4. Programming Languages - III

[1]

[1] M. Ganesh, MATH440/540, SPRING 2018

[2] Arrays

- **First we consider syntax to create, read, write, and manipulate one-dimensional arrays**
- **Integer valued, real valued, and character arrays, say `int_1d_arr`, `real_1d_arr`, and `name_1d_arr`, of respective dimension 100, 1000, and 45, may be declared as follows:**
- ```
  INTEGER, DIMENSION(100) ::  int_1d_arr
  REAL, DIMENSION(1000) ::  real_1d_arr
  CHARACTER(len = 25), DIMENSION(45) ::  name_1d_arr
  ```
- **Array constants may be created created by placing the arrays values within `(/` and `/)`. (In F2003, replace `(/` and `/)` with `[` and `]` .)**
- **For example, to create `Fib_arr` with first five Fibonacci numbers, use**
  `INTEGER, DIMENSION(5) ::  Fib_arr = (/ 1, 1, 2, 3, 5 /)`
- **The 100-dimensional array `int_1d_arr` may be created as**
  ```
  int_1d_arr = (/ (i**2, i = 1, 100) /)
  int_1d_arr = (/ ((i*j, i = 1,20,1), j = 2,10,2) /)
  ```

[2] M. Ganesh, MATH440/540, SPRING 2018
  ```

- [3]**Last two statements use implied `DO` loops**
- **The declaration part `DIMENSION(1000)` assumes that the associated variables use indices from** 1 **to** 1000
- **In general, `DIMENSION(low_val:high_val)` can be used for variables with indices from `low_val` to `hig_val`**
- **The statements**

```
INTEGER, PARAMETER ::  low_bound = -1000
INTEGER, PARAMETER ::  up_bound = 1000
INTEGER, PARAMETER ::  MINE = SELECTED_REAL_KIND(15,30)

REAL (KIND=MINE), DIMENSION(low_bound:up_bound) ::  real_arr
```

  are useful for creating a 2001**-dimensional array `real_arr` with reference to, for example, second element in the array being `real_arr(-999)` and** 1001**th entry being `real_arr(0)`**

- **The values of the entire array may be preassigned without a loop as**

```
real_arr = 0.0_MINE      !All 2001 entries of the array are zero
```

[3]M. Ganesh, MATH440/540, SPRING 2018

- **Several parts of the array `real_arr` can be separately created using the colon operator (with `start:end:incr` form; default `incr = 1`):**

```
real_arr(-1000:-1) = -1.0_MINE          !First 1000 values are -1

real_arr(:-1) = -1.0_MINE          !Same as the previous statement

real_arr(0:10:2) = 22.0_MINE !Value 22 for 1001, 1003, ...1011 elements

sel_subs = (/ 1 7 9 /)          !Some arbitrary subscripts (assume decl.)

sel_rad = (/ 0.0 0.1 0.6 /)      !Some arbitrary data (assume declared)

real_arr(sel_subs) = SIN(sel_rad) !Elementary functions work on arrays

real_arr(11:1000) = 2.d0*real_arr(-1000:-11) !Assign values in bulk

real_arr(11:)  = ABS(real_arr(-1000:-11)) + EXP(real_arr(11:1000))
```

- **[4]The colon operator is useful for reading/writing data in bulk without loops and for creating and manipulating multi-dimensional arrays:**

```
REAL (KIND=MINE), DIMENSION(100) ::  new_arr
new_arr = real_arr(101:200)
WRITE(14,FMT=27) (new_arr((i-1)*10+1:i*10), i = 1,10,1)
27 FORMAT(3X, 10G20.5E4)
```

- **Two-dimensional arrays, say, a 10×10 matrix `new_mat` and a 7×5 matrix `rect_mat` may be declared as:**

```
REAL (KIND=MINE), DIMENSION(10,10) ::  new_mat
REAL (KIND=MINE), DIMENSION(-3:3,7:11) ::  rect_mat
```

- **It is standard in F90+(although not required) compilers to store/read/write two-dimensional arrays in column major order. (Assume this below.)**

- **That is, storage of `new_mat` is in one-dimensional array form with order `new_mat(1,1)`,`new_mat(2,1)`, ...`new_mat(10,1)`,`new_mat(1,2)`, ...**

- **`RESHAPE` functions are useful to reshape arrays, say, 1-D to 2-D arrays:**

