inquiry functions.
  - Transformational functions.
  - Non-elemental subroutines.
- All the intrinsic *functions* are pure (no side effects).

Inquiry functions for any type

- ASSOCIATED(pointer[,target]):
  - When target is absent, returns the value true if the pointer is associated with a target and false otherwise.
  - If target is present it returns true if the pointer is associated with the specific target.
- PRESENT(a):
  - May be called in a subprogram that has an optional argument. It returns true if the argument is present.
- KIND(x):
  - Has type default integer and value equal to the kind type parameter value of x (x: integer, real, complex, logical)

Elemental numeric functions

- Elemental functions that may convert:
  - **ABS(a)**: returns the absolute value of an argument of type integer, real, or complex. The result is of type integer if a is integer and otherwise it is real.
  - AIMAG(z): returns the imaginary part of the complex value *z*. the type is real.
  - AINT(a[,KIND]): truncates a real value *a* towards zero to produce a real that is a whole number.
  - ANINT(a[,KIND]): returns the nearest whole number to the real value *a*.

AINT(-0.9) = 0.0
ANINT(-0.9) = -1.0

Elemental numeric functions

- Elemental functions that may convert:
  - CEILING(a[,KIND]): returns the least integer greater than or equal to its real argument.
  - CMPLX(x[,y][,KIND]): converts *x* or *(x*,*y)* to complex type.
  - FLOOR(a[,KIND]): returns the greatest integer less than or equal to its real argument.

CEILING is similar to AINT() but returns an integer
FLOOR is similar to ANINT() but returns an integer.

Elemental numeric functions

- Elemental functions that may convert:
  - **INT(a)**: converts to integer type. The argument a may be:
    - INTEGER: in which case INT(a) = a.
    - REAL: truncate towards zero.
    - COMPLEX: real part is truncated towards zero.
  - **NINT(a)**: returns the integer value that is nearest to the real *a*.
  - **REAL(a[,KIND])**: converts to real type.

Elemental numeric functions

Example:

```
INTEGER, PARAMETER :: MK = KIND(1.0E0)
REAL(MK) :: x,dx,Lx
INTEGER :: i,N
N = 10; Lx = ACOS(-1.0_MK) ! ACOS(-1.) = π
dx = Lx/REAL(N-1,MK)
DO i=1,N
    x = REAL(i-1,MK)*dx
```

Elemental numeric functions

- Elemental functions that may not convert:
  - CONJG(z): returns the conjugate of the complex z.
  - DIM(x,y): returns MAX(x-y,0.) for arguments that are both integer or both real.
  - **MAX(a1,a2,[,a3,…])**: returns the maximum of two or more integer or real values.
  - **MIN(a1,a2,[,a3,…])**: returns the minimum of two or more integer or real values.

Elemental numeric functions

- Elemental functions that may not convert:
  - **MOD(a,p)**: returns the reminder of: *a* modulo *p*, that is: *a* – INT(*a/p*)**p*. **The value of *p* must NOT be zero**; *a* and *p* must be both integer or both real.
  - MODULO(a,p): as MOD() but replacing INT with FLOOR.
  - SIGN(a,b): returns the absolute value of *a* times the sign of *b*. *a* and *b* must both be integer or both real.

Elemental mathematical functions

- ACOS(x): returns the arc cosine (inverse cosine) function value for real values *x* such that *|x|≤1*, expressed in radians in the range: 0 ≤ acos(x) ≤ pi.
- ASIN(x): returns the arc sine, in the range: -pi/2 to pi/2.
- ATAN(x): returns the arc tangent (inverse tangent) in the range: -pi/2 to pi/2.
- **ATAN2(y,x)**: returns the arc tangent function value for pairs of real, x and y. The range is –pi ≤ atan2(y,x) ≤ pi. **The value of x and y must both not be zero !**

Elemental mathematical functions

- COS(x): returns the cosine function for an argument of type real or complex that is treated as a value in **radians**.
- COSH(x): returns the hyperbolic cosine function for a real argument *x*.
- EXP(x): returns the exponential function value for a real or complex argument *x*.
- LOG(x): returns the natural logarithm for a real or complex argument *x*. In the real case, *x* must be positive. In the complex case *x* must not be zero.
- LOG10(x): returns the common (base 10) logarithm.

Elemental mathematical functions

- SIN(x): returns the sine function for a real or complex argument (radians).
- SINH(x): returns the hyperbolic sine function for a real argument.
- SQRT(x): returns the square root function value for a real or complex argument *x*. If *x* is real its value must not be negative.
- TAN(x): returns the tangent function for a real argument (radians).
- TANH(x): returns the hyperbolic tangent function.

Elemental character and logical functions

- Character-integer expressions:
  - ACHAR(i): is of type default character with length one and returns the character in the position of the **ASCII** collating sequence that is specified by the integer: *i* with $i \in [0,127]$.
  - CHAR(i[,KIND]): is of type character and length one and returns the character in the position of the ***processors*** collating sequence.
  - IACHAR(c): is of type default integer and returns the position in the ASCII collating sequence of the default character c.
  - ICHAR(c): (as IACHAR but in the seq. of the processor).

