C2115
Practical introduction to supercomputing

Lesson 7

Petr Kulhánek

kulhanek@chemi.muni.cz

National Centre for Biomolecular Research, Faculty of Science
Masaryk University, Kamenice 5, CZ-62500 Brno

Content

FORTRAN

- ### Introduction
  history of Fortran language, Hello world!, compilers, compilation, compiler options
- ### Syntax
  program, differences from F77, variables, control structures, I/O, arrays, functions, procedures
- ### Exercises
  simple programs, calculation of a definite integral
- ### Literature

Introduction

History

Fortran (abbreviation of words FORmula and TRANslator) in informatics is an imperative programming language designed by IBM for **scientific calculations and numerical applications**.

Source: wikipedia

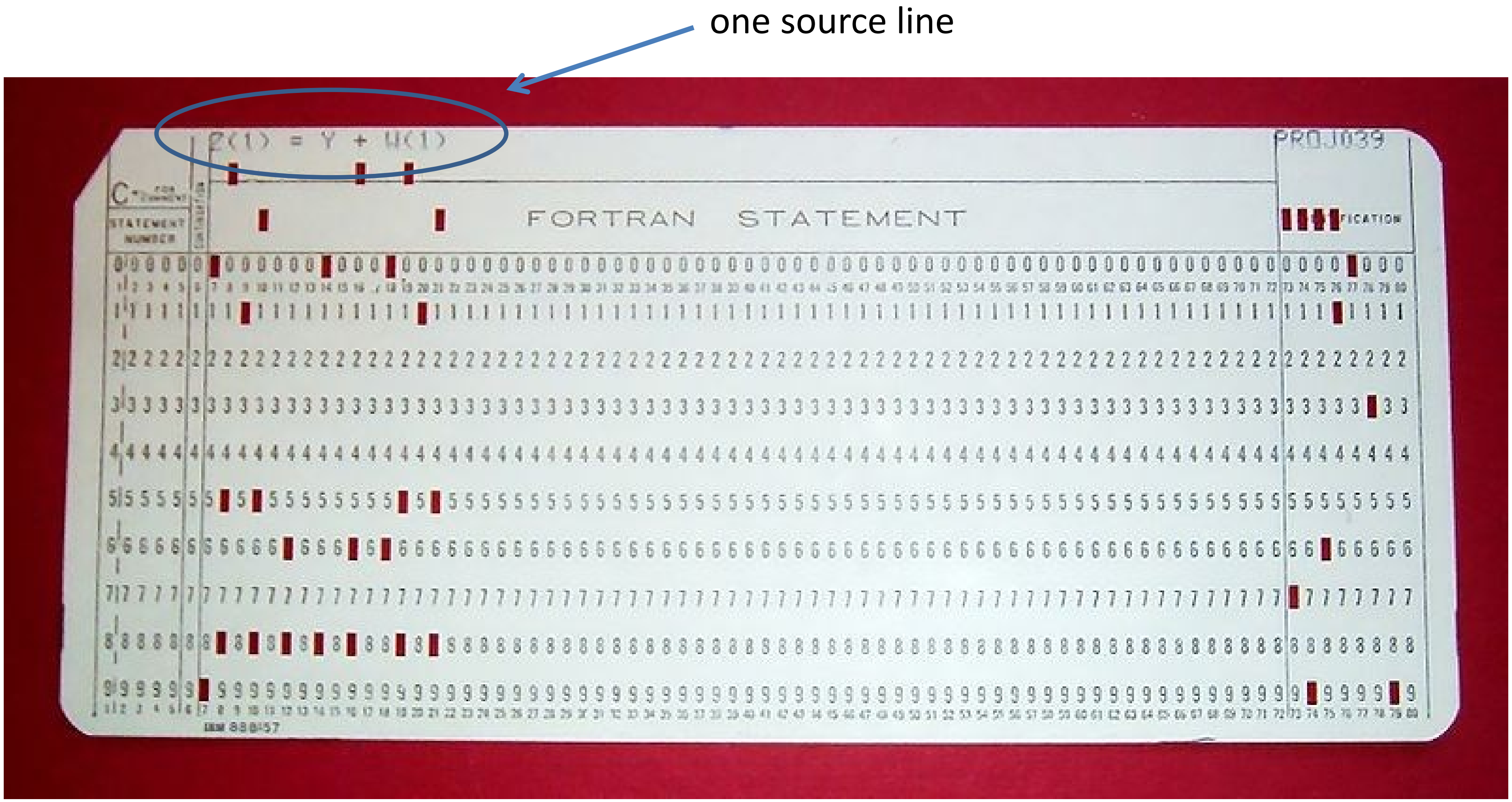
Language version:

Fortran 77
Fortran 90
Fortran 95
Fortran 2003
Fortran 2008

Several libraries are written in this language Compilers are able to create **highly optimized code**.

