Theory for WORK 8 onwards

Program units

All programs written in the Fortran language can be organized into program units. A program unit is considered a sequence of specifications and instructions that can be written to a separate source file and compiled. Of course, several program units can also be written to a source file, and the order in which they are written is not important, except for modules (modules must be compiled before the program units that use them, so they must appear before them in the source file, so that by the time the unit that uses the module is compiled, the module is already compiled).
Usually each program unit starts with a definition and ends with the END mark followed by the specification of the corresponding program unit type. Program unit names must be unique and must comply with the criteria for symbolic names (cannot contain spaces or non-permitted characters, must start with a letter and cannot be longer than 32 characters, and for older versions of Fortran it is recommended to limit it to 6 characters).
No all program units may contain executable instructions, there are program units that may only contain specifications relating to entities used by other program units. There are 4 types of program units in Fortran:

- Main program (required in any application and may contain executable instructions),
- External procedures (subroutines, functions – may contain executable instructions),
- Modules (may not contain executable instructions, only possibly in embedded module procedures)
- Data blocks (cannot contain executable instructions, only specifications).

Each application created using Fortran must contain a single main program (this will be launched at the start of the run). External procedures are subroutines and functions that are defined separately. There are several types of procedures, but only external ones are considered program units. Modules are pre-compiled units (must be compiled before the program units that use them), usually containing only entity specifications. Data blocks contain specifications about entities and may also contain data initializations. The difference between data blocks and data files is the content of the specifications that require compilation (data files contain only values, no specifications in Fortran, so do not require compilation). The main program and procedures can contain executable statements, data blocks and modules can only contain entity specifications (with the exception that modules can also contain executable statements if these statements are part of module procedures).

Main program

Cannot be missing from any application and no application can contain more than 1 main program. This is the only program unit where specifying the type of program unit is optional. A main program cannot self-reference (directly or indirectly). The syntax of a main program is as follows (with comments):

[PROGRAM *name*]	If the keyword PROGRAM is used, then the name must also be specified (which must be unique and will be considered global - meaning it will be "seen" from all program units). Without the keyword PROGRAM no name can be specified, in such cases the default MAIN name for the program will be considered. Any program unit that starts with specifications or comments (or compilation directives via the OPTIONS keyword) will be considered main program.
[*specifications*]	The keywords INTENT, OPTIONAL, PUBLIC and PRIVATE may not be used in specifications. The entity specifications in all program units must precede the executable instructions.
[*executable statements*]	ENTRY and RETURN keywords may not be used.
[CONTAINS *internal procedures*]	Several internal procedures (subroutines and functions) may be defined successively..
END [PROGRAM [*name*]]	The final marking must be at least the END keyword. It may also be followed by the keyword PROGRAM, but the *name* may only be specified

if explicitly defined at the beginning of the program unit.

Example:	Explanations:
`END`	Main program with default name MAIN, without content.
`PRINT *,"Hello!"`	Main program that will only display the text `Hello!` on the
`END PROGRAM`	monitor.
`PROGRAM test`	Main program named TEST, that will call subroutine SUB1
`INTEGER C, D`	(contained as an internal procedure, along with the FUNC function).
`…`	Call subroutine named SUB1.
`CALL sub1`	
`…`	
`CONTAINS`	Marking the contained procedures.
`SUBROUTINE sub1`	Defining subroutine SUB1 as an internal procedure.
`…`	
`PRINT *, func(X,Y)`	Printing the result of the FUNC function for the current values of arguments X and Y.
`…`	
`END SUBROUTINE sub1`	End marking for internal procedure SUB1.
`FUNCTION func(X,Y)`	Defining the FUNC function as an internal procedure.
`…`	
`END FUNCTION func`	End marking for internal procedure FUNC.
`END PROGRAM test`	End mark for main program TEST with name indication (although `END` was sufficient).

Procedures

May be subroutines or functions, but only those defined as external procedures are program units. Procedures can be self-referencing (directly or indirectly) and have implicit interfaces (but interfaces can also be explicitly specified, via interface blocks). The types of procedures existing in Fortran are as follows:

- External procedures (subroutines and functions that are not part of another program unit);
- Internal procedures (subroutines and functions that are part of a main program or another procedure);
- Module procedures (procedures defined within modules);
- Intrinsic procedures (subroutines and functions predefined in the Fortran language);
- Dummy procedures (usually a dummy argument specified as a procedure, or listed as a procedure reference);
- Statement function (a computational procedure defined by a single statement, which may be referred to by its symbolic name).

