Theory for WORK 4 onwards

Arrays

The declaration of arrays can be done by the type specification, or by the specifying DIMENSION, COMMON (eliminated starting with Fortran 90), ALLOCATABLE, respectively POINTER or TARGET (starting only from Fortran 95, while in Fortran 90 there is the possibility to define them as "derived" type).

The characteristics of any array are:

- Type (any intrinsic or derived type),
- Rank (the number of "dimensions", e.g. a vector has rank 1, a matrix has rank 2, etc. – the maximum rank is 7 in Fortran),
- Extents (”lower” and ”upper” limits for each "dimension" separately, the "lower" means the initial value of the respective indices, and the "upper" means the final value of the respective indices),
- Size (results from the total number of elements),
- Shape (results from rank and extents).

Arrays of identical shape are "conformable" (meaning that certain operations can be performed on their elements, without explicitly specifying each element's positional indices). A scalar conforms to any array, regardless of the array’s shape.

The syntax for declaring an array by the DIMENSION attribute (static memory allocation):

[*Type*,] DIMENSION(*extents*)[,*attribute*] :: *array_list*

or

Type[,*attribute*] :: *array_name*(*extents*)[,*array_name*(*extents*)…]

The syntax for declaring an array by the ALLOCATABLE statement (dynamic memory allocation):

[*Type*,] ALLOCATABLE(:[,:]…)[,*attribute*] :: *array_list*

Note: for each rank, a position marked by the ":" character in the round bracket after the ALLOCATABLE keyword is reserved. For dynamic memory allocation the POINTER or TARGET attributes can also be used, the syntax of the declaration by POINTER or TARGET being similar to the syntax for ALLOCATABLE, only the keyword used differs (POINTER or TARGET will be written instead of ALLOCATABLE).

When using dynamic memory allocation via ALLOCATABLE, POINTER or TARGET, the ALLOCATE function will be used in the source file to actually allocate the required space to the arrays:

ALLOCATE(*array_name*(*extents*)[,*array_name*(*extents*)…])

In the case of dynamic memory allocation, it must be taken into account that at the end of the program run, the control over the memory blocks allocated for arrays (within the program) will also end, so that successive runs can lead to a situation where the working memory is completely occupied by areas allocated to arrays that can no longer be controlled. To avoid these situations, memory allocated dynamically within a program must be freed within the same program (before losing control of the memory area) using the function:

DEALLOCATE(*array_list*)

It may also be necessary to free allocated memory blocks in order to allocate different memory blocks (of different sizes) to the same arrays within a program. The allocation status can be tested using the ALLOCATED(*array_list*) function, e.g. within a simple logical IF (the syntax is shown in the control statements) used to free up the memory allocated to specific arrays:

IF(ALLOCATED(*array_list*)) DEALLOCATE(*array_list*)

Examples:	Explanations:
DIMENSION A(10,2,3),L(8)	The array named A has rank 3 (3 "dimensions", in total 10x2x3=60 positions for elements) and will be implicitly of type

	REAL. The array named L has rank 1 (8 positions) and implicitly of type INTEGER (due to the first letter of the name).
ALLOCATABLE X12(:,:),B(:)	The array named X12 has rank 2 (the reservation of each "dimension" is marked with the : character), and the array B will be a vector, having rank 1. Both arrays will be of type REAL by default. The actual number of positions in each array will be specified later.
POINTER C(:,:,:)	The array named C has rank 3 (the reservation of each "dimension" is marked with a : character) and will be of type REAL, POINTER by default. The actual number of positions (the extents) in the array will be specified later.
REAL,DIMENSION (3,3) :: D,E	Arrays D and E will be of type REAL with rank 2 and conform to each other (having identical shape).
INTEGER MAT(2:11,3)	The MAT array is of type INTEGER, with rank 2, having a total of 30 element positions. At the first rank the lower limit is 2 and the upper limit is 11 (position indices being incremented from 2 to 11), and at the second rank the lower limit is 1 (default) and the upper limit is 3.

Storing arrays in memory is done by positioning the elements in a row, incrementing the position indices successively in their order. Here is an example for array D (mentioned above, with rank 2 and size 3x3=9 positions):

D								
(1,1)	(2,1)	(3,1)	(1,2)	(2,2)	(3,2)	(1,3)	(2,3)	(3,3)

As can be seen, the indice on the first position is incremented (from the initial value, which is the lower limit, to the upper limit), then the next index, and so on...