Standard math libraries: BLAS, LAPACK and others at http://www.netlib.org

History



Source: wikipedia

Hello world!

hello.f90

```
program Hello

write(*,*) 'Hello world!'

end program
```

Compilation:

```
$ gfortran hello.f90 -o hello
```

Starting:

```
$ ./hello
```

Assembler compilation:

```
$ gfortran hello.f90 -S
```

$\longrightarrow$ `hello.s`

Exercise 1

1. Create a hello.f90 file. Compile it with gfortran compiler. Verify the function of the created program.

Compilers

GNU GCC

Compiles: **gfortran**
License type: GNU GPL (freely available)
URL: http://gcc.gnu.org/wiki/GFortran

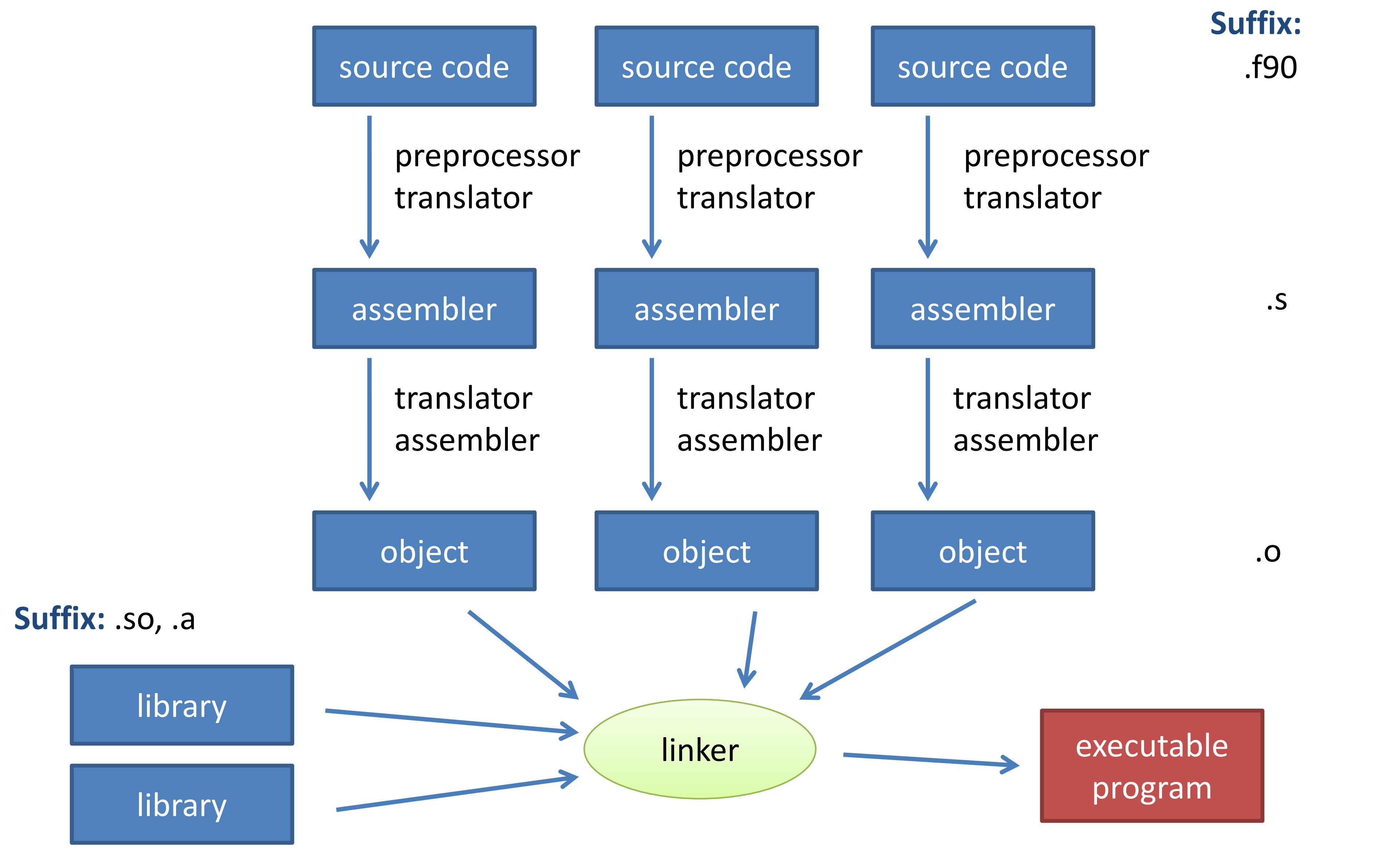
Intel® Composer XE

Compiler: **ifort**
Type of license: (a) commercial (available at MetaCentrum, meta modules: intelcdk)
(b) free for personal use after registration (linux)
URL: http://software.intel.com/en-us/articles/intel-composer-xe/

The Portland Group

Compiler : **pgf90**, **pgf77**
License type: commercial (available in MetaCentrum, meta modules: pgicdk)
URL: http://www.pgroup.com/

Compilation...



Useful compiler options

Compiler options:

-o	name of the resulting program
-c	translates source code into object code
-S	compiles the source code into assembler
-Ox	the level of optimization of the resulting program, where x=0 (none), 1, 2, 3 (the highest)
-g	inserts additional information and code for debugging the program run (slows down the program run)
-lname	linking of library *name* to the final program
-Lpath	path to libraries that are not in standard ways

Compiler options (ifort):

-trace all	controls ranges of arrays, use of uninitialized variables, etc.

Programs written in Fortran

Gaussian

http://www.gaussian.com/
Commercial program for quantum chemical calculations.

AMBER

http://www.ambermd.org/
Academic software for molecular simulations using molecular mechanics and hybrid QM/MM methods. Programs **sander** and **pmemd** are written in Fortran.

CPMD

http://www.cpmd.org/
Academic software designed for molecular simulations using methods of density functional.

