

# Introduction to structured programming with Fortran

<https://forge.uclouvain.be/barriat/learning-fortran>



**Pierre-Yves Barriat**

**November 13, 2024**

CISM/CÉCI Training Sessions

# Fortran : shall we start ?

- You already know one computer language ?
- You understand the very basic programming concepts :
  - What is a variable, an assignment, function call, etc. ?
  - Why do I have to compile my code ?
  - What is an executable ?
- You (may) already know some Fortran ?
- How to proceed from old Fortran, to much more modern languages like Fortran 90/2003 ?

# Why learn Fortran ?

- Because of the execution **speed** of a program
- Well suited for numerical computations :  
more than 45% of scientific applications are in Fortran
- **Fast** code : compilers can optimize well
- Optimized **numerical libraries** available
- Fortran is a **simple** language and it is (kind-of) **easy to learn**

# Fortran is simple

- **We want to get our science done! Not learn languages!**
- How easy/difficult is it really to learn Fortran ?
- The concept is easy:  
*variables, operators, controls, loops, subroutines/functions*
- **Invest some time now, gain big later!**

# History

## FORMula TRANslation

invented 1954-8 by John Backus and his team at IBM

- FORTRAN 66 (ISO Standard 1972)
- FORTRAN 77 (1978)
- Fortran 90 (1991)
- Fortran 95 (1997)
- Fortran 2003 (2004) → "standard" version
- Fortran 2008 (2010)
- Fortran 2018 (11/2018)

# Starting with Fortran 77

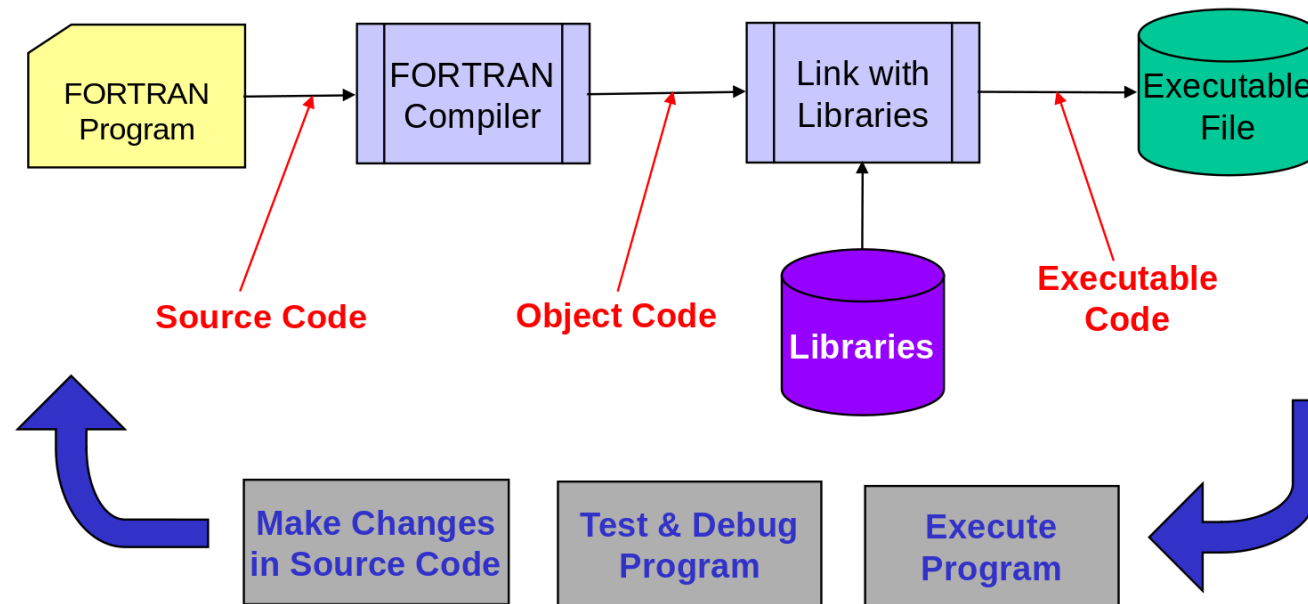
- Old Fortran provides only the absolute minimum !
- Basic features
  - data containers (integer, float, ...), arrays, basic operators, loops, I/O, subroutines and functions
- But this version has flaws
  - no dynamic memory allocation, old & obsolete constructs, "spaghetti" code, etc.
- Is that enough to write code ?

# Fortran 77 → Fortran >90

- If Fortran 77 is so simple, why is it then so difficult to write good code ?
- Is simple really better ?
  - ⇒ Using a language allows us to express our thoughts (on a computer)
- A more sophisticated language allows for more complex thoughts
- More language elements to get organized
  - ⇒ Fortran 90/95/2003 (recursive, OOP, etc)

# How to build a FORTRAN program

FORTRAN is a compiled language (like C) so the source code (what you write) must be converted into machine code before it can be executed (e.g. Make command)

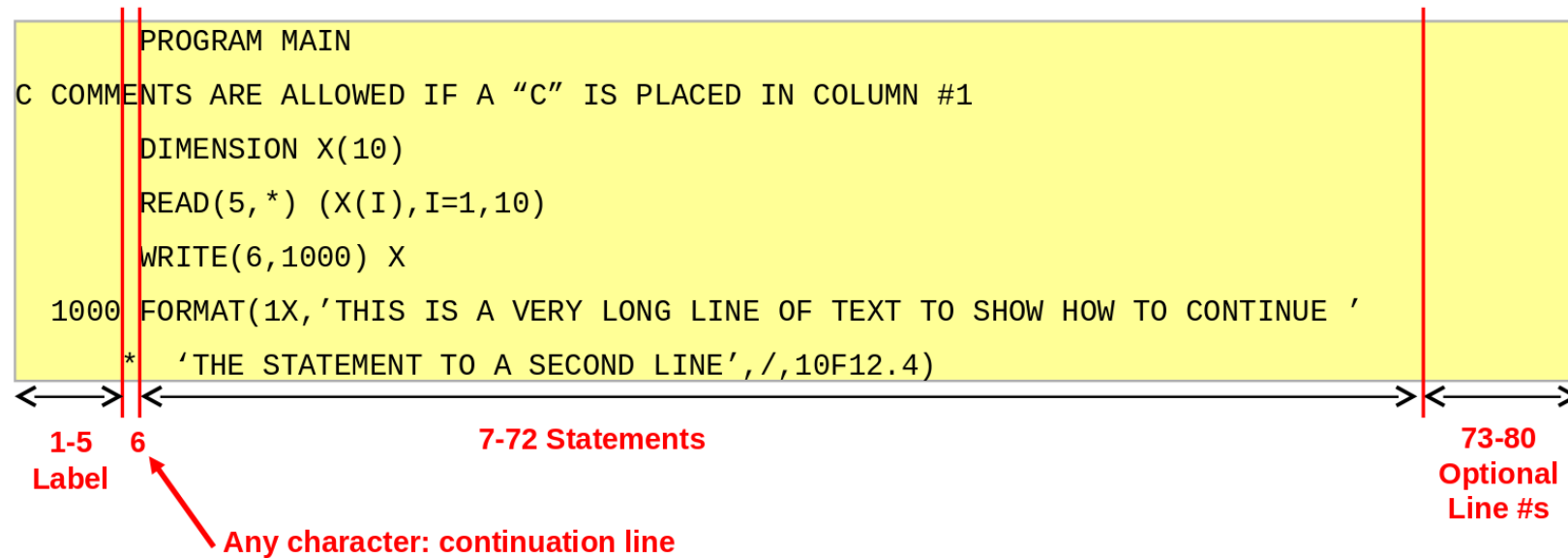


Fortran 77 source code [hello\\_world.f90](#)



# FORTRAN 77 Format

This version requires a fixed format for programs



- max length variable names is 6 characters
- alphanumeric only, must start with a letter
- character strings are case sensitive

# FORTRAN >90 Format

Versions >90 relaxe these requirements:

- comments following statements ( `!` delimiter)
- long variable names (31 characters)
- containing only letters, digits or underscore
- max row length is 132 characters
- can be max 39 continuation lines
- if a line is ended with ampersand ( `&` ), the line continues onto the next line
- semicolon ( `;` ) as a separator between statements on a single line

# Program organization

Most `FORTRAN` programs consist of a main program and one or more subprograms

There is a fixed order:

```
Heading  
Declarations  
Variable initializations  
Program code  
Format statements  
  
Subprogram definitions  
(functions & subroutines)
```

# Data type declarations

Basic data types are :

- `INTEGER` : integer numbers (+/-)
- `REAL` : floating point numbers
- `DOUBLE PRECISION` : extended precision floating point
- `CHARACTER*n` : string with up to **n** characters  
    `character(len=n)`
- `LOGICAL` : takes on values `.TRUE.` or `.FALSE.`

`INTEGER` and `REAL` can specify number of bytes to use

Default is: `INTEGER*4` and `REAL*4`

`DOUBLE PRECISION` is same as `REAL*8`

or `REAL(8)`

```
program variables_example
  integer :: a = 5
  real :: b = 3.14
  character(len=10) :: name = "Fortran"
  logical :: is_programming_fun = .true.
end program variables_example
```

# Arrays

Arrays of any type must be declared with the `dimension` attribute in **F77**

```
! declare and initialize a vector
integer, dimension(5) :: nums = (/1, 2, 3, 4, 5/)
! declare a 3 x 5 array
real, dimension nums(3,5)
! declare a vector with 30 characters strings in each element
character(30), dimension name(50)
```

In **>F90**, you *can* remove the `dimension` attribute

```
! declare a 3 x 5 array
real nums(3,5)
```

Arrays can be multi-dimensional (up to 7 in F77) and are indexed using ( )

| `my_array(3)` or `force(4,2)`

Indices are by default defined as `1...N`

We can specify index range in declaration

| `INTEGER arr(0:11)` : `arr` is dimensioned from `0-11` (12 elements)

Whole array reference (only in >F90): `arr(:)=-8` assigns -8 to all elements in `arr`

| Avoid `arr=-8` assignement

Arrays are stored in column order (1st column, 2nd column, etc) so accessing by **incrementing row index first** usually is **fastest**

```
real, dimension(1000, 1000) matrix

do j = 1, 1000          ! Column loop (outer loop)
  do i = 1, 1000        ! Row loop (inner loop)
    matrix(i, j) = matrix(i, j) * 2.0
  end do
end do
```

**column major** ordering in Fortran vs. **row major** ordering in C

A compiler (with sufficient optimization flags) may re-order loops automatically



# Implicit vs Explicit declarations

By default, an implicit type is assumed depending on the first letter of the variable name:

- A-H, 0-Z define REAL variables
- I-N define INTEGER variables

Can use the IMPLICIT statement:

```
IMPLICIT REAL (A-Z)
```

makes all variables REAL if not declared

## Good habit

Force explicit type declarations :

```
IMPLICIT NONE
```

User must explicitly declare all variable types

# Assignment statements

**Old** assignment statement: `<label> <variable> = <expression>`

- `<label>` : statement label number (1 to 99999)
- `<variable>` : FORTRAN variable  
(max 6 characters, alphanumeric only for standard FORTRAN 77)

## Expression:

- Numeric expressions: `var = 3.5*cos(theta)`
- Character expressions: `day(1:3) = 'TUE'`
- Relational expressions: `flag = ans .GT. 0`
- Logical expressions: `flag = F1 .OR. F2`

# Numeric expressions

Arithmetic operators: precedence: `**` (*high*) → `-` (*low*)

Operator	Function
<code>**</code>	exponentiation
<code>*</code>	multiplication
<code>/</code>	division
<code>+</code>	addition
<code>-</code>	subtraction

Numeric expressions are up-cast to the highest data type in the expression according to the precedence:

*(low)* logical → integer → real → complex *(high)*

and smaller byte size *(low)* to larger byte size *(high)*

## Examples:

Fortran source code [01\\_arith.f90](#)

Fortran source code [02\\_sphere.f90](#)

# Character expressions

Only built-in operator is **Concatenation** defined by `//`

```
'ILL'//'- '//'ADVISED'
```

`character` arrays are most commonly encountered

- treated like any array (indexed using `:` notation)
- fixed length (usually padded with blanks)

## Example:

```
CHARACTER FAMILY*16  
FAMILY = 'GEORGE P. BURDELL'  
  
PRINT*, FAMILY(:6)  
PRINT*, FAMILY(8:9)  
PRINT*, FAMILY(11:)  
PRINT*, FAMILY(:6)//FAMILY(10:)
```

```
GEORGE  
P.  
BURDELL  
GEORGE BURDELL
```

# Relational expressions

Two expressions whose values are compared to determine whether the relation is true or false

- may be numeric (common) or non-numeric

character strings can be compared

- done character by character
- shorter string is padded with blanks for comparison



Operator	Relationship
<code>.LT.</code> or <code>&lt;</code>	less than
<code>.LE.</code> or <code>&lt;=</code>	less than or equal to
<code>.EQ.</code> or <code>==</code>	equal to
<code>.NE.</code> or <code>/=</code>	not equal to
<code>.GT.</code> or <code>&gt;</code>	greater than
<code>.GE.</code> or <code>&gt;=</code>	greater than or equal to

# Logical expressions

Consists of one or more logical operators and logical, numeric or relational operands

- values are `.TRUE.` or `.FALSE.`
- need to consider overall operator precedence

can combine logical and integer data with logical operators but this is tricky  
**(avoid!)**

F77 Operator	>F90 Operator	Example	Meaning
<code>.AND.</code>	<code>&amp;&amp;</code>	<code>A .AND. B</code>	logical <code>AND</code>
<code>.OR.</code>	<code>  </code>	<code>A .OR. B</code>	logical <code>OR</code>
<code>.EQV.</code>	<code>==</code>	<code>A .EQV. B</code>	logical equivalence
<code>.NEQV.</code>	<code>/=</code>	<code>A .NEQV. B</code>	logical inequivalence
<code>.XOR.</code>	<code>/=</code>	<code>A .XOR. B</code>	exclusive <code>OR</code> (same as <code>.NEQV.</code> )
<code>.NOT.</code>		<code>.NOT. A</code>	logical negation

# Unconditional **GO TO** in FORTRAN 77

- Syntax: **GO TO** label
- Unconditional transfer to labeled statement

```
10  -code-  
    GO TO 30  
    -code that is bypassed-  
30  -code that is target of GOTO-  
    -more code-  
    GO TO 10
```

- **Problem** : leads to confusing "*spaghetti code*" 🌟

# Spaghetti Code in F77 (and before)

Use of `GO TO` and arithmetic `IF` 's leads to bad code that is very hard to maintain

```
10  IF (KEY.LT.0) GO TO 20
    TEST=TEST-1
    THETA=ATAN(X,Y)
    GO TO 30
20  TEST=TEST+1
    THETA=ATAN(-X,Y)
30  CONTINUE
```

# IF ELSE IF statement

Basic version:

```
IF (KSTAT.EQ.1) THEN  
  CLASS='FRESHMAN'  
ELSE IF (KSTAT.EQ.2) THEN  
  CLASS='SOPHOMORE'  
ELSE IF (KSTAT.EQ.3) THEN  
  CLASS='JUNIOR'  
ELSE IF (KSTAT.EQ.4) THEN  
  CLASS='SENIOR'  
ELSE  
  CLASS='UNKNOWN'  
ENDIF
```

# Loop statements (old versions)

**do** loop: structure that executes a specified number of times

## *Spaghetti Code Version*

```
      K=1  
10    PRINT*, A(K)  
      K=K+1  
      IF (K.LE.10) GO TO 10  
20    CONTINUE
```

## *Fortran 77 Version*

```
      DO 100 K=1, 10  
      PRINT*, A(K)  
100   CONTINUE
```

# Loop Statements (>F90)

```
DO K=1, 10  
  WRITE(*, *) A(K)  
END DO
```

can include a third parameter to specify increments, including negative values

```
R=10  
DO WHILE (R.GE.0)  
  VOL=2*PI*R**2  
  R=R-1  
END DO
```

loop will not execute at all if `logical_expr` is not true at start



# Comments on Loop Statements

## To exit a loop

- in old versions: use a `GO TO` statement
- in new versions: use an `EXIT` statement

you cannot transfer out of multiple nested loops with a single `EXIT`  
use named loops if needed : `myloop : do i=1,n` and then `EXIT myloop`

## To skip to next loop cycle

- in new versions **only** : use a `CYCLE` statement

# File I/O

Much of early FORTRAN was devoted to reading input data from "cards" and writing to a line printer

Today, most I/O is to and from a file: it requires more extensive I/O capabilities standardized until FORTRAN 77

**I/O** = communication between a program and the outside world

- opening and closing a file with `OPEN` & `CLOSE`
- data reading & writing with `READ` & `WRITE`
- can use **unformatted** `READ` & `WRITE` if no human readable data are involved (much faster access, smaller files)

```
program io_example

  integer file_id = 105
  character(len=*) file_name = "data.txt"

  open(unit=file_id, file=file_name)
  write(file_id, *) "Hello, file!"
  close(file_id)

end
```

Device number `*` is by default the screen (or *standard output* - also 6)

device numbers 1-7 are defined as standard I/O devices

# FORMAT statement

data type	format descriptors	example
integer	iw	<code>write(*,'(i5)') int</code>
real ( <i>decimal</i> )	fw.d	<code>write(*,'(f7.4)') x</code>
real ( <i>exponential</i> )	ew.d	<code>write(*,'(e12.3)') y</code>
character	a, aw	<code>write(*,'(a)') string</code>
logical	lw	<code>write(*,'(l2)') test</code>
spaces & tabs	wx & tw	<code>write (*,'(i3,2x,f6.3)') i, x</code>
linebreak	/	<code>write (*,'(f6.3,/,f6.3)') x, y</code>

# WRITE statement

```
WRITE(*,1000) A,B,KEY
1000 FORMAT(F12.4,E14.5,I6)
```

```
|-----+-----0-----+-----0-----+-----+-----|
      1234.5678   -0.12345E+02      12
```

Each `WRITE` produces one or more output lines as needed to write out `variable_list` using `format` statement

`variable_list` can include implied `DO`

```
write(105,1000) (A(I),I=1,10)
```

# READ statement

```
      READ(105,1000) A,B,C  
1000 FORMAT(3F12.4)
```

Each **READ** reads one line of data

any remaining data in a line is dropped if not translated in **variable\_list**

**variable\_list** can include implied **DO**

```
read(105,1000) (A(I), I=1, 10)
```

- input items can be integer, real or character
- characters must be enclosed in `' '` (or `" "`)
- input items are separated by commas
- input items must agree in type with variables in `variable_list`
- each `READ/WRITE` processes a new record (line)

## Example

Fortran 90 source code [04\\_plot.f90](#)

# Advanced example (>F90)

```
character(len=*) :: x_name
integer          :: ierr, iSize, guess_unit
logical          :: itsopen, itexists
!
inquire(file=trim(x_name), size=iSize, number=guess_unit, opened=itsopen, exist=itexists)
if ( itsopen ) close(guess_unit, status='delete')
!
open(902,file=trim(x_name),status='new',iostat=ierr)
!
if (iSize <= 0 .OR. .NOT.itexists) then
    open(902,file=trim(x_name),status='new',iostat=ierr)
    if (ierr /= 0) then
        ...
        close(902)
    endif
    ...
endif
```



# NAMelist

It is possible to pre-define the structure of input and output data using `NAMelist` in order to make it easier to process with `READ` and `WRITE` statements

- Use `NAMelist` to define the data structure
- Use `READ` or `WRITE` with reference to `NAMelist` to handle the data in the specified format

This is not part of standard F77 but it is included in >F90

## NAMELIST - cont'd

On input, the `NAMelist` data must be structured as follows:

```
&INPUT  
  THICK=0.245,  
  LENGTH=12.34,  
  WIDTH=2.34,  
  DENSITY=0.0034  
/
```

Fortran 90 source code [05\\_namelist.f90](#)

Namelist file [05\\_namelist.def](#)

# Internal `WRITE` statement

Internal `WRITE` does same as `ENCODE` in F77 : **a cast to string**

```
INTEGER J,K  
CHARACTER(50) CHAR50  
J=1  
K=2  
WRITE(CHAR50,*) J,K
```

Results:

```
CHAR50= '    1    2'
```

# Internal **READ** statement

Internal **READ** does same as **DECODE** in F77 : **a cast from string**

```
INTEGER K  
REAL A, B  
CHARACTER(80) REC80  
REC80(1)='1.2'  
REC80(2)='2.3'  
REC80(3)='-5'  
READ(REC80, *) A, B, K
```

Results:

```
A=1.2, B=2.3, K=-5
```

# Structured programming

Structured programming is based on subprograms (functions and subroutines) and control statements (like `IF` statements or loops) :

- structure the control-flow of your programs (e.g. give up the `GO TO` )
- improved readability
- lower level aspect of coding in a smart way

It is a **programming paradigm** aimed at improving the quality, clarity, and access time of a computer program

# Functions and Subroutines

Subprograms allow **structured coding**

**FUNCTION** : returns single explicit function value for given function arguments

it's also a variable → so must be declared !

**SUBROUTINE** : any values returned must be returned through the arguments

no explicit subroutine value is returned !

- Subprograms are not recursive in F77
- Subprograms use a separate namespace (variables are local)

Subprograms should (must) include at least one `RETURN`

`FUNCTION` example:

```
REAL FUNCTION AVG3(A, B, C)
REAL A, B, C
AVG3=(A+B+C)/3
RETURN
END
```

Use:

```
AV = WEIGHT*AVG3(A1, F2, B2)
```

`FUNCTION` type is implicitly defined as REAL

Subroutine is invoked using the `CALL` statement

```
SUBROUTINE AVG3S(A, B, C, AVERAGE)
REAL A, B, C, AVERAGE
AVERAGE=(A+B+C)/3
END
```

Use:

```
CALL AVG3S(A1, F2, B2, AVR)
RESULT = WEIGHT*AVR
```

Any returned values must be returned through argument list



# Arguments

Arguments in subprogram are **dummy** arguments

arguments used in invocation are called "actual" or "real"

- passed by **reference** (memory address) if given as *symbolic*
  - the subprogram can then alter the actual argument value since it can access it by reference
- passed by **value** if given as *literal* (so cannot be modified)

```
CALL AVG3S(A1, 3.4, C1, QAV)
```

2nd argument is passed by value and others by reference

Arguments used in invocation (by calling program) may be *variables, array names, literals, expressions or function names*

Using symbolic arguments (variables or array names) is the **only way** to return a value (result) from a `SUBROUTINE`

It is considered **BAD coding practice**, but functions can return values by changing the value of arguments

this type of use should be strictly **avoided** !

The `INTENT` keyword (>F90) increases readability and enables better compile-time error checking

```
SUBROUTINE AVG3S(A, B, C, AVERAGE)
  IMPLICIT NONE
  REAL, INTENT(IN)      :: A, B
  REAL, INTENT(INOUT)   :: C      ! default
  REAL, INTENT(OUT)     :: AVERAGE

  A = 10                  ! Compilation error
  C = 10                  ! Correct
  AVERAGE=(A+B+C)/3      ! Correct
END
```

compiler uses `INTENT` for error checking and optimization

# FUNCTION versus Array

REMAINDER(4,3) could be a 2D array or it could be a reference to a function

If the name, including arguments, **matches an array declaration**, then it is taken to be an array, **otherwise**, it is assumed to be a FUNCTION

Be careful about implicit versus explicit type declarations with FUNCTION

```
PROGRAM MAIN
  INTEGER REMAINDER
  ...
  KR = REMAINDER(4,3)
  ...
END

INTEGER FUNCTION REMAINDER(INUM, IDEN)
  ...
END
```

# Examples

Fortran 77 source code [03\\_histogram.f](#)

Fortran 90 source code [03\\_histogram.f90](#)

# Arrays with Subprograms

Arrays must be passed by reference to subprogram

How do you tell subprogram how large the array is ?

answer varies with FORTRAN version and vendor (dialect)...

- when an array element, e.g. `A(1)` , is used in a subprogram invocation , it is passed as a reference (address), just like a simple variable
- when an array is used by name in a subprogram invocation, it is passed as a reference to the entire array.

the array must be appropriately dimensioned (and this can be tricky...)

# Arrays - dynamic allocation

Using `ALLOCATABLE` on declaration, and using `ALLOCATE` and `DEALLOCATE` later

```
integer :: m, n
integer, allocatable :: idx(:)
real, allocatable :: mat(:, :)
m = 100 ; n = 200
allocate( idx(0:m-1))
allocate( mat(m, n))
...
deallocate(idx , mat)
```

It exists many array intrinsic functions

SIZE, SHAPE, SUM, ANY, MINVAL, MAXLOC, RESHAPE, DOT\_PRODUCT,  
TRANSPOSE, WHERE, FORALL, etc

# COMMON & MODULE Statement

A variable declared in a `Main` program can be made accessible to subprograms

- `COMMON` statement allows variables to have a more extensive scope
  - | can group into **labeled** `COMMON`
- with > F90, it's better to use a `MODULE` subprogram
  - | this can be selective (don't have to share all everywhere) with `ONLY`

Fortran 77 source code [06\\_common.f](#)

Fortran 90 source code [06\\_module.f90](#)



# Hands-on

Fortran 90 source code [07\\_plot\\_newton.f90](#)

Fortran 90 source code [07\\_newton.f90](#)

Text file [08\\_ChristmasTree.txt](#)

# Data Type Declarations

FORTRAN >90 allows user derived types

```
TYPE my_variable
  character(30)      :: name
  integer            :: id
  real(8)            :: value
  integer, dimension(3,3) :: dimIndex
END TYPE variable

type(my_variable) var
var%name = "salinity"
var%id   = 1
```

# Subprograms type

`MODULE` are subprograms that allow modular coding and data encapsulation

The interface of a subprogram type is **explicit** or **implicit**

Several types of subprograms:

- `intrinsic` : explicit - defined by Fortran itself (trigonometric functions, etc)
- `module` : explicit - defined with `MODULE` statement and used with `USE`
- `internal` : explicit - defined with `CONTAINS` statement inside (sub)programs
- `external` : implicit (but can be manually (re)defined explicit) - e.g. **libraries**

Differ with the **scope**: what data and other subprograms a subprogram can access

# MODULE type



```
MODULE example
  IMPLICIT NONE
  INTEGER, PARAMETER :: index = 10
  REAL(8), SAVE      :: latitude
CONTAINS
  FUNCTION check(x) RESULT(z)
    INTEGER :: x, z
    ...
  END FUNCTION check
END MODULE example
```

```
PROGRAM myprog
  USE example, ONLY: check, latitude
  IMPLICIT NONE
  ...
  test = check(a)
  ...
END PROGRAM myprog
```

# internal subprograms

```
program main
  implicit none
  integer N
  real X(20)
  ...
  write(*,*), 'Processing x...', process()
  ...
contains
  logical function process()
    ! in this function N and X can be accessed directly (scope of main)
    ! Please note that this method is not recommended:
    ! it would be better to pass X as an argument of process
    implicit none
    if (sum(x) > 5.) then
      process = .FALSE.
    else
      process = .TRUE.
    endif
  end function process
end program
```

# external subprograms

- `external` subprograms are defined in a separate program unit
- to use them in another program unit, refer with the `EXTERNAL` statement
- compiled separately and linked

**!!! DO NOT USE THEM:** modules are much easier and more robust !

They are only needed when subprograms are written with different programming language or when using external libraries (such as BLAS)

It's **highly** recommended to construct `INTERFACE` blocks for any external subprograms used

# interface statement

```
SUBROUTINE nag_rand(table)
  INTERFACE
    SUBROUTINE g05faf(a,b,n,x)
      REAL, INTENT(IN)      :: a, b
      INTEGER, INTENT(IN)  :: n
      REAL, INTENT(OUT)    :: x(n)
    END SUBROUTINE g05faf
  END INTERFACE
  !
  REAL, DIMENSION(:), INTENT(OUT) :: table
  !
  call g05faf(-1.0,-1.0, SIZE(table), table)
END SUBROUTINE nag_rand
```

# Fortran Compiler and libraries

Examples:

```
module load netCDF-Fortran/4.5.3-gompi-2021b
gfortran -ffree-line-length-none \
-o OceanGrideChange.exe 07_OceanGrideChange.f90 \
-I${EBROOTNETCDFMINFORTRAN}/include -L${EBROOTNETCDFMINFORTRAN}/lib -lnetcdff
```

```
module load netCDF-Fortran/4.5.3-impi-2021b
ifort -O3 \
-o OceanGrideChange.exe 07_OceanGrideChange.f90 \
-I${EBROOTNETCDFMINFORTRAN}/include -L${EBROOTNETCDFMINFORTRAN}/lib -lnetcdff
```

Fortran 90 source code [09\\_OceanGrideChange.f90](#) with the input file [09\\_input.nc](#)



# Conclusions

- Fortran in all its standard versions and vendor-specific dialects is a rich but confusing language
- Fortran is a modern language that continues to evolve
- Fortran is still ideally suited for numerical computations in engineering and science
  - most new language features have been added since F95
  - "High Performance Fortran" includes capabilities designed for parallel processing