F. Zsongor GOBESZ

C. Mihai MOCAN

Programming in Fortran G95 for beginners

(Theory for lab works)

U.T.PRESS
Cluj-Napoca, 2025
ISBN 978-606-737-804-7

F. Zsongor GOBESZ C. Mihai MOCAN

Programming in Fortran G95 for beginners

(Theory for lab works)

U.T.PRESS
Cluj-Napoca, 2025
ISBN 978-606-737-804-7

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Conf.dr.ing. Ciprian Pavel Oprișa
Conf.dr.ing. Adrian Coleșa

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2025 Editura U.T.PRESS
Reproducerea integrală sau parţială a textului sau ilustraţiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-804-7

CONTENT

Foreword

Creating a program on a computer makes sense only in the situation where the amount of calculations would exceed manual possibilities. Such situations can arise when many similar problems need to be solved, complex and laborious calculations need to be performed, or large amounts of data need to be processed. Creating a computer program requires some specific knowledge. The efficiency and performance of the program will depend not only on the platform on which it will run, but also on the knowledge and experience of those who collaborate to create it. In the following, we will refer to the creation of simple applications, using a high-level programming language (close to natural language), namely Fortran. The programming stages are generally summarised by 3 phases: conception (the logical level of problem solving with the development or choice of the appropriate algorithm), coding (transcription of the algorithm into a machine-accessible programming language, to create the program), testing and implementation (verifying correctness with test data and fine-tuning the program). These stages can be covered by the following steps:

- recognise and define the problem (to know the initial data);
- selection and description of the proposed method (how to obtain the results);
- translate the description of the method into a programming language (create the source file);
- making the program by compiling (translating the source file into machine code, generating the object image) and link-editing (filling the object image with parts from the language library, generating the executable file);
- running and testing the created software.

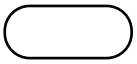
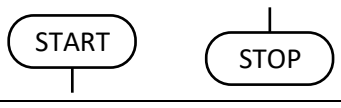
The contents of this handbook have been written to guide first year Civil Engineering students in the subject of "Computer Programming and Programming Languages", but it can also be useful for beginners who want to get started with the Fortran language. The first part illustrates some concepts regarding the use of structured flowcharts to describe certain methods, followed by a brief overview of the Fortran language and mention of some freely available development environments. Information on writing source files is followed by a more detailed presentation of the basic aspects of the Fortran 95 language syntax (using the G95 compiler) for writing simple programs. Finally, there are some illustrative exercises (with transcriptions of flowcharts and source files).

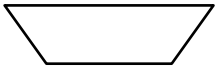
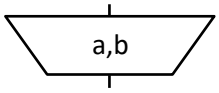
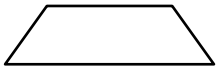
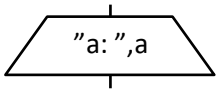
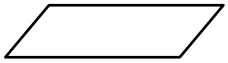
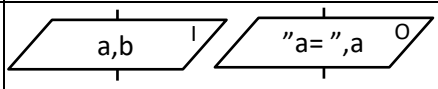
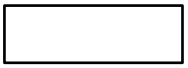
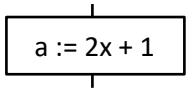
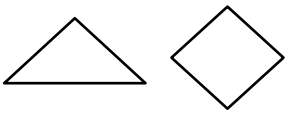
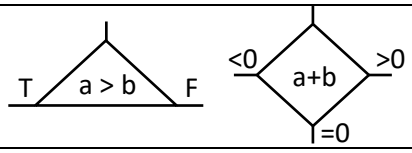
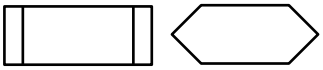
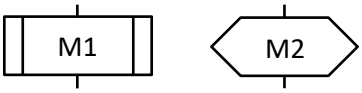
Structured flowcharts

A flowchart (logic scheme) is a graphical tool that can be used to represent the steps of an algorithm in the form of blocks (symbols) connected by lines. In order to use this tool in a structured way, some principles should be known, such as:

- Flowcharts are drawn and read from top to bottom (exceptions are marked with arrows).
- Blocks can have only one entry point (except for the start block, which has only one exit), and the number of exit points depends on the type of block: modules and those representing input / output operations or attribution have only one exit, conditional ones depending on the type of expression (logical ones have 2, for true or false; arithmetic ones have 3, for negative, zero or positive), and the ending block has no exit point.
- In the construction of structured flowchart, sequentially chained module-like parts (with a single entry point and a single exit point) are used as far as possible.

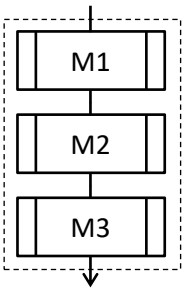
A module can contain anything, provided it has only one entry and one exit. The contents of a module must be detailed separately, where appropriate. The symbols used in the composition of flowcharts are shown in the following table:

Use	Symbol	Variants (examples)
Start block (marked START) or end block (marked STOP)		

Input block (marked with the elements to be read)		
Output block (marked with elements to be written)		
Alternative to input/output block (note whether input or output)		
Attribution block (contains a single expression, whose value is attributed to the variable on the left)		
Decision blocks (output is branched according to the type of expression evaluated, conditions are also marked)		
Module or procedure block (marked with module name)		
Inner connector (for interrupt or continuation within the same page, marked correspondingly)		
External connector (for interrupt or continuation between different pages, with matching mark)		

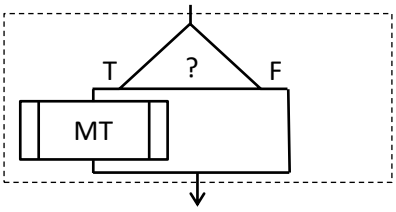
The principles of structured programming were published by Edsger W. Dijkstra[1], exemplifying the following 3 levels in order of complexity:

1. Sequential chaining (the exit from one module will be the entry to the next module):



2. Decision-making structures (only one branch is used when traversing):

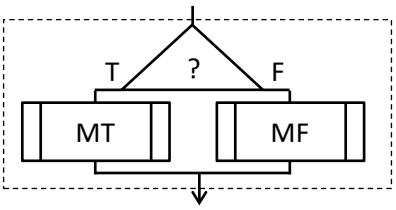
   2.1. Simple logical decision with void branch, the false case branch being empty.



”if ? do MT”

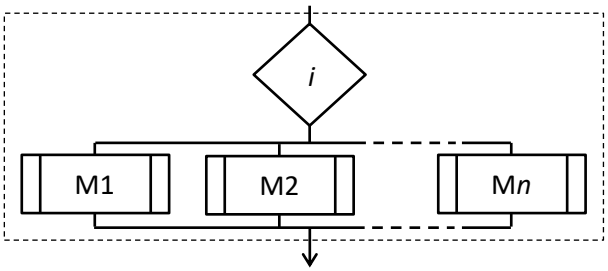
[1] E. W. Dijkstra: “Notes on Structured Programming” (Report) 70-WSK-03, Technical University of Eindhoven, The Netherlands, 1970. via E.W. Dijkstra Archive. Center for American History, University of Texas at Austin, USA. https://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF

2.2. Common logical decision.



”if ? then MT else MF”

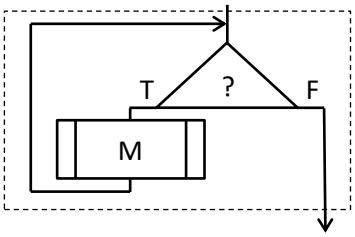
2.3. Generalized decision or choice.



”case *i* of (M1; M2; …; M*n*)”

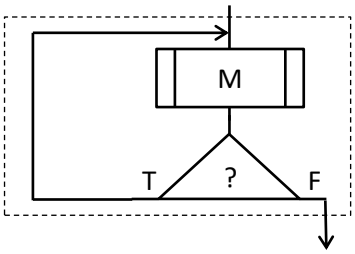
3. Repeating structures (loops):

3.1. Preconditioned loop.



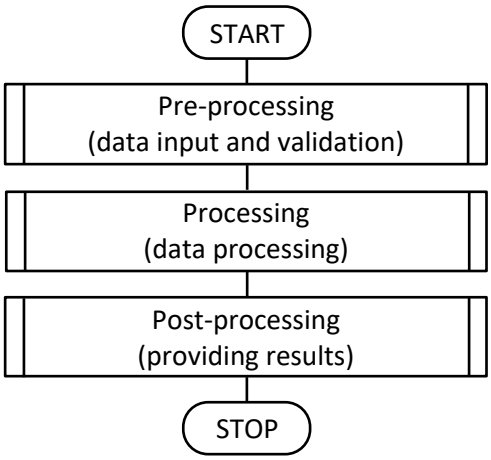
”while ? do M”

3.2. Postconditioned loop.

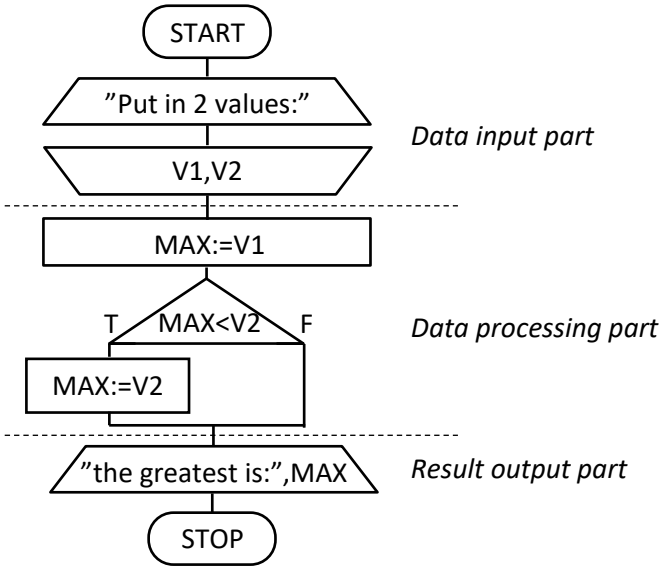


”repeat M until ?”

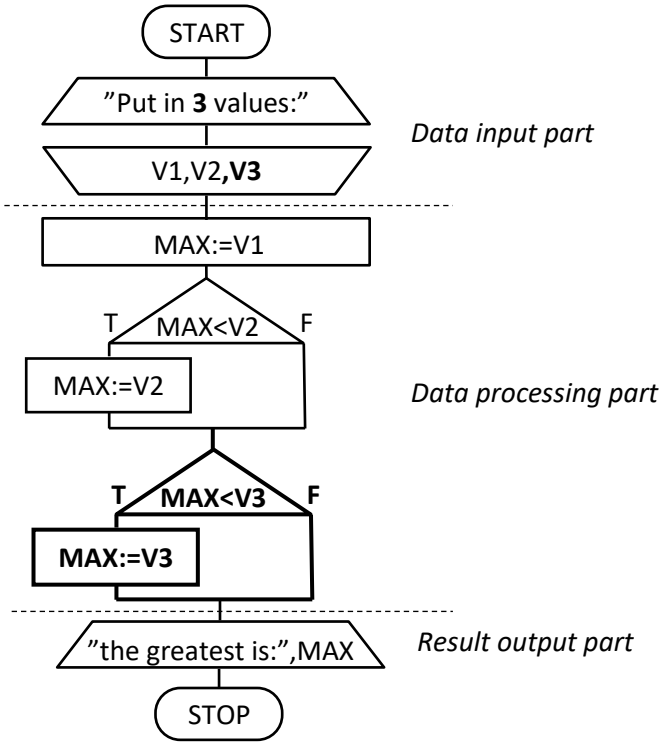
In most programming languages, there are statements (instructions) corresponding to these basic structures. To make it easier to understand (verify) the content of source files and to reduce the execution time of statements, it is recommended to use structures starting from the simplest to the most complicated ones. Applying the principle of structuring, the algorithm of a program can be described by a generic flowchart consisting of 3 modules:



This makes it easier to split up and test applications across development teams, and in case of updates or changes to source files, the separate approach to modules makes tasks easier. Here is an example for displaying the maximum value from the contents of two variables:



As can be seen, if this algorithm is to be modified to compare multiple values, the data input part will be slightly modified, the processing part will be completed by replicating the simple logical decision structure with a void branch, and the result output part will remain unchanged. Here is a variant for 3 values (the modifications are marked in bold):



Briefly about Fortran

The first version of this programming language was created by a team from IBM under the leadership of John W. Backus, being released in 1957 under the name "IBM Mathematical Formula Translating System" (in short: FORTRAN, from the combination of FORmula TRANslation words), this being the first high-level programming language (close to natural language). In 1958 IBM published a revised version, called FORTRAN II, which provided support for procedural programming by introducing specifications for subroutines and functions. Because of its popularity, IBM decided to remove the features that limited the use of the language on IBM systems, and in 1964 released a variant called FORTRAN IV that could run on any computer. The FORTRAN 66 version appeared in 1966, as a result of the standardization carried out by the American Standards Association (ASA, the precursor of ANSI), being the first programming language defined by a standard. The "ANSI FORTRAN" committee (known as "X3J3") began developing a new version in 1969, and the result was FORTRAN 77, the most widely used version of the language.
The next version was expected to be released in the 1980s (Fortran 8X), but it was released only in 1991, introducing the free form and became known as Fortran 90, opening a path for HPF (High Performance Fortran). In 1997, the standard for Fortran 95, the first object-oriented version, was published.
Compared to C++ (an object-oriented language that supports polymorphism and inheritance), Fortran has introduced some similar features (through modules and derived types), but has no automatic inheritance. On the other hand, Fortran is easier to learn and use for scientific computing than C++, having native support for complex values, multidimensional arrays, etc., which C++ lacks. Fortran 2003 represents a significant turn in object-oriented features, also ensuring interoperability with C/C++, and in 2010 Fortran 2008 was released with new provisions (sub-modules, co-arrays, the contiguous attribute, etc.) and having implemented parallel processing with distributed memory. After Fortran 2018, which was a revision of the previous version with additional support for parallel processing, Fortran 2023 is the latest standardized version with even more features.
Here is a quoted fragment from the FAQ section of https://fortran-lang.org/, for those interested in the usefulness of this language: "*What is Fortran used for? Fortran is mostly used in domains that adopted computation early–science and engineering. These include numerical weather and ocean prediction, computational fluid dynamics, applied math, statistics, and finance. Fortran is the dominant language of High Performance Computing and is used to benchmark the fastest supercomputers in the world*."