All procedures have an interface, which is usually defined by default. A procedure interface refers to the properties of a procedure with which it interacts, or to the calling program unit. The interface may also be explicitly defined, through interface blocks. With the exception of data blocks, all program units may contain interface blocks.

External procedures may contain internal procedures, but internal and module procedures cannot contain internal procedures. Internal procedures are in the section preceded by the `CONTAINS` keyword and have access to all entities in the containing program unit (HOST). Their name cannot be used as an argument to another procedure (there are variants of Fortran that allow this, e.g. Intel Visual Fortran) and they cannot contain separate entry points (via the `ENTRY` specification).
Subroutines are invoked by the `CALL` statement or by a defined assigned statement. Subroutines do not return a value directly, but values may be transferred by known arguments or variables between the calling program unit and the subroutine. The return from a subroutine to the calling program unit is done by the `RETURN` statement, whose syntax is as follows:

`RETURN` [*number*]	The `RETURN` keyword may be followed by a *number* or numeric expression

whose value must be of type `INTEGER` (signifying the reserved position in the list of arguments by which the calling program unit will be returned).

Functions are invoked by name or by a defined operator. Normally they return a single result value (through the function name) after evaluation. The return from a function will default to the program unit in which the function reference was used, but the `RETURN` statement (shown above) can also be used to specify different return points from the function endpoint.

Entrance to a procedure (by `CALL` instruction in the case of subroutines, by name invocation in the case of a function) can also be made at a position other than the start of the procedure, by using the `ENTRY` specification, whose syntax is:

`ENTRY` *name* [`(`*arguments*`)`]	The statement may be specified in the content of external procedures (it cannot be used in internal procedures), being part of the body of the procedure, and the *name* is the name of the entry point in the procedure (different from the name of the procedure) by which that part of the procedure will be invoked. In such cases the statements preceding the `ENTRY` specification in the procedure definition will be ignored when the procedure is activated (execution of the statements in the procedure will start from the first statement following the specified entry point).

It is generally recommended to avoid the use of entry points in procedures, for clarity of source files. Arguments that are specified when defining a procedure (or an entry point in an external procedure) are considered notional, in the sense that at the time of procedure definition their values are not known, only their type. Arguments that are specified when invoking a procedure are considered effective, because in addition to knowing their type, their actual values are usually known. The order and type of the actual arguments (used at the call) must coincide with the order and type of the notional arguments (used when defining the procedure), but the name of the notional arguments may differ from the name of the effective arguments.

When defining procedures, in front of the keyword specifying the type of procedure, some characteristics can also be specified, such as::

`ELEMENTAL`	Where it is desired that the procedure be applied to only one element in an array at a time.
`PURE`	To avoid possible side-effects (on the value of the entities used). In the case of functions declared `PURE`, the `INTENT` options for arguments and function names will not be used (in subroutines there is no such restriction). In addition, a procedure declared `PURE` will only be able to use other `PURE` procedures.
`RECURSIVE`	As mentioned, direct or indirect recursion (self-reference) is allowed for functions and subroutines. If this feature is specified, when defining the procedure, the line declaring the type of the procedure (after the list of dummy arguments) may be completed with `RESULT(`*name_r*`)` to specify a different name `(`*name_r*`)` from the original name of the procedure, this different name being used for recursion.
`MODULE`	To specify a module procedure (can only be used within modules).

<u>Subroutines:</u>
In addition to the intrinsic subroutines existing in the Fortran language, other subroutines may be defined as needed. The syntax for defining a subroutine is as follows:

`SUBROUTINE` *name* [`(`*arguments*`)`]	Before the `SUBROUTINE` keyword, a procedure characteristic (`ELEMENTAL`, `PURE`, `RECURSIVE`) can be specified and the arguments are optional (they are only specified if value transfer between the calling program unit and the subprogram is desired). Arguments are considered notional in the sense that at the time of subprogram definition their values are not known, only their type. Reserved placeholders can also be used as arguments (see `RETURN` examples below).