Exemplifying with a matrix like: $\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{vmatrix}$, we could say that the storage of elements in memory is done according to the columns.

Initializing the elements of an array by using the DATA specification:

Examples:	Explanations:
DIMENSION A10(10,10)	Declaring an array named A10 (default type REAL), having a total of 10x10=100 element positions.
DATA A10/100*1.0/	Initialization by name: all 100 elements in array A10 will be given the value 1.0.
DATA A(1,1),A(10,2),A(5,5)/2*3.3,2.0/	Initialisation by elements: the elements at positions (1,1) and (10,2) are given the value 3.3, and the element at position (5,5) is given the value 2.0.
DATA ((A(i,j),i=1,5,2),j=1,3)/9*3.5/	Initialisation by cycle: the elements in positions (1,1), (3,1), (5,1), (1,2), (3,2), (5,2), (1,3), (3,3) and (5,3) are each given a value of 3.5. The indice *i* starts with a value of 1 and reaches a final value of 5 with increments of 2.

Note: the DATA specification is a declarative statement, so it must be passed before any executable statement. The following are some examples of executable statements (attribution statement).

Examples:	Explanations:
L=10 *! Equivalent to:*	L is the 8-position array, and the number 10 is a scalar value. Since a scalar conforms to any array,

`L(1)=10;L(2)=10;L(3)=10;L(4)=10` `L(5)=10;L(6)=10;L(7)=10;L(8)=10`	all 8 positions in the array L will be given the value 10.
`L=L*2` `! Equivalent to:` `L(1)=L(1)*2;L(2)=L(2)*2;L(3)=L(3)*2` `L(4)=L(4)*2;L(5)=L(5)*2;L(6)=L(6)*2` `L(7)=L(7)*2;L(8)=L(8)*2`	All 8 elements in the array L will have the value multiplied by 2 (since the scalar 2 is "conform" to the array L).
`D=-1.2` `E=-2.*D`	Each element in array D will be assigned the value -1.2. Arrays D and E are conform (they have the identical 2x3 shape), therefore each element in array E will receive the value 2.4 (resulting from multiplying -1.2 by -2).

String sections

The syntax for referencing a subset (part of an array with rank 1, i.e. of a string):

array_name ([*start*] : [*stop*][: *increment*])

Examples:	Explanations:
`REAL,DIMENSION(6) :: VA` `INTEGER,DIMENSION(0:5) :: VB`	The VA and VB arrays declared with 6 positions each – the position indice in the case of the VA array can take values from 1 to 6 inclusive, with an increment of +1, and in the case of the VB array from 0 to 5 (also 6 positions).
`VA(3:5)=1.0`	The elements at positions 3, 4, and 5 of the VA vector are given a value of 1.0.
`VB(1:5:2)=1`	The elements at positions 1, 3, and 5 (the indice starts at 1 and goes up to 5 with step 2) of the VB vector are given a value of 1.
`CHARACTER(LEN=8) :: TIT="ALanDALa"`	the string named TIT will have 8 positions and will be initialized with the quoted characters (1 character per 1 position).

The following references to sections of the entity named TIT (from the previous example) mean the (quoted) characters in the right column::

`TIT(2:4)`	"Lan" - the characters in positions 2-4 (including),
`TIT(5:5)`	"D" - the character at position 5,
`TIT(:5)`	"ALanD" - characters up to position 5,
`TIT(5:)`	"DALa" - characters from the 5th position,
`TIT(:)`	"ALanDALa" - equivalent to the reference of TIT,
`TIT(10:)`	String of null length characters (no characters from position 10),
`TIT(5:10)`	The last position in the string is 7 (`LEN=7`), and 10 > `LEN`. Such a reference is not allowed, it will generate error!

Intrinsic functions for character strings:

`LEN(`*string*`)` - returns the length (number of characters) of the specified *string*.

`INDEX(`*substring*,*string*`)` - returns the (start of) position of the *substring* in the *string*, or 0 if not.

`TRIM(`*string*`)` - returns the string without the trailing blank characters.

Flow control statements

Some of these instructions have already been presented in previous lab-works, being repeated to have a grouping of all control instructions in a single subsection.

Instructions for stopping execution

`STOP` [*stop_code*] — Terminates execution, stopping the program from running. If *stop_code* is specified, then it will be displayed (*stop_code* can be an integer, or a quoted string, it is used in case of more than one stop possibility, to identify the branch being run).