Creating programs

In order to create a Fortran program, you need a text editor (preferably ASCII) and a suitable package containing a compiler plus link editor (builder). The G95 compiler includes some features of Fortran 2003 in addition to Fortran 95 and can be installed on Windows with MinGW (by running the g95-MinGW.exe file), although its development was discontinued in 2013. After the installation, you can create a source file in the C:\MinGW folder (such as test.f95), containing the following two lines:

```
print *,"OK"
end
```

Make sure to save the file with the ".f95" extension. To test the functionality of G95, open a console window and type:

```
cd C:\MinGW
g95 -c test.f95
```

If nothing is displayed, then the object image (test.o file) was created without errors after compiling, from which the executable file can be created by the following command:

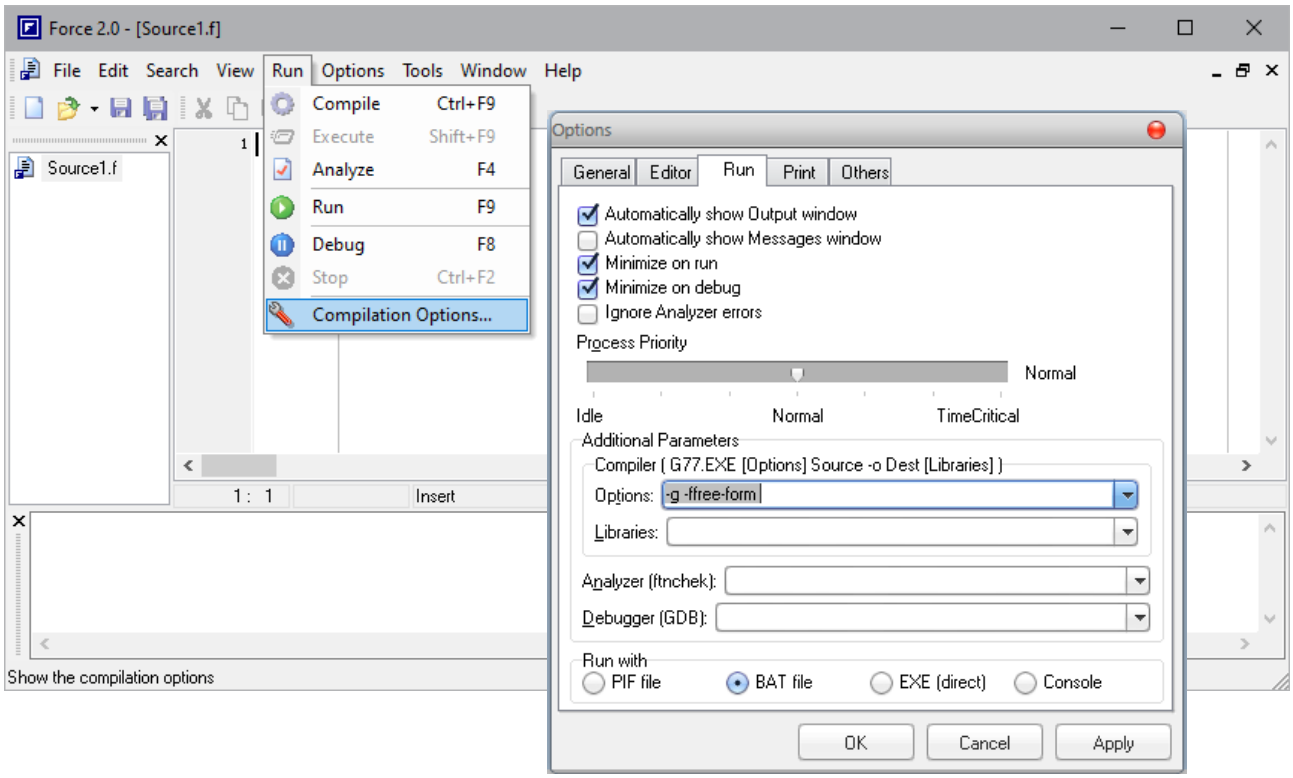
```
g95 test.f95 -o test.exe
```

To run the created program (the test.exe file) type `test` and the `OK` letters should appear on the screen.
If you want to learn more about the options of the G95 compiler and the facilities it provides, take a look at the G95Manual.pdf file in the C:\\MinGW\doc folder.
For other operating systems, you can use GNU Fortran (GFortran 95), which is included in the GCC package.

However, it is more convenient to use an integrated development environment (IDE), and for Windows, our choice is the Force package with the G95 compiler included. To get it, go to the project's web site[2] and in the "Downloads" section select the version marked in the figure below.

Filename	Size	Description
Force209GFortranSetup.exe	10.6 MB	Force 2.0.9 plus GNU Fortran (GFortran)
Force209G95Setup.exe	3.55 MB	Force 2.0.9 plus G95 Fortran (G95)
Force209G77Setup.exe	2.03 MB	Force 2.0.9 plus GNU Fortran 77 (G77)
Force3beta3Setup.exe	2.19 MB	Force 3.0 b3 plus GNU Fortran 77 (G77)

After installation, the G95Manual.pdf file will be located in the C:\Program Files (x86)\Force 2.0\doc folder. Before the first run on Windows 10 and 11, the properties of the C:\Program Files (x86)\Force 2.0 folder must be changed to give full control to the administrator (and possibly the current user), otherwise the initial settings will not be saved. To be able to use the free form of the source file in Force, after installation you will need to add the "`-ffree-form`" option (as illustrated) in the Compilation Options and experiment with the convenient running mode (through batch command file, run directly the executable, or in a console window).



Of course it is possible to choose other variants, such as Code::Blocks with Fortran[3], Geany[4] (no Fortran compiler included, but you can install G95 separately on Windows, or GFortran on other operating systems), etc. There are even variants that run online through a browser (such as GDB online[5], myCompiler[6], Ideone[7], Jdoodle[8], and many more). You can also visit https://fortranwiki.org/ and https://fortran-lang.org/ for additional options and resources.

[2] Force Fortran - The Force Project. https://force.lepsch.com/
[3] Code::Blocks IDE for Fortran | CBFortran. https://cbfortran.sourceforge.io/
[4] Geany - The Flyweight IDE. https://www.geany.org/
[5] GDB Online Fortran Compiler. https://www.onlinegdb.com/online_fortran_compiler
[6] myCompiler – Create a new Fortran program. https://www.mycompiler.io/new/fortran
[7] Online Compiler and IDE. https://ideone.com/
[8] Online Fortran Compiler. https://www.jdoodle.com/execute-fortran-online/

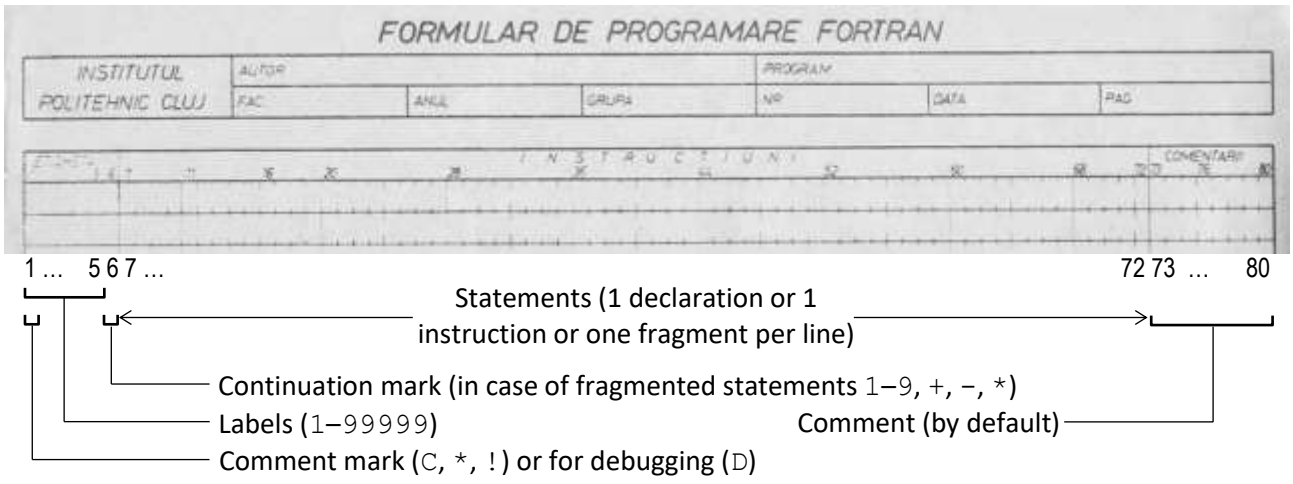
Source file structure

A source file can contain one or more program units (these will be presented later), or fragments of them (in the form of sections). The source file can be created with any text editor, provided that it results in character content (aka ASCII file).
The character set usable in the Fortran language contains the alphanumeric characters (the 26 upper and lower case letters of the English alphabet: `a–z`, `A–Z`, and the digits: `0–9`), plus 4 symbols for arithmetic operations (`+`, `–`, `*`, `/`) and a set of special characters (blank or space, horizontal tab, comma, period, apostrophe, open and closed parentheses, as well as the following characters: `=`, `$`, `&`). The Fortran 90 language extended this list with the following allowed special characters: `_`, `!`, `:`, `;`, `"`, `%`, `<`, `>`, `?`, `^` and `#`. The comma has the role of separating elements within a list, while the period (dot) is the decimal separator.
Symbolic names are used to name variables, different program parts, and to identify functions. If the conventions of older versions of the language allowed the use of only 8 characters (consisting of alphanumeric characters and the special character "`$`"), Fortran 95 allows the use of 31 characters (consisting of alphanumeric characters, the special character "`$`" and the special character "`_`"), but the first character must always be a letter. Program unit and section names are considered global and must be unique throughout the source, and entity names must be unique within the same program unit. The Fortran language is not case sensitive for symbolic names. The editing mode of the source file can be in fixed form (Fortran 77), tabular form or free form (the latter being introduced by Fortran 90 and allowed by following versions of the language).

Regardless of the horizontal structure (fixed, free or tabular form), the vertical structure of a source file must respect the following sequence of specifications: declarations (concerning the program unit, the entities used), body (containing the statements to be executed at runtime) and final marker. If the source file contains only a segment of a program unit, any of the three parts (declarations, body, final marker) may be missing, but the order must be respected. In such cases, the contents of such a source file shall be included (using the `INCLUDE` specification) in the source file corresponding to the program to be built before compilation.

Fixed form

This corresponds to the editing structure based on punched cards and old template sheets (like the one in the image), considering 80 characters as the maximum length of a line (record), having the following structure (below the image of the template sheet, the relevant column numbers and the content allowed for each field are marked):



The labels are integers of at most 5 digits, with a reference role within the program section, marking the statements before which they appear (in the respective line). Their use is optional and subject to some restrictions (only lines with executable statements can be labelled and labels cannot exceed the range of

columns 1–5). For a label to be valid, its value must be in the range 1–99999. If a line should be marked as a comment, the letter "C" or the character "*" (respectively "!" starting with Fortran 90) should be written in the first column, which will cause the structure and content of the line to be ignored during compilation. Some compilers also allow the use of the character "D" to mark the current line in the first column as a comment, thus allowing the optional compilation (interpretation) of these lines in case of debugging the source.
In Fortran 77 and earlier, only one statement was written per line, but since Fortran 90 it is allowed to write more than one statement on a line (using the character ";" to separate them).
If the space between columns 7 and 72 of the current row is not enough to write the desired statement, it can be extended by marking in column 6 on the following rows the continuation of the previous ones, by numbers (only from the range 1–9), letters or by one of the characters: +, −, * (starting with Fortran 90, any character except the 0 digit can be used). The number of continuation lines allowed also depends on the compiler chosen. Fortran 77 allowed 99 fragments (1 initial line and 98 continuation lines), but the Fortran 90 standard allows only 19 fragments in fixed form (and 39 fragments in free form), while Fortran 95 allows up to 90 continuation lines in fixed form (and only 31 continuation lines in free form). Some compilers allow the line interpretation range to be extended up to column 80 (even 132, starting with Fortran 90), but as standard any content in the range of columns 72-80 is considered comment by default and as such is ignored by the compiler.

Free form

Does not have the restrictions described above, the statements are not limited to any particular fitting on the line columns, each line can contain up to 132 characters. Instead, spaces are significant, and in some cases act as separators (for names, constants, keywords, or as spacers between labels and statements). This form has only been introduced since Fortran 90 (but Fortran 90 also supports fixed and tabular formats). In the free form, the comment is indicated by the character "!" (starting from any column) or by the letter "C" written in the first column (beware of specifications and names starting with this letter, do not write them from the first column), while the "&" character marks the break of a statement (at the end) which will be continued on the next line. It is allowed to write more than one statement on a line if they are separated by the "**;**" character (which is ignored at the end of a line, of course).

Tabular form

It is actually a variant of both fixed and free form, and is so called because of the use of the horizontal tab character at the beginning of lines. If this <*Tab*> character is the first on a line, then the line contains a statement (declaration or instruction, or maybe a marker). If this first character is followed by a non-zero digit, the digit marks a continuation fragment of the previous line and must be followed by a space to separate it from the continuation content. The <Tab> character may be preceded only by a comment mark or a label. Line lengths must not exceed column 72 for fixed form and column 132 for free form.

> Notice: In the following chapters, syntax (writing rules) and examples are given where other characters are used. The square brackets are not part of the syntax, but mark the optionality of the included content, and the consecutive dots (...) mark repeatable elements. Italic sequences mark elements that replace content in the positions in which they appear. Given that the G95 Fortran compiler will be used in the classes, the specifications and statements presented will be for this variant and old ones, not part of the standard, will be marked grey.

Entity types

In Fortran, every entity has a type, either implicit, or explicitly declared. There are intrinsic types and derived types (defined by the programmer using intrinsic types or previously defined derived types). Intrinsic types are `INTEGER` (integer numbers), `REAL` (real numbers, with a decimal part), `COMPLEX` (complex numbers, viewed as pairs of numbers with a decimal part), `LOGICAL` (logical values, there are only two, the constants `.TRUE.` and `.FALSE.` constants), `CHARACTER` (character or string), and `BYTE` (8-bit value, used in older versions of the language). Explicit type declaration of entities can be done according to the following syntax:

type[(*kind*)][[, *attribute*]… : :] *entity_list*

The keywords for *type* are `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER` (in some versions of Fortran there is also `BYTE`), or `TYPE(`*name*`)`, where *name* refers to a type previously defined by the programmer. The *kind* specifies the number of bytes used for storage, optionally preceded by the keyword `KIND=`, in the case of `CHARACTER` type `LEN=` (or `*` in older syntax, without parentheses). This value depends on the type of entities, but there are also compiler dependent default values (usually 4 bytes for `REAL` type entities and 2 or 4 bytes for `INTEGER` type entities). Explicit values can be: `1`, `2` or `4`, eventually `8` for the `INTEGER` and `LOGICAL` types; `4` or `8`, (eventually `16`) for the `REAL` and `COMPLEX` types. Single characters and `BYTE` type entities are stored on 1 byte, so their storage length cannot be changed explicitly (if the *kind* is specified for `CHARACTER` type, it defaults to the number of characters in the string). `INTEGER(1)` and `LOGICAL(1)` type entities will also be stored on 1 byte.