```
new_mat = RESHAPE(new_arr, (/10,10/) )
```

[4]M. Ganesh, MATH440/540, SPRING 2018

- [5]**If the last statement is executed the first column of** `new_mat` **are the first ten values of** `new_arr` **and the second column of** `new_mat` **are** `new_arr(11:20)` **and so on**
- **The general form of** `RESHAPE` **is** `RESHAPE( given_array, shape)`**, where** `given_array` **contains the data to be reshaped according to the form specified in the one-dimensional (rank-1) array** `shape`
- **Recall that we stored** `new_arr` **in a file using the unit number** 14
- **We may initialize** `new_mat` **by opening the file with unit number, say,** 15 **and we may read the data and assign to** `new_mat` **without loop as** `READ(15,*) new_mat`
- **This will lead to**
  ith row of the file correspond to the ith column of `new_mat`**,** $i = 1, \cdots, 10$
- **Instead, if we want the data in the ith row of the file correspond to the ith row of** `new_mat`**,** $i = 1, \cdots, 10$**, use instead**
  `READ(15,*) (( new_mat(i,j), j =1,10), i = 1,10)`

[5]M. Ganesh, MATH440/540, SPRING 2018

- **[6]The colon operator is useful to refer to various parts of the parts and to create new arrays for sub-blocks of the existing arrays:**

- `new_arr = new_mat(5,:)  !Use 5th row to create a 10-dim array`

- `new_arr1 = new_mat(:,8)  !Use 8th column to create a 10-dim array`

- `rect_arr = new_mat(3:9,2:10:2)  !Use row 3 to 9 and even columns`

- **Multi-dimensional arrays can be declared and manipulated similarly.**
  `REAL (KIND=MINE), DIMENSION(10,10,20) ::  new_mult_arr`

- **Standard storage of these arrays are similar to 2-D arrays:**

  `new_mult_arr(1,1,1),new_mult_mat(2,1,1), ...new_mult_arr(10,1,1),`

  `new_mat(1,2,1), new_mult_mat(2,2,1), ...new_mult_arr(10,2,1) ...`

[6]M. Ganesh, MATH440/540, SPRING 2018

- [7]**So far, we have used static memory allocations using explicit-shape arrays. (Avoid this in HPC codes; instead use the following.)**
- **It is efficient (and necessary in HPC, due to memory limitations) to use dynamic memory allocation using deferred-shape arrays.**
- **Dynamic memory allocations require three statements: declaration, allocation, and deallocation**
- **(i) Deferred-shape declaration includes data type and rank of arrays:**

```
REAL, ALLOCATABLE, DIMENSION(:)  ::  real_1d_arr
REAL (KIND=MINE), ALLOCATABLE, DIMENSION(:,:)  ::  new_mat, rect_mat
```

- **(ii) Allocation includes list of arrays to be allocated and may include `STAT=` (that if present will return 0 value if allocation is successful; a positive number if not successful; in HPC use this):**

```
ALLOCATE ( real_1d_arr(10000) )
ALLOCATE (new_mat(100,100), STAT=mat_all_stat )
ALLOCATE (rect_mat(-3:3,7:11), STAT=rect_mat_stat )
```

- **(iii) All allocated arrays should be deallocated:**

