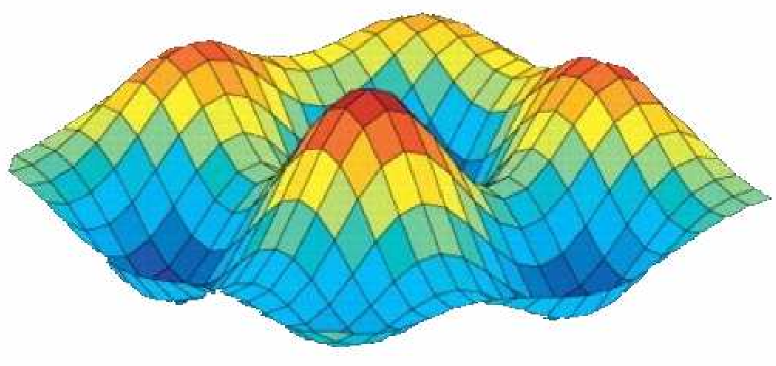
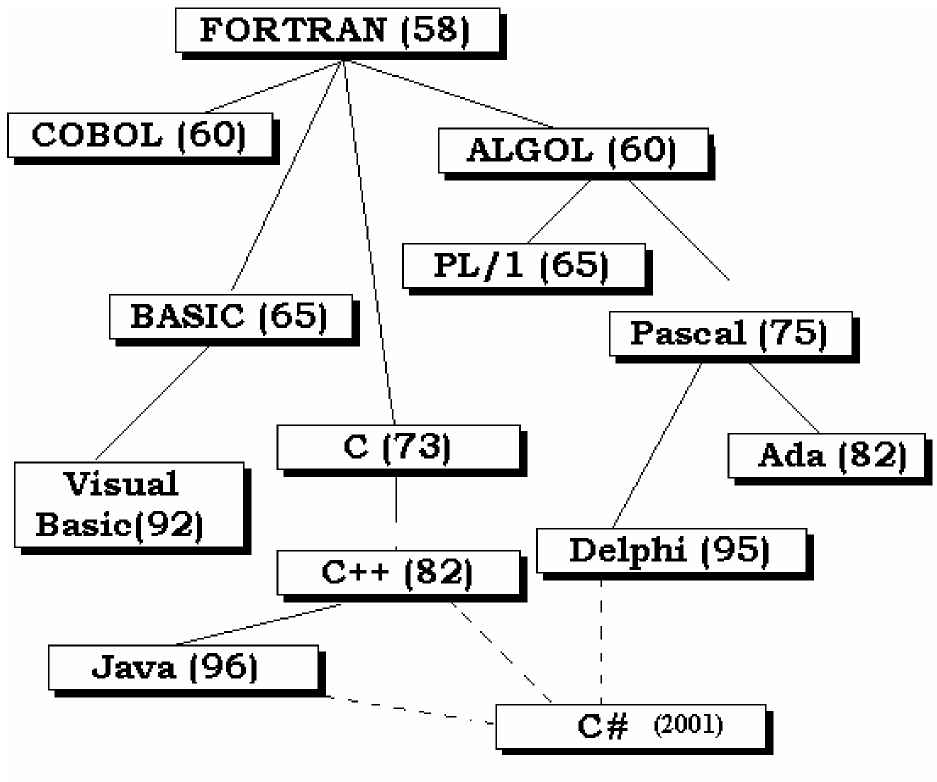
# Part III: Introduction to FORTRAN



# Introduction to FORTRAN language

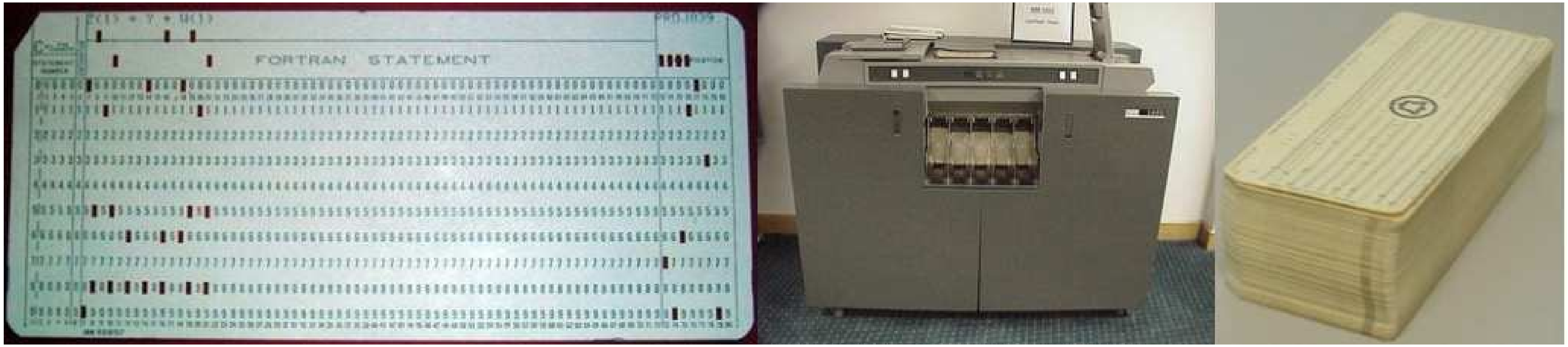
**FORTRAN** = **For**mula **Tran**slator

1st language of *high level*, developed from 1954 by John Backus (at IBM).



