This is still true!

- Write your code properly!
- Indent your code! (use emacs!)
- Provide comments!
- Compile with a makefile!
A first example

```fortran
program sqrt_two
  implicit none
  !--- variables
  integer::n
  real::u
  !--- initialisation
  u=1.0
  !--- loop
  do n=1,10
    u=u/2.0+1.0/u
  end do
  !--- printing
  print*,’approximation of sqrt(2): ’,u
end program sqrt_two
```

```
$ gfortran -o prog sqrt2.f90 ; ./prog
approximation of sqrt(2): 1.4142135
```
Fortran: a very old language

- first compiler in 1957
- 1978: Fortran 77, still used in well known libraries.
  - Strong limitations on the format (compatibility with 80 column cards),
  - Variables encoded on 6 characters (first letter defining the type)
- Fortran 90: A modern language
  - free format,
  - modular programmation,
  - important tool for scientific computing: main part of industrial codes are written in Fortran.
- Newer ones since (95, 2000 - including object programmation,...)
Structure of a F90 program

```
program program_name
  variable declarations
  instructions
end program program_name
```

<table>
<thead>
<tr>
<th>declaration</th>
<th>affectation example</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer :: n,i</td>
<td>n=10</td>
</tr>
<tr>
<td>real :: x,y</td>
<td>x=1.0e-5</td>
</tr>
<tr>
<td></td>
<td>y=1.0</td>
</tr>
<tr>
<td>complex :: z</td>
<td>z=cmplx(1.0,2.5e3)</td>
</tr>
<tr>
<td>character :: c</td>
<td>c='d'</td>
</tr>
<tr>
<td>logical :: b</td>
<td>b=.true.</td>
</tr>
</tbody>
</table>

Always use **implicit none** at the begginging of the declaration bloc.
Tables

- Any type can be used with table thanks to dimension:
  - integer, dimension(-2:2) :: dim
    (one dimensional, indices between -2 and 2)
  - real, dimension(3,0:4,20) :: tab
    (3D ; 1st index between 1 and 3, 2nd between 0 and 4...)
  - character, dimension(3,4:6) :: tch
    (NB: maximum 7 dimensions)

- Access to an element:
  - dim(-1)=10
  - tab(2,3,20)=2.3e-5
  - tch(1,5)=’a’
Identifiers

- Sequence between 1 and 31 characters in:
  - small and capital letters (*no accents* !),
  - digits,
  - _

- First character must be a letter,

- Capital and small letter are the same !

Good identifiers:
- constant__gaz
- pi2
- Rk54
- PI2
- ToTOo0tT0o

Bad identifiers:
- casé
- with space
- there_is_more_than_31_characters
- _underscore_first
- 1_digit_first
- withAsharp#
Comments and maximum lines

- Anything after a `!` is a comment
- Maximum line is 132 characters
- Longer line? cut the line in several
  - put a `&` at the end of the line,
  - starts the next one with a `&`. 
Logical expressions and conditional instruction

<table>
<thead>
<tr>
<th>test</th>
<th>meaning</th>
<th>test</th>
<th>meaning</th>
<th>test</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equality test</td>
<td>&lt;</td>
<td>strictly less than</td>
<td>.and.</td>
<td>and</td>
</tr>
<tr>
<td>/=</td>
<td>different</td>
<td>&gt;=</td>
<td>more or equal</td>
<td>.or.</td>
<td>or</td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly more than</td>
<td>&lt;=</td>
<td>less or equal</td>
<td>.not.</td>
<td>not</td>
</tr>
</tbody>
</table>

```fortran
[name:] if (logical expression) then
  instructions
[else if (logical expression) then
  instructions ]
[else
  instructions ]
end if [name]
```

(optional name for the structure to avoid confusion in case of nested if)
character(len=30)::country

... 

language: select case(country)

case('france','quebec','switzerland','belgium')
  print* 'Bonjour'
case('united-kingdom','usa')
  print* 'hello'
case default
  print* 'Unavailable language'
end select [language]
Iterative instruction

\[
\begin{align*}
\text{[name:] } \textbf{do} \ & \text{ variable=beginning, end[, step]} \\
& \text{ instructions} \\
\textbf{end do} \ [\text{name}] 
\end{align*}
\]