`PAUSE` [*pause_code*] — It can be used up to Fortran 90, since Fortran 95 it has been removed (it can be replaced by a display statement followed by a read). Temporarily suspends the running of the program, displaying (if specified) also the *pause_code* (can be an integer, or a quoted string). The <*Enter*> key must be pressed to continue running. In all cases it will cause the message to be displayed:
`PAUSE statement executed. Hit Return to continue.`
If a *pause_code* has also been specified, it will be displayed between the words "`PAUSE`" and "`statement`" in the above message.

Jump instructions

There are 3 variants of jump instructions (all use the keywords `GO TO` or `GOTO`), with the following syntax:

`GOTO` *label* — This is the unconditional jump, *label* is where to jump to when this statement is executed. The *label* must be "carried" by an executable instruction (it must be written in front of the instruction to be jumped to during execution).

`GOTO (`*label_list*`)[,]`*expression* — This is the computed jump. Evaluating the *expression* will give the position of the label in the *label_list* that will be used to perform the jump. Obviously, the resulting value of the *expression* must be a strictly positive integer. If this number is negative, zero or greater than the number of elements in the *label_list*, the jump will not be performed.

`GOTO` *var*`[[,] (`*label_list*`)]` — This is the assigned jump, where *var* is the name of an `INTEGER` entity to which the value of the desired label must first be assigned (`ASSIGN` *e* `TO` *var*). In newer versions of Fortran, the value of the label can be attributed or read instead of assignment.
The jump will only be performed if the value of *var* exists in the *label_list* (so by specifying labels in the *label_list* you can condition the jump to be performed or not).
If the *label_list* is not specified (`GOTO` *var*), the statement works like the unconditional jump.

Conditional statements

There are several types, some of which also have structured variants (introduced with Fortran 90), their syntax being as follows.

`IF (`*arithmetic_expression*`)` e_1, e_2, e_3 — Arithmetic decision (arithmetic `IF`) involves testing the value of the result of *arithmetic_expression* against zero, specifying 3 labels (not necessarily different). If the result from *arithmetic_expression* is strictly negative, a jump will be made to label e_1, if the result is null (the value 0) to label e_2, and in case of a strictly positive result to label e_3.

`IF(`*logical_expression*`)` *instruction*	Unstructured logical decision (simple logical `IF`), with empty branch, allows a single *instruction* to be specified. This statement will only be executed if *logical_expression* evaluates to true (with the value .`TRUE`.). Otherwise (resulting in .`FALSE`. for *logical_expression*) the statement will be ignored.
`IF(`*logical_expression_1*`) THEN` *instructions_1* [`ELSE IF(`*logical_expression_i*`) THEN` *instructions_i*] [`ELSE` *instructions_x*] `ENDIF`	The structured logical decision (structured logical `IF`) can be empty-branched (the variant in which only the `IF`, `THEN` and `ENDIF` keywords appear), or not. The `ELSE IF` keywords can also be written together as `ELSEIF` in some variants of the Fortran language. If several `ELSE IF` sequences are specified, the *logical_expression_i* must be distinct for each sequence, without the possibility of simultaneous fulfillment of several expressed conditions (the mention is also valid for *logical_expression_1*). If *logical_expression_1* results with the value .`TRUE`., those specified in the *instructions_1* block will be executed. Otherwise, if *logical_expression_1* returns .`FALSE`., the first *logical_expression_i* (if specified) that returns the value .`TRUE`. will be considered, leading to the execution of what is specified in the corresponding *instructions_i* block. Only if all preceding logical expressions returned .`FALSE`. those specified in the *instructions_x* block will be executed.
`SELECT CASE(`*expression*`)` [`CASE(`*criteria_ set_i*`)` *instructions_i*] [`CASE DEFAULT` *instructions_x*] `END SELECT`	The generalized condition testing allows the value of any *expression* to be tested. Care must be taken that each *criteria_ set_i* specified is clear, and without overlaps between them! The `CASE DEFAULT` branch will only be considered (performing *instructions_x*) if the conditions specified in all previous *criteria_ set_i* are not met.

Structured variants can contain other structured statements (structures), but **without intersecting them**. The contained structured statements must begin and end within the same block (marked in the preceding syntaxes with *instructions_1*, *instructions_i*, and *instructions_x*).