[*specifications*]	In all program units, entity specifications must precede executable instructions.
[*executable statements*]	They may contain ENTRY specifications (for defining entry points) and RETURN instructions (for returning to the program unit from which the subroutine was called).
[CONTAINS *Internal_procedures*]	Several internal procedures (subroutines and functions) can be defined in succession, but only in the case of a subroutine defined as an external procedure. For internal procedures this section cannot appear.
END [SUBROUTINE [*name*]]	The final marking must be at least the END keyword for subroutines defined as an external procedure. It may also be followed by the keyword SUBROUTINE, possibly also by name. In the case of internal procedures the end marker must contain at least both keywords END SUBROUTINE.

Calling a subroutine is done by the CALL statement, whose syntax is as follows:

CALL *name* [(*arguments*)]	Arguments are specified if they exist in the subroutine definition. On call these arguments are considered effective, in the sense that at the time the subroutine is called, along with their type and their values, they are usually known. The order of the effective arguments (from the subprogram call) must match the order of the notional arguments (from the subroutine definition) as type, but different names may be used.

Examples:	Explanations:

`! main program`
`CALL hi`
`END PROGRAM`
 	
`! subroutine`
`SUBROUTINE hi`
`  PRINT *,"Hello!"`
`END SUBROUTINE salut`
	Main program that will only call the HI subroutine defined as an external procedure, and the subroutine will only display the text Hello! on the monitor.
	In the example below the run will stop in the subprogram.
	It can also be seen that at the end of the main program mark (END PROGRAM) it was not possible to specify the name of the main program as it was not defined.

`! main program`
`CALL hi`
`END`
 	
`! subroutine`
`SUBROUTINE hi`
`  PRINT *,"Hello!"`
`  RETURN`
`END`
	The previous example modified by inserting the RETURN statement in the definition of the subroutine HI. In this case, after calling the subroutine and displaying the text Hello! on the monitor, it will return to the main program and the run will stop at the end of the main program.
	It can also be seen that the END marking for the subroutine defined as an external procedure is sufficient.

`! subroutine with entry point`
`SUBROUTINE sign`
`  PRINT *,"positive value"`
`  RETURN`
`ENTRY negative`
`  PRINT *,"strictly negative value"`
`  RETURN`
`END`
 	
`! calling program unit`
	Example with an entry point named NEGATIVE in the subprogram named SIGN.
	If the value of scalar N is negative, then NEGATIVE is called, which is not a subroutine, but an entry point in the subroutine SIGN. As an effect, the executable instructions preceding the specification of the NEGATIVE entry point in the SIGN subroutine shall be ignored and the message strictly negative value shall be printed on the display, after which it

```
…
IF(N < 0) THEN
   CALL negative
ELSE
   CALL sign
ENDIF
…
END
```

shall return to the calling program unit.
If the value of scalar N is not negative, then the SEMN subroutine is called and the instructions are executed until the first RETURN is encountered (the message `positive value` is displayed and then the calling program unit is returned).
Of course, the specification of an entry point only conditions the start from which the instructions are executed, not the end (if `RETURN` had not been specified before the NEGATIVE entry point, when calling the SIGN subroutine after the `positive value` message was displayed, the `strictly negative value` message would also be displayed).

```
! calling program unit
     …
     CALL verif(A,B,*10,*20,C)
     PRINT *,”negative value”
     GOTO 30
10   PRINT *,”null value”
     GOTO 30
20   PRINT *,”positive value”
30   CONTINUE
     …
     END

! subroutine as external procedure
     SUBROUTINE verif(X,Y,*,*,Z)
     …
     IF(X*Y-Z) 50,54,55
50   RETURN
54   RETURN 1
55   RETURN 2
     END
```