## History

- **Fortran 66**:  1st standardisation




-2d standard: **Fortran 77** : syntax IF-THEN-ELSE

-3d standard: **Fortran** 90 and **95:**  file format .f90, modern instructions (modules, dynamic tables, ...)

-4th standard: **Fortran 2003:** numerous revisions (pointers, inter-operability with the language C, Object-Oriented programming, ...)

- 5th standard: **Fortran 2008 :**  new instructions for parallelism

# Avantages of Fortran

Language developed for **scientific calculation** (« number crunching »)

Generates **optimized high-performance** code:
- compiled language
- automatic optimizations

**Simple and flexible syntax**
Problems with high memory-space cost: easy dynamic management

Good support of **mathematics**:
- numerous kinds of data (complex numbers, vectors, matrices, …) and math functions available as standard
- compact syntax for matrix calculation (→ automatic parallelism)

Since Fortran 2003:
- Object-Oriented programming
- inter-operability with the language C (and Python)

## Disadvantages of Fortran

The language allows many obsolete instructions (supports 4 standards!) still encountered frequently in codes.

Badly adapted to anything other than scientific computing (graphical interface, access to databases, ...)

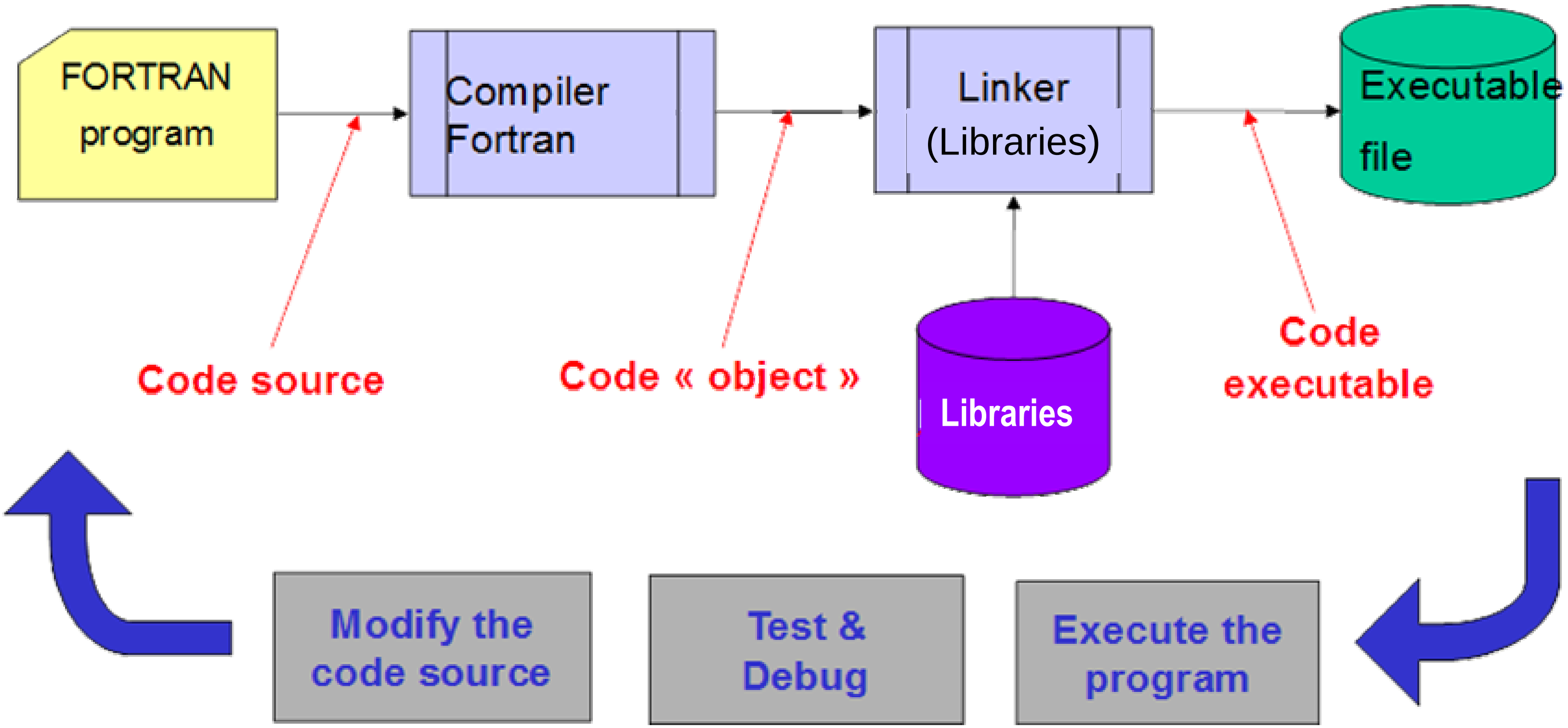
Quite small user community (compared to C)

## Advantage or disadvantage

The language evolves slowly. (→compatibility, stability)

Fortran (like C) is a compiled language.

The code-source (what we write) should be converted into machine language before being executed.



# Basic instructions of Fortran and first code

- Overview of the main instructions

- Illustration of good (and bad) programming practices

Note:
The language has become very complex, it is impossible to cover all the possibilities of Fortran in this course.

# §1 Structure of a program

**Source code**: contained in a text file with the extension **.f90**

Example: MyProgram.f90

```
program < Name of the program >
   :
   :      Declaration instructions
   :


   :
   :      Executable instructions
   :
end program < Name of the program >
```

**Name of the program**
(without extension f90)
max 31 alphanumerical characters,
the 1st one is a letter

Declarations of variables,
tables and other used objects
(modules).