Other software: Turbomole, DALTON, CP2K, ABINIT and others...
http://en.wikipedia.org/wiki/List_of_quantum_chemistry_and_solid_state_physics_software

Syntax

F77 dialect

- fixed format
- column 1, if it starts with the letter C, is a comment.
- columns 1-6 is devoted to labels (for I/O formats, loops)
- column 6, if it contains a character *, is continuation of previous line
- columns 7-72 contain line of the program

```
123456789012345678901234567890012345678912345678901234567890123456789001234567890

C this is a comment
      implict none
      real       f
      integer    a, b
C ---------------------------
C sum numbers a and b
      a = a + b
C long line
      f = a*10.0 + 11.2*b
     *+ (a+b)**2
100   format(I10)
      write(*,100) a
```

Source files

- Fortran 90 and higher uses free syntax (commands no longer need to be column-aligned, as was the case with Fortran 77).
- Allowed source file name suffixes: **.**fpp, .**f90**, .f95, .f03,.f08
- Fortran is not case-sensitive
- It is not advisable to use a tab to indent.
- Comments can start anywhere, to start a comment an exclamation mark is used !.
- The maximum line length is limited (typically 132 characters). The ampersand character is used to write longer expressions &.

```
implicit none
real            :: f
integer         :: A, B
! ---------------------------
! Add numbers A a B
A = A + B
f = A*10.0 + 11.2*B &
    + (A+B)**2     ! Long line
```

Preprocessor

- Source file can contain directives of CPP preprocessor (used by C and C++ languages)

```
#include <file>
#include “file"
#ifdef
#ifundef
#if
#else
#endif
#define
```

and more ...

- Processing of the file by the preprocessor can be forced by selecting the compiler, or by changing the file ending to: .fpp, .FPP, F90, .F95, .F03, .F08

http://gcc.gnu.org/onlinedocs/gfortran/Preprocessing-Options.html

Section Program

```
program Hello
! definition of variables

! program itself
write(*, *)  ' Hello world! '

! End program
end program
```

direction of program execution

The program can be terminated prematurely by a command **stop**.