```
DEALLOCATE (real_1d_arr, new_mat, rect_mat, STAT=deall_stat)
```

[7]M. Ganesh, MATH440/540, SPRING 2018

- [8]**Recall that we allocated matrices:** `new_mat(100,100)`, `rect_mat(-3:3,7:11)`. **Suppose that for** $i = -3:3$, `rect_mat(`i`,7:11) =` i^2
- **Fortran 2003 provides flexibility in dynamic memory allocation**
- **Two simple useful additions in Fortran 2003 within** `ALLOCATE` **statements are: (i)** `SOURCE=` **(that allows allocation and initialization of a new array to be of the same shape and value specified in source) and (ii)** `ERRMSG=` **(a string that will not change if allocation is successful; otherwise changed to an appropriate error message)**
- **For example, in a Fortran 2003 code, we may use**
  `ALLOCATE(rect_mat1, SOURCE=rect_mat+17, STAT=rect,ERRMSG = my_msg )`
  to allocate a 7×4 **matrix** `rect_mat1` **with** `rect_mat1(`i`,7:11) =` $i^2 + 17$
- **A substantial improvement in Fortran 2003 is automatic allocation, reallocation, and deallocation of new arrays of a specific rank (that is 1-D, or 2-D, ...array) using existing allocated arrays of same rank**
- **For example, after** `REAL, ALLOCATABLE, DIMENSION(:,:) :: flex_mat` **in Fortran 2003, without allocation** `flex_mat` **may take shape of any allocated matrix within the program:**
  `flex_mat = new_mat` **and after several statements** `flex_mat = rect_mat`

[8]M. Ganesh, MATH440/540, SPRING 2018

[9]F90+ has several intrinsic inquiry functions to get properties of arrays:

- `ALLOCATED(new_mat) !.TRUE. if allocated; in general ALLOCATED(array)`
- `SHAPE(rect_mat)  !Ans:  7 4; in general SHAPE(array)`
- `LBOUND(rect_mat)  !Ans:  -3 7; in general LBOUND(array, dim)`
- `LBOUND(rect_mat,1)  !Ans:  -3; using format LBOUND(array, dim)`
- `UBOUND(rect_mat)  !Ans:  3 11; in general UBOUND(array, dim)`
- `UBOUND(rect_mat,2)  !Ans:  11 ; using format UBOUND(array, dim)`
- `SIZE(rect_mat)  !Ans:  28 (total #); in general SIZE(array, dim)`
- `SIZE(rect_mat,2)  !Ans:  4; in general SIZE(array, dim)`

[9]M. Ganesh, MATH440/540, SPRING 2018

[10]F90+ has several useful intrinsic functions that act on arrays:

- `ALL(l_arr) !.TRUE. if all values in logical array l_arr are true`
- `ANY(l_arr) !.TRUE. if any of the values in l_arr are true`
- `COUNT(l_arr)  !Number of .TRUE. values in the array l_arr`
- `DOT_PRODUCT(1d_arr1, 1d_arr2) !dot product of equal sized vectors`
- `MATMUL(mat1, mat2)  !Matrix multiplication of appropriate size matrices`
- `TRANSPOSE(my_mat)  !Transpose of a 2-D matrix my_mat`

[10]M. Ganesh, MATH440/540, SPRING 2018

[11]**The following transformational functions can be used with or without an additional logical `mask` (= `log_mask`) array to avoid operations on certain elements in arrays that do not obey a chosen property in `log_mask`.**

If `mask` is not present in the following commands, operations are performed on all of the elements in the argument array.

- `MAXLOC(array,mask)! Loc. of max. val. of elem. with mask property`
- `MAXVAL(array,mask) ! Max. val. of elements in array with mask prop.`
- `MINLOC(array,mask)! Loc. of min. val. of elem. with mask property`
- `MINVAL(array,mask) ! Min. val. of elements in array with mask prop.`
- `PRODUCT(array,mask) ! Product of elements in array with mask property`
- `SUM(array,mask) ! SUM of elements in array with mask property`

[11]M. Ganesh, MATH440/540, SPRING 2018

- [12]**Elementary functions that mathematically allow only scalar arguments can be applied directly to arrays in F90+ without using loops**

- **Examples of such functions are** `SIN, COS, TAN, ASIN, ACOS, ATAN, ...` `ABS, LOG, LOG10, SQRT, ...`
- **One approach to apply such functions only for a certain elements in an array is to use a combination of** `DO` **and** `IF` **statements**
- `WHERE` **statements facilitate avoiding loops in such cases:**

```
WHERE(ABS(new_mat) <= 1.D0)
                my_inv_sin_mat = ASIN(new_mat)
ELSEWHERE(new_mat < -1.d0)
                my_inv_sin_mat = -9999999
ELSE
                my_inv_sin_mat = 9999999
ENDWHERE
```

[12]M. Ganesh, MATH440/540, SPRING 2018

[13]The general form of WHERE construct:

```
name_where: WHERE <logical expression>_1 THEN
              <statements>_1
                ...
            ELSEWHERE <logical expression>_2
              <statements>_2
                ...
            ELSE
              <statements>_3
                ...
            END WHERE    name_where:
```

- **<statements>$_i$ are executed if <logical expression>$_i$ is .TRUE.**
- **More than one ELSEWHERE branch is permitted.**
- **The ELSEWHERE and ELSE branches may be omitted and statement lists may be empty. The name_value may be omitted or may be added next to ELSEWHERE and ELSE**

[13]M. Ganesh, MATH440/540, SPRING 2018

- [14]**One useful component in HPC is a facility to able to process operations on arrays in any order selected for and by individual processors**
- **The `DO` type loop structures must be executed in strict order**
- **In F90+ this can be avoided by using `FORALL` construct on arrays**
- **The general form of `FORALL` construct:**

```
name_anyorder: FORALL (index1 = ..., index2 = ..., ...., log_expr)
                        <statement>_1
                        <statement>_2
                          ...
                    END FORALL  name_anyorder
```

- **More than one counting indices and logical expressions (that is, `index2 = , index3 = , ..., log_exp`) may be omitted**
- **`<statement>`$_1$ is executed for all specified indices before `<statement>`$_2$ is executed for all indices and so on**
- **The `name_anyorder` may be omitted**

[14]M. Ganesh, MATH440/540, SPRING 2018

[15]Examples:

```
INTEGER, PARAMETER ::  n = 100
REAL, ALLOCATABLE, DIMENSION(:,:)  ::  A, B, C
ALLOCATE (A(n,n), B(n,n), C(n,n))
FORALL (i = 1:n:2, j = 2:n:2)
              A(i,j) = ........
              B(i,j) = ........
              C(i,j) = A(i,j) + B(i,j)
END FORALL
evenodd:    FORALL (i = 2:n:2, j = 1:n:2, i /= j)
               A(i,j) = ........
                B(i,j) = ........
                C(i,j) = A(i,j) - B(i,j)
            END FORALL  evenodd
diag:  FORALL (i = 1:n)
              A(i,i) = .....
              B(i,i) = sqrt(A(i,i))
              C(i,i) = A(i,i) +2.d0*B(i,i)
        END FORALL  diag
```

[15]M. Ganesh, MATH440/540, SPRING 2018

[16] Procedures

- **Three types:**
  - ⋆ **Internal procedure (within a host program unit)**
  - ⋆ **External procedure (separate program unit; can be compiled, debugged, tested independently of other procedures)**
  - ⋆ **Module procedure (separate unit to share information, such as initial values and definitions between various program units)**
- **Unless specified below, by procedures we refer to external procedures**

- **Two subclass of procedures:**
  - ⋆ **Subroutines. (Useful to obtain multiple variable results using `CALL` in main and other program units.)**
  - ⋆ **Functions. (Useful to obtain single variable value by evaluating in main and other program units.)**
- **Except for simple calculations, identify various sub tasks in an algorithm (especially those that can be used for various other problems) and use several procedures to implement the algorithm**

[16] M. Ganesh, MATH440/540, SPRING 2018

[17] Subroutines

- **General form:**

```
SUBROUTINE name_of_routine(list_of_variable_arguments)
                          ...
                        <Declaration section>
                          ...
                        <Execution section>
                          ...
END SUBROUTINE name_of_routine
```

- **The `list_of_variable_arguments` are dummy arguments**
- **Suppose `list_of_variable_arguments` is: `a, b, c, d`. Suppose that (i) `a, b` are dummy 1-D variables to pass only input data (not to be changed inside the routine) ; (ii) 2-D array `c` is used for both passing input data and returning results; and (iii) 2-D array `d` is used only to return results, specify your intent in declaration:**

```
REAL, DIMENSION(:), INTENT(IN) ::  a, b
REAL, DIMENSION(:,:), INTENT(INOUT) ::  c
REAL, DIMENSION(:,:), INTENT(OUT) ::  d
```

[17] M. Ganesh, MATH440/540, SPRING 2018

- [18]**Consider a test routine**

```
SUBROUTINE test_routine(cons,a,b,c,d)
USE ini_data_mod      !use data from module procedure ini_data_mod
IMPLICIT NONE
INTEGER, INTENT(IN) ::  cons
REAL, DIMENSION(:), INTENT(IN) ::  a, b
REAL, DIMENSION(:,:), INTENT(INOUT) ::  c
REAL, DIMENSION(:,:), INTENT(OUT) ::  d !For strings use LEN = *
REAL, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) ::  loc_temp
INTEGER ::  loc_temp_stat
...
...
c =  ...
...
ALLOCATE(loc_temp(1000,1000), STAT=loc_temp_stat)
loc_temp =  ...
d =  ...
DEALLOCATE(loc_temp)
END SUBROUTINE test_routine
```

[18]M. Ganesh, MATH440/540, SPRING 2018

- [19]**The dummy variables in the `list_of_variable_arguments` should not be allocated or deallocated in subroutines. (Exception F2003 ...)**
- **Local variables required to do some calculations in a subroutine may be allocated and deallocated within the subroutine**
- **To call a procedure within its procedure (recursive), use**
  `RECURSIVE SUBROUTINE name_of_routine(list_of_variable_arguments)`
- **To make some of the variables in `list_of_variable_arguments` optional use `OPTIONAL` and in the routine use `PRESENT` to declare some default value or ...., to deal with calls using fewer argument variables:**

```
INTEGER, INTENT(IN), OPTIONAL ::  cons
IF ( PRESENT(cons) ) THEN     ···     ELSE     ··· END IF
```

- **To return back from a procedure (if some constraints are violated) use `RETURN` in the procedure**
- **The `test_routine` is aware of only shape of the dummy arrays. In such procedures, for proper array references from any calling program unit to subroutines, it is important that all calling program units have explicit `interface` to all calling subroutines:**

[19]M. Ganesh, MATH440/540, SPRING 2018