In the program unit from which the VERIF subroutine is called, in the list of effective arguments appear the scalar entities of type `REAL` (due to the implicit rule) A, B, the reserved positions (by the `*` mark) with labels 10 and 20, respectively the scalar entity C (also of type `REAL` due to the implicit rule). These arguments correspond in order (and type) to the notional arguments that were specified when defining the subroutine: X and Y (of type `REAL` by default), then 2 reserved positions (each marked by `*`) and Z (of type `REAL` by default).
When the VERIF subroutine is called, the value from A will be transferred to X, the value from B to Y, and the value from C to Z in the subroutine. When the subroutine comes to test the value resulting from the arithmetic expression, the appropriate label is chosen from the list (in the case of a strictly negative result it jumps to label 50, in the case of a null result to label 54, and in the case of a strictly positive result to label 55). If jumping to the instruction with label 50, the return to the calling unit will be made to the actual arguments of the `CALL` instruction (the value in A will be updated from the value of X, B from Y, and C from Z) and the first instruction that follows will be executed (displaying the `negative value` text) and then jumping to the instruction with label 30. So the value transfer will also be from the subroutine to the calling program unit (no other options being specified by `INTENT`) and `RETURN` means ”normal” return.
If the arithmetic condition in the subroutine results in jumping to the instruction with label 54, the return to the calling unit will be done by activating the first reserved position in the list of notional arguments, which in the list of actual arguments corresponds to `*10`, consequently the first instruction executed after the return will be the one with label 10 (the text `null value` will be displayed after which it will jump to the instruction with label 30). So `RETURN 1`

	means return through the first reserved position. If the arithmetic condition in the subroutine results in jumping to the instruction with label 55, the return to the calling unit will be done by activating the second reserved position (due to the value 2 specified in `RETURN`) in the list of notional arguments, which in the list of actual arguments corresponds to `*20`, consequently the first instruction executed after the return will be the one with label 20 (the text `positive value` will be displayed after which the instruction with label 30 will continue). So `RETURN 2` means return through the second reserved position.

Functions:
n addition to the intrinsic functions existing in the Fortran language, it is possible to define different functions. There are several categories of functions: defined as external procedures (program units), defined as internal or module procedures (contained by other program units), defined as an statement (in a single specification expression). The use of functions is done by specifying the name and arguments (if a function has no arguments, then the name will be followed by empty brackets) within instructions. You can pass values to functions via arguments (as with subroutines, except that unlike subroutines, with functions the parentheses enclosing the arguments are mandatory, even if they are not arguments), but functions will return a result via their name, not their arguments! With this in mind, an expression calculating the result of the function must be mandatory in the definition of a function.
When defining a function, in addition to keywords specifying characteristics (ELEMENTAL, PURE, RECURSIVE, MODULE), the type of the function can also be specified (in the case of those defined as external procedures, only intrinsic types can be used). The syntax for defining a function as a procedure is as follows:

[*type*] FUNCTION *name* ([*arguments*])	Before the FUNCTION keyword, a function characteristic (ELEMENTAL, PURE, RECURSIVE) and a type (INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER, BYTE) can be specified, and the arguments are optional (they are specified only if value transfer between the calling program unit and the function is desired), but the argument delimiting parentheses are mandatory. Arguments are considered notional in the sense that at the time of function definition their actual values are not known, only their type. In the case of self-reference (RECURSIVE), the definition must be completed at the end of this line with RESULT(*name_r*).
[*specifications*]	In all program units, entity specifications must precede executable instructions.
[*executable statements*]	They may contain ENTRY specifications (for defining entry points) and RETURN instructions (for returning to the program unit from which the function was called). **It must also contain an expression to obtain the result of the function**!
[CONTAINS *Internal procedures*]	Several internal procedures (subroutines and functions) can be defined in succession, but only in the case of a function defined as an external procedure. For internal procedures this section cannot appear.
END [FUNCTION [*name*]]	The final marking must be at least the END keyword for functions defined as an external procedure. It may also be followed by the keyword FUNCTION, possibly also by *name*.

In the case of internal procedures the end marker must contain at least both keywords `END FUNCTION`.

Invoking a function is done by using the name and actual arguments (if any) in a statement, in the form:

… *name* ([*arguments*])	Arguments are specified if they exist in the function definition (if they do not, then the parentheses will be empty). On call these arguments are considered effective, in the sense that at the time the function is invoked, along with their type and values, they are usually known. The order of the effective arguments (at function invoking) must match the order of the notional arguments (from function definition) as type, but different names may be used.

Examples:	Explanations:

`! main program`
`   INTEGER on2`
`10  PRINT *,"numar: "`
`   READ *,i`
`   IF(i==0) STOP`
`   PRINT *,on2(i)`
`   GOTO 10`
`END`
`! on2 function definition`
`INTEGER FUNCTION on2(nr)`
`on2=nr/2`
`END`
	Main program that will invoke the ON2 function defined as an external procedure, and display the result of this function for the value of the effective argument I (on the monitor). The program will stop only if the value read for I is null.
	When the function is invoked (to print the result) it will transfer the value of I to NR (from the function definition), and the returned result will be obtained by the name of the ON2 function.
	It can also be seen that the `END` marking for the function defined as an external procedure is sufficient.