**Definition of the subprograms**
(subroutines and functions)

This part is optional

# §1 Structure of a program

Example:

```
! Program to calculate a square root
program square_roots
```

A character **!** means that the remainder of the line is a comment

```
implicit none
integer :: i,nroots=3
real :: value,root
```

Declarations

Instructions

```
print *, “Let’s calculate ”,nroots, “ square roots”
do i=1,nroots
        print *, ‘Give a number :’
        read *, value
        if (value > 0) then
                root=sqrt(value)
                print *, “The number ”,value, “ has the root :”,root
        else
                print *, “The number ”,value, “ has no roots”
        end if
end do

end program square_roots
```

max 132 characters per line

Comment your codes, *but not evident things*!
Indent your codes!

# §1 Structure of a program

## Text formatting (free format)

- Capitalization (upper / lower case) is not important
- Spaces between keywords do not count.
- Maximum 132 characters per line, but

a line ending in & (ampersand) continues on the next line
and the & character can be reproduced on the next line

```
integer::n,resultat
```

⇔

```
integer :: n, &
           resultat
```

⇔

```
integer :: n, &
           & resultat
```

- A semicolon (;) separates two statements on the same line

```
print *, 'Give a number';read *, n
```

# §1 Structure of a program

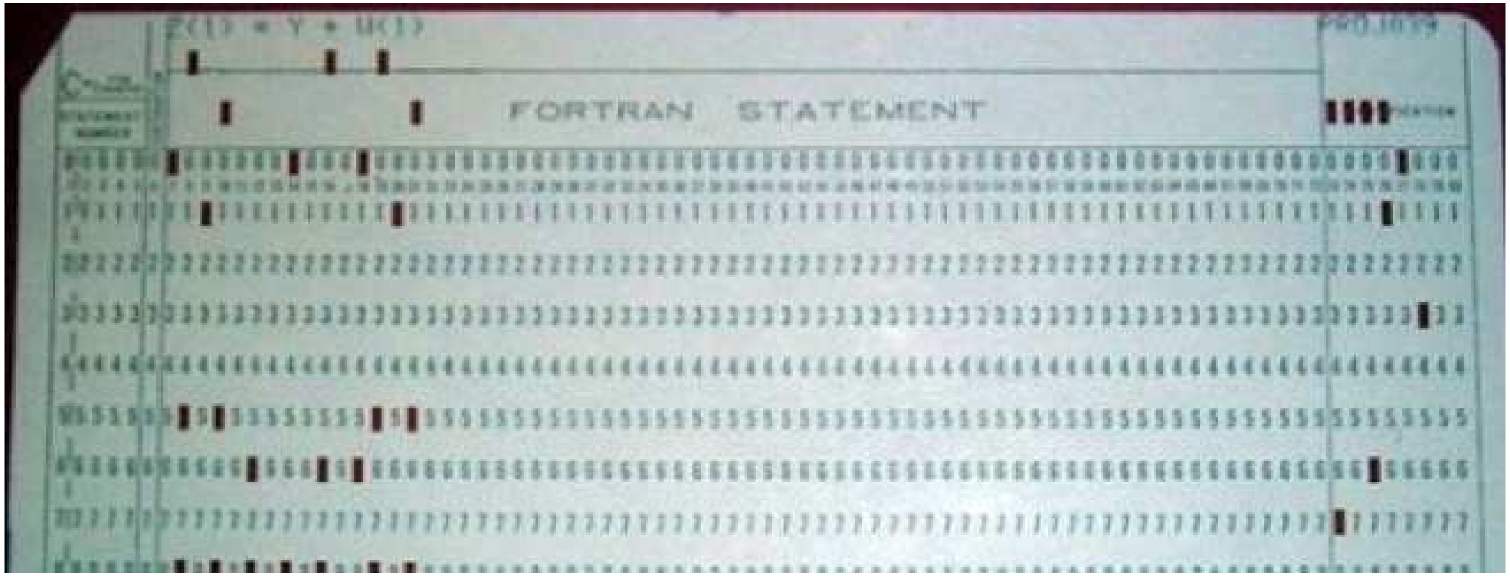
## Text formatting

**Free format (Fortran 90 and +)**: file extension .f90
**Fixed format (Fortran 77)**: file extension .f

```
      PROGRAM MAIN
C COMMENTS ARE ALLOWED IF A “C” IS PLACED IN COLUMN #1
      DIMENSION X(10)
      READ(5,*) (X(I),I=1,10)
      WRITE(6,1000) X
 1000 FORMAT(1X,’THIS IS A VERY LONG LINE OF TEXT TO SHOW HOW TO CONTINUE ’
     *  ‘THE STATEMENT TO A SECOND LINE’,/,10F12.4)
```

1-5 Label | 6 | 7-72 Statements | 73-80 Optional Line #s

Any character: continuation line



# §1 Structure of a program

## **Names of variables** (and other identifiers: constants, procedures, …)

- They are composed of a series of alphanumerical characters
  - the 1st one is a letter
  - limited to 31 characters
- Concerning the letters
  - the underline character (_) is considered as a letter

```
total_value _total value_2_8
```

FORTRAN does not distinguish between uppercase and lowercase, except in strings of characters

```
Root    ⇔      root
'Root' ≠ 'root'
```