Examples:	Explanations:
`  CHARACTER r` `    …` `1 WRITE(*,*)'Enter the values: '` `    …` `  WRITE(*,*)'Restart? (Y/N): '` `  READ(*,*)r` `  IF(r.EQ.'y'.OR.r.EQ.'Y') GOTO 1` `    …`	Declare an entity of type character (1 position) An executable instruction with label 1 Reading a character and storing it in R, then testing the value by a simple logical `IF` and perhaps an unconditional jump to the instruction with label 1.
`  CHARACTER r` `    …` `1 WRITE(*,*)'Enter the values: '` `    …` `  WRITE(*,*)'Restart? (Y/N): '` `  READ(*,*)r` `  IF(r=='y'.OR.r=='Y') THEN`	The previous example, using a structured logical `IF` (without the `ELSE` branch) instead of a simple logical `IF`, and `.EQ.` replaced by ==.

`    GOTO 1` `  ENDIF`	
`   IF(x+1)3,1,6` `3  WRITE(*,*)"negative result"` `   GOTO 2` `1  WRITE(*,*)"null result"` `   GOTO 2` `6  WRITE(*,*)"positive result"` `2  CONTINUE`	Test the result of the numerical expression `x+1`, using an arithmetic `IF`, and depending on the result, display whether it is negative, zero or positive.
`IF(x+1<0) THEN` `   WRITE(*,*)"negative result"` `   ELSE IF(x+1==0) THEN` `    WRITE(*,*)"null result"` `    ELSE` `      WRITE(*,*)"positive result"` `ENDIF`	The previous example, using a structured logical `IF` instead of an arithmetic `IF`.
`    IF(x/2)3,3,6` `3   WRITE(*,*)"result <=0"` `    GOTO 2` `6   WRITE(*,*)"result >0"` `2   CONTINUE`	Testing the value resulting from the evaluation of the numerical expression `x/2`, with an arithmetic `IF`, then display it depending on the result, if it is less than or equal to zero or strictly positive.
`SELECT CASE(x/2)` `    CASE(:0)` `    WRITE(*,*)"result <=0"` `    CASE DEFAULT` `    WRITE(*,*)"result >0"` `END SELECT`	The previous example, but using a `SELECT CASE` structure with an arithmetic expression instead of an arithmetic `IF`. The criteria specified by `(:0)` means all numeric values up to and including zero.
`SELECT CASE(x/2<=0)` `    CASE(.TRUE.)` `    WRITE(*,*)"result <=0"` `    CASE(.FALSE.)` `    WRITE(*,*)"result >0"` `END SELECT`	The previous example, using a `SELECT CASE` structure with a logical expression. The value following the evaluation of a logical expression can be `.TRUE.` or `.FALSE.` (only one of the two logical values).

Instructions for loop cycles (repetitions)

The instructions for making loops are structured (those where the end of the structure is marked instead of the label with END_*name* being introduced with Fortran 90). Being structured instructions (structures), they can contain other structures (for example, loop within loop, or structured decision within loop, or loop within decision structure, etc.), but they cannot be intersected. Structured statements must begin and end within the same (*statement*-marked) block in the syntaxes below.

`DO` *label* [[`,`]*loop_control*]
instructions
label last_executable_statement

It would correspond to a post-conditional loop, with the caveat that if *loop_control* was specified, it would be evaluated first (as noted below). With this structured statement, if other loops are included in the loop having the same body, it is allowed to use a single *label* to mark the end of the structures (no intersection is considered).
If the last specification in the loop body is not an executable statement (like an `ENDIF` tag, or something similar), the neutral `CONTINUE` statement (shown below) can be used.

`DO` [*loop_control*]

The difference from the previous variant consists in the end marking

instructions
ENDDO

ENDDO (some variants of Fortran also accept END DO).

The syntax for *loop_control* is as follows:

loop_counter=*initial_value*, *end_value*[, *increment_step*]

Interpreting this is done by assigning *initial_value* to the *loop_counter* and checking if it is below *end_value* (if it is not, the loop will be ignored without executing any instruction from the body of the loop). After a first step through the *instructions* in the loop body, the *loop_counter* is changed by the value specified at *increment_step*. If *increment_step* is not specified, it will be taken as +1 by default. It checks that the value in the *loop_counter* has not exceeded the *end_value*, in order to resume the execution of the *instructions* in the body of the loop again. The exit from the loop will be made when the *loop_counter* will get a value above the *end_value*. Explicit modification (by statements) in the loop body of any *loop_control* component is not allowed.