Variables

```
implicit none

logical                   :: f
integer                   :: a, g
real                      :: c, d
double precision          :: e
character(len=30)         :: s
```

turns off automatic variable declaration

real number in simple precision

real number in double precision

string (text)

maximum string length and markích

Alternative entries:

```
real(4)          :: c, d
real(8)          :: e
```

We define variables at the beginning of a program, function, or procedure.

Variables

```
implicit none
logical                   :: f
!-------------------------
f = .TRUE.
write(*,*) f
f = .FALSE.
write(*,*) f
```

```
implicit none
real                      :: a,b
!-------------------------
a = 1.0
b = 2.0
b = a + b
write(*,*) a, b
```

```
implicit none
character(len=30)     :: s
!-------------------------
s = 'some text'
write(*,*) trim(f)
```

We always initialize variables
(i.e., we assign them a default value).

trim function truncates the string to the right (removes blanks)

Variables

```
implicit none
real            :: a = 1.0
real            :: b
!-------------------------
b = 2.0
b = a + b
write(*,*) a, b
```

~~`:: a =`~~

We NEVER initialize a variable during their declaration.

permitted construction, which translates as

```
real,save     :: a = 1.0
```

similar to keyword "**static** " from C and C++

Mathematical operations

Operators:

+	addition
-	subtraction
*****	multiplication
/	division
******	power

Without direct support:

MOD(n, m) modulo (**n % m** from C language)

```
real             :: a, b, c
!-------------------------
a = 1.0
b = 2.0
c = 4.0
b = a + b
b = a * b / c
c = a ** 2 + b ** 2
```

Loops 1

```
do variable = initial_value, end_value [, step]
       command1
       command2
       ...
end do
```

Variable can only be **integer**.

```
integer                    :: i
!-------------------------
do i = 1, 10
   write(*,*) i
end do
```

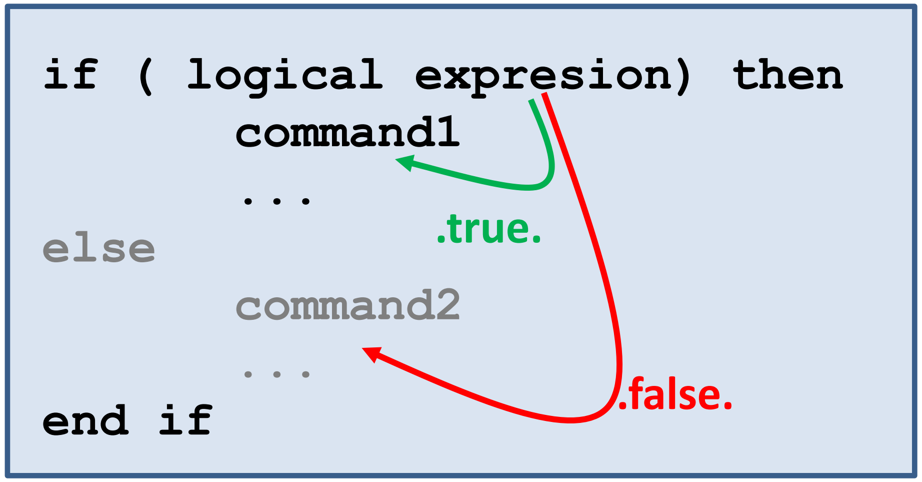
List numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```
integer                    :: i
!-------------------------
do i = 1, 10, 2
   write(*,*) i
end do
```

List numbers: 1, 3, 5, 7, 9

Run of cycles can be controlled by **loop** (similar to continue from C) and **exit** (similar to break).

Conditions



```
if ( logical expresion) then
        command1
        ...
else
        command2
        ...
end if
```

```
integer                    :: i = 7
!-------------------------
if ( i .gt. 5 ) then
   write(*,*) ,i is greater than 5'
end if
```

Logical operators:

.and.	logical yes
.or.	logical or
.not.	negation

Comparison operators (numbers):

.eq.	equals
.ne.	does not equal
.lt.	lower than
.le.	lower than or equal to
.gt.	greater than
.ge.	greater than or equal to

Comparative operators (logical):

.eqv.	equivalence
.neqv.	inequivalence

Loops 2

```
do while ( logical_expression )
       command1
       command2
       ...
end do
```

loop cycles as long as **logical_expression** returns .**true**.