The following can be specified as an *attribute*:

- `ALLOCATABLE` for arrays with dynamically allocated memory or `DIMENSION(`*limits*`)` for arrays with statically allocated memory (will be presented later),
- `EXTERNAL` for entities redefined by the programmer or `INTRINSIC` for entities predefined in Fortran,
- `INTENT(`*direction*`)` for input/output purpose (where *direction* can be `IN` for input, `OUT` for output, default `INOUT`),
- `PARAMETER` for constant values,
- `PUBLIC` for visible entities, `PRIVATE` for local entities (only accessible in the current program unit),
- `POINTER` for indicators or `TARGET` for targets,
- `OPTIONAL` for temporary entities, `SAVE` for stored entities.

If no *attribute* is specified, the `::` separator can be omitted (it only serves to delimit the list of keywords on the left, from the *entity_list* on the right of the specification).

For numeric entities there is an implicit rule regarding their type, which (of course) can be changed or cancelled with the following syntax of the `IMPLICIT` statement:

`IMPLICIT` *type* `(`*c*[, *c*]…`)` [, *type* `(`*c*[, *c*]…`)`]…

where *type* must be an intrinsic type specifier (or previously defined derived type) and *c* stands for a letter or range of letters in alphabetical order. To cancel any implicit rule, write:

```
IMPLICIT NONE
```

When cancelling the implicit rule, the types of all entities must be explicitly declared. According to the predefined implicit rule in Fortran, entities whose name starts with one of the letters I, J, K, L, M, or N will be of type `INTEGER`, and the rest will be of type `REAL`. Consequently, unless this rule is changed or cancelled, type declarations can be omitted while respecting the rule.

Examples:	Explanations:
`IMPLICIT INTEGER(B,f-H,k)`	All entities whose name begins with one of the letters B, F, G, H or K will be of type `INTEGER` (regardless of whether they are written in uppercase or lowercase).
`IMPLICIT REAL(n),COMPLEX(A-C)`	All entities whose name begins with the letter N will be of type `REAL`, and those whose name begins with one of the letters A, B, or C will be of type `COMPLEX`.

`IMPLICIT NONE` `INTEGER I,j,K` `REAL X,Y`	The implicit rule has been cancelled and the types of all entities must be explicitly defined. The one named I, J and K will be of type `INTEGER`, and the ones named X and Y will be of type `REAL`. As no attributes were specified, the `::` separator was omitted (only the right is list).
`REAL(KIND=8) Di,e33` *`! Equivalent to:`* `REAL(8) dI,E33`	The entities (variables) named DI and E33 are of type `REAL` and are stored on 8 bytes each (in older versions of Fortran, the type `DOUBLE PRECISION` was used in such a case). As you can see, it doesn't matter if the names of the entities are written in lower or upper case.
`COMPLEX(KIND=8) xC,Y1` *`! Equivalent to:`* `COMPLEX(8) Xc,y1`	The entities (variables) named XC and Y1 are of type `COMPLEX` and are stored on 8 bytes each (in older versions of Fortran, the `DOUBLE COMPLEX` type was used in such case). Since we are dealing with complex values consisting of pairs of values (the "real" part and the "imaginary" part), 16 bytes will actually be used for each entity.
`INTEGER(2),INTENT(IN) :: Q`	The Q entity is of type `INTEGER`, stored on 2 bytes and used only for input values. Since an attribute (`INTENT`) is also specified, it is mandatory to use the `::` characters to separate the left list from the one right list, even if there is only one element on the right.
`REAL,PARAMETER :: pi=3.14159`	The entity named PI is of type `REAL` and with a constant (unchangeable) value of 3.14159.
`EXTERNAL :: SIN`	The entity named SIN is declared as a variable with the default type `REAL` (because the name starts with the letter S). In this situation the SIN name will not be usable for the intrinsic trigonometric function in Fortran.
`REAL,POINTER,PRIVATE :: p,Q1`	The entities named P and Q1 will be pointers of type `REAL`, accessible only in the current program unit.

The definition of a derived type is done according to the syntax:

`TYPE` *name*
specifications
`END TYPE`[*name*]

Once defined, such derived types can be used to specify the type of entities by replacing the *type* keyword with `TYPE(`*name*`)` in the explicit type declaration. Reference to a component in such a derived type can be made using the % selector, in the form *parent*`%`*component*[`%`*subcomponent*...], as will be illustrated in an example below.

When the entity type is explicitly declared, initial values can also be attributed. The attribution can be done within the *entity_list* or separately, through the `DATA` statement. The syntax of this statement is as follows:

`DATA` *variable_list*`/`*value_list*`/`[[`,`]*variable_list*`/`*value_list*`/` …]

where for each entity from the *variable_list* there must be a corresponding value from the *value_list* (enclosed between "`/`" characters), in order of succession from left to right.

Examples:	Explanations:
`TYPE comp` `  CHARACTER(LEN=24) name` `  INTEGER day` `  CHARACTER(3) month` `  INTEGER :: year=2023` `END TYPE`	The derived type named COMP is defined as consisting of two character strings (NAME having 24 positions and MONTH 3) and two integers (DAY and YEAR, the latter being also initialized with value 2023). Note the optionality of the `LEN=` keyword, as it is not used for the MONTH string.

`TYPE(comp) r23,r24` `CHARACTER at,stars*3` `INTEGER m1,m2,m3` `DATA at,m1,m2,m3/"@",2*1,5/` `DATA stars/"***"/,r24%year/2024/` `DATA r24%month,r24%day/"AUG",12/`	Entities named R23 and R24 will have the type defined above. Declaration of `CHARACTER` type entities: AT will contain 1 character and STARS will contain 3 characters (an old syntax was used instead of LEN=3), followed by the declaration of `INTEGER` type entities M1, M2 and M3. The variable named AT gets character `@`, the variables M1 and M2 get the value 1 (2 pieces, for the 2 entities), and M3 gets the value 5, after which the STARS string is also initialised with the `***` characters. The YEAR, MONTH and DAY components of entity R24 will be given the values 2024, AUG and 12.

Expressions

Expressions can be arithmetic (numeric), string (character), logical, or initialisation and specification (from Fortran 90), and consist of operators, operands, and parentheses. An operand is a value represented by a constant, variable, array or array element, or resulting from the evaluation of a function. Operators can be intrinsic (implicitly recognised by the compiler and of global in nature, so always available to all sequences of code) or user-defined (when an operator is explicitly described as a function by the programmer). Depending on how they work, we can talk about unary operators (acting on a single operand) and binary operators (acting on a pair of operands). Unary operators take precedence over binary operators. Evaluating an expression always produces a single result, which can be used for attribution or as a reference. The type of value resulting from the evaluation of a numeric expression depends on the type of operands and their rank. If the operands within the expression have different ranks, the resulting value will be of the type of the operand with the highest rank (unless an operation involves a complex value and one in double precision, the result in such situations being of double complex type). When checking the correctness of a combined numerical expression, it is recommended to take into account the type of partial values resulting during the evaluation. Expressions can be arithmetic (numeric), string (character), logical, or initialization and specification (starting with Fortran 90).
There are homogeneous expressions (where the operators and operands are of the same type) and non-homogeneous expressions (where the operators and operands are of several types). The evaluation priority of operators within non-homogeneous expressions is as follows (in descending order):

- defined unary operators and functions;
- numeric operators (in the following order: `**`, `*` or `/`, `+` or `−`);
- concatenation operator for strings (characters);
- relational operators (with equal priority);
- logical operators (in order: `.NOT.`, `.AND.`, `.OR.`, `.EQV.` or `.NEQV.` or `.XOR.`).

Arithmetic expressions

As their name suggests, represent numerical calculations, made up of arithmetic operators and operands, giving a numerical result that must be defined mathematically (division by zero, raising a base of zero value to a zero or negative power, or raising a base of negative value to a real power are invalid operations). The term numeric operand can also include logical values, since they can be treated as integers in a numerical context (the logical value `.FALSE.` corresponds to the `INTEGER` value `0`). The numeric operators are: `**` (exponentiation), `*` (multiplication), `/` (division), `+` (addition), `−` (subtraction). In an arithmetic expression with several operators, the parts enclosed in parentheses (from the inside to the outside) and the functions are always evaluated first, with the evaluation priority of the intrinsic operators being as follows: exponentiation, multiplication and division, unary plus and minus, addition and subtraction. Operators with the same priority are evaluated from left to right. By local effect, unary operators can affect this rule, generating exceptions in the case of compilers that accept such expressions.

Fortran expression	Math. formula
`(3*X**2+1)/(2*Y)-1` `(3*X**2+1)/2/Y-1`	$\frac{3x^2+1}{2y}-1$
`X/(-5)*Y`	$\frac{x}{-5}y$
`X**(-Y)*3`	$x^{-y}3$

Fortran expression	Math. formula
`(3*X**2+1)/2*Y-1`	$\frac{3x^2+1}{2}y-1$
`X/(-5*Y)` `X/(-5)/Y`	$\frac{x}{-5y}$
`X**(-Y*3)`	x^{-y3}

String (character) expressions

They can be composed using the `//` concatenation operator (in older versions of Fortran using the `+` intrinsic operator) or using programmer-defined functions, applied to `CHARACTER` type constants or variables. Evaluating such an expression produces a single string value. Concatenation is performed by joining the character contents from left to right, without parentheses affecting the result. Blanks (spaces) contained in the operands are also included in the result.

Logical expressions

They consist of logical or numeric operands combined with logical and/or relational operators. The result of a logical expression is normally a logical value (equivalent to one of the logical literal constants `.TRUE.` or `.FALSE.`), but logical operations applied to integer numeric values will still result in integer values, being performed bit by bit in order corresponding to the internal representation of those values. Logical operations cannot be performed directly on values of type of `REAL`, `COMPLEX` or `CHARACTER`, but these types of values can be handled using relational operands within logical expressions. The relational and logical operators are as follows:

Relational operators		
Syntax	Meaning	Older syntax*
`<`	Less Then	`.LT.`
`<=`	Less or Equal to	`.LE.`
`==`	Equal	`.EQ.`
`/=`	Not Equal	`.NE.`
`>`	Greater Than	`.GT.`
`>=`	Greater or Equal to	`.GE.`

* Older versions (marked with dots in the last column) are also allowed to be used.

Logical operators	
Syntax	Meaning
`.NOT.`	Logical negation (logical complement) returns `.TRUE.` if the operand has the value `.FALSE.` and returns `.FALSE.` if the operand has the value `.TRUE.`.
`.AND.`	Logical conjunction returns `.TRUE.` only if both operands have the `.TRUE.` value, otherwise returns `.FALSE.`.
`.OR.`	Logical disjunction returns `.TRUE.` if one of the operands has the `.TRUE.` value, otherwise returns `.FALSE.`.
`.EQV.`	Logical equivalence, results true if both operands have the same value, if they have different values then results `.FALSE.`.
`.NEQV.`	Logical inequality returns `.TRUE.` if the operands are different, and `.FALSE.` if they are the same.
`.XOR.`	Exclusive logical disjunction (eXclusive OR), similar effect to logical inequality (`.NEQV.`).

The relational operators have equal priority (they are executed from left to right, but before the logical and after the numerical), and the logical operators are executed in the order of their evaluation priority. Relational operators are binary (they act on two operands), as are logical operators, except for the logical negation operator (`.NOT.`), which is unary.

Specification and initialisation expressions

These can be considered those that contain intrinsic operations and constant parts, or a whole scalar expression. As their name suggests, they are used to initialise values (for example, the index to control an implicit cycle) or to specify properties (for example, to declare array bounds or string lengths).

Constants used in expressions

Operands can be variables (only named entities can have variable values) or constant values. Constant values are specified according to their type, as shown in the following table:

Constant type	Examples:	Explanations:
Character string	`"Bla 3-1a"` `"anii '80"` `'anii ''80'`	Printable characters are quoted. If there are apostrophes or quotation marks within a character string, either the inner apostrophe can be doubled (see the third string), or the other character is used for as a delimiter.
Decimal number	`231` `50.66` `-.13e2` `256.`	The decimal separator is the dot, negative values are indicated by the minus sign. Non-significant digits can be omitted (the first value is an integer and the last three values are real). The third value is -13.0 (`e2` means $\times 10^2$)
Binary number	`B"1001"` `b"1011"` `B'1100'`	When quoting the value after the `B` mark, only the digits 0 or 1 are allowed (max. 256 positions). The minus sign before the `B` mark has no effect and is not accepted in the quoted content (there are no such negative values). Quotations can be made either with quotation marks or with apostrophes (without combining them).
Octal number	`O"152"` `O'223'` `o"107"`	Only the digits 0 to 7 can be used (max. 86 positions) in the value that is quoted after the `O` mark. As before, the minus sign in front has no effect and is not allowed inside.
Hex number	`Z"15F"  X"15f"` `Z'1B0'  x'1B0'` `z"A28"  x"a28"`	The digits 0 to 9 and letters A to F can be used (max. 64 positions) by quoting the value after the `Z` or `X` mark. As before, the minus sign in front has no effect and is not accepted inside.
Hollerith	`1H&` `3H123` `12Hla "Taverna"` `12Hab"1 x'+#.%@`	They are constants that can contain any printable character. Their syntax is: *n*`H`*string*, where *n* is the number of characters (positions in the string), `H` is the Hollerith mark and *string* stands for the content. Although these constants were originally defined to contain up to 2000 characters, the number of characters can be between 1 and 32767 ($2^{15}-1$) on 32-bit platforms, or between 1 and 2147483647 ($2^{31}-1$) on 64-bit platforms. Cannot be used as Format descriptor starting from Fortran 90.