If *loop_control* is not specified, the exit can be done with the EXIT statement or an "infinite" loop can be made (it can be stopped by pressing the <*Ctrl*> and <*C*> keys simultaneously, causing the program to be exited by forced interruption).

DO *label* [,] WHILE (*logical_expression*)
instructions
label last_executable_statement

It would correspond to a preconditioned loop. The *instructions* in the loop body will be executed only if *logical_expression* evaluates to .TRUE. (and the loop will only run as long as this value exists). When *logical_expression* becomes .FALSE., the loop will be exited.
If the last specification in the loop body is not an executable statement (like an ENDIF tag, or something similar), the neutral CONTINUE statement (shown in an example) can be used.

DO WHILE (*logical_expression*)
instructions
ENDDO

The difference from the previous variant consists in the end marking ENDDO (some variants of Fortran also accept END DO).

In addition to these statements, there are also some control statements that can be used to repeat or exit the above described loops.

CYCLE

Causes execution of previous instructions in a loop to resume, without going through all the statements in the loop body.

EXIT

Allows leaving the body of a loop (loop exit).

[*label*] CONTINUE

It is an executable statement with no effect. The meaning of use is only to wear the *label*.

Examples:	Explanations:
`DO i=1,10` `  WRITE(*,*)i` `ENDDO` `WRITE(*,*)i` `! Equivalent to:` `DO 8 i=1,10` `8  WRITE(*,*)i` `  WRITE(*,*)i`	Cycle for displaying the *loop_counter* value (*i*), in the version with ENDDO, or, using the label 8 to mark the end of the loop body.
`  DO i=1,n` `  DO j=i+1,n` `  REZ(i,j)=1.0/(i+j)` `  ENDDO` `ENDDO`	Loop inside a loop, in the variant with ENDDO (the first ENDDO is for the loop with counter *j*, considered internal) and in the variant of using a label (20) to mark the end of the loop body. It can be observed that in the

``! Equivalent to: DO 20 i=1,n DO 20 j=i+1,n 20 REZ(i,j)=1.0/(i+j) ! Equivalent to: DO 11 i=1,n DO 20 j=i+1,n 20 REZ(i,j)=1.0/(i+j) 11 CONTINUE```	second variant only one label was used (it is not considered a structure intersection in such situations). It can also be observed that the use of the value of the *loop_counter*s is allowed, but without their explicit modification. Of course, the `CONTINUE` statement (mentioned above) can also be used in such situations. The inner loop will be the one with label 20 (the last open structure must be the first closed one).
`` DO READ *,N IF(N==0) EXIT ENDDO```	A loop variant without control will exit the cycle due to the `EXIT` statement (if a null value has been entered for N).
`` DO i=1,4 PRINT *,i IF(i > 2) CYCLE PRINT *,i ENDDO PRINT *,'finished...'```	The following will be displayed on the screen: 1 1 2 2 3 4 finished... The `CYCLE` statement will cause the loop to resume (without executing the statements that follow it) from the moment the value of *i* exceeds 2.
`` CHARACTER*132 LINE READ ('A'),LINE i=1 DO WHILE (LINE(i:i)==" ") i=i+1 ENDDO```	A character string (named LINE) is defined with 132 positions (the old Fortran 77 syntax was used) and the characters are read in a single instruction (using the descriptor for alphanumeric values). As long as spaces (blank characters) are encountered starting from the beginning of the string, the *i* value will be incremented, which is also used to specify the position of the characters in the string (see substrings). Finally, *i* will contain the position of the first non-blank character in the LINE string (total number of blanks +1).

For input/output operations, implicit cycles can be used (similar to the examples for the `DATA` specification), as shown below:

Examples:	Explanations:
``DIMENSION A(10,10) READ *,"no. of lines in matrix A: ",nl READ *," no. of columns in matrix A: ",nc DO i=1,nl PRINT *,"elements on line ",nl," :" READ *,(A(i,j),j=1,nc) ENDDO PRINT *,"matrix A:" PRINT *, ((A(i,j),", ", j=1,nc), i=1,nl)```	Declaration of an array with 10x10=100 positions Using an implicit loop to read elements from a row in an array. Interpretation: read A(*i,j*) while the position indice *j* starts from the value 1 and reaches (incremented at each step by *+1*) the value of *nc*. Display the elements of array A, one by one, followed by the characters ",". Attention, in this case the display will be continuous (without line separation). The loop with counter *j* is inside the loop with counter *i*.