## §2 Types of variables

**There are 5 basic types:**

**INTEGER** – **integer number** (positif, zero or negatif)

- 16 bits (2 bytes): $-32.768 \le i \le 32.767$
- 32 bits (4 bytes): $-2.147.483.648 \le i \le 2.147.483.647$

**REAL(kind=4)** – **real number** (simple precision)

coded on 4 bytes (=32 bits): $1.2\ 10^{-38} \le |x| \le 3.4\ 10^{38}$
Precision : 6 significant digits

**REAL(kind=8)** – **real number** (double precision)

coded on 8 bytes (=64 bits): $2.2\ 10^{-308} \le |x| \le 1.8\ 10^{308}$
Precision : 15 significant digits

**COMPLEX** – complexe number

**LOGICAL** – boolean: takes the values **.TRUE.** or **.FALSE.**

**CHARACTER(len=n)** – string of characters (n characters)

Exemples:

```
integer         :: i = 5
real(kind=4)    :: epsilon = 1e-5
real            :: x          ! initial value undetermined
character(len=26)  :: message = “Hello”
```

**REAL(kind=10)** – **real number** (extended precision)
**REAL(kind=16)** – **real number** (quadruple precision)

# §3 Types of data

## Attributes

The declarations can use the following attributes:

| | |
|---|---|
| PARAMETER | : Symbolic constant |
| DIMENSION | : Size of a table |
| SAVE | : Static object |
| EXTERNAL | : External procedure |
| INTRINSIC | : Intrinsic procedure |

Examples:

```
real, parameter :: pi = 3.1415965      ! def of a constant
real :: angle = 2*pi
```

# §3 Types of data

***Ancient notation (Fortran 77)***

```
real*4   value                          !idem to real(kind=4)
real*8   value_precise                  !idem to real(kind=8)
double precision value_precise          !idem to real(kind=8)
```

**Specific notation for the numerical values "double precision"**, i.e. coded on 64 bits (kind=8)

| Example: | simple precision | **double precision** | | | |
|---|---|---|---|---|---|
| | `2.5E-8` | `2.5D-10` | | | |
| | `1.2` | `1.2_8` | variants: | `dble(1.2)` | [conversion] |
| | | | | `1.2d0` | [notation F77] |

Code to change the precision of all variables

```
integer, parameter :: r=4   !single line to modify
real(kind=r) :: x
x = 2.5_r                   !num constant coded on r=4 or 8 bytes
```

# §3 Types of data

**Complex numbers**

Notation: (real part, imaginary part)

Example 1:
```
complex :: i = (0,1)
```

Example 2:
```
integer, parameter :: r = 8
complex(kind=r) :: z
complex(kind=r), parameter :: i = (0_r, 1_r)

z = (3.5_r,8e-6_r)*i          !multiplication by i
```

In the expressions, one can use data of different types
```
real             :: x
double precision :: y
complex(kind=8)  :: z
y = 5*x
z = z+y
```

# §3 Types of data

**Implicit declaration versus explicit one**

The variables which names start by I, K, L, M, N are considered by default as integer,
the variables starting by other letters are considered as real.

This is more harmful than helpful !

Solution:

Always use the implicit none instruction.
The compiler then requires all variables to be declared.

# Compilation options

Use:

gfortran -Wall -Wuninitialized *code.f90*

```
program main
implicit none
integer :: j
print *, “ j= “,j
end program main
```

The value of j is not given!

# §4 Operators

- **arithmetic**

| Operator | Function |
|---|---|
| ** | power |
| / | division |
| * | multiplication |
| - | substraction |
| + | addition |

Priority of the operation

Examples: **`3.42 + (x-y)/SIN(angle) – x*y/r**3`**

**`2**3**2  ⇔  2**(3**2) = 512`** (**: priority from right to left)

**Optimisation of the arithmetic operations**

```
x = a + b*c
y = b*c
```

! The multiplication is not re-calculated for the 2d time

Do not seek to optimize yourself this kind of operation.
The compiler will do the job better than you!

# §4 Operators

- **arithmetic (continued)**

**Trap** (language Fortran and language C)

Which value takes x at the end of the following code?

```
t = 1.0
x = (1/2)*9.81*t**2                !This formula is incorrect !
```

The result of the division of two integers is an integer (the entire part of the quotient).

# §4 Operators

- **of comparison**

| Old notation | New notation | Meaning |
|---|---|---|
| .LT. | < | inferior to |
| .LE. | <= | inferior or equal to |
| .GT. | > | superior to |
| .GE. | >= | superior or equal to |
| .EQ. | == | equal to |
| .NE. | /= | different from |

Examples: **IF (x >= y)** ⇔ **IF (x .gt. y)**
**IF (i/=j)**

Compare the real numbers via **IF (x == y)** is a BAD idea!

Use instead IF (abs(x-y) < 1e-7), e.g.