Infinite form, we quit via \textbf{exit}:

\[
\begin{align*}
\text{[name:] } \textbf{do} \\
& \text{ instructions} \\
& \textbf{if} \ (\text{logical expression}) \ \textbf{then} \\
& \quad \textbf{exit} \ [\text{name}] \\
& \textbf{end if} \\
& \text{ instructions} \\
\textbf{end do} \ [\text{name}] 
\end{align*}
\]

Conditional loop:

\[
\begin{align*}
\text{[name:] } \textbf{do while} \ (\text{logical expression}) \\
& \text{ instructions} \\
\textbf{end do} \ [\text{name}] 
\end{align*}
\]
Standards input and output

- Read on the standard input:
  ```fortran
  read*,variable
  ```

- Printing a message and/or a value of a variable:
  ```fortran
  print*,’hello’
  print*,x
  print*,’there remains’,s,’seconds’
  ```
More details

- Constant variable: `parameter` (no function call)
  
  ```fortran
  real, parameter:: pi=3.14159265, half=1.0/2.0
  real, parameter:: pi=acos(1.0) !– not good
  ```

- (intern) subprograms:
  - bloc of instructions used several times,
  - before the last line `end program` `program_name`,
  - after the keyword `contains`,
  - there are two kind:
    - subroutine
    - function
  - Any variable of the main program can be used (exception: parameters of the subprogram)
  - Local variables (to the subprogram) cannot be used outside.
Principle

```fortran
program program_name
  variable declarations
  instructions
contains
  subroutine subroutine_name
    ...
  end subroutine subroutine_name
  function function_name
    ...
  end function function_name
end program program_name
```

Using subprograms:
- subroutine: use `call`
  ```fortran
call subroutine_name(parameters)
  ```
- function: as a mathematical function
  ```fortran
  varname = function_name(parameters)
  ```
Subprogram parameters

- **Subroutine:**
  - parameters are optionals,
  - declarations inside the subroutine,
  - one usually adds
    - `intent(in)` for input data,
    - `intent(out)` for output data,
    - `intent(inout)` for mixed parameters,
    (this enables the compiler to detect more errors)

- **Function:** same as subroutine, except . . .
  - input parameters are mandatory,
  - returns a result, stored in a variable that has the same name as the function.
Result of a function

- Declaration of the returned type before the name of the function:

```
real function maxi(t)
  ...
end function maxi
```

- or in a declaration:

```
function maxi(t)
  ...
  real :: maxi
  ...
end function maxi
```

- different name than the function one, use `result`:

```
function maxi(t) result(y)
  ...
  real :: y
  ...
end function maxi
```
Examples

program exsubrout
  implicit none
  real:: avg, maxi
  real,dimension(100):: tab
  call random_number(tab)
  call sp (tab, avg, maxi)
  print*, avg, maxi
contains
  subroutine sp(t,moy,max)
    implicit none
    real, dimension(100), intent(in)::t
    real,intent(out):: moy, max
    integer :: i
    max = t(1) ; moy = t(1)
    do i =2 ,100
      if (t(i) > max) then
        max = t(i)
      end if
      moy = moy + t(i)
    end do
    moy = moy/100
  end subroutine sp
end program exsubrout

program exfct
  implicit none
  real :: maximum
  real,dimension(100) :: tab
  call random_number (tab)
  maximum = maxi (tab)
  print*, maximum
contains
  function maxi (t)
    implicit none
    real, dimension(100), intent(in)::t
    integer :: i
    real :: maxi
    ! - function declaration
    maxi = t (1)
    do i =2 ,100
      if ( t(i) > maxi ) then
        maxi = t (i)
      end if
    end do
  end function maxi
end program exfct

adrien.poteaux@lifl.fr
Using partial tables

```fortran
program exsubrout
    ...
    real,dimension(100):: tab
    ...
contains
    subroutine sp(t,n,moy,max)
        ...
        integer,intent(in) :: n
        real, dimension(n), intent(in)::t
        real,intent(out):: moy, max
        integer :: i
        max = t(1) ; moy = t(1)
        do i = 2, n
            ...
        end do
        moy = moy/n
    end subroutine sp
end program exsubrout
```

Also possible:

- n=size(t)
- implicit sizes (only for parameters !): real, dimension(:), intent(in)::t
Dynamic allocation

- Up to now: static tables (constant number of elements for the allocation process, as in C)
- If we do not know the number of elements when compiling?

```fortran
real, dimension (:,:), allocatable :: A
```

- We provide only the dimensions!
- Later on, we need to allocate memory:

```fortran
integer :: n,m
... 
read*,n,m
allocate(A(1:n,1:m))
```