`! main program`
`   INTEGER on2`
`10  PRINT *,"numar: "`
`   READ *,i`
`   IF(i == 0) STOP`
`   PRINT *,on2(i)`
`   GOTO 10`
`CONTAINS`
`! on2 function definition`
`FUNCTION on2(nr)`
`INTEGER on2`
`on2=nr/2`
`END FUNCTION on2`
`END`
	Previous example modified by making the function definition an internal procedure. Although it was possible to define the function by specifying the type `INTEGER` as in the previous case, it was chosen to specify the type separately.
	In this case the function end marking must also contain the keyword `FUNCTION` (next to `END`), the mention of the function name is optional there.

`! main program`
`   INTEGER on2,nr`
`! on2 function definition`
`on2(nr)=nr/2`
`! executabile statements`
`10  PRINT *,"numar: "`
`   READ *, i`
`   IF(i==0) STOP`
`   PRINT *,on2(i)`
`   GOTO 10`
`END PROGRAM`
	The previous example modified by transforming the function definition into a statement. It can be seen that in this variant the function name is followed by the notional argument in the definition line. The statement function definition is not an executable instruction, so it must appear in the specification area.

`PROGRAM factorial`
`INTEGER f,i`
`PRINT *,"i: "`
`READ *,i`
`PRINT *,"factorial of ",i,":",f(i)`
	A "classic" example of a function defined as a self-referring (recursive) procedure for calculating the factorial value of a number.

```
END
! recursive function definition
RECURSIVE FUNCTION f(i) RESULT(fa)
INTEGER f,fa
IF(i==1) THEN
   fa=1
ELSE
   fa=i*f(i-1)
ENDIF
END
```

Note that in this case, since RECURSIVE is specified, the specification RESULT (*name_r*) is also mandatory, *name_r* being the name of the function used for self-referencing (recursion) in the description. Although the function is named F, the name FA (the one specified for RESULT) is used for the calculation of the result of the function in its definition.

```
PROGRAM array_function
PRINT *,'a,b,c: '
READ *,a,b,c
PRINT *,func(a,b,c)
CONTAINS
! internal procedure
     FUNCTION func(x1,x2,x3)
     DIMENSION func(3)
     func(1)=x1
     func(2)=x2
     func(3)=x3
     END FUNCTION
END
```

A quick example of a function defined as an internal procedure and as an array. Although a function normally returns a single result (a single scalar value), in the case of defining it as an internal procedure, you can also create an array function (which will return a result as an array).
When the function is called, the arguments are passed in the specified order (X1 corresponds to the value in A, X2 corresponds to B, X3 corresponds to C) and the result is obtained by the name of the function (in this case, 3 different values). For each position in the function FUNC - array with 3 positions: FUNC(1), FUNC(2) AND FUNC(3) - the results are calculated. The first item in the FUNC array will take the value from X1, the second will take the value from X2 and the third will take the value from X3. Thus, when the result of FUNC(A,B,C) is printed, 3 consecutive different values will be displayed on the monitor.

Modules

These are program units that usually contain specifications and definitions that can be made accessible to other program units. They may also contain explicit interfaces (via interface blocks) to an external procedure or DUMMY procedure. The syntax of a module definition is as follows:

MODULE *name*	It is obligatory to give a *name*, it is global and it is unique!!
[*specifications*]	Cannot contain: AUTOMATIC, ENTRY, FORMAT, INTENT, OPTIONAL and no defined or intrinsic functions.
[CONTAINS *Module procedures*]	Executable statements can only occur within module (internal) procedures.
END [MODULE [*name*]]	It is sufficient to specify only the END keyword (if the module has not been named, there is no *name* to specify).

A module can only be used after compilation by specifying its use in the target program unit with the:
USE *name* (where *name* is the name by which the module was defined).

Exemple:

```
MODULE prim
   INTEGER,PARAMETER :: A,B
   REAL E22(5,5)
END
! using it in program units
SUBROUTINE P21
   USE prim
  …
```

Explicații:

A module defined as PRIM that contains only a few data specifications.