These are useful for converting the case of letters

Elemental character and logical functions

- Lexical comparison functions (ASCII):
  - LGE(string_a,string_b): returns the value TRUE if string_a follows or is equal to the string_b.
  - LGT(string_a,string_b): returns the value TRUE if string_a follows string_b.
  - LLE(string_a,string_b): returns the value TRUE if string_b follows or is equal to string_a.
  - LLT(string_a,string_b): returns the value TRUE if string_b follows string_a.

Comparison: in terms of ASCII seq.

String-handling elemental functions

- ADJUSTL(string): adjust left to return a string of the same length by removing all leading blanks and inserting the same number of trailing blanks.
- ADJUSTR(string): adjust right to return the a string of the same length by removing all trailing blanks and inserting the same number of leading blanks.
- INDEX(string,substring[,BACK]): has type default integer and returns the starting position of substring as a substring of string, or zero if it does not occur. If back is absent or FALSE the starting position of the first such substring is returned. Otherwise the last occurrence.

String-handling elemental functions

- LEN_TRIM(string): returns a default integer whose value is the length of string without trailing blanks.
- SCAN(string,set[,back]): returns a default integer whose value is the position of a character of string that is in set, or zero if there is not such character.
- VERIFY(string,set[,back]): returns the default integer value 0 if each character in string appears in set, or the position of a character of string that is not in set.

Non-elemental string-handling functions

- String-handling inquiry functions:
  - LEN(string): returns a scalar default integer holding the number of characters in string.
- String-handling transformational functions:
  - REPEAT(string,ncopies): forms the string consisting of the concatenation of ncopies copies of string.
  - TRIM(string): returns string with all trailing blanks removed.

```
CHARACTER(LEN=85) :: string
string(1:4) = ‘abcd’
PRINT*,LEN(string) ! will return 85
```

Numerical inquiry and manipulation functions

- Numerical inquiry functions:
  - DIGITS(x): for real or integer *x*, returns the default integer whose value is the number of significant digits in *x*.
  - EPSILON(x): for real *x*, returns a real result with the same type parameter as *x* that is almost negligible compared with the value ONE
  - HUGE(x): for real or integer *x*, returns the largest possible value of *x*.
  - MAXEXPONENT(x): for real *x*, returns the default integer e_max, the maximum exponent in x.
  - MINEXPONENT(x) - similar to MAXEXPONENT.

HUGE(1.0e0) ~ 3e38   HUGE(1.0d0) ~ 1.8e308

Numerical inquiry and manipulation functions

- Numerical inquiry functions:
  - PRECISION(x): for real or complex x, returns a default integer holding the equivalent decimal precision of x (typical 6 and 15 for single and double precision)
  - RADIX(x): for real or integer x, returns the default integer that is the base in the number x (typical 2).
  - RANGE(x): for integer, real, or complex x, returns a default integer holding the equivalent decimal exponent range of x (fx. 37 and 307 for single and double precision real).
  - TINY(x): for real x, returns the smallest possible number represented by x.

Numerical inquiry and manipulation functions

- Transformational functions for kind values
  - SELECTED_INT_KIND(r): returns the default integer scalar that is the kind type parameter for an integer data type **able to** represent all integer values *n* in the range -10r < n < 10r.
  - SELECTED_REAL_KIND([p][,r]): returns the default integer scalar that is the kind type parameter value for a real data type with decimal precision (as returned by PRECISION()) **at least** *p*, and decimal exponent range (as returned by RANGE()) of **at least** *r*.

Bit manipulation procedures

- Eleven procedures based on the US Military Standard MIL_STD 1753.
- Inquiry function:
  - BIT_SIZE(i)
- Elemental functions:
  - BTEST(i,pos)
  - IAND(i,j)
  - IBCLR(i,pos)
  - IBITS(i,pos,len)
  - IBSET(i,pos)

Bit manipulation procedures

  - IEOR(i,j)
  - IOR(i,j)
  - ISHFT(i,shift)
  - ISHFTC(i,shift[,size])
  - NOT(i)
- Elemental subroutine:
  - CALL mvbits(from,frompost,len,to,topos)

Vector and matrix multiplication functions

- DOT_PRODUCT(a,b):
  - Returns the dot product between two the vectors. Requires two arguments each of rank one and the same size.
- MATMUL(A,B):
  - Perform matrix multiplication. Three cases:
    - SHAPE(A) = (n,m); SHAPE(B) = (m,k); result: (n,k)
    - SHAPE(A) = (m); SHAPE(B) = (m,k): result: (k)
    - SHAPE(A) = (n,m); SHAPE(B) = (m): result: (n)

Transformational functions that reduce arrays

- There are seven transformational functions that perform operations on arrays (fx. summing their elements).
- Single argument case:
  - ALL(mask): returns TRUE if all the elements of the logical array mask is true.
  - ANY(mask): … if any of the elements is true.
  - COUNT(mask): returns the number of true elements in mask.
- Optional argument: DIM is a scalar integer. If present the operation is applied to all rank-one sections that span through dimension DIM.