- Change dimensions?

```fortran
deallocate(A)
read*,n,m ! assuming n < m
allocate(A(n:m,n:m))
```

- As in C, *free the memory*! (with `deallocate`)
Automatic allocation in a subprogram

```fortran
subroutine autom(m,n)
  ...  
  integer :: m,n
  real, dimension (m,n) :: A
```

- Only for a subprogram (we provide dimensions as parameters)
- Faster than allocation, easier to write
- One cannot check if we have enough memory to allocate the table

**Conclusion**: use it for subprograms frequently used; prefer classical allocation for big sizes not allocated too much time.
Operations on table: conformity

- The idea: use $A = B + C$ instead of a loop as

```fortran
  do i=1,n
    do j=1,n
      A(j,i) = B(j,i) + C(j,i)
    end do
  end do
```

- Tables must have the same shape!
  - $A(10,4)$ and $B(3:12,0:3)$ have the same shape,
  - $A(10,4)$ and $B(3:12,3)$ do not,
  - $A(1:1,1:1)$ and $B(1:1)$ do not either.

- Scalar is assumed having a correct shape for any table ($A = A + 1$ adds 1 to each element of A).

- Warning! $C = A \times B$ is *not* the matrix product of A and B!
Initialisation / “hand” tables

- Given \( n \) variables of the same type: \( v= (/\text{var1}, \cdots, \text{varn}/) \),

- Implicit loops:
  - \( v= (/i^2, i=1, 4/) \) ! same as \( v= (/1, 4, 9, 16/) \)
  - \( v= (/i+j, j=1, 2), i=1, 3/) \) ! (2, 3, 3, 4, 4, 5)
  - \( v= (/i, i=1, 11, 2/) \) ! (1, 3, 5, 7, 9, 11)

- From a vector to a multidimensional table:

  \[
  \text{reshape} (\text{vector}, \text{table_form})
  \]

  \[
  \begin{array}{c}
  \text{real, dimension}(2,3)::A \\
  A= \text{reshape}(v, (/2,3/))
  \end{array}
  \]

  is the matrix
  \[
  A = \begin{pmatrix}
  1 & 5 & 9 \\
  3 & 7 & 11
  \end{pmatrix}
  \]

  filled \textit{column by column}
Table sections and data storage

**real, dimension**(10,20)::A

| A(2:6,1:20) | ! - lines from 2 to 6 |
| A(2:6,:)     | ! - same |
| A(i,:)       | ! - i-th line |
| A(:,j)       | ! - j-th column |
| A(2:6,1:4)   | ! - submatrix (lines between 2 and 6, columns between 1 and 4) |

• Whatever the dimensions, elements are stored in a unidimensional table:

| a(1,1) | a(2,1) | a(3,1) | ⋮      | a(n,1) | ⋮      | a(n-1,m) | a(n,m) |

• when scanning a multidimensionnal table, the loop on the first dimension must be the intern one (see practical work).

• (otherway around in C).
Opening a file: \texttt{open(unit=FD, file=’filename’) }

- file opened at the first line (created if not existant),
- FD is a \textit{file descriptor} (constant different from 5 - standard input - and 6 - standard output) ; more details next week,
- if no \texttt{file} keyword, create the file \texttt{fort.FD} (FD the constant).

Read in the file: \texttt{read(FD,* ) \textit{variables}_of_the_good_type}

Write in the file: \texttt{write(FD,* ) var1, var2, ’anything’, var3}

Closing the file: \texttt{close(FD)}

Binary files: \texttt{open} with parameter \texttt{form=’unformatted’} ; \texttt{open} and \texttt{write} without the parameter \texttt{*}

One can provide some format (\texttt{fmt=’description’})
Modular programmation

- Extern procedures: subprograms written *outside* the main program or in another file.
- By default: types are not checked during compilation!
- The solution: modules. In a file `module_name.f90`:

```fortran
module module_name
  implicit none
contains
  subroutine subroutine_name
    ...
  end subroutine
end module module_name
```

- In programs using it: `use module_name` (before `implicit none`)
- also used to share constants, variables, new types…
Separated compilation

- **Global compilation:**
  
  \$ gfortran -o exec_name file1.f90 file2.f90

- **Separated compilation:**
  
  \$ gfortran -c file1.f90 \# create file1.o
  \$ gfortran -c file2.f90 \# create file2.o
  \$ gfortran -o exec_name file1.o file2.o \#link edition

- file1.f90 contains a module toto \? file toto.mod created

- *Order of compilation is important:* always compile files containing modules before files using this module.