If it is written in the same source file as the program unit that will use it (e.g. subroutine P21 and function FU33), the module must be placed before the program unit, so

`END`
`FUNCTION FU33(A,X)`
`   USE prim`
`  …`
`END`
	that when the contents of the source file are compiled, by the time `USE PRIM` is reached, the module has already been compiled!

`MODULE cal_M`
`TYPE element`
`   PRIVATE`
`   INTEGER C,D`
`END TYPE`
`  …`
`INTERFACE`
`   FUNCTION calculate(R)`
`   REAL :: calculate`
`   REAL,INTENT(IN) :: R(:)`
`   END FUCTION`
`END INTERFACE`
`END MODULE cal_M`
	A module called CAL_M, in which a derived type called `ELEMENT` (default `PUBLIC`, so visible from all program units) has been defined, with the C and D components declared `PRIVATE` (visible only from the module).
	
	After specifying the derived type, there follows an interface block for the CALCULATE function, where the argument R is a vector used only for input (passing values to the CALCULATE function), both the CALCULATE function (the value resulting from the expression specified elsewhere in the function definition) and R being of type `REAL`.

An older and more complex example (from https://www.star.le.ac.uk/~cgp/f90course/f90.html#tth_sEc6) with a module that could be used to simulate the operation of a console window (VT100 or X-TERM window) controlled by ESC (ASCII) sequences, similar to ANSI.SYS in DOS, also containing module procedures:

```
MODULE vt_mod
  IMPLICIT NONE
! specifying the code for <ESC> as a constant value named ESC
  CHARACTER(1),PARAMETER :: esc=ACHAR(27)
! initialising variables for 80 columns and 24 rows on the screen
  INTEGER,SAVE :: nr_c=80,nr_r=24
  CONTAINS
! clear the display and move the cursor to the top left
  SUBROUTINE clear_disp
   CALL write_str(esc//”[H”//esc//”[2J”)
  END SUBROUTINE clear_disp
! set the new width to 80 or 132 columns
  SUBROUTINE set_w(col)
   INTEGER, INTENT(IN) :: col
   IF (col>80) THEN
   ! switch to 132 columns
     CALL write_str(esc//”[?3h”)
     nr_c=132
  ELSE
   ! switch to 80 columns
     CALL write_str(esc//”[?3l”)
     nr_c=80
   ENDIF
  END SUBROUTINE set_w
! get the actual width
  SUBROUTINE get_w(col)
   INTEGER,INTENT(OUT) :: col
   col=nr_c
  END SUBROUTINE get_w
! for internal use only
  SUBROUTINE write_str(string)
   CHARACTER,INTENT(IN) :: string
   WRITE(*,"(1X,A)",ADVANCE="NO")string
  END SUBROUTINE write_str
END MODULE vt_mod
```

This module can be used with the following specification variants (examples):

`USE vt_mod`	Use the entire contents of the module.
`USE vt_mod,ONLY:clear_disp`	Use only the module procedure CLEAR_DISP from procedures.
`USE vt_mod,wide=>get_w`	Use the whole module, but temporarily replacing the name of the module procedure GET_W with the new name WIDE.

Block Data units

These program units are intended to provide the possibility of initialising entities in common blocks (shared memory areas), but are considered obsolete because the COMMON specification has been removed since the Fortran 90 standard (but the G95 compiler supports it and in the absence of this specification will issue a warning message). Blocks contain entity specifications, possibly with initialization of some data (not in the case of POINTER and TARGET), but cannot contain executable instructions. The syntax of a data block definition is as follows:

BLOCK DATA [*name*]	Giving a *name* is optional, mostly for the clarity of the source files. If more than one block is defined, only one can be unnamed.
[*specifications*]	May contain: COMMON (depending on the compiler), INTRINSIC, STATIC, USE (only for named constants), DATA (for data initialisations), PARAMETER (for constants), TARGET and POINTER (but no initialisations), DIMENSION (for arrays), *type* (keywords for intrinsic data types), TYPE (with user-defined type names and definitions), RECORD and STRUCTURE (for records), EQUIVALENCE, IMPLICIT, SAVE.
END [BLOCK DATA [*name*]]	It is sufficient to specify only the END keyword (if the data block has not been named, then there is no *name* to specify).