Intrinsic functions (for expressions)

Intrinsic functions are specific to the libraries used, and have predefined (reserved) symbolic names. Some of them are not part of the standard kit of the programming environment, since they are not found in all variants of the Fortran language. The fact that the names of these functions are reserved means that there should be no entities with names that coincide with those of the intrinsic functions. Also, the names of these functions are not recommended to appear in a list of an `EXTERNAL` statement, which leads to the cancellation of their intrinsic definition. In such cases, by including their names in lists of the `INTRINSIC`

statement, they can be used in procedures defined as program units (user-defined subroutines or functions). The general syntax of functions is as follows:

function_name(*a*,[*a*]…)

where *function_name* is the symbolic name of the function and *a* represents the argument(s). Some intrinsic functions are given in the following table:

Function:	*function_name*:	Result:
|x|	ABS(*x*)	The absolute value (modulus) of the specified X argument.
A×B	MATMUL(*A*,*B*)	Matrix resulting from the multiplication of matrices A and B.
A^T	TRANSPOSE(*A*)	Returns the transpose of matrix A.
A_{max}	MAXVAL(*A*)	Returns the maximum value in array A.
A_{max}(pos)	MAXLOC(*A*)	Returns the first position of the maximum value in array A.
A_{min}	MINVAL(*A*)	Returns the minimum value in array A.
A_{min}(pos)	MINLOC(*A*)	Returns the first position of the minimum value in array A.
arccos(x)	ACOS(*x*)	The arccosine of the X argument expressed in radians.
arcsin(x)	ASIN(*x*)	The arcsine of the X argument expressed in radians.
arctg(x)	ATAN(*x*)	The arctangent of the X argument expressed in radians.
character	ACHAR(*x*)	Returns the character at position X in the code table.
complex-i	AIMAG(*x*)	The imaginary part of a complex number X.
complex-r	REAL(*x*)	The real part of a complex number X.
cos(x)	COS(*x*)	The cosine value of the X argument expressed in radians.
cosh(x)	COSH(*x*)	The hyperbolic cosine of the X argument.
e^x	EXP(*x*)	The exponential value of the Euler constant (e=2.71828...).
ln(x)	LOG(*x*)	The value of the natural logarithm of the X argument.
log(x)	LOG10(*x*)	The logarithm with base 10 of the X argument.
length	LEN(*string*)	The number of characters in the STRING considered argument.
max(x,y,...)	MAX(*value_list*)	The maximum value among the items contained in the argument list.
min(x,y,...)	MIN(*value_list*)	The minimum value among the items contained in the argument list.
random	RAN(*x*)	Returns a pseudorandom number between 0 and 1.
rest of div.	MOD(*x1*,*x2*)	The remainder of the argument division (X1/X2, with the sign of X1).
round	NINT(*x*) ANINT(*x*)	The value of the X argument rounded to the nearest integer. The rounded value of the X argument to zero decimal places.
sin(x)	SIN(*x*)	The value of the sine of the X argument expressed in radians.
sinh(x)	SINH(*x*)	The hyperbolic sine of the X argument.
size	SIZE(*array*[,*ri*])	Returns the size of the array (by RI rank, if specified).
$\sqrt{x}$	SQRT(*x*)	The square root (radical) of the X argument.
substring	INDEX(*string*,*ss*)	The starting position of the SS substring in the first argument STRING.
ΣA	SUM(*array*[,*ni*])	Returns the sum of the values in the array (by RI rank, if specified).
tg(x)	TAN(*x*)	The tangent of the X argument expressed in radians.
tgh(x)	TANH(*x*)	The hyperbolic tangent of the X argument.
truncate	INT(*x*) AINT(*x*)	The truncated value of the argument X to the nearest integer. Truncated value of argument X with zero decimals.

Input and output (I/O) statements

Read operations are called inputs (I) and write or display operations are called outputs (O). For sequential inputs, the READ statement can be used, with the following syntax variants:

READ *f*[,*input_list*]

when reading from the default logical unit (usually the console, so the keyboard), where *f* is the format specifier (shown later). A more general variant has the following sintax:

READ([UNIT=]*u*[,[FMT=]*f*][,ERR=e_1][,END=e_2][,IOSTAT=*var*][,ADVANCE=*opt*])[*input_list*]

where the UNIT= keyword can be omitted if it is the first parameter and *u* is the value of the logical unit number (the value is * for the default logical unit, i.e. console), the FMT= keyword can be omitted if it is the second parameter or if it is you do not want to use a format specifier *f* (the case of reading without format), e_1 is the label of an executable statement to jump to if the end of file (EOF) is encountered or if there are no values to read, e_2 is the label of an executable statement to jump to if a read error is encountered, and *var* is the name of an INTEGER variable in which the success / failure of the read operation would be recorded (successful reads result in 0, unsuccessful reads result in higher values marking error codes, -1 means EOF while -2 means EOR). When using ADVANCE=, *opt* can be "YES" (advance to next line after reading, default) or "NO" (means no advance to the next line). The entities in which the read values are to be stored form the *input_list*. If there is no *input_list*, the only effect of the statement is to temporarily stop the execution of the program (until the <*Enter*> key is pressed).
There are also other variants, such as internal reading (to convert characters into integers corresponding to the positions in the character table), direct reading (to jump to the position number of a record in a fixed formatted logical unit), or keyed reading (in the case of indexed files).

The following statements can be used for sequential output operations:

PRINT *f*[,*output_list*]

when writing to the default logical unit (usually the console, hence the monitor display), where *f* is the format specifier, or

WRITE([UNIT=]*u*[,[FMT=]*f*][,ERR=e_1][,IOSTAT=*var*][,ADVANCE=*opt*])[*output_list*]

where the notation is the same as for reading (without END=e_2, as it makes no sense for writing). The entities whose values are to be written make up the *output_list*. If *output_list* is missing, an empty line is written.
There are also other variants, such as internal writing (to convert integers to characters, according to the positions in the character table), direct writing (jumping to the position number of a record in a fixed formatted logical unit), or rewriting a record. Writing to indexed files uses sequential writing with format specifier, where key fields are among the entities in *output_list*.
When the * symbol is used as a format specifier (i.e. default format), the value type in the entity list is usually taken into account. For so-called long values, such as REAL(8) or DOUBLE PRECISION, REAL(16), COMPLEX(8) or DOUBLE COMPLEX, COMPLEX(16), the default format cannot be used, so a format specification appropriate to the type must be used.

Examples:	Explanations:
READ * *! Equivalent to:* READ(*,*)	Apparent reading (no input). Waiting for the <*Enter*> (carriage return) key to be pressed to continue.
READ *,I,j *! Equivalent to:* READ(*,*)i,J	Two numerical values (of type INTEGER) entered from the keyboard are read and stored in variables I and J respectively. The two values can be entered separately (the program will continue only after both values have been entered) or on the same line, separated by a comma (or a blank).
PRINT * *! Equivalent to:* WRITE(*,*)	A blank line will be displayed on the screen (similar to the effect of the <LF> character).
PRINT *,"Max= ",max *! Equivalent to:* WRITE(*,*)"Max= ",MAX	The quoted string will be displayed (without the quotation marks), followed by the content (value) of the MAX variable.

Format specification and descriptors

Format descriptors are like templates applied to input or output data. They are usually used through the format specification, which has the following syntax:

label FORMAT(*descriptor_list*)

however, descriptors can also appear in quoted form within read or write statements. There are two categories of descriptors: for data editing and for controlling formatting. They will be presented below in separate tables, with examples, using the following notations:

n – number of pieces;
w – descriptor length (total number of positions in the respective field);
m – minimum number of positions requested (of the total number), has effect on output only;
d – number of positions for the decimal part (of the total number);
e – number of positions for the exponent (of the total number);
c – character, respectively [*c*...] other optional characters;
□ – space (blank character) displayed in the examples.

Table of descriptors used for data editing (in alphabetical order):

Syntax:	Destination:	Examples and comments:		
[*n*]A[*w*]	Alphanumeric data (CHARACTER)	Input: ABC_D ABC_D ABC_D	Format: A5 A5 A5	Entity type: Value: CHARACTER(1): D CHARACTER(3): C_D CHARACTER(6): ABC_D□
		Value: ABC ABCDE ABCDEFG	Format: A5 A5 A5	Output (5 positions): □□ABC ABCDE ABCDE
[*n*]B*w*[.*m*]	Binary numeric data	Input: 1001 1001 1001	Format: B4 B2 2B2	Value (in decimal form): 9 (all 4 positions read) 2 (only the first 2 positions read) 2 și 1 (2 distinct values)
		Value: 13 0 0	Format: B5 B2 B2.2	Output: □1101 □0 00
		If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).		
[*n*]D*w*.*d*	Numerical data in double precision: REAL(8) i.e. DOUBLE PRECISION, or COMPLEX(8), i.e. DOUBLE COMPLEX	Input: 123.456E3 12345678 123.45678	Format: D9.3 D6.2 D7.3	Value (double precision): 123456.0D+0 1234.56D+0 123.456D+0
		As can be observed, *w* positions are read from the input, of which *d* positions for the decimal part (from the decimal separator to the right – if there is no decimal separator at the input, then the decimal part will result considering *d* positions at the end of the *w* read). The ”D+0” mark at the end only indicates that the values will be obtained in double precision.		
		Value: 123456.789 0.0363 -0.5555	Format: D11.2 D10.3 D10.3	Output: □□□0.12D+06 □0.363D-01 -0.556D+00
		The display will result in *w* positions, of which *d* positions for the decimal part, but it should be noticed that 1 position will be consumed for the sign of the value, 1 more for the decimal separator (dot), 1 position for the letter of the descriptor (D), the last 3 positions for the sign and value of the exponent. If we consider that the first significant digit will be the first decimal place, it follows that it is advisable that *w*−*d* > 6. If this condition is		

		not met, format overflow will occur (asterisks will be displayed on the *w* positions).
[*n*]E*w*.*d*[E*e*]	Numeric data in exponential format (REAL or COMPLEX)	Input: Format: Value: □□123.45□□ E10.2 123.45 123456789 E9.3 123456.789 123.456D3 E9.3 123456.0 (simple precision!) As with the previous descriptor, *w* positions are read from the input, of which *d* positions for the decimal part (from the decimal separator to the right – if there is no decimal separator at the input, the decimal part will result considering *d* positions at the end of the *w* read). In case of reading double precision values with this descriptor (or with other usable descriptors except D), a value converted to single precision will be obtained.
		Value: Format: Output: 123456.789 E11.5 0.12345E+06 -0.5555 E12.3E3 □-0.556E+000 0.0363 E5.2 ***** (format overflow!) The display will result in *w* positions, of which *d* positions for the decimal part, but it should be noticed that 1 position will be consumed for the sign of the value, 1 more for the decimal separator (dot), 1 position for the letter of the descriptor (E), the last 3 positions for the sign and value of the exponent. If we consider that the first significant digit will be the first decimal, it turns out that *w*−*d* > 6 [+(*e*−2)] (where *e* is the number of digits of the exponent). If this condition is not met, format overflow will occur (asterisks will be displayed on the *w* positions).
[*n*]EN*w*.*d*[E*e*]	Numeric data in exponential ”engineering” format (REAL or COMPLEX)	Input: Format: Value: 123.45E+03 EN10.2 12345.0 -12345678 EN9.3 -12345.678 123.456D3 EN9.3 123456.0 (simple precision!)
		Value: Format: Output: 123456.789 EN11.2 □123.46E+03 -0.5555 EN7.1 ******* (format overflow!) 0.0363 EN12.3 □363.000E-04 When displayed, the decimal point will be after the first 3 digits.
[*n*]ES*w*.*d*[E*e*]	Numeric data in exponential ”scientific” format (REAL or COMPLEX)	Input: Format: Value: □□□1.234E+03 ES12.3 1234.0 -10.234E-03 ES11.3 -0.010234
		Value: Format: Output: 123456.789 ES11.2 □□□1.23E+05 -0.5555 ES10.3 -5.555E-01 0.0363 ES12.3 □□□3.630E-02 On display the decimal point will be after the first significant digit.
[*n*]F*w*.*d*	Numeric data (REAL, F stands for ”Float”)	Input: Format: Value: 12345678 F8.5 123.45678 -12345678 F8.2 -1234.56 24.77E+2 F8.2 2477.0
		Value: Format: Output: 2.3547188 F8.5 □2.35472 325.03 F5.2 ***** (format overflow!) -0.2 F5.2 -0.20
[*n*]G*w*.*d*[E*e*]	Intrinsic type data (G	Input: Format: Value: -0.05566 G10.3 -0.05566

	stands for ”Generic”)	123456	G10.3	123.456
		123456.789	G10.3	123456.79
		Value:	Format:	Output:
		-45.66	G11.3	□-4.566E+01
		123456	G10.3	□□□□123456
		123456.78	G10.3	□0.123E+06
		Remarks: It can be used for any of the intrinsic type values. If 0 is specified for *w*, the actual value of *w* will be chosen by the processor (in such cases only G0 or G0.*d* may be specified). If *w* is different from 0, then the value for *d* must also be specified. In the case of INTEGER, CHARACTER and LOGICAL values the value specified by *d* will be ignored, the descriptor will behave as the one corresponding to these values (I, A and L).		
[*n*]I*w*[.*m*]	Integer numeric data (INTEGER)	Input:	Format:	Value:
		-1234	I4	-123
		□□□123	I6	123
		1234.6	I6	Error! (not INTEGER)
		Value:	Format:	Output:
		0	I3	□□0
		0	I3.0	□□□
		1	I3.2	□01
		-123	I3	*** (format overflow!)
		1.2	I4	Error! (not INTEGER)
[*n*]L*w*	Logical data	Input – logical values written in the following forms are accepted, including lowercase (not just uppercase): .TRUE. or .T or T , or if the first characters in the input are .T or T (for true); respectively .FALSE. or .F or F, or if the first characters in the input are .F or F, or the content is from space/ blanks (for false).		
		Value:	Format:	Output:
		.TRUE.	L7	□□□□□□T
		.FALSE.	L1	F
		□□□	L3	□□F
		Only 1 character (T or F) will be output regardless of the length *w* specified.		
[*n*]O*w*[.*m*]	Integer octal numeric data (with base in 8)	Input:	Format:	Value (decimal):
		1111	O2	9
		1111	O4	585
		□11□	O4	9
		191	O3	Error! (9 is not octal)
		12	O0	Error! (*w* must be positive)
		Value (decimal):	Format:	Output:
		11	O6.4	□□0013
		-11	O6	****** (format overflow!)
		-11	O12	□37777777765
		1.5	O11	□7760000000
		81	O0	121
		If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).		
[*n*]Z*w*[.*m*]	Integer hexadecimal numeric data (with base in 16)	Input:	Format:	Value (decimal):
		A2F	Z3	2607
		-A2F□	Z5	-2607

		3.A2F	Z5	Error! (invalid decimal point)
		Value (decimal):	Format:	Output:
		3033	Z5	□□BD9
		16	Z5.4	□0010
		-10	Z8	FFFFFFF6
		1.1	Z8	3F8CCCCD
		2.5	Z0	40200000
		If *w*=0, as many positions as required to display the value will be used at the output (*w*=0 is not allowed at the input).		
'c[c...]' or "c[c...]"	Quoted alphanumeric constants (CHARACTER)	Input: – (cannot be used for input).		
		Format:		Output:
		'aBc''DD:'	(double apostrophe inside)	aBc'DD:
		"aBc""DD:"	(double quote mark inside)	aBc"DD:
		'aBc"DD:'	(quote mark inside)	aBc"DD:
		"abcD'#"	(apostrophe inside)	abcD'#
		"ab'cD'#"	(quote within quote!)	Error!