Transformational functions that reduce arrays

- MAXVAL(A): returns the maximum value of the real or integer array A.
- MINVAL(A): returns the minimum value of the real or integer array A.
- PRODUCT(A): returns the product of the elements of an integer, real or complex array A.
- SUM(array): returns the sum of the elements of an integer, real, or complex array A.

• Additional optional argument: logical MASK.

Array inquiry functions

– ALLOCATED(A): returns true if the allocatable array A is currently allocated, otherwise false.
– LBOUND(A[,DIM]): when DIM is absent, returns a rank-one default inter array holding the lower bonds of the A. If DIM is present it returns a scalar integer holding the lower bonds for the array dimention DIM.
– SHAPE(A): returns a rank-one scalar default array holding the shape of the array A.
– **SIZE(A[,DIM])**: returns a scalar default integer that is the size of the array A or extend along dimension DIM if DIM is present.
– UBOUND(A[,DIM]): as LBOUND.

Array construction and manipulation functions

- The merge elemental function:
  - MERGE(tsource,fsource,mask): merges the arrays tsource and fsource. The result is tsource where mask is true otherwise fsource.

Array construction and manipulation functions

- Packing and unpacking arrays:
  - PACK(A,mask[,vector]): when vector is absent, returns a rank-one array containing the elements of array A corresponding to true elements of mask in array element order.
  - UNPACK(V,mask,field): returns an array of the type and type parameters of V and shape of mask. Field must be of the same type and type parameters as vector and must either be scalar of be of the same shape as mask. The result equal V when mask is true – otherwise field.

Array construction and manipulation functions

- Reshaping an array:
  - RESHAPE(source,shape[,pad][,order]): returns an array with shape given by the rank-one array shape, and type parameters those of the array source. The elements of shape must be non-negative. If pad is present it must be an array of the same type and type parameters as source. If pad is absent or of zero size, the size of the result must not exceed the size of source. If order is absent, the elements of the result, in array element order, are the elements of source in array element order followed by copies of pad in array element order. If order is present, it must be a rank-one integer array with the permutations of the result.

Array construction and manipulation functions

- Transformational function for replication
  - SPREAD(A,DIM,ncopies): replicates the array A ncopies time in the direction DIM
- Array shifting functions
  - CSHIFT(A,shift[,DIM]): performs a circular shift of the array elements in A
  - OESHIFT(array,shift[,boundary][,DIM]): as CSHIFT but with boundary elements replacing the circular shift.
- Matrix transpose:
  - TRANSPOSE(A): performs a transpose of matrix A.

Transformational functions for geometric location

– MAXLOC(array[,mask]): returns a rank-one default integer array of size equal to the rank of array. Its value is the sequence of *subscripts* on an element of maximum value (optionally masked).
– MAXLOC(array,DIM[,mask]) – as above but omits the dimension DIM.
– MINLOC(array[,mask]): similar to MAXLOC.
– MINLOC(array,DIM[,mask]): similar to MAXLOC.

Transformational functions for pointer disassociation

– NULL([mold]): returns a disassociated pointer.

Non-elemental intrinsic subroutines

- Real-time clock:
  - DATE_AND_TIME([date][,time][,zone][,values]):
    - Date: is a scalar character variable holding the data in the form CCYYMMDD.
    - Time: is a scalar character variable holding the time in the form HHMMSS.SSS (SSS: milliseconds)
    - Zone: is a scalar character variable that is set to the difference between the local time and Coordinate Universal Time (UTC, also known as Greenwich Mean Time) in the form: Shhmm (S: sign).
    - Values: is a rank-one integer array of size at least 8 holding the sequence of values: CCYY, MM, DD, Shhmm, HH, MM, SS, SSS.

Non-elemental intrinsic subroutines

- System clock:
  - SYSTEM_CLOCK([count][,count_rate][,count_max]):
    - Count: is a scalar default integer holding a processor-dependent value based on the current value of the processor clock.
    - Count_rate: is a scalar default integer holding the number of clock counts per second.
    - Count_max: is a scalar default integer holding the maximum value that count may take.

Non-elemental intrinsic subroutines

- CPU time:
  - CALL CPU_TIME(time):
    - Time: is a scalar real that is assigned a processor-dependent approximation to the processor time in seconds, or a processor-dependent negative value if there is not clock.

Example:

```
REAL :: t1,t2
CALL CPU_TIME(t1)
… ! Code to be timed
CALL CPU_TIME(t2)
WRITE(*,*) ‘CPU time: ‘,t2-t1,’ seconds’
```

Non-elemental intrinsic subroutines

- Random numbers:
  - CALL RANDOM_NUMBER(harvest):
    - Returns a pseudorandom number from the uniform distribution over the range $0 \le x < 1$ or an array of such numbers. Harvest has intent OUT, may be an array, and must be of type REAL.
  - CALL RANDOM_SEED([size][put][get]):
    - size, INTENT(OUT): scalar default integer that the processor sets to the size n of the seed array.
    - put, INTENT(IN): default integer array of rank one and size n that is used by the processor to reset the seed.
    - get, INTENT(OUT): default integer array of rank one and size n that is processor sets to the current value of the seed.