```
PROGRAM main_program  !Example program unit using test_routine
USE ini_data_mod      !module procedure to be discussed below
USE other_data_mod  !USE ....  should be first, before IMPLICIT NONE
IMPLICIT NONE
INCLUDE ‘‘mpif.h’’      !In case of MPI2.*, USE mpi is allowed
INTEGER, PARAMETER ::  n = 100
INTEGER ::  my_rank, num_procs, avail_procs, my_cons, my_alloc_stat
REAL, ALLOCATABLE, DIMENSION(:)  ::  u,v,w
REAL, ALLOCATABLE, DIMENSION(:,:)  ::  x,y,z
!  INTERFACE PART
INTERFACE
     SUBROUTINE test_routine(cons,a,b,c,d)
     USE ini_data_mod
     IMPLICIT NONE
     INTEGER, INTENT(IN) ::  cons
     REAL, DIMENSION(:), INTENT(IN) ::  a, b
     REAL, DIMENSION(:,:), INTENT(INOUT) ::  c
     REAL, DIMENSION(:,:), INTENT(OUT) ::  d
     END SUBROUTINE test_routine
  END INTERFACE
```

```
  !  EXECUTABLE PART
...
...
ALLOCATE (u(n), STAT= my_alloc_stat)
ALLOCATE (v(-n:n), STAT= my_alloc_stat)
ALLOCATE (x(n,n), STAT= my_alloc_stat)
ALLOCATE (y(5*n,n+100:n+200), STAT= my_alloc_stat)
...
u =  ...
...
v =  ...
...
x =  ...
...

!  call the subroutine samp_routine to get y value and change x
CALL test_routine(my_cons, u, v, x, y)

ALLOCATE (w(n), STAT= my_alloc_stat)
```

```
ALLOCATE (z(n,n), STAT= my_alloc_stat)

...

w = 2*x(n,:) ...
...

z = MATMUL(x,x)  ...

...
WRITE(...) ...

... FORMAT ...

DEALLOCATE(u,v,w,x,y,z)

END PROGRAM main_program
```

- [20]`INTERFACE` is required to make sure that F90+ compilers are aware of the properties (type, rank, ...) of the list of dummy variables in `test_subroutine` while we compile `main program` (before `CALL`)
- This way compilers can easily check and make sure that each of the actual variable in `CALL` statements satisfy all the properties of the associated dummy variable (otherwise will declare error and abort)
- Consequently, actual variable pointers point to the correct first value of the array in the memory. The "make aware" approach is necessary to avoid any out of bound array access and wrong results
- For such an explicit `INTERFACE`, some of the declaration part from subroutines should be repeated in all calling programming units
- Such a repetition cannot be avoided if required subroutines are to be accessed from a built-in library collection within a HPC system, such as LAPACK, BLAS, FFT etc. (F2003 `IMPORT` helps to avoid ...)
- However, if we have a direct read-write access to a subroutine, such a repetition can be avoided, for example, by placing the routine inside a module with `CONTAINS` and in the calling units `USE` the module to avoid repetition. For example, we may place `test_routine` in a `test_module`:

[20]M. Ganesh, MATH440/540, SPRING 2018

```
MODULE test_module
CONTAINS
  SUBROUTINE test_routine(cons,a,b,c,d)
  USE ini_data_mod      !use module procedure ini_data_mod
  IMPLICIT NONE
  INTEGER, INTENT(IN) ::  cons
  REAL, DIMENSION(:), INTENT(IN) ::  a, b
  REAL, DIMENSION(:,:), INTENT(INOUT) ::  c
  REAL, DIMENSION(:,:), INTENT(OUT) ::  d
  REAL, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) ::  loc_temp
  INTEGER ::  loc_temp_stat
  ...
  c =  ...
  ...
  ALLOCATE(loc_temp(1000,1000), STAT=loc_temp_stat)
  loc_temp =  ...
  d =  ...
  DEALLOCATE(loc_temp)
  END SUBROUTINE test_routine
END MODULE test_module
```

- **[21]Using `test_module` the `INTERFACE` block in previous version of `main_program` may be avoided as follows:**

```
PROGRAM main_program_no_interface  !Program unit using test_module
USE ini_data_mod      !module procedure to be discussed below
USE other_data_mod
USE test_module    !Avoid INTERFACE for calling test_routine
IMPLICIT NONE
INCLUDE ‘‘mpif.h’’      !to be discussed - MPI programming
INTEGER, PARAMETER ::  n = 100
INTEGER ::  my_rank, num_procs, avail_procs, my_cons, my_alloc_stat
REAL, ALLOCATABLE, DIMENSION(:)  ::  u,v,w
REAL, ALLOCATABLE, DIMENSION(:,:)  ::  x,y,z
!  EXECUTABLE PART
...
!  call samp_routine contained in test_module to get y and change x