Table with control descriptors:

Syntax:	Meaning:	Examples and comments:
BN BZ	BLANK NULL BLANK ZERO	BN will have the effect of ignoring the spaces in the number fields. BZ will have the effect of "replacing" the spaces in the numeric fields with 0 digits. Input: Format: Value: □1□□ BN,I4 1 □1□□ BZ,I4 100 1□23 BZ,I4 1023
*k*P	Power *k* is a scaling factor, with value in the range [-128, +127]	Allows the interpretation of numeric values with decimals, using the scale factor *k* for descriptors D, E, F and G when these values do not explicitly contain an exponent. On inputs, a positive *k* value will have the effect of moving the decimal separator to the left, and a negative value to the right (on outputs, the effect will be the reverse). The descriptor P need not necessarily be separated by a comma from the descriptor to which it refers, but must precede it. For example, the following specifications will have the same effect, the scale factor being associated with the first real number descriptor following it in the list (E10.3): (2P,I4,E10.3,F8.2) (I4,2P,E10.3,F8.2) (I4,2PE10.3,F8.2) Input: Format: Value: □□□37.614□ 3PE10.5 0.037614 □□□37.614□ -3PE10.5 37614.0 123.45 2PF8.3 1.2345 123.45 -2P,F8.3 12345.0 Value: Format: Output: -170.139 1P,E10.3 -1.701E+02 -170.139 -1PE10.3 □-0.02E+04
S SP SS	Sign Sign Positive Sign Suppress	SP will cause the + sign to be displayed in front of positive values and SS will inhibit it. S acts as a switch between SP and SS.
T*n* TLn	Tab Tab Left	With descriptor T, the position *n* in a line is indicated, from which reading (or to which writing) is desired.

TR*n*	Tab Right *n* – tab position	Assuming that the following string will be typed from the keyboard: 123456789ABC to be read with the sequence: `CHARACTER(3) C1,C2` `READ(*,5) NR,C1,C2` `5 FORMAT(T7,I3,T1,A3,T10,A3)` the values will result: NR=789; C1="123" și C2="ABC". TR*n* allows specifying the n^{th} position to the right from the current position and TL*n* to the left (*n* being a positive number). When using TL, if *n* is greater than or equal to the current position, then positioning will be done on the first character in the row.
[*n*]X	Determine the jump over *n* positions in the current line	On input it will cause *n* positions to be ignored, and on output it will have the effect of printing *n* spaces (if it appears at the end of the descriptor list, then it has no effect. In the example the effect is highlighted by marking □ on display): Source code: `PRINT 4` `READ 3,nr` `PRINT 4,nr` `3 FORMAT(2X,I2)` `4 FORMAT("number:",1X,I2)` Display: number: Input: 1234 Display: number:□34
$ \	Suppress the jump to a new line (suppress <*LF*>).	It will cause the cursor to remain at the last current position (<*LF*> is short for *Line_Feed*). The $ variant is newer, but not part of the standard, and the \ variant is the older one (the G95 compiler supports both). Whichever variant is used, the descriptor must be the last in the list to which it belongs. Source code: `PRINT 5,"nr:"` `READ *,nr` `PRINT 4,nr` `5 FORMAT(A,$)` `4 FORMAT("number:",1X,I2)` Display+Input (12): nr:12 Display: number:□12
[*n*]/	Induces *n* new line jumps (induces *n* pieces of <*LF*>)	It can also be used without n, e.g. (3/) is equivalent to (///), without the need for separating commas. In the following example it will insert a new line feed before displaying "number", then insert 2 more new line feeds: Source code: `PRINT 5,"nr:"` `READ *,nr` `PRINT 4,nr` `5 FORMAT(A,$)` `4 FORMAT(/"number:",2/,3X,I2)` Display+Input (12): nr:12 Display: □ number: □ □□□12
:	Ends descriptor control in the absence of input/output list items	In the following example, in the absence of items to display, the descriptor will cause the "j2" part to be ignored: Source code: `PRINT 1,3` `PRINT 2,14` `1 FORMAT("i",I2,1X,"i2",I2)` `2 FORMAT("j",I2,:,1X,"j2",I2)` Display: i□3i2□□ j14

The format specification may also be composed by using string (character) expressions. The following example shows how it might apply for N pairs of descriptors of the form (I2,1X), assuming 1< N <9:

Example:	Explanations:
`CHARACTER fm(10)` `INTEGER j(9)` `PRINT *,"nr. (1-9): "` `READ(*,*)n` `k=48+n` `fm="("//ACHAR(k)//"(i2,1x))"` `PRINT *,"the ",n," values: "` `READ *,(j(i),i=1,n)` `WRITE(*,fm)(j(i),i=1,n)` `END`	Declare the FM string with 10 positions (to be used as format specification). The variable J will have 9 positions (will be a vector) and will contain the values to be displayed with `I2` type descriptors. The quoted string is displayed and on reading the number entered (of desired pieces) is stored in variable N. The position number in the code table is composed of the digit corresponding to the quantity (from variable N), obtaining the character of the digit representing this value. An alphanumeric string is constructed by concatenation and using the intrinsic function `ACHAR` (which returns the character at position K in the code table), which is assigned to the variable FM, which is the format descriptor with N pairs of `I2` (two-byte integer) and `1X` (space) fields for the N values. The quoted string (including the value of N) is displayed, then the values corresponding to the N positions of the vector J are read (by implicit loop). The N positions of the vector J are displayed (also by implicit loop) using the format specification stored in the FM variable as an alphanumeric string.

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 index, and the "upper" means the final value of the respective index),
- 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.

Declaring an array involves either allocating memory areas for each array element at the time of program creation (static allocation method), or allocating memory areas only for the array rank at the time of program creation, with the actual memory allocation for the array elements occurring during program execution (dynamic allocation method).

If memory usage is to be optimised (less space means fewer addresses, resulting in faster execution), then it would be desirable not to allocate unused space for arrays. This can be achieved by dynamically allocating memory at runtime, specifying only the really needed size of the arrays.

Static allocated memory

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

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

or

Type[,*attribute*] :: *array_name_1*(*extents*)[,*array_name_i*(*extents*)…]

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).
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 index 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 index *i* start with a value of 1 and reach 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(1)=10;L(2)=10;L(3)=10;L(4)=10` `L(5)=10;L(6)=10;L(7)=10;L(8)=10`	L is the 8-position array, and the number 10 is a scalar value. Since a scalar conforms to any array, 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 string section (i.e a part of an array with rank 1):

string_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 index 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 index 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 (one character for each 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 (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!	

Some 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.

Dynamic allocated memory

By dynamic memory allocation, 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).

Allocatable arrays

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

[*Type*,]ALLOCATABLE [,*attribute*… ::] *array_name_1*(:[,:]…)[,*array_name_i*(:[,:]…)…]

Note: for each rank, a position marked by the ":" character in the round bracket after the *array_name* 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 dynamic memory allocation is used 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_1*(*extents*)[,*array_name_i*(*extents_i*)…])

In such situations, 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:
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 of rank

	1. Both arrays are of type REAL by default. The actual number of positions in each array will be specified later.
REAL,ALLOCATABLE,DIMENSION(:) :: n,m	Arrays N and M will have the same rank (1). In this case the type of the entities (REAL) must also be specified.
REAL,ALLOCATABLE :: v(:),m(:,:) … ALLOCATE(v(10),m(0:9,-2:7)) … DEALLOCATE(v,m) … IF(ALLOCATED(m)) THEN DEALLOCATE(m) ENDIF … ALLOCATE(m(3,3)) … DEALLOCATE(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). 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! Freeing the memory spaces allocated to the previous 2 arrays. 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 a new size of memory to the M array, this time 3x3=9 positions. Release the memory allocated to the M array before the end of the program 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 even without prior allocation of memory space. The space in 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 starting from Fortran 95 (similar to the use of the ALLOCATABLE specification already presented), in the case of Fortran 90 such arrays can be created by the derived type specification (exemplified in the following).

Example:	Explanations:
POINTER C(:,:,:) REAL,TARGET :: kt(:)	The array named C has rank 3 (the reservation of each "dimension" is marked with a ”:” character) and will be of type REAL (by default), POINTER. The array named KT has rank 1 (like a vector) and is declared explicitly as type REAL (because the name starts with letter K). The actual number of positions (the extents) in these arrays will be specified later.

TYPE p_array REAL,DIMENSION(:),POINTER :: tp END TYPE … TYPE(p_array),ALLOCATABLE :: vp(:) … READ(*,*)n,m … ALLOCATE(vp(n)) DO i=1,n ALLOCATE(vp%tp(m)) ENDDO … DEALLOCATE(vp) …	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. 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. 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. Release allocated space, as with 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.

Examples:	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. When the POINTS subroutine is activated (and 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)	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 (as 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.

ENDDO END FUNCTION END	

Flow control statements

This category includes conditional statements, jump and loop (repetition) statements, as well as those for stopping or suspending the execution of a program.

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*) *statement*	Unstructured logical decision (simple logical IF), with empty branch, allows a single *statement* 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 *statements_1* [ELSE IF(*logical_expression_i*) THEN *statements_i*] [ELSE *statements_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 as ELSEIF in many 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 *statements_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 *statements_i* block. Only if all preceding logical expressions returned .FALSE. those specified in the *statements_x* block will be executed.
SELECT CASE(*expression*) [CASE(*criteria_ set_i*) *statements_i*] [CASE DEFAULT *statements_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 *statements_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 *statements_1*, *statements_i*, and *statements_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 statement 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 statement with label 1.
`  CHARACTER r` `    …` `1 WRITE(*,*)'Enter the values: '` `    …` `  WRITE(*,*)'Restart? (Y/N): '` `  READ(*,*)r` `  IF(r=='y'.OR.r=='Y') THEN` `    GOTO 1` `  ENDIF`	The previous example, using a structured logical `IF` (without the `ELSE` branch) instead of a simple logical `IF`, and `.EQ.` replaced by `==`.
`   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.` (just one of the two logical values).

Jump statements

The variants of jump statements (using the keywords `GO TO` or as `GOTO`) have the following syntax:

`GOTO` *label* `GO TO` *label*	The unconditional jump, *label* is where to jump to when this statement is executed. The *label* must mark an executable statement

(it must be written in front of the statement to be jumped to during execution).

`GOTO (`*label_list*`) [,]`*expression*
`GO TO (`*label_list*`) [,]`*expression*

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 value. If this value is negative, zero or greater than the number of elements in the *label_list*, the jump will not be performed.

Loop statements

The statements for making loops are structured in Fortran 95 (those where the end of the structure is marked instead of the label with END_*name* being introduced with Fortran 90). Being structured statements (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. Variants marked with a label at the end are inherited from previous versions of Fortran.

`DO` *label* [[,]*loop_control*]
statements
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`, or something similar), the neutral `CONTINUE` statement (shown below) can be used.

`DO` [*loop_control*]
statements
`ENDDO`

The difference from the previous variant consists in the end marking `ENDDO` (many 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 statement from the body of the loop). After a first step through the *statements* 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 *statements* 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 stop by forced interruption).

`DO` *label* [,] `WHILE (`*logical_expression*`)`
statements
label last_executable_statement

It would correspond to a preconditioned loop. The *statements* 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*) *statements* 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 statements 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 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` `! 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`	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 second variant only one label was used (it is not considered a structure intersection in such situations). It can also be observed that using the value of the *loop_counter*s is allowed, but it is forbidden to explicitly change their value. 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`	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 statement (using the descriptor for alphanumeric values).

`DO WHILE (LINE(i:i)==" ")` `  i=i+1` `ENDDO`	As long as spaces (blank characters) are encountered starting from the beginning of the string, the value *i* 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 loops 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 inside an explicit loop to read elements from a row of an array. Interpretation: read A(*i,j*) while the position index *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 "," character. The loop with counter *j* is inside the loop with counter *i*. In this example the display of the values will be in line.

Statements to stop 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, being removed from the Fortran 95 standard (it can be replaced by an empty read statement), but is supported by G95. Stops the execution of the program temporarily and displays (if specified) the *pause_code* (can be an integer, or a quoted string). The <*Enter*> key must be pressed to resume execution. In all cases, it will cause the display of the message:
`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.

Using logical units (peripherals and files)

The internal or external parts of a computing system used for input (reads) and output (writes) operations, respectively for data storage, are considered physical units. These are accessible in the Fortran language as logical units (default or explicit) corresponding to those physical units. Input (reads) and output (writes) statements are performed through logical units. The default logical unit (marked with the value `*` in statements that require specification) is the console, i.e. the assembly consisting of the keyboard and display (monitor screen) – for inputs the keyboard is considered, and for outputs the display. The logical units that must be explicitly specified are files, respectively peripherals (printer, magnetic tape drive, etc.),

with an integer value assigned to them. The indication of the logical unit to which an input / output statement refers is done by this numerical value. Some values also have predefined logical units in older Fortran, such as 1, 2, 3 and 4 for files named FOR00*n*.DAT (where the corresponding digit from 1 to 4 will appear instead of the character *n* in the file name), or 5 for input devices (card reader, keyboard, etc.) and 6 for output devices (printer, display, etc.). Allocating these references (numbers) to logical units can be done explicitly by the OPEN statement. The syntax of this executable statement is as follows:

OPEN(*parameter*[,*parameter*]…)

where *parameter* can be a *keyword*, or of the form *keyword*=*value* (each parameter may be specified only once within the list in parentheses).

Table with the parameters of the OPEN statement in alphabetical order:

keyword	*value*	Explanations	Default value
ACCESS=	"SEQUENTIAL" "DIRECT" "APPEND" "STREAM"	Setting how to access the logical unit: - sequential, - direct, - adding records at the end, - stream access (access by position).	"SEQUENTIAL" (row by row).
ACTION=	"READ" "WRITE" "READWRITE"	How to use the logical unit: - only to read from it, - only to write in it, - reading and writing.	"READWRITE" (read and write)
BLANK=	"NULL" "ZERO"	Interpretation of blanks: - spaces (no conversion), - 0 (conversion to digits for numbers).	"NULL" (no conversion).
CONVERT=	"NATIVE" "SWAP" "LITTLE_ENDIAN" "BIG_ENDIAN"	Allows specifying a numeric format (for conversion / interpretation) for unformatted data: - native (no conversion), - switch (between LITTLE_ENDIAN and BIG_ENDIAN), - the last two explicitly specify the encodings.	"NATIVE" (no conversion).
DECIMAL=	"COMMA" "POINT"	Decimal separator character	"POINT".
DELIM=	"NONE" "APOSTROPHE" "QUOTE"	Specifying the delimiter character (for CHARACTER type constants) for I/O operations: - without delimiter, - the apostrophe character, - the quotation mark character.	"NONE" (no delimiter).
ERR=	*label*	Statement *label* to jump to in case of error when opening the logical unit.	Not implicit, no jump by default.
FILE=	*string*	Specify the file to be used as the logical unit. The file	It depends on the logical unit and the

		specifier is considered a string, so it is delimited by apostrophe or quotation marks if quoted, and if contained by a CHARACTER, entity, the name of that entity should be specified.	operating system.
FORM=	”FORMATTED” ”UNFORMATTED”	Format of the logical unit (file) accessed: - with format, - without format.	Depends on the value of the ACCESS keyword. If it is ”DIRECT” then ”FORMATTED” will be considered, otherwise ”UNFORMATTED” will be considered.
IOSTAT=	*variable*	Returns a scalar INTEGER value in the variable, indicating the success (or failure) of accessing the logical drive. If the logical unit is opened successfully, the value of the variable is 0.	No default value.
PAD=	”YES” ”NO”	Specifies whether a record is filled with spaces (blank characters) when the format requires more positions than the value entered, or not filled.	”YES” (blanks are used when necessary).
POSITION=	”ASIS” ”REWIND” ”APPEND”	Specifies the positioning in a file: - as is, - back to the beginning, - add at the end.	”ASIS” (current position).
RECL=	*number*	Record length in bytes for the logical unit in case of direct access, or maximum length in the case of sequential access (*number* should be a positive integer value).	Depends on the value specified for the ACCESS keyword.
SHARE=	”COMPAT” ”DENYNONE” ”DENYWR” ”DENYRD” ”DENYRW”	Controls how other processes can access the logical unit simultaneously: - compatible (not shared), - without restrictions, - no writing from other program, - no reading from other program, - exclusive, no access from other program.	”DENYNONE” (no restrictions).