# §4 Operators

- **logical**

| Operator | Example | Meaning |
|---|---|---|
| .AND. | A .AND. B | logical AND |
| .OR. | A .OR. B | logical OR |
| .NEQV. | A .NEQV. B | logical inequivalence |
| .XOR. | A .XOR. B | exclusive OR (same as .NEQV.) |
| .EQV. | A .EQV. B | logical equivalence |
| .NOT. | .NOT. A | logical negation |

Examples:

```
IF (x>y+3 .and. i==j)      !Comparisons are made before the logical operators
IF (.not. i==j)            !⇔  IF (i/=j)
```

Action of the operator .NOT. On a boolean expression `l`:

| l | .NOT.l |
|---|---|
| .true. | .false. |
| .false. | .true. |

# §4 Operators

## Truth tables

| $l_1$ | $l_2$ | $l_1$ .AND. $l_2$ | $l_1$ .OR. $l_2$ |
|---|---|---|---|
| .true. | .true. | .true. | .true. |
| .true. | .false. | .false. | .true. |
| .false. | .true. | .false. | .true. |
| .false. | .false. | .false. | .false. |

| $l_1$ | $l_2$ | $l_1$ .EQV. $l_2$ | $l_1$ .NEQV. $l_2$ |
|---|---|---|---|
| .true. | .true. | .true. | .false. |
| .true. | .false. | .false. | .true. |
| .false. | .true. | .false. | .true. |
| .false. | .false. | .true. | .false. |

# §5 Tests

- **Test “IF - ELSE”**

```
[nom_bloc: ] IF( exp1 )THEN
                 bloc1
             [ELSE IF( exp2 )THEN [nom_bloc]
                 bloc2
                 ...
             [ELSE [nom_bloc]
                 blocn]]
              END IF [nom_bloc]
```

☞ nom_bloc a label

☞ $\mathtt{exp}_i$ an expression of type **LOGICAL**,

☞ $\mathtt{bloc}_i$ a series of FOTRAN instructions

In the simplest case:

`IF (exp) instruction`

# §5 Tests

## •Multiple tests

Multiple branching based on the value of an expression

```
[ nom_bloc: ] SELECT CASE(expression)
               [ CASE(liste) [ nom_bloc ]
                    bloc1]
                      ...
               [ CASE DEFAULT[ nom_bloc ]
                    blocn]
              END SELECT[ nom_bloc ]
```

- `nom_bloc` a label
- `expression` an expression of type `INTEGER`, `LOGICAL` ou `CHARACTER`,
- `liste` a list of constants of the same type as that of `expression`,
- $\mathtt{bloc}_i$ a series of FOTRAN instructions

# §5 Tests

```
PROGRAM structure_case
        integer :: mois, nb_jours
        logical :: annee_bissext
                 ...
        SELECT CASE(mois)
          CASE(4, 6, 9, 11)
            nb_jours = 30
          CASE(1, 3, 5, 7:8, 10, 12)
            nb_jours = 31
          CASE(2)
        !-----------------------------------
             fevrier: select case(annee_bissext)
                case(.true.)
                  nb_jours = 29
                case(.false.)
                  nb_jours = 28
             end select fevrier
        !-----------------------------------
          CASE DEFAULT
            print *, ' Numéro de mois invalide'
        END SELECT
END PROGRAM structure_case
```

# §6 Cycles

There are many kinds of iterative cycles having each the form :

```
[ block name ] DO [ loop control      ]
                   block
                 END DO [ block name ]
```

☞ `block name` is a label

☞ `loop control` defines the conditions of executing and stopping the cycle

☞ `block` is a series of FOTRAN instructions

# §6 Cycles

**1st form: cycle DO with a counter**
**DO variable = beginning, end, step**

```
PROGRAM iteration_do
       INTEGER i, somme, n
              ...
! affectation de n
       somme=0
       DO i=1,n,2
         somme=somme+i
       END DO
              ...
END PROGRAM iteration_do
```

Cycle on values of i from 1 to n
with a step of 2

## §6 Cycles

**2d form:** **DO WHILE (expression)**

Executing of the code while the expression is valid.

To avoid infinite cycles, the expression should be able to become false in the block.

Example:

```
! Sum of the numbers from 1 to 100
sum = 0
i = 1
do while (i<=100)
   sum = sum + i
end do
```

# §6 Cycles

**3d form:** **DO** (with exit via the command `exit`)

Execution in cycle till getting the command EXIT

Example:

```
x = 0.0
do
   x = x + 1.0/3
   if (x > 10) EXIT
end do
```

For readability, prefer loops **do** with a counter or **do while**

<u>Note</u>: the command STOP interrupts all the program.

Example:

`STOP 12` (<— interrupts the program giving “error 12”).

# §6 Cycles

**Instruction CYCLE**

CYCLE abandons the current iteration processing and moves on to the next

Example:

```
INTEGER :: year
DO
! Ask the user to give a year:
   read *, year
   IF (year < 0) EXIT
! Refuse the leap years:
   IF( ((year/4*4     == year) .AND. &
        (year/100*100 /= year)) .OR. &
        (year/400*400 == year) ) CYCLE
   print *, “This is a non-leap year ”
   !…
END DO
```

# §7 Math functions

## Basic

| | |
|---|---|
| `ABS(x)` | absolute value of x (x integer or real) |
| `SQRT(x)` | square root (√x) |
| `SIN(x), COS(x), TAN(x)` | sinus, … (x in **radian**) |
| `SINH(x), COSH(x), TANH(x)` | hyperbolic sinus, … |
| `ASIN(x), ACOS(x), ATAN(x)` | arcsinus, … |
| `EXP(x)` | exponential |
| `LOG(x), LOG10(x)` | natural logarithm, in base 10 |
| `MOD(a,p)` | modulo: rest of the entire division of a by p |
| `MAX(x1,…,xn), MIN(…)` | maximum/minimum of x1, …, xn |

## Complex numbers

| | |
|---|---|
| `REAL(z), AIMAG(z)` | real/imaginary part of z |
| `ATAN2(z)` | argument of the complex number (in [-π,π]) |