CALL test_routine(my_cons, u, v, x, y)
...
END PROGRAM main_program_no_interface
```

[21]M. Ganesh, MATH440/540, SPRING 2018

- **[22]Another approach to avoid `INTERFACE` is to use only explicitly declared arrays in procedures: (add dummy bounds - avoid this ....)**

```
SUBROUTINE test_routine(cons,a,b,c,d,n,m,l,j,k)
USE ini_data_mod      !use data from module procedure ini_data_mod
IMPLICIT NONE
INTEGER, INTENT(IN) ::  cons
REAL, DIMENSION(n), INTENT(IN) ::  a, b
REAL, DIMENSION(m,l), INTENT(INOUT) ::  c
REAL, DIMENSION(j,k), INTENT(OUT) ::  d
REAL, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) ::  loc_temp
INTEGER ::  loc_temp_stat
...
c =  ...
...
ALLOCATE(loc_temp(1000,1000), STAT=loc_temp_stat)
loc_temp =  ...
d =  ...
DEALLOCATE(loc_temp)
END SUBROUTINE test_routine
```

[22]M. Ganesh, MATH440/540, SPRING 2018

[23] Modules

- **Useful for sharing common data between various program units:**
- **Example:**

```
MODULE name_mod  !If saved as name_mod.f90 after compilation,
IMPLICIT NONE  !say mpif90 -c name_mod.f90, leads to name_mod.mod
SAVE   !All variables below; otherwise, say, REAL, SAVE ::  ...
!Some MPI variables
INTEGER ::  my_rank, ierror, avail_cores
INTEGER, DIMENSION(:), ALLOCATABLE ::  labor, work1, work2
!  Some common variables
INTEGER, PARAMETER ::  MY_KIND = SELECTED_REAL_KIND(15,30)
INTEGER ::  fixed_cons, specific_constant
INTEGER, PARAMETER ::  size_vec
REAL (KIND=MY_KIND), DIMENSION(size_vec) ::  fixed_real_vec
COMPLEX (KIND=MY_KIND), DIMENSION(size_vec) ::  fixed_comp_vec
CHARACTER (LEN = 5), ALLOCATABLE, DIMENSION(:)  ::  char_array
REAL, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) ::  work_space
END MODULE name_mod
```

[23] M. Ganesh, MATH440/540, SPRING 2018

[24] Function procedure

- **Returns a single variable to main executing programming units**
- **The function name should be the name of the single return variable**
- **The type of the variable/function-name should be declared either as a type declaration or in the declaration section of the function program, especially if `IMPLICIT NONE` is used. (Use `IMPLICIT NONE`.)**
- **Use `INTENT(IN)` to avoid function procedures to change input**
- **Should have an assign statement with name of the function on LHS**
- **`INTERFACE` and related details discussed earlier for `SUBROUTINE` apply for function procedures. (Replace `SUBROUTINE` and its associated declaration parts with `FUNCTION` and its associated declaration part.)**
- **Recall that while executing a procedure actual arguments are passed to the procedure as pointers to specific memory locations**
- **If procedures (say functions) are to be used/passed as arguments in/from, say, a subroutine/main program, then use `EXTERNAL` for the procedure arguments in the subroutine and main program:**

```
REAL, EXTERNAL ::  my_func          EXTERNAL ::  my_routine
```

[24] M. Ganesh, MATH440/540, SPRING 2018

- **Example:**

```
REAL FUNCTION test_function(a,b,c,d)  !May move REAL to declaration
USE ini_data_mod  !In program units use, say, val = test_function(u,v,x,y)

IMPLICIT NONE
INTEGER, INTENT(IN) ::a
INTEGER, PARAMETER ::  master = 0
REAL, DIMENSION(:), INTENT(IN) ::  b
REAL, DIMENSION(:,:), INTENT(IN) ::  c
CHARACTER*(*), DIMENSION(:,:), INTENT(IN) ::  d
REAL, ALLOCATABLE, DIMENSION(:,:), INTENT(OUT) ::  loc_temp
INTEGER ::  loc_temp_stat
...
...
ALLOCATE(loc_temp(1000,1000), STAT=loc_temp_stat)
loc_temp =  ···
test_function =  ··· !This statement is required to return test_function