STATUS=	"OLD" "NEW" "REPLACE" "SCRATCH" "UNKNOWN"	State of the logical unit (of the file) when opened: - existing (if it does not exist, an error is obtained), - new (if already exists, generate error), - overwriting a file, - temporary (deleted after closing), cannot be used with FILE=, - unknown (opened if exists, created if does not).	"UNKNOWN" (opened if exists, created if does not).
UNIT=	*number*	The logical unit number (associated with the desired file or device) being accessed (*number* is a positive integer). The logical unit number can be specified without the UNIT= keyword if it is the first parameter in the parentheses.	No default value.

Disconnection of the logical unit (in the case of files it means closing them) can be specified by the CLOSE executable statement, the syntax of which is as follows:

CLOSE(*parameter*[,*parameter*]…)

where *parameter* is of the form *keyword*=*value* (each *parameter* can be specified only once within the list in parentheses).

The CLOSE statement will also cause the <EOF> (end-of-file) to be recorded (written) when the unit is disconnected (file is closed).

Table of parameters in the CLOSE statement (in alphabetical order):

keyword	*value*	Explanation	Default value
ERR=	*label*	The label of the statement to jump to in case of an error when disconnecting the logical unit (*label* is a positive integer).	Not implicit, no default jump.
IOSTAT=	*variable*	Returns a scalar INTEGER value in the variable, which indicates the success (or failure) of closing the logical unit. If the logical unit was successfully closed, the value of the variable is 0.	No default value.
STATUS=	"KEEP" "DELETE"	Option to keep (save) or delete the file.	"KEEP" (save the file) if STATUS= was not "SCRATCH".
UNIT=	*number*	The logical unit number (associated with the desired file or device) to disconnect (*number* is a positive integer). The logical unit number can be	No default value.

		specified without the UNIT= keyword if it is the first parameter in the parentheses.	

Caution: A file opened with the STATUS="SCRATCH" specification cannot be saved or printed (displayed), such an attempt will generate an error at runtime, and if ACTION="READ" was specified when opening, the file cannot be deleted on disconnection (close). A read-only file does not necessarily need to be closed, but a file whose contents have been changed (written to) must be closed using the CLOSE statement, otherwise it may be stuck with inaccessible contents when the program finishes. Writing through a buffer, if it has not been explicitly emptied (by the effect of the CLOSE statement), then it is not certain that all records have been transferred, and at the end of the program run there will be no one to manage the contents of the buffer (resulting in the computer's memory being filled with unnecessary data).

Example:	Explanations:
`OPEN(3,FILE="TEST.DAT",STATUS="OLD")` `READ(3,*)n,m` `CLOSE(3)`	Open the existing TEST.DAT file associated with logical unit number 3, then read the values of variables N and M from this file and disconnect the logical unit (close the file).
`DIMENSION A(10,10)` `  CHARACTER(12) name` `PRINT *,"data file name: "` `3  READ(*,"(A)")name` `OPEN(1,FILE=name,STATUS="OLD",ERR=9)` `! get the number of rows for A` `  READ(1,*) nl` `! get the number of columns for A` `  READ(1,*) nc` `! read the elements, one row at a time` `  DO i=1,nl` `  READ(1,*)(A(i,j),j=1,nc)` `  ENDDO` `  …` `OPEN(2,FILE="R.DAT",STATUS="UNKNOWN")` `! Write a title to the R.DAT file` `  WRITE(2,*)"Array A:"` `! Write the elements, line by line:` `  DO i=1,nl` `  WRITE(2,*)(A(i,j),",",j=1,nc)` `  ENDDO` `CLOSE(2)` `  …` `STOP` `9  PRINT *,"file not found!"` `GOTO 3` `  …`	Declare an array of 10x10=100 positions and the NAME entity with 12 positions (characters). Note CHARACTER, the letter C in the first column marks comment! Read the file name into the NAME variable. Open the (existing) file associated with logical unit 1, from which the data will be read (see the comments in the adjacent column marked with an exclamation mark). If the file does not exist, it jumps to the statement labelled 9. Use an implicit loop (J=1,NC) inside an explicit loop (I=1,NL) to read elements from an array. Opening the R.DAT file associated with logical unit 2 (if the file does not exist, it is created, and if it exists, it is opened and its contents are overwritten). Write the elements of the array A, line by line, with a comma after each element. Close the R.DAT file (disconnect logical unit number 2). Logical unit number 1 has not been modified and will be automatically disconnected when the program ends. If the data file is not found, after displaying the specified message, an attempt will be made to read its name again (jumping to the statement labelled 3).

There are other additional statements for handling files, such as:

Syntax:	Explanations:
BACKSPACE([UNIT=]*u*[,ERR=*label*][,IOSTAT=*var*]) or BACKSPACE *u*	Repositions the file on the previous record in case of sequential access.
ENDFILE([UNIT=]*u*[,ERR=*label*][,IOSTAT=*var*]) or ENDFILE *u*	Write the end of file marker in the current position to the file associated with logical unit number *u* (accessed via a previous OPEN statement, the rest of the file being truncated).
REWIND([UNIT=]*u*[,ERR=*label*]) or REWIND *u*	Repositioning to the beginning of the file associated (previously by the OPEN statement) with logical unit number *u*.

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 statements 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).
Not all program units can contain executable statements, there are program units that can 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 statements),
- External procedures (subroutines, functions – may contain executable statements),
- Modules (may not contain executable statements, only possibly in embedded module procedures)
- Data blocks (cannot contain executable statements, only specifications).

Each application (Fortran program) 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, but 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 statements.
[*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	An empty main program unit with default name MAIN.
PRINT *,"Hello!" END PROGRAM	Main program that will only display the text Hello! on the monitor.
PROGRAM test INTEGER C, D … CALL sub1 … CONTAINS SUBROUTINE sub1 … PRINT *, func(X,Y) … END SUBROUTINE sub1 FUNCTION func(X,Y) … END FUNCTION func END PROGRAM test	Main program named TEST, which will call the subroutine SUB1 (included as an internal procedure along with the FUNC function). Calling the subroutine named SUB1. Marking the contained procedures. Defining subroutine SUB1 as an internal procedure. Printing the result of the FUNC function for the current values of arguments X and Y. End mark for the internal procedure SUB1. Defining the FUNC function as an internal procedure. Marking the end of the internal procedure FUNC. The end mark for the main program TEST (with the name specified, although an END would have sufficed).

Procedures

These 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.

Entering a procedure (by CALL statement in case of subroutines, or by name in case of a function) can also be done at a position other than the start of the procedure, 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 — When it is desired to apply the procedure 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 that are declared PURE, the INTENT options should not be used for arguments and function names (there is no such restriction in subroutines). In addition, a procedure which is declared as 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).

Subroutines

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).
[*specifications*]	In all program units, entity specifications must precede executable statements.
[*executable statements*]	They may contain ENTRY specifications (for defining entry points) and RETURN statements (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 hi	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 screen. 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	The previous example modified by inserting the RETURN statement in the definition of the subroutine HI. In this case, after calling the subroutine and

`` ! subroutine SUBROUTINE hi PRINT *,"Hello!" RETURN END ```	displaying the text `Hello!` on the screen, 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 or null value" RETURN ENTRY negative PRINT *,"strictly negative value" RETURN END ! calling program unit … IF(N < 0) THEN CALL negative ELSE CALL sign ENDIF … END ```	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 statements 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 shall return to the calling program unit. If the value of scalar N is not negative, then the SEMN subroutine is called and the statements are executed until the first RETURN is encountered (the message `positive value` is displayed and then the control returns to the calling program unit). Of course, the specification of an entry point only conditions the start from which the statements 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, then 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 statement with label 50, the return to the calling unit will be made to the actual arguments of the `CALL` statement (the value in A will be updated from the value of X, B from Y, and C from Z) and the first statement that follows will be executed (displaying the `negative value` text)

	and then jumping to the statement 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 statement 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 statement 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 statement with label 30). So `RETURN 1` means turning back through the first reserved position. If the arithmetic condition in the subroutine results in jumping to the statement 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 statement executed after the return will be the one with label 20 (the text `positive value` will be displayed after which the statement with label 30 will continue). So `RETURN 2` means turning back through the second reserved position.

User defined functions

In 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 a 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 statements. 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*), where *name_r* is the entity through which the result will be used in the function.
[*specifications*]	In all program units, entity specifications must precede executable statements.
[*executable statements*]	They may contain `ENTRY` specifications (for defining entry points) and `RETURN` statements (for returning to the program unit from which the function was called). **Attention:** it must also contain an expression that results in the value 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`.

The syntax for defining a function statement is as follows:

[*type*] *name*([*arguments*])=*expression*	Before the `FUNCTION` keyword, a *type* (intrinsic or derived) can also be specified, and specifying the arguments is optional (they are specified only if value transfer is desired between the calling program unit and the function), but the parentheses in which the arguments would be are mandatory. The arguments are considered notional, in the sense that at the time of function definition, their actual values are not known, only their type.

Calling a function is done by its name and specifying the effective arguments (if any) within 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 *,"number: "` `  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 *,"number: " 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 being optional there.
`` ! main program INTEGER on2,nr ! on2 function definition on2(nr)=nr/2 ! executable statements 10 PRINT *,"number: " 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 statement, so it must appear in the specification area.
`` PROGRAM factorial INTEGER f,i PRINT *,"i: " READ *,i PRINT *,"factorial of ",i,":",f(i) 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 ```	A "classic" example of a function defined as a self-referring (recursive) procedure for calculating the factorial value of a number. 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).

Examples:	Explanations:
MODULE prim INTEGER,PARAMETER :: A,B REAL E22(5,5) END *! using it in program units* SUBROUTINE P21 USE prim … END FUNCTION FU33(A,X) USE prim … END	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 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 (adapted after 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)
! initializing 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_width(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_width
! get the actual width
  SUBROUTINE get_width(col)
   INTEGER,INTENT(OUT) :: col
   col=nr_c
  END SUBROUTINE get_width
! 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,columns=>get_width`	Use the whole module, but temporarily replacing the name of the module procedure GET_WIDTH with the new name COLUMNS.

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 statements. 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.

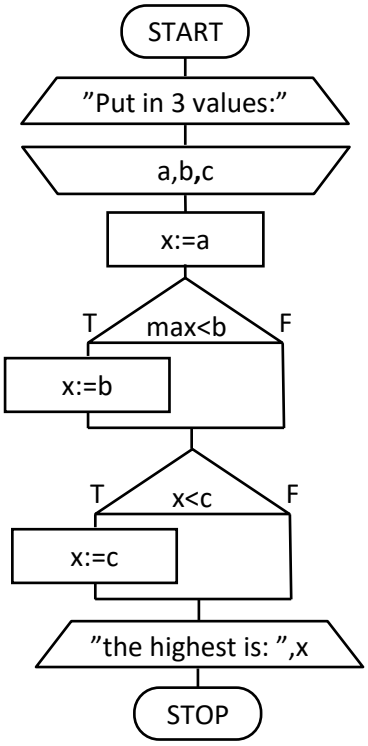
END [BLOCK DATA [*name*]] It is enough to use only the END keyword (if the data block has not been named then there is no *name* to specify).

Example:	Explanations:
! main program CHARACTER(6) Actor COMMON /zone1/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*/*components_list*[[,]…]
! data block for initialisation BLOCK DATA DIMENSION x(4) COMMON /zone1/x,name DATA x/3*1.,5/ CHARACTER(6) :: name="Adrian" END	Definition of a data block by 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.