```
double precision     :: a
!-------------------------
a = 0.0
do while ( a .le. 5 )
   write(*,*) a
   a = a + 0.1
end do
```

Lists numbers from 0 to 5 with 0.1 step

Loop execution can be controlled by commands **cycle** (similar to continue from language C) and **exit** (similar to break).

Functions and procedures

Function is part of program that can be **repeatedly** called from different places in the code. **Procedure** is similar to the function, but unlike the function **does not return a value**. Proper use of functions and procedures increases program readability and reduces duplicate code.

```
program Hello
! definition of variables
…
! Program itself
! calling of functions or procedures
…
! end program
…
contains

! definition of functions or procedures

end program
```

Functions and procedures can be called both from the program itself and from the functions and procedures themselves.

Arguments of functions and procedures are **transmitted by reference**.

Definition of function

```
function my_function(a, b, c) result(x)
implicit none
double precision :: a, b, c   ! arguments (parameters) of function
double precision :: x         ! result of function
! ----------------------
 integer :: j                 ! local variable
! ------------------------------------------------ ----------
! Function itself
x = a + b + c
end function my_function
```

Alternative notation:

```
double precision function my_function(a, b, c)
…
my_function = a + b + c
end function my_function
```

Definition of procedures

```
subroutine my_procedure(a, b, c)
implicit none
double precision :: a, b, c  ! arguments (parameters) of procedure
! ---------------------
 integer :: j                ! local variable
! ----------------------------------------------- ---------
! procedure itself
and = a + b + c
end subroutine my_procedure
```

The access properties of function and procedure arguments can be changed using a keyword **intent**. The default access property is intent(**inout**).

```
double precision, intent(in)      :: a      ! argument can only be read
double precision, intent(out)     :: b      ! argument can only be writtendouble
double precision, intent(inout)   :: c      ! argument can be worked with any way
```

Calling functions and procedures

Function calls:

```
double precision :: a
double precision :: d
! ----------------------------
a = 5.0
d = my_function_2(a)
write (*, *) d
```

~~double precision :: a~~
~~double precision :: d~~
~~! ----------------------------~~
~~a = 5.0~~
~~my_functions_2(a)~~

Calling procedures:

```
double precision :: a
double precision :: d
! ----------------------------
a = 5.0
d = 2.0
call my_procedure_3(a, d)
```

Result of the function **must be used**.

Passing arguments by reference

```
double precision :: a
double precision :: d
!----------------------------
a = 5.0
d = 2.0
write(*, *) d
call my_procedure_3(a, d)
write(*, *) d
```

2

?

```
subroutine My_procedure_3 (a, b)
implicit none
double precision :: a, b  ! arguments (parameters)
!--------------------------------------------------------
! procedure itselft
b = a + b
end subroutine my_procedure_3
```

Passing arguments by reference

```
double precision :: a
double precision :: d
! ----------------------------
a = 5.0
d = 2.0
write(*, *) d
call my_procedure_3(a, d)
write(*, *) d
```

2

7

In C, the value would be 2.

```
subroutine My_procedure_3(a, b)
implicit none
double precision :: a, b  ! arguments (parameters)
!------------------------------------------------ ----------
! procedure itself
b = a + b
end subroutine my_procedure_3
```

Standard functions and procedures

Mathematical functions:

sin(x)	
cos(x)	
sqrt(x)	square root
exp(x)	
log(x)	**natural** logarithm
log10(x)	decimal logarithm

Random numbers:

call **random_seed**()	initializes random number generator
call **random_number**(number)	sets variable **number** to a random number in the interval <0.0; 1.0)

Measuring time:

call **cpu_time**(time)	sets the value of the variable **time** for program run time in seconds (with microsecond resolution)

Array

Statically defined arrays:

```
double precision  :: a(10)
double precision  :: d (14,13)
```

One-dimensional array of 10 elements.

Two-dimensional array of 14x13 elements. (14 rows and 13 columns)

Dynamically declared arrays:

```
double precision,allocatable          :: a(:)
double precision,allocatable          :: d (:, :)
! ------------------------------------------------ -----
! allocation of memory for the array
allocate(a(10000), d(200,300))

! use array

! free memory
deallocate(a,d)
```

One-dimensional array

Two-dimensional array.

Field dimensions can also be defined using integer variables.

Working with array

```
double precision :: a(10)
double precision :: d(14,13)
integer          :: i
!-----------------------------------

a(:) = 0.0    ! can also be written as a = 0.0

do i=1, 10
    write(*,*) i,  ' - ty prvek pole je ' , a(i)
end do

a = d(:,1)  ! write first column from
            ! matrix d into vector a
a(5) = 2.3456
d(1,5) = 1.23
write(*,*) d(1,5)
```

Array elements are indexed from one.*

Field size can be determined by function **size**.

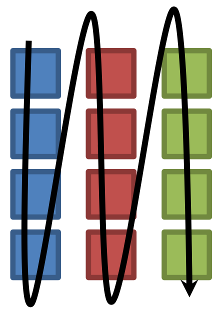
*however, the index ranges for each dimension can be changed

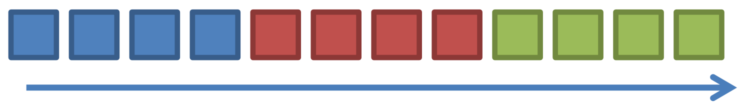
Array - memory model

Fortran

a(i,j)

Elements follow each other in columns (column based).



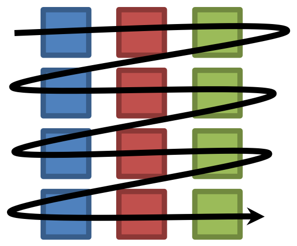


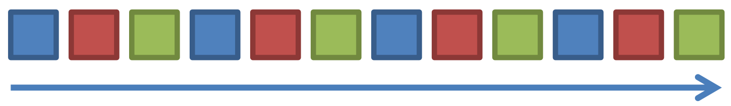
arrangement of matrix elements in memory

C/C++

A[i][j]

Elements follow each other in rows (row based).





If we call functions from BLAS or LAPACK libraries, we must consider different indexing of multidimensional arrays.

Array - memory model

Fortran

```
double precision :: d(10,10)
double precision :: sum
integer          :: i,j
!-----------------------------------

sum = 0.0d0
do i=1, 10
    do j=1,10
        sum = sum + d(j,i)
    end do
end do
```

index change is fastest for rows

C/C++

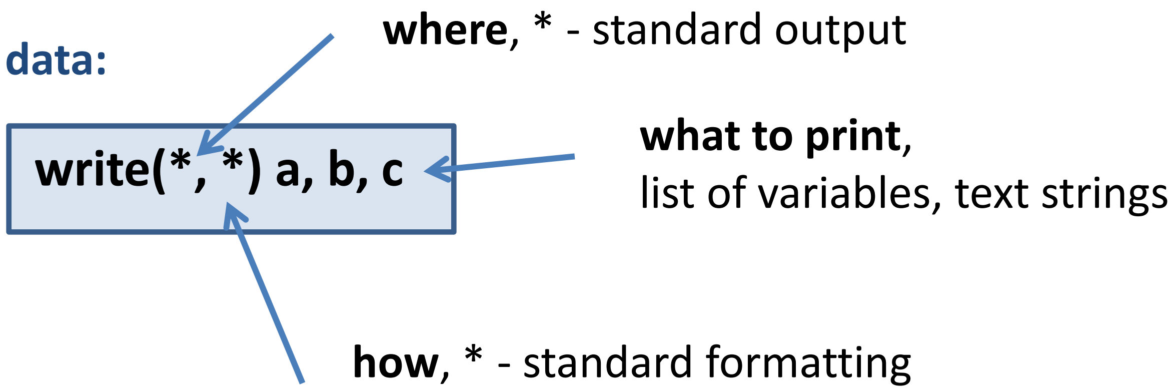
```
double* d[];
double   sum;
//-----------------------------------

sum = 0.0;
for(int i=0; i < 10; i++){
    for(int j=0; j < 10; j++){
        sum += d[i][j];
    }
}
```

index change is fastest for columns

Note: presented arrangement does not affect the function but the execution speed

I/O operations