## Special functions (FORTRAN 2008)

| | |
|---|---|
| `ERF(x), ERFC(x)` | error function |
| `BESSEL_J0(x), BESSEL_Y0(x), …` | Bessel functions |
| `GAMMA(x)` | gamma function |

## Conversion

| | |
|---|---|
| `INT(x)` | integer part |
| `NINT(x)` | nearest integer by rounding |

# §8 Strings of characters

Examples:
```
‘The result is’
“It’s erroneous.”
“Hello!”
```

Examples of declarations: 8 characters (=8 bytes)

```
character(len=8) :: inputFile, outputFile
character(len=*), parameter :: format=“(f4.2,’a’)”
```

**Comparison** of two strings of characters: == and /=

**Concatenation** (putting “end-to-end”, “in parallel”)**:** //

```
character(len=*) :: s1 = “Hel”
character(len=*) :: s2 = “lo”
character(len=50) :: entireword

entireword = s1 // s2
```

# §9 Input/output and formats

**Free-format output:**

* means that the format is determined automatically

```
print *, "The calculation is finished"
print *, 'counter=', i, 'sin(x)=', sin(x)
```

format

**Formatted output** `print "('Energy = ',F5.1,'eV')", energy`

gives the output: `Energy = -45.0 eV`

| Format | Example | Comment |
|---|---|---|
| `I4` | `-555` | integer displayed on 4 characters |
| `sp,I4` | `+555` | (sp: always display the sign +) |
| `F8.3` | `   1.234` | real (decimal notation), 8 characters with 3 decimals |
| `E8.3` | `.123E+01` | (exponential notation) |
| `G8.3` | `1.23` | (automatic notation dec/exp) *example x=1.234* |
| | `.288E+10` | *example x=2.88E9* |
| `A8` | `texte` | string of characters |
| `L1` | `T` | boolean (logical): T or F *(example:* L7 *displays .false.)* |

Variant: `write(*,'(i4)',ADVANCE=NO) entire_nb`

# §9 Input/output and formats

**Formatted output (continued)**

`7X` - leave 7 spaces

`/` - go to the line

`2E15.8` - repeat 2 times the format E15.8

`3(2X,F5.1)`- repeat 3 times “2 spaces and format F5.1”

# §9 Input/output and formats

**Interactive input (from the keyboard)**

```
read *, variable
read “(I4,I4,I4)”, i, j, k
read “(3I4)”      , i, j, k
```

reading 3 integers
(the user enters e.g. “1 2 3”)

repetition: 3 times

# §9 Input/output and formats

**Files**

- opening:

choose a channel number
Do not choose neither 5 (keyboard) nor 6 (screen)!

```
open(unit=10,file=“data”)
```

*Note: If the file does not exist, it is created.*

- opening with error handling :

```
integer :: codeError
open(unit=10,file=“data”,iostat= codeError)
if (codeError /= 0) then
   print *, “Error no:, codeError
end if
```

- reading/writing:

```
write(10,fmt=‘(I5)’) N_particles
read (10,*,iostat=io)    Length
```

- closing: `close(10)`

`io` is a variable of type integer.
`io == 0` if the file is read correctly
`io ≠ 0` if there is an error (for example, end of file)

# §9 Input/output and formats

## Other operations with the files

- go to the beginning of the file: `rewind(unit=10)`
- go back one line: `backspace(unit=10)`

# Tables

# §10 Tables

Variable containing a vector, a matrix or a multi-dimensional matrix

- **Definition of a variable table** containing 100 elements (numbered from **1** to 100)

```
real(kind=r), dimension(100) :: vector
```

To get the 16th element of the table: `vector(16)`

```
dimension(0:9)          ! table with 10 elements numbered starting from 0
dimension(100,0:2)      ! 2-dimensional table containing 100 x 3 elements
```

- **Definitions**: **rank** of a table: number of dimensions
  **extents** of a table : number of elements for each dimension
  **size** of a table : total number of elements

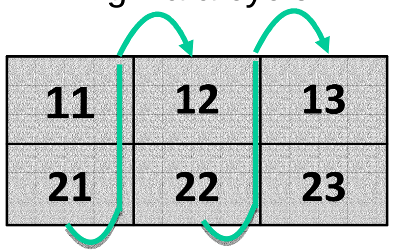
Filling a table with initial values

```
real, dimension(5) :: tableau
tableau = [-2.0, 22.0, 33.0, 44.0, 1e8]   ! syntax F2003
tableau = [(2*i, i=1,5)]                   ! filling via a cycle
```

Filling a 2D table:

```
integer, dimension(2,3) :: mat
mat = reshape([11,21,12,22,13,23],[2,3])
```

mat:



# §10 Tables

- **Display all the elements of a table**

```
print *, table
```

The display is done by varying the index 1 first, then the index 2, etc.