Exercises

Transcribing some logical schemes (flowcharts) into Fortran

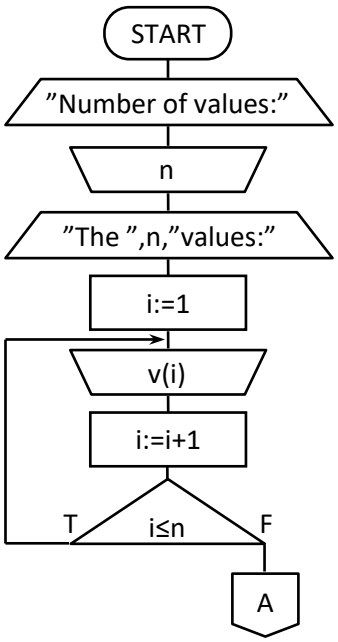
1. The highest value from a, b and c



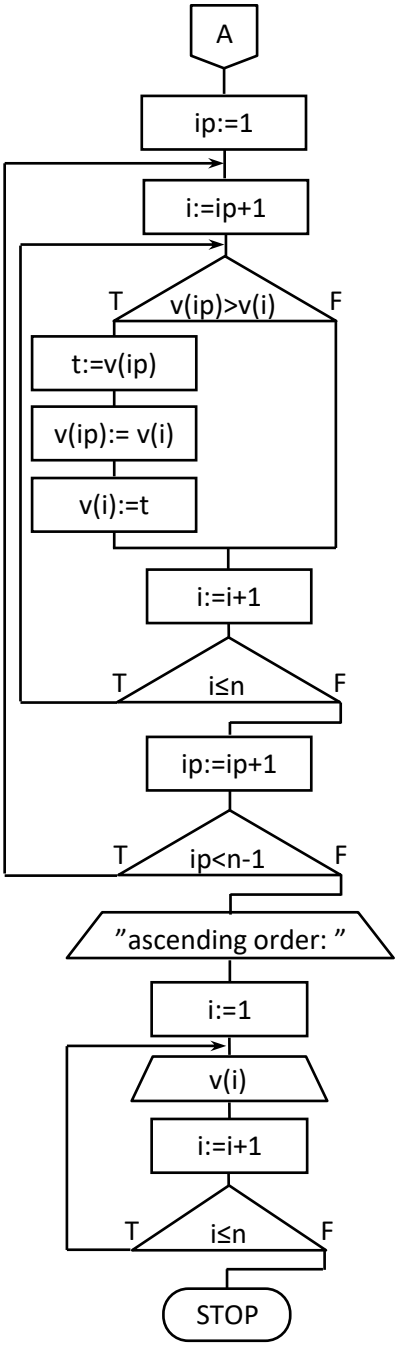

```
! 1st variant, elementary translated
      PRINT *,"Put in 3 values:"

      READ *,a,b,c

      x=a


      IF(x<b) THEN
        x=b
      ENDIF


      IF(x<c) THEN
         x=c
      ENDIF

      PRINT *,"the highest is: ",x
! marking the end of the program unit
      END
```

```
! 2nd variant, structured
      PRINT *,"Put in 3 values:"
      READ *,a,b,c
      x=a
      IF(x<b) x=b
      IF(x<c) x=c
      PRINT *,"the highest is: ",x
      END
```

2. The ascending ordering of values
   a) Pivot method

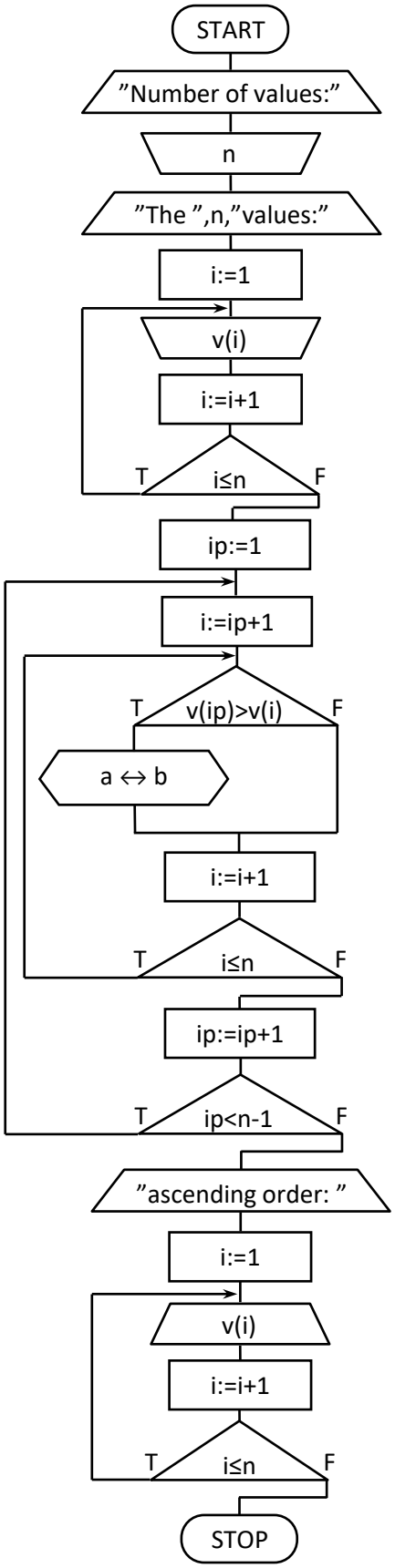



```
! 1st variant using static allocation and jumps
      DIMENSION v(30)
! the instructions from the logic scheme
      PRINT *,"Number of values:"

      READ *,n

      PRINT *,"The ",n,"values:"

      i=1

2     READ *,v(i)

      i=i+1


      IF(i<=n) GOTO 2

! continued on the next page
```

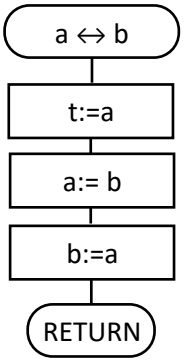



```
! continued from the previous page

      ip=1

21    i=ip+1

22    IF(v(ip)>v(i)) THEN
        t=v(ip)
        v(ip)=v(i)
        v(i)=t
      ENDIF
      i=i+1
      IF(i<=n) GOTO 22

      ip=ip+1

      IF(ip<=n-1) GOTO 21

      PRINT *,”ascending order:”

      i=1

23    PRINT *,v(i)

      i=i+1

      IF(i<=n) GOTO 23
! marking the end of the program unit
      END
-------------------------------------------------------------------------
!2nd variant, with dynamic allocation and loops
      ALLOCATABLE v(:)
      PRINT *,”Number of values:”
      READ *,n
      ALLOCATE(v(n))
      PRINT *,”The ”,n,”values:”
      READ *,(v(i),i=1,n)
      DO ip=1,n-1
       DO i=ip+1,n
        IF(v(ip)>v(i)) THEN
            t=v(ip)
            v(ip)=v(i)
            v(i)=t
        ENDIF
       ENDDO
      ENDDO
      PRINT *,”ascending order:”
      PRINT *,(v(i),i=1,n)
      DEALLOCATE(v)
      END
```

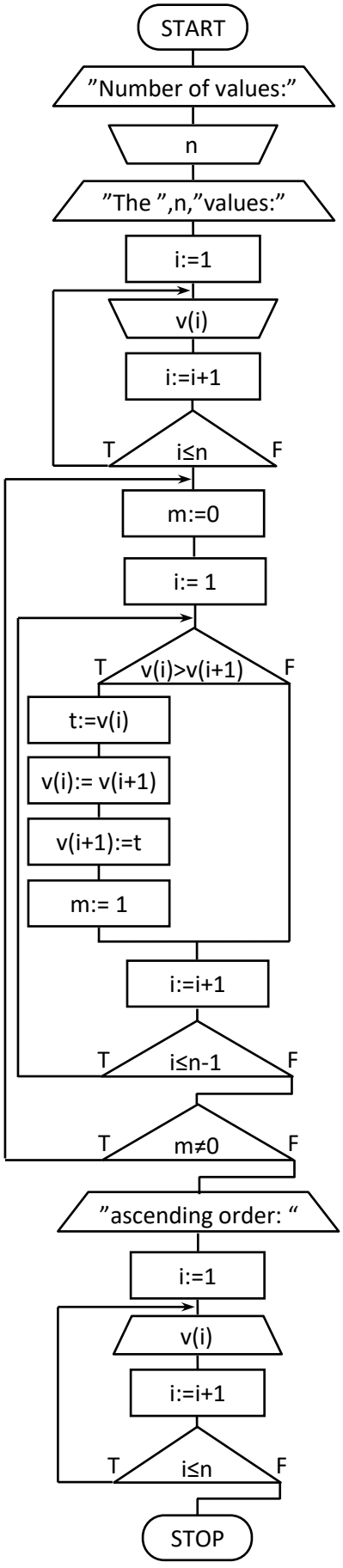
- variant with a modul transcribed as procedure




```
! 3rd variant, with internal procedure
  DIMENSION v(30)
! the statements from the logic scheme
  PRINT *,"Number of values:"

  READ *,n

  PRINT *,"The ",n,"values:"



  READ *,(v(i),i=1,n)

  i=i+1



  DO ip=1,n-1

    DO i=ip,n

      IF(v(ip)>v(i)) THEN

        CALL change(v(ip),v(i))

      ENDIF


    ENDDO



  ENDDO

  PRINT *,"ascending order: "


  PRINT *,(v(i),i=1,n)

  CONTAINS
! the internal procedure ->
      SUBROUTINE change(a,b)
      t=a
      a=b
      b=t
      RETURN
      END SUBROUTINE

 END
```



a) Marking method




```
! 1st variant using static allocation and jumps
      DIMENSION v(30)
! the statements from the logic scheme
PRINT *,"Number of values:"
      READ *,n
      PRINT *,"The ",n,"values:"
      i=1
2     READ *,v(i)
      i=i+1
      IF(i<=n) GOTO 2
23    m=0
      i=0
22    IF(v(ip)>v(i)) THEN
        t=v(ip)
        v(ip)=v(i)
        v(i)=t
        m=1
      ENDIF
      i=i+1
      IF(i<=n-1) GOTO 22
      IF(m/=n) GOTO 23
      PRINT *,"ascending order:"
      i=1
23    PRINT *,v(i)
      i=i+1
      IF(i<=n) GOTO 23
! marking the end of the program unit
      END
```

```
! 2nd variant using dynamic allocation and loops
      ALLOCATABLE v(:)
      PRINT *,"Number of values:"
      READ *,n
      ALLOCATE(v(n))
      PRINT *,"The ",n,"values:"
      READ *,(v(i),i=1,n)
23    m=0
      DO i=1,n-1
        IF(v(i)>v(i+1)) THEN
            t=v(i)
            v(i)=v(i+1)
            v(i+1)=t
        ENDIF
      ENDDO
      IF(m/=0) GOTO 23
      PRINT *,"ascending order:"
      PRINT *,(v(i),i=1,n)
      DEALLOCATE(v)
      END
```

Source file examples

1. Multiplying the terms of a matrix by a scalar value

```
      REAL  matrix(10,10)
      PRINT 1,"Number of lines and columns of the matrix (max.10x10):"
      READ *,nl,nc
      PRINT 1,"Put in the terms of the matrix:"
      READ *,((matrix(i,j),j=1,nc),i=1,nl)
      PRINT 1,"The scalar value:"
      READ *,s
      matrix=matrix*s
      DO i=1,nl
        PRINT *,(matrix(i,j)," ",j=1,nc)
      ENDDO
1     FORMAT(A,$)
      END
```

Variant reading the matrix from the A.TXT file:

```
      REAL  matrix(10,10)
      OPEN(1,FILE="A.TXT",STATUS="OLD",ERR=9)
      READ(1,*)nl,nc
      READ(1,*)((matrix(i,j),j=1,nc),i=1,nl)
      PRINT "(A,$)","The scalar value:"
      READ *,s
      matrix=matrix*s
      DO i=1,nl
        PRINT *,(matrix(i,j)," ",j=1,nc)
      ENDDO
9     PRINT *,"file A.TXT not found!"
      END
```

The structure of the A.TXT file: *number_of_lines*, *number_of_columns*
terms_of_the_matrix

For example, for the $\begin{vmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{vmatrix}$ matrix the A.TXT file can be created with the following content:

```
3,3
1,1,1,2,2,2,3,3,3
```

or:

```
3,3
1,1,1
2,2,2
3,3,3
```

2. The sum of the terms in a chosen column of a matrix

```
      REAL  matrix(10,10)
      CHARACTER r
      PRINT *,"Number of lines and columns of the matrix (max.10x10):"
      READ *,nl,nc
      PRINT *,"Put in the terms of the matrix:"
      READ *,((matrix(i,j),j=1,nc),i=1,nl)
1     PRINT *,"Column number:"
      READ *,ncol
      sum=0.
      DO i=1,nl
        sum=sum+matrix(i,ncol)
      ENDDO
      PRINT *,"The sum is: ",sum
      PRINT *,"Choose another column? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

Variant with dynamic memory allocation and reading the matrix from the A.TXT file:

```
      REAL,ALLOCATABLE :: matrix(:,:)
      CHARACTER r
      OPEN(1,FILE="A.TXT",STATUS="OLD",ERR=9)
      READ(1,*)nl,nc
      ALLOCATE(matrix(nl,nc))
      READ(1,*)((matrix(i,j),j=1,nc),i=1,nl)
1     PRINT *,"Column number:"
      READ *,ncol
      sum=0.
      DO i=1,nl
        sum=sum+matrix(i,ncol)
      ENDDO
      PRINT *,"The sum is: ",sum
      PRINT *,"Choose another column? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      DEALLOCATE(matrix)
9     PRINT *,"file A.TXT not found!"
      END
```

Note: the structure of the A.TXT file is like in the previous example.

3. Transpose of a matrix

```
      REAL,ALLOCATABLE :: matrix(:,:),mtransposed(:,:)
      CHARACTER r
1     PRINT *,"Number of lines:"
      READ *,nl
      PRINT *,"Number of columns:"
      READ *,nc
      ALLOCATE(matrix(nl,nc),mtransposed(nc,nl))
      DO i=1,nl
      PRINT *,"Terms on line ",i,":"
      READ *,(matrix(i,j),j=1,nc)
      ENDDO
      DO i=1,nl
        DO j=1,nc
      mtransposed(j,i)=matrix(i,j)
        ENDDO
      ENDDO
      PRINT *,"transposed matrix:"
      DO i=1,nc
        PRINT *,(mtransposed(i,j)," ",j=1,nl)
      ENDDO
      DEALLOCATE(matrix,mtransposed)
      PRINT *,"restart? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

4. Multiplying two square matrices, using data files

```
      REAL,ALLOCATABLE :: mat1(:,:),mat2(:,:),matres(:,:)
      CHARACTER r,filename_in(12),filename_out(12)
1     PRINT *,"Data input file:"
      READ *,filename_in
      OPEN(1,FILE=filename_in,STATUS='OLD',ERR=9)
      READ(1,*)n
      ALLOCATE(mat1(n,n),mat2(n,n),matres(n,n))
      READ(1,*)((mat1(i,j),j=1,n),i=1,n)
      READ(1,*)((mat2(i,j),j=1,n),i=1,n)
      matres=0.
      DO i=1,n
        DO j=1,n
          DO k=1,n
            matres(i,j)=matres(i,j)+mat1(i,k)*mat2(k,j)
          ENDDO
        ENDDO
      ENDDO
      PRINT *,"Output file:"
      READ *,filename_out
      OPEN(2,FILE=filename_out,STATUS='UNKNOWN',POSITION='APPEND')
      DO i=1,n
        WRITE(2,*)(matres(i,j)," ",j=1,n)
      ENDDO
      CLOSE(2)
      DEALLOCATE(mat1,mat2,matres)
      STOP
9     PRINT *,"Non-existent file name! Retry? (Y/N):"
      READ *,r ;IF(r=="Y".OR.r=="y") GOTO 1
      END
```

The structure of the data file:

n	(number of lines= number of columns)
terms_of_mat1	(n×n pieces)
terms_of _mat1	(n×n pieces)

For example, for the $\begin{vmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{vmatrix}$ and $\begin{vmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 3\end{vmatrix}$ matrices the data file can be created with the following content:

```
3
1,1,1,2,2,2,3,3,3
1,0,0,0,2,0,0,0,3
```

or:

```
3
1,1,1
2,2,2
3,3,3
1,0,0
0,2,0
0,0,3
```

5. Solving a quadratic equation of the form: $a \cdot x^2 + b \cdot x + c = 0$

```
      CHARACTER r
      INTEGER a,b,c
      PRINT *,"solving the equation a.x2+b.x+c=0"
1     PRINT *,"put in the coefficients a, b, c: "
      READ *,a,b,c
      SELECT CASE(a)
       CASE(0)
        IF(b==0) THEN
         IF(c==0) THEN
          PRINT *,"x can have any value"
         ELSE
          PRINT *,"mistake, c cannot be different from 0"
         ENDIF
        ELSE
         IF(c==0) THEN
          PRINT *,"non-quadratic equation, the solution is x=",0.
         ELSE
          PRINT *,"non-quadratic equation, the solution is x=",-c/b
         ENDIF
        ENDIF
       CASE DEFAULT
        delta=b**2-4*a*c
        IF(delta<0) THEN
         PRINT *,"complex roots"
         PRINT *,"x1=",-b/2/a,"+i(",SQRT(-delta)/2/a,")"
         PRINT *,"x2=",-b/2/a,"-i(",SQRT(-delta)/2/a,")"
        ELSEIF(delta==0.) THEN
         PRINT *,"identical roots, x1=x2=",-b/2/a
        ELSE
         PRINT *,"x1=",(-b+SQRT(delta))/2/a
         PRINT *,"x2=",(-b-SQRT(delta))/2/a
        ENDIF
      END SELECT
      PRINT *,"Restart? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

6. Solving a linear system of 2 equations with 2 unknowns (x and y): $\begin{cases} a \cdot x + b \cdot y = c \\ d \cdot x + e \cdot y = f \end{cases}$

```
      CHARACTER r
1     PRINT 2,'Coefficients a, b and c from equation "ax+by=c": '
      READ *,a,b,c
      PRINT 2,'Coefficients d, e and f from equation "dx+ey=f": '
      READ *,d,e,f
      delta=a*e-b*d
      IF(delta==0) THEN
       IF(b*f==c*e) THEN
         PRINT *,"indeterminate compatible system"
       ELSE
         PRINT *,"incompatible system"
       ENDIF
      ELSE
       PRINT *,"x=",(c*e-b*f)/delta,"y=",(-c*d+a*f)/delta
      ENDIF
2     FORMAT(A,\)
      PRINT 2,"Retry? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

7. Simulating a lotto draw with pseudorandom numbers

```
      CHARACTER r
      ALLOCATABLE nr(:)
1     PRINT 2," How many numbers are drawn: "
      READ *,n
      PRINT 2," from how many: "
      READ *,nmax
      ALLOCATE(nr(nmax))
      DO i=1,nmax
       nr(i)=i
      ENDDO
      man=RAND(TIME())
      DO i=1,n
       id=INT(RAND(0)*(nmax-i+1))+i
       man=nr(i)
       nr(i)=nr(id)
       nr(id)=man
      ENDDO
      DO i=1,n
       DO j=i,n
        IF(nr(i)>nr(j)) THEN
         man=nr(i)
         nr(i)=nr(j)
         nr(j)=man
        ENDIF
       ENDDO
      ENDDO
      PRINT *,(nr(i),i=1,n)
      DEALLOCATE(nr)
2     FORMAT(A,$)
      PRINT 2,"Retry? (Y/N):"
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

8. Simulating the simultaneous throwing of a pair of dice

```
      CHARACTER r
      DIMENSION n(2)
! initializing the pseudorandom number generator
      man=RAND(TIME())
1     PRINT 2,”the dice rolled:”
      DO i=1,2
       n(i)=INT(6*RAND(0))+1
       WRITE(*,”(1X,I1)”,ADVANCE=”NO”)n(i)
      ENDDO
! advance to a new row
      PRINT *
2     FORMAT(A,$)
      PRINT 2,”Retry? (Y/N):”
      READ *,r
      IF(r==”Y”.OR.r==”y”) GOTO 1
      END
```

9. Reading strings from the keyboard and displaying them using pointers

```
! defining a node type entity (with self-reference)
      TYPE node
        CHARACTER(60) row
        TYPE(node),POINTER :: next
      END TYPE
! defining the pointers that will be used
      TYPE(node),POINTER :: front,back,position
      CHARACTER(60)buffer
      CHARACTER r
1     NULLIFY(front,back)
      PRINT *,”press <Enter> to finish”
! recording typed strings
      DO
        WRITE(*,”(A,$)”)”type anything: ”
        READ(*,”(A)”)buffer
! exiting the cycle when only the <Enter> key is pressed
        IF(buffer==””) EXIT
        IF(.NOT.ASSOCIATED(front)) THEN
         ALLOCATE(front)
         back=>front
        ELSE
         ALLOCATE(back%next)
         back=>back%next
        ENDIF
        back%row=buffer
        NULLIFY(back%next)
      ENDDO
! displaying the typed strings
      position=>front
      DO WHILE(ASSOCIATED(position))
        WRITE(*,*)position%row
        position=>position%next
      ENDDO
      PRINT *,”Restart? (Y/N):”
      READ *,r
      IF(r==”Y”.OR.r==”y”) GOTO 1
      STOP
      END
```

10. Calculating the area and perimeter of a rectangle, right triangle or semicircle after selecting an option (using Hollerith constants, subroutines, entry points, and label returns)

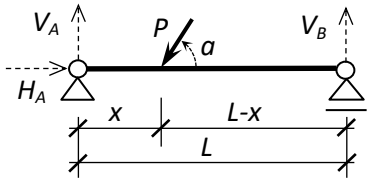
```
      CHARACTER op
1     PRINT *,40HArea and perimeter calculation
      PRINT *,40Hchoose one of the options:
      PRINT *,40H D - for rectangle
      PRINT *,40H T - for right triangle
      PRINT *,40H S - for semicircle
      PRINT *,40H X – to exit the program
2     PRINT 3,9H option:
3     FORMAT(A,\)
      READ *,op
      SELECT CASE(op)
      CASE(”D”,”d”) ; CALL d(*2)
      CASE(”T”,”t”) ; CALL t(*2)
      CASE(”S”,”s”) ; CALL s(*2)
      CASE(”X”,”x”)
       STOP
      CASE DEFAULT
       PRINT *,”invalid option”
       GOTO 2
      END SELECT
      GOTO 2
      END

      SUBROUTINE d(*)
      PARAMETER (pi=3.14159)
3     FORMAT(A,\)
      PRINT 3,”the lengths of the two sides: ”
      READ *,a,b
      CALL test(a,b,*4)
      PRINT *,”Area=”,a*b,”Perimeter=”,(a+b)*2
      RETURN
      ENTRY t
       PRINT 3,”the lengths of the two perpendicular sides: ”
       READ *,a,b
       CALL test(a,b,*4)
       PRINT *,”Area=”,a*b/2,”Perimeter=”,a+b+SQRT(a**2+b**2)
       RETURN
      ENTRY s
       PRINT 3,”base length (diameter): ”
       READ *,a
       CALL test(a,1.,*4)
       PRINT *,”Area=”,pi*(a/4)**2,”Perimeter=”,pi*a/2
       RETURN
4     RETURN 1
      CONTAINS
         SUBROUTINE test(a,b,*)
          IF(a==0.or.b==0) THEN
            PRINT *,”Cannot calculate with null value”
            RETURN 1
          ELSEIF(a<0.or.b<0) THEN
            PRINT *,”negative value?!”
          ENDIF
          RETURN
         END SUBROUTINE test
      END
```

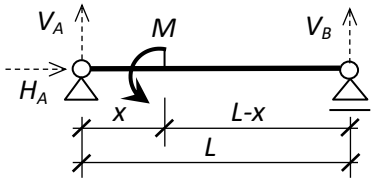
Variant using 2 functions with entry points instead of the 3 subroutines:

```
      CHARACTER op
1     PRINT *,40HArea and perimeter calculation
      PRINT *,40Hchoose one of the options:
      PRINT *,40H D - for rectangle
      PRINT *,40H T - for right triangle
      PRINT *,40H S - for semicircle
      PRINT *,40H X – to exit the program
2     PRINT 3,9H option:
3     FORMAT(A,\)
      READ *,op
      SELECT CASE(op)
      CASE(”D”,”d”) ; PRINT 3,”the lengths of the two sides: ”
       READ *,a,b ; CALL test(a,b,*2)
       PRINT *,”Area=”,da(a,b),”Perimeter=”,dp(a,b)
      CASE(”T”,”t”)
       PRINT 3,”the lengths of the two perpendicular sides: ”
       READ *,a,b ; CALL test(a,b,*2)
       PRINT *,”Area=”,ta(a,b),”Perimeter =”,tp(a,b)
      CASE(”S”,”s”) ; PRINT 3,”base length (diameter): ”
       READ *,a ; CALL test(a,1.,*2)
       PRINT *,”Area=”,sa(a),”Perimeter =”,sp(a)
      CASE(”X”,”x”)
       STOP
      CASE DEFAULT
       PRINT *,”invalid option”
       GOTO 2
      END SELECT
      GOTO 2
      CONTAINS
         SUBROUTINE test(a,b,*)
          IF(a==0.or.b==0) THEN
            PRINT *,”Cannot calculate with null value”
            RETURN 1
          ELSEIF(a<0.or.b<0) THEN
            PRINT *,”negative value?!”
          ENDIF
          RETURN
         END SUBROUTINE test
      END

      FUNCTION da(a,b)
      da=a*b ; RETURN
      ENTRY dp(a,b)
      dp=(a+b)*2 ; RETURN
      ENTRY ta(a,b)
      ta=a*b/2 ; RETURN
      ENTRY tp(a,b)
      tp=a+b+SQRT(a**2+b**2)
      END

      FUNCTION sa(a)
      PARAMETER (pi=3.14159)
      sa=pi*(a/4)**2 ; RETURN
      ENTRY sp(a)
      sp=pi*a/2
      END
```

11. Determining the reactions of a simply supported beam subjected to a point load, and calculation of the forces in a cross-section on the beam axis:

Sketches with notations:





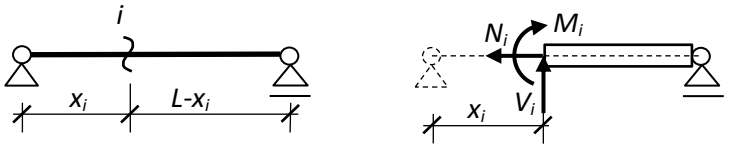
Formulas used: $V_A = P \cdot \sin(a) \cdot \frac{(L-x)}{L}$; $H_A = P \cdot \cos(a)$; $V_A = \frac{M}{L}$; $H_A = 0$; $V_B = -\frac{M}{L}$

$V_B = P \cdot \sin(a) \cdot \frac{x}{L}$

```
      INCLUDE "f.txt"
1     PRINT *,"Calculation of the reactions at the ends of a simply", &
       "supported beam, loaded with a force or moment"
      PRINT *
3     PRINT *,"for a force load, press P"
      PRINT *,"for a moment load, press M"
      PRINT 2,"load type (P/M): "
      READ *,t
      PRINT 2,"the length ""L"" of the beam [m]: "
      READ *,l
5     PRINT 2,"distance ""x"" [m]: "
      READ *,x
      IF(x<0.OR.x>l) THEN
       PRINT *,"the load is not on the beam!"
       GOTO 5
      ENDIF
      SELECT CASE(t)
       CASE("P","p")
        PRINT 2,"force intensity ""P"" [kN]: "
        READ *,p
        PRINT 2,"angle ""a"" to the beam axis [degrees]: "
        READ *,a
        va=p*SIN(a*pi/180)*(l-x)/l
        ha=p*COS(a*pi/180)
        vb=p*SIN(a*pi/180)*x/l
        ra=SQRT(va**2+ha**2)
       CASE("M","m")
        PRINT 2,"moment intensity ""M"" [kN.m]: "
        READ *,m
        va=m/l ; ha=0. ; vb=-m/l
       CASE DEFAULT
        PRINT *," invalid option! "
        GOTO 3
      END SELECT
      PRINT 4,"VA= ",va,"kN; HA= ",ha,"kN; VB= ",vb,"kN"
      IF(.NOT.(a==0.OR.a==90)) PRINT 4,"RA= ",ra,"kN at", &
       ATAN(va/ha)*180/pi,"degrees"
      CALL FINT(l,x,t,va,ha,p,m,vb,a)
      PRINT 2," Restart? (Y/N): "
      READ *,r
      IF(r=="Y".OR.r=="y") GOTO 1
      END
```

The FINT subroutine for the calculation of the internal forces in a chosen cross-section:

Illustration:




```
SUBROUTINE FINT(l,x,t,va,ha,p,m,vb,a)
INCLUDE "f.txt"
PRINT 2,'"xi" for the cross-section:'
READ *,xi
SELECT CASE(t)
 CASE("P","p")
  IF(xi<x) THEN
   fn=ha
   fv=va
   fm=va*xi
  ELSE
   fn=0.
   fv=va-p*SIN(a*pi/180)
   fm=va*xi-p*SIN(a*pi/180)*(xi-x)
  ENDIF
 CASE("M","m")
  IF(xi<x) THEN
   fn=ha
   fv=va
   fm=va*xi
  ELSE
   fn=0.
   fv=va
   fm=va*xi-m
  ENDIF
END SELECT
PRINT 4,"Ni= ",fn,"kN; Vi= ",fv,"kN; Mi= ",fm,"kN.m"
RETURN
END
```

The contents of the F.TXT file included in the program (this file must be in the same folder as the source files):

```
      CHARACTER r,t
      REAL l,m
      DATA pi/3.14159/
2     FORMAT(1X,A,$)
4     FORMAT(1X,3(A,F8.2),A)
```

Resources

The Home of Fortran Standards (JTC1/SC22/WG5): https://wg5-fortran.org/

The Fortran programming language: https://fortran-lang.org/

Fortran Wiki: https://fortranwiki.org/

Fortranplus | Fortran Information. https://www.fortranplus.co.uk/fortran-information/

The G95 Project / Running G95 (options, error codes): https://g95.sourceforge.net/docs.html

Some online available books and tutorials

Metcalf M., Reid J. K.: *Fortran 90/95 Explained*, Oxford University Press, 1996. https://archive.org/details/fortran9095expla0000metc

Sandu A.: *Lecture Notes. Introduction to Fortran 95 and Numerical Computing. A Jump-Start for Scientists and Engineers*. Michigan Technological University, 2001. https://www-eio.upc.edu/lceio/manuals/Fortran95-manual.pdf

van Mourik T.: *Fortran 90/95 Programming Manual*. University College London, 2005. https://www-eio.upc.edu/lceio/manuals/Fortran95-manual.pdf

Nicholson J. A.: *Introduction to Programming using FORTRAN 95*, 2011. https://www.fortrantutorial.com/documents/IntroductionToFTN95.pdf

Learn – Fortran Programming Language. https://fortran-lang.org/en/learn/

Fortran Tutorial – Free Guide to Programming Fortran 90/95. https://www.fortrantutorial.com/

TutorialsPoint - Fortran Tutorial: https://www.tutorialspoint.com/fortran/index.htm

The Irish Centre for High-End Computing (ICHEC): Fortran Tutorial. https://www.ichec.ie/academic/national-hpc/documentation/fortran-tutorial