Example:	Explanations:

`! main program`
`  CHARACTER(6) Actor`
`  COMMON /zona1/a,b,c,d,Actor`
`  INTEGER :: s1=2`
`  PRINT *,"s1:",s1`
`  PRINT *, a,b,c,d`
`  PRINT 2,Actor`
`   …`
`2 format("Actor: ",A)`
`   …`
`END`
	The COMMON specification is used to designate by name and composition a common memory area, addressable from any program unit (by specifying the common block name). The syntax for specifying a common block is:
	COMMON /*name*/*entity_list*[[,]…]

`! data block for initialisation`
`BLOCK DATA`
` DIMENSION x(4)`
` COMMON /zona1/x,name`
` DATA x/3*1.,5/`
` CHARACTER(6) :: name="Adrian"`
`END`
	Definition of a data block specifying and initialising some data. Due to the COMMON specification, the entities X (4-digit vector) and NAME, which are part of the common block called ZONE1, will occupy the same memory area as A, B, C, D and ACTOR (containing the string Adrian), provided that the storage size of the corresponding entities is identical.

Dynamically allocated memory

As described in the section on arrays, they can be declared by specifying a type or by specifying DIMENSION, COMMON (depending on the compiler), ALLOCATABLE, also POINTER or TARGET (only since Fortran 95). A known (and unchangeable during the execution of the program) size in computer memory is allocated to an array by the type specification alone, or by using DIMENSION with the bounds set corresponding to each extent (rank) of an array. This size is a maximum size and need not be used in full (fewer positions in the table can be used). If memory usage is to be optimised (less space means fewer addresses, resulting in higher speed), then it would be desirable not to allocate unused space. This can be achieved by dynamically allocating memory at runtime, specifying only the size of the tables that are really needed. Thus, when the program is written, only the number of extents (or rank) of the array is reserved into memory (creating the possibility of generating addresses for possible locations), and the actual allocation of memory space to the array takes place only when the statements that require this have been reached. Of course, the programmer has to bear in mind that in this way it is not the operating system that manages the memory allocated to the array, but the program, so the release of this memory must also be controlled by statements. If this aspect is ignored, then after each execution of the program, areas of memory will remain occupied and uncontrolled (this phenomenon is called ”memory leakage”), which, after repeated executions, can lead to the working memory being filled up, making it difficult or even blocking the operation of the computer.

Dynamically allocated memory can be achieved in one of three ways:

- Allocatable arrays (using the ALLOCATABLE specification),
- Pointer or target arrays (via the POINTER or TARGET specification – since Fortran 95),
- Automatically allocated arrays (by passing data to procedures).

<u>Allocatable arrays:</u>
When using the ALLOCATABLE specification, the rank (number of extents) of the array must be reserved accordingly, and the lower and upper bounds (limits) of the array can be set at any time within the program (if they have not already been set). The ALLOCATABLE specification cannot be combined with the COMMON, DATA, EQUIVALENCE or NAMELIST specifications.
Allocatable arrays can only be used between procedures if memory has been allocated for them beforehand (limits have been set for each rank), but to avoid ”memory leaks” the space allocated for them must be freed (deallocated) before the end of the procedure in which memory has been allocated. Multiple allocations to an array are not allowed (to test the allocation status, the intrinsic ALLOCATED function can be used, which returns the logical value .TRUE. if the array already has allocated space). The DEALLOCATE intrinsic function can be used to free the allocated memory of an array, and the ALLOCATE intrinsic function can be used to allocate memory.

Example:	Explanations:
`REAL,ALLOCATABLE :: v(:),m(:,:)` …	Two allocatable arrays have been declared, the vector V with rank 1 (reserved by the : character) and the matrix M with rank 2 (i.e. 2 dimensions).
`ALLOCATE(v(10),m(0:9,-2:7))` …	10 positions have been allocated for the vector V and 10x10=100 positions for the matrix M (from 0 to 9 inclusive for the first extent and from -2 to 7 inclusive for the second extent). Remember to close the brackets for the ALLOCATE function!
`DEALLOCATE(v,m)` …	Free the memory allocated to the previous 2 arrays.

`IF (ALLOCATED(m)) THEN`
`      DEALLOCATE(m)`
`ENDIF`
…
	Use a logical expression to check the state of array M to avoid double allocation (not allowed). The memory space is released (by DEALLOCATE) only if the intrinsic function ALLOCATED indicates (by returning the

	logical value .TRUE.) that there is already previously allocated memory space. If the intrinsic function ALLOCATED returns the logical value .FALSE., it means that no memory space is allocated to the specified array and therefore there is no need to release the memory (the intrinsic function DEALLOCATE is ignored).
`ALLOCATE(m(3,3))` …	Allocate a new size of memory to the M array, this time 3x3=9 positions.
`DEALLOCATE(m)` …	Release the memory allocated to the M array before the end of the programme unit.