Example:

```
dimension(2,2)
```

| | |
|---|---|
| **11** | **12** |
| **21** | **22** |

—> display: 11 21 12 22

- **Assignment of values**

```
real, dimension(10) :: a
a = 0.0                       ! All the elements are put to 0
a(3) = 555                    ! The 3d element equals 555
a(5:10) = 1.0                 ! Elements are put to 1 in a part of the table
```

General syntax: beginning:end:step

If omitted: 1

If omitted: end of the table

If omitted: beginning of the table

# §10 Tables

**Examples of calculations with tables:** (automatic parallelism/vectorization!)

- Algebraic operation on multiple components:

```
a(1:10) = b(5,11:20) + 4.0*c(10:1:-1)
```

The matrix multiplication is not written as `a*b` but `matmul(a,b)`

- Permutation of columns

```
integer, dimension(4) :: perm = [3, 1, 4, 2]
real, dimension(4)    :: departure, arrival
arrival = departure(perm)
```

permutes the columns of `departure` to obtain `arrival`

- Conditionnal operations: WHERE

```
where (logical_expression)
   array1 = expression
elsewhere
   array2 = another_expression
endwhere
```

Example:

```
where (a >= 10.0)
      a = 20 - a
elsewhere
      b = -a
endwhere
```

# §10 Tables

**Command** **forall** : loops running through the indices

Example 1: `forall (i=1:3)  mat33(i,i) = 1`

Example 2:

```
forall(i=1:N, j=1:N, a(i,j)/=0.0) &
    b(i,j) = 1.0/a(i,j)
```

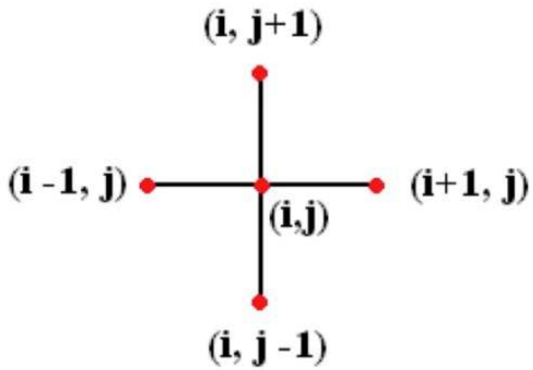
Exemple 3:

The command `forall` allows to program an operation like

$$A_{i,j} = (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})/4$$

(A: matrix n x n)

to be done on some elements of a table




```
real, dimension(n,n) :: A

forall (i=2:n-1, j=2:n-1)
    A(i,j) = 0.25*(A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
end forall
```

# §10 Tables

## Intrinsic functions

| | |
|---|---|
| `DOT_PRODUCT(a,b)` | scalar product |
| `MATMUL(a,b)` | matrix product |
| `TRANSPOSE(a)` | transposed matrix |
| `NORM2(a)` | norm of a vector (FOTRAN2008) |
| `SUM(a)` | sum of the elements |
| `SUM(a,2)` | sum of the elements of the 2d column |
| `PRODUCT(a)` | product of the elements |
| | |
| `MAXVAL(a), MINVAL(a)` | maximal/minimal value |
| `MAXLOC(a), MINLOC(a)` | position of the maximal/minimal element |
| | |
| `ALL(logical_expression)` | true if the expression is true for all elements |
| `ANY(logical_expression)` | true if the expression is true at least for one element |
| `COUNT(logical_expression)` | number of elements for which the expression is true |

logical expression involving a table,
for example a > 0

## §10 Tables

**Dynamic allocation of tables**

Reserve / release the memory area of a table
allows to create tables of variable size (size indicated by a variable)

```
real, dimension(:,:,:), allocatable :: tableTemp
! table of rank 3 (without memory space currently associated)

n = 10; m = 5
ALLOCATE(tableTemp(100,n,m))    ! the table is now usable
   ! operations on the table:
DEALLOCATE(tableTemp)           ! the memory space is free now
```

Make sure to respect the limits of the tables!

```
real, dimension(20) :: a
real :: b = 0
do i=1,21
a(i) = i       ! table overflow when i = 21
enddo          ! activate if necessary the option -fbounds-check of the compiler
! Now b = 21 !!!          !
```

# Subroutines and functions

# §11 Subroutines

The procedures such as subroutines and functions are used **to structure** long programs and avoid duplicate instructions

```
program main
   …
   call do_this(data1)
   call do_this(data2)
   …
   x = fonc1(…)

contains
   subroutine do_this(…)
      …
   end subroutine do_this

   function fonc1(…)
      …
   end function fonc1

end program
```

Definitions of the subroutines and functions ***internal*** to the main program

The **internal** procedures have access to the variables of the main program (unless they are obscured by the declaration of a local variable )

The use of global variables (shared between the main program and the subroutines) is **not recommended** .
(—> place the procedures in a module)

## §11 Subroutines

```
subroutine convert(x,y,rho,theta,arg5)
   real, intent(in)    :: x,y
   real, intent(out)   :: rho,theta
   real, intent(inout) :: arg5
   integer :: i                        ← local variable
   integer, SAVE :: N                  ← local variable which keeps its value during the successive calls of the subroutine
   …
   executable instructions
end subroutine convert
```

**Attributes intent** (optional attributes but recommended to improve readability)

intent(in): **input variable** only
The subroutine can not change the value of this variable

intent(out): **output variable** only
The input value can not be used by the subroutine

intent(inout): **input and output variable**
The subroutine uses the value contained in the input variable, and modifies it during its execution

Note: A subroutine which has no side effect (no modification of global variables) is said pure.
A pure function can only modify the arguments out and inout and nothing else.

# §11 Subroutines

```
subroutine convert(x,y,rho,theta,arg5)
   …
   …
   if (logical_condition) return
   …
   …
end subroutine convert
```

*model arguments*
*(dummy arguments)*

Examples of calling a subroutine:

*current arguments*

```
call convert(10.0, 22.0, r,  alpha,  temp)
call convert(50.0, -10., r2, alpha2, temp)
```

or with arguments key-words:

```
call convert(10.0,y=22.0,arg5=temp,rho=r,theta=alpha)
```

*the order is not important here*

The command return allows completion of a subroutine run before reaching the end

# §12 Functions

A function is used to calculate a value from one or more arguments.