DEALLOCATE(loc_temp)
END FUNCTION test_function
```

- [25]**For recursive functions use**
  `RECURSIVE FUNCTION rec_fn_name(...)`

- **`PURE FUNCTION fn_name(...)` is useful for creating a function `fn_name` with no side effects, such as modifying input arguments and any other visible data (such as those in modules)**

- **Similarly `PURE SUBROUTINE routine_name(...)` has no side effects except that it can modify only variables with `INTENT(OUT` and `INTENT(INOUT)`**

- **Procedures with side no effects are useful for parallel computations if processors are allowed to take a combination of control indices and execute in any order, such as those facilitated by `FORALL` statements**

- **A function procedure may be placed inside a main program unit using `CONTAINS`. Such a procedure is called an internal procedure:**
- **Example:**

[25]M. Ganesh, MATH440/540, SPRING 2018

```
PROGRAM main_program  !Example program unit using test_routine
DECLARATION SECTION
SOME EXECUTION PARTS
...
...
value = my_internal_func(...)
WRITE(*,*) value
...
...
CONTAINS    !Want to include my_internal_func
PURE FUNCTION my_internal_func(...)
USE ini_data_mod
IMPLICIT NONE
REAL (KIND = ....)  ::  my_internal_func ···
...
my_internal_func = ...
PURE FUNCTION fn_name(...)
END FUNCTION my_internal_func
END PROGRAM main_program
```

[26] M. Ganesh, MATH440/540, SPRING 2018

- [27]**There are several other useful tools in F90+:**
  - ⋆ **Derived data types. (Structures, structure constructors ....)**

```
TYPE ::  my_type ........  END my_type  !Many additions in F2003
```

  - ⋆ **Several advanced features of procedures and modules in F2003**
  - ⋆ **Pointers and dynamic data structures:**

```
REAL, POINTER ::  my_poi   REAL, TARGET ::  val   my_poi => val
```

  - ⋆ **Object oriented programming (Fortran 2003)**

[27]M. Ganesh, MATH440/540, SPRING 2018

[28] MAKEFILE, BLAS, LAPACK, scaLAPACK, FFT, ...

- **Suppose that we have a module file `func_mod.f90` that is used by three distinct main programs `q1_ans.f90`, `q2_ans.f90`, and `q3_ans.f90` (corresponding to three questions in an assignment) in a directory**

- **Suppose that we want to create three executable files `q1_exe`, `q2_exe`, and `q3_exe` (one for each of the questions) in the directory using the compiler gfortran**

- **Create a file in the directory, with name `makefile` using details in the next page**

- **After creating the `makefile`, type the command `make` in the directory to produce the three exeutable files**

- **To create the executable only for the first question, type the command `make q1_exe` (and similarly for other questions)**

- **To remove the executables and *.mod files, type `make clean`**

[28] M. Ganesh, MATH440/540, SPRING 2018

```
# Set compiler commands
#-------------------------------------
FC90 = gfortran      #Change if necessary
FLAGS = -O3
#-------------------------------------
q1_files = func_mod.f90 q1_ans.f90
q2_files = func_mod.f90 q2_ans.f90
q3_files = func_mod.f90 q3_ans.f90

all: q1_exe q2_exe q3_exe

q1_exe: $(q1_files)
        $(FC90)  $(FLAGS)  $(q1_files) -o $@

q2_exe: $(q2_files)
        $(FC90)  $(FLAGS)  $(q2_files) -o $@

q3_exe: $(q3_files)
        $(FC90)  $(FLAGS)  $(q3_files) -o $@

clean: rm  q1_exe  q2_exe q3_exe   *.mod
```

- [29]**Suppose that we have several procedures (modules, functions, subroutines, old/downloaded stuff) and a main program in a directory that in combination is an implementation of an algorithm to solve a project**
- **Suppose that the old stuff do not need the modules and let**
  `MODULES -- my_mod.f90, share_data_mod.f90`
  `SUBROUTINES -- my_routine.f90, routine_diff.f90, routine_int.f90`
  `FUNCTIONS -- my_func.f90, my_func1.f90`
  `SOME OLD F77 and C STUFF -- old1.for, old2.f, old3.c`
  `MAIN PROGRAM -- my_main.f90`
- **Suppose that our task is to produce an executable file, say, `project_exe` using all these files and compilers `gfortran, gcc, g++` or (`pgf90,pgcc, pgCC` or `ifort, icc, icpc`) or associated parallel MPI compiler commands: `mpif90, mpicc, mpiCC` on Sayers Lab computers or on Mio/AuN**
- **It is convenient to write a file called `makefile` to produce `project_exe` using the command `make` in various computing environments. Below, if required, replace `mpif90` with any of the above compiler options.**

[29]M. Ganesh, MATH440/540, SPRING 2018