- Use a makefile ! (same way than for C)
Makefile: one example

```
FORTRAN = gfortran # the compiler we use
CFLAGS  = -Wall -pedantic -c
LDFLAGS =

all : strassen.out

strassen.out: multiplication.o print.o strassen.o
   $(FORTRAN) -o $@ $^ $(LDFLAGS)

multiplication.mod multiplication.o: multiplication.f90
   $(FORTRAN) -o $*.o $(CFLAGS) $<

print.mod print.o: print.f90
   $(FORTRAN) -o $*.o $(CFLAGS) $<

strassen.o: strassen.f90 multiplication.mod print.mod
   $(FORTRAN) -o $@ $(CFLAGS) $<

.PHONY: all clean fclean

# clean functions

clean:
   rm -f *.o *.mod *~

fclean: clean
   rm -f strassen.out
```

More details on (constant) tables

```
program example
  implicit none
  real, dimension(8,8) :: A
  call toto(A,8,8)
contains
  subroutine toto(B,LB,n)
    integer, intent(in) :: n, LB
    real, dimension(LB,LB), intent(in) :: B
    write(*,*) B(2,3)
  end subroutine
end program example
```

- What is given to toto on line 4 is the *address* of A!
- The program will print $A(2,3)$
One can provide submatrices!

```fortran
program example
  implicit none
  real, dimension(8,8) :: A
  call toto(A(5,5),8,4)
contains
  subroutine toto(B,LB,n)
    integer, intent(in) :: n, LB
    real, dimension(LB,LB), intent(in) :: B
    write(*,*) B(2,3)
  end subroutine
end program example
```

- Parameter of toto on line 4 is the *address* of $A(5,5)$, i.e.
  $B=A+((5-1)*8+(5-1))*4$ (not taking care of the size of a real later on)
- The program will print $A(6,7)$:
  $B(2,3)=B+(3-1)*LB+(2-1)=B+17=A+53=A+(7-1)*8+(6-1)=A(6,7)$
One example

```
subroutine add(lambda,A,B,C,n,LA,LB,LC)
    ! LA, LB and LC the computer size of A,B and C
    ! n the mathematical size of the operation
    ! C=A+lambda*B (lambda = 1 or -1 for our purpose)
    integer, intent(in) :: n, LA,LB,LC,lambda
    real, dimension(LA,LA), intent(in) :: A
    real, dimension(LB,LB), intent(in) :: B
    real, dimension(LC,LC), intent(out) :: C
    integer :: i, j
    do i=1,n
        do j=1,n
            C(j,i)=A(j,i)+lambda*B(j,i)
        end do
    end do
end subroutine add
```
Recursive subprograms

**recursive subroutine** name(arg_list)

**recursive function** proc_name(arg_list) **result**(res_name)

- **result** is mandatory for recursive functions
- one example (there are better ways to compute it):

```fortran
recursive function fibonacci (n) **result** (u)
    integer, intent (in) :: n
    integer :: u
    select case (n)
        case (0)
            u = 0
        case (1)
            u = 1
        case default
            u = fibonacci (n-1) + fibonacci (n-2)
    end select
end function fibonacci
```
New types?

• Definition:

```fortran
  type type_name
    field_declarations
  end type type_name
```

(no parameter or allocatable in a field)

• Declaration:

```fortran
  type type_name [,parameter_list] :: variable_name
```

• One example:

```fortran
  type student
    character (len=20) :: name
    integer :: number
    real :: mark
  end type student
  type (student) :: toto,tata
  toto%name='maxime'
toto%number=1
toto%mark=10
```
Number precision: key word **kind**

- **Some examples:**

  ```fortran
  real(kind=8) :: x ! real encoded on 8 bytes
  integer(kind=2) :: n ! integer encoded on 2 bytes
  real(kind=kind(1.d0)) :: x ! double precision real on any computer
  ```

- **Also working on constants:**

  ```fortran
  34.56021495_8 ! real encoded on 8 bytes
  123456_2 ! integer encoded on 2 bytes
  2_'toto' ! string, each character encoded on two bytes
  ```

- **Similar:**

  ```fortran
  integer, parameter :: two=2, eight=8
  34.56021495_eight
  123456_two
  two_'toto'
  ```