Pointer/Target arrays:

A POINTER does not contain data, but points to a scalar or array where data can be stored. The scalar or array to which a POINTER points must have the TARGET attribute. Unlike allocatable arrays, a POINTER (or TARGET) array can be passed to a procedure without prior allocation of memory space. The memory for such an array is not actually allocated until the program is executed. The syntax for specifying these arrays is similar to that of allocatable arrays, with the exception that POINTER arrays usually require an explicit interface (for internal procedures, the interface is known). Since the specification of POINTER (and TARGET) arrays is only possible from Fortran 95 (similar to the use of the ALLOCATABLE specification already presented), in the case of Fortran 90 variants such arrays can be created by the derived type specification (which will be exemplified in the following).

Example:	Explanations:

`TYPE p_array`
` REAL,DIMENSION(:),POINTER :: tp`
`END TYPE`
…
	A derived type named P_ARRAY has been declared, containing a component of type REAL with the attribute POINTER as a vector (array of rank 1) named TP.

`TYPE(p_array),ALLOCATABLE :: vp(:)` …	This derived type P_ARRAY is used to declare another allocatable array called VP, also in vector form (array of rank 1). This means that each element of VP will have in its composition an array of type REAL with the attribute POINTER in the form of a vector (array of rank 1) called TP.

`READ(*,*) n,m`
…
`ALLOCATE(vp(n))`
`DO i=1,n`
`      ALLOCATE(vp%tp(m))`
`ENDDO`
…
	Assuming that the values of the scalar entities of type INTEGER N and M are known (in the adjacent example by reading), the desired storage space (in the adjacent example N positions) for the array VP, respectively the desired storage space (in the adjacent example M positions) for each component of type TP in VP. Thus, each element of the POINTER VP array will have M positions, which means that the VP array will have a total of NxM positions.

`DEALLOCATE(vp)`	Release of the allocated space, as in the case of allocatable arrays.

Automatically allocated arrays:

Automatically allocated arrays are variables allowed only within procedures (subroutines and functions), and the lower and upper bounds for each pre-reserved extent (reserved rank) are set at the time of the procedure call. These arrays cannot be initialised (their elements cannot contain initial values) and values cannot be passed through such arrays between procedures.

Example:	Explanations:

`SUBROUTINE points(nr,pos)`
`INTEGER,INTENT(IN) :: nr`
`REAL,INTENT(OUT) :: pos`
`REAL :: zone(nr),zone_2(2*nr)`
  …
	A subroutine named POINTS has been specified with arguments NR and POS (whose value is known at the time the subroutine is entered). The argument NR is of type `INTEGER` and is only used as a value to pass to the POINTS subroutine. POS is of type `REAL` and is only used to pass a value from the POINTS subroutine to the program unit calling the subroutine.
	At the time of activation of the POINTS subroutine (when the known NR value is passed to the subroutine), the `REAL` arrays named ZONE and ZONE_2 are automatically allocated memory space (defined size).

`PROGRAM array_function`
`ALLOCATABLE X(:)`
`PRINT *,"n: "`
`READ *,n`
`ALLOCATE(X(n))`
`PRINT *,"the ",n," values: "`
`READ *,(X(i),i=1,n)`
`PRINT *,func(n,X)`
`DEALLOCATE(X)`
`CONTAINS`
    `FUNCTION func(k,X)`
    `DIMENSION func(k),X(k)`
    `DO i=1,k`
    `func(i)=X(i)`
    `ENDDO`
    `END FUNCTION`
`END`
	A more complex example with a function defined as an internal procedure and as an automatic array (extending an earlier example from the function walkthrough). The array X passed to the FUNC function (along with the size of N) benefits from dynamic memory allocation. The memory allocated to the array X is freed before the program ends. When the function is called, the arguments are passed and the result is obtained by the function name (in this case, N different values).
	The function (array) will automatically have K positions (corresponding to the N values passed at the time of the call). Each element in the FUNC array receives the value of the corresponding position in the X array.