```
# Makefile to produce the executable project_exe
#-----------------------------------
# Set compiler commands
#-----------------------------------
FC90 = mpif90       #Change if necessary
FC77 = mpif77       #If need to compile f77 codes:  *.for and *.f
CC = mpicc          #If some C codes to be compiled and wrapped with f90+
FLAGS = -O3
CFLAGS = -O3
MODS = my_mod.mod share_data_mod.mod
# If need to use lapack, blas, and library routines in */lib directory
LIBS = -l lapack -l blas -L$(MY_DIR)/lib $(LINK_FLAGS)
DBG =       #leave blank or add some debug command if required
LINK_FLAGS =    #to be discuss below:  AMD or INTEL libraries; LA, FFT
MY_DIR = /ux/mx/ci/some_name/some_more/some_more1  #Assuming Mio/AuN
#-----------------------------------
# List objects
#-----------------------------------

OBJS =  my_mod.o share_data_mod.o my_routine.o \
routine_diff.o my_func.o my_func1.o old1.o  old2.o old3.o
```

```
#-------------------------------------
# set object rules to get *.o files
#-------------------------------------
%.o: %.f90
        $(FC90) -c $(FLAGS) $(DBG) -o $@ $<
%.o: %.for
        $(FC77) -c $(FLAGS) $(DBG) -o $@ $<
%.o: %.f
        $(FC77) -c $(FLAGS) $(DBG) -o $@ $<
%.o: %.c
        $(CC) -c $(CCFLAGS)  -I$(MY_DIR)/include  $@ $<
# Use "-I" part above if, say, some header files are in MY_DIR/include
#-------------------------------------
# Primary rules
#-------------------------------------
all:   my_main.o $(OBJS)
       $(FC90) $(DBG) -o project_exe my_main.o $(OBJS) $(LIBS)
clean:   #Use command  "make clean" to remove old *.o and *.mod files
        rm *.o
        rm *.mod
```

```
#-------------------------------------
# modules
#-------------------------------------
my_mod.mod: my_mod.f90
            $(FC90) $(DBG) -c my_mod.f90

share_data_mod.mod: share_data_mod.f90
            $(FC90) $(DBG) -c share_data_mod.f90
#-------------------------------------
# other rules
#-------------------------------------
my_routine.o: my_rountine.f90  $(MODS)
routine_diff.o: routine_diff.f90 $(MODS)
my_func.o: my_func.f90 $(MODS)
my_func1.o: my_func1.f90 $(MODS)
old1.o: old1.for
old2.o: old2.f
old3.o: old3.c

# END OF MAKEFILE
```

- [30]**A well known class of subroutines for solving problems in Linear Algebra (system of equations, factorizations - LU, Cholesky, QR, SVD, Schur etc. - eigenvalue problems, etc.) is:**
  Linear Algebra PACKage (LAPACK)
- **For example LAPACK routines** `SGESV` **and** `DGESV` **solve** $Ax = b$ **in single- and double-precisions respectively for a general real matrix** A**.** (`CGESV`, `ZGESV` **are their counterparts for complex matrices.)**
- **Similarly, for** `x = S, D, C, Z`**, the LAPACK routines** `xGETRF` **are useful to compute the LU factorization of** A **and then** $Ax = b$ **can be solved using the factorization with LAPACK routines** `xGETRS`
- **Visit** `http://www.netlib.org/lapack/` `http://www.netlib.org/blas/` **and** `http://www.netlib.org/scalapack/` **to learn details of how to call such subroutines in your code**
- **LAPACK routines call Basic Linear Algebra Subprograms (BLAS)**
- **The ScaLAPACK (or Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory parallel computers. It is currently written in a SPMD style using explicit message passing for interprocessor communication**

[30]M. Ganesh, MATH440/540, SPRING 2018

- [31]**Both LAPACK and BLAS are installed on Mio/AuN and on Sayers Lab machines and can be linked as specified in the makefile**
- **The INTEL (MKL) and AMD (ACML) libraries include these packages (and well optimized respectively for INTEL and AMD processors) and several other useful routines for FFT etc.**
- **Visit** `http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm` **and** `http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx` **to learn details of using these packages**
- **LAPACK is installed on Mio/AuN and Sayers Lab machines and INTEL-MKL libraries on Mio/AuN (all with INTEL processors)**
- **For LAPACK subroutines (on Sayers Lab machiens), use** `LINK_FLAGS`**:** `LINK_FLAGS = -llapack` **in the makefile. For INTEL-MKL routines (on Mio/AuN), use** `LINK_FLAGS = -mkl`
- **For further specalized routines for linear algbebra problems, visit** `http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html` `http://www.cise.ufl.edu/research/sparse/umfpack/`

[31]M. Ganesh, MATH440/540, SPRING 2018