Definition of a function:

```
function polynomial(x,y)
   real :: polynomial
   real, intent(in) :: x,y
   …
   polynomial = x**2 + x*y + 3*y**2
end function polynomial
```

Example of use:

```
valeur = polynome(3., -5.)
```

Calling a function is done without the keyword call (≠ calling a subroutine)

Notes:

- The function and its return value have the same name

- Even if a function is allowed in principle to modify the value of arguments, this is strongly discouraged (use a procedure rather than a function)

# §13 Transmit a table as an argument

Easiest way: declare the array argument with an implicit profile

(*assumed-shape array*)

```
program main
   implicit none
   real, dimension(3,3)  :: mat33
   real, dimension(100,5) :: tab
   call sub(mat33)             !  displays    3    3
   call sub(tab)               !  displays  100    5
contains
   subroutine sub(table)
       real, dimension(:,:) :: table  ! table of implicit profile
             print *, shape(table)        ! displays the extents
       end subroutine
end program
```

The compiler is responsible here to transmit to the procedure the information about the table's extents (for each dimension)

# §14 Transmit a function as an argument

In general, it is necessary to define the "interface" of the function
(=its arguments and type of return value )

```
program main
   implicit none
   call integr(fonct,0.0,1.0,integral)
   …
contains
   function fonct(x)
     real, intent(in) :: x
     real :: fonct
     fonct = (ln(x)/x)**2
   end function
end program
```

```
subroutine integr(f,a,b,res)
   real, intent(in)  :: a,b
   real, intent(out) :: res
   interface
      function f(x)
         real, intent(in) :: x
         real :: f
       end function
   end interface
   val_1 = f(a)
   val_2 = f(b); …
end subroutine
```

## §15 Procedures with optional arguments

A procedure (subroutine or function) may have optional arguments

```
function erf(x,accuracy)
   real, intent(in) :: x
   real, intent(in), optional :: accuracy
   …
   if (present(accuracy)) then
      …
   else
      …   ! do not use accuracy here
   end if
end function
```

This function can be called via

```
value = erf(x=3.)
value = erf(x=3.,accuracy=1E-6)
```

# Modules

# §16 Modules

In addition to internal procedures, we can define **external procedures**.
These procedures are placed in one or more separate files.

**Interest:**

- easy reuse of procedures by other programs
- code structuring (modularity)
- separate compilation of the different files

program.f90

```
program myProg
use hefty
!declaration of variables
!end of the declarations

  …
end program
```

moduleHefty.f90

```
module hefty
  … ! definitions of variables/constants
contains
  …       ! definitions of
          ! - subroutines
          ! - functions
end module
```

**Compilation**: `gfortran moduleHefty.f90 program.f90`

# §16 Modules

**General structure of a module**

```
module Integration
implicit none
public
   real :: variable !  global variable (accessible in the main program)
private
   real :: epsilon                    !  local  variable for the module

contains
   subroutine integrate(a, b, nbPoints, result)
      …
   end subroutine integrate

   …
end module
```

The quantities defined in a module are automatically accessible from outside (→ global variables) unless they have the attribute "private".

# §16 Modules

**Example 1**: file of constants

In the main program:

```
use Constants
```

or

```
use Constants, only : pi, c
```

```
module Constants

!Math
   real(kind=8),parameter :: pi = 3.14159265359_8

!Physics
   real(kind=8),parameter :: c = 3D8         !speed of light
   real(kind=8),parameter :: e = -1.6D-19    !elementary charge
      …

end module
```

# §16 Modules

**Example 2**: global variable declaration file

```
module GlobalVariables

      integer              :: NbParticles
      integer, parameter :: Nmax = 100
      real                 :: ElecField(Nmax, Nmax, Nmax,3)

end module
```

Use then `use VariablesGlobales` at the beginning of each file

Restrict the use of global variables as much as possible!
=> Prefer to explicitly pass all arguments to procedures.

# §16 Modules

**Example 3**:

```
module Matrices
implicit none
contains

    subroutine printmat(table2D)
       real, dimension(:,:) :: table2D
        integer :: i
        do
i=lbound(table2D,dim=1),ubound(table2D,dim=1)
            print *, table2D(i,:)
        enddo
    end subroutine

end module
```

# §16 Modules

**Example 4**:

```
module ModuleHisto          ! Note: the name histogram is already used by the function
    implicit none
    integer, parameter:: ww = 4
contains
    function histogram(data,histoMin,histoMax,nbBins)
        real(kind=ww), dimension(:), intent(in) :: data
        real(kind=ww), intent(in) :: histoMin,histoMax
        integer, intent(in) :: nbBins
        real(kind=ww),dimension(:),allocatable :: histogram

        integer :: nbValues, bin, i

        nbValues = size(data)
        allocate(histogram(nbBins))
        histogram = 0

        do i=1,nbValues
            bin = int(1 + (data(i)-histoMin)/(histoMax - histoMin) * nbBins)
            if (bin > nbBins) &
                bin = nbBins
            histogram(bin) = histogram(bin) + 1
        enddo

        !normalisation
        do i=1,nbBins
            histogram(i) = histogram(i)/nbValeurs
        enddo

    end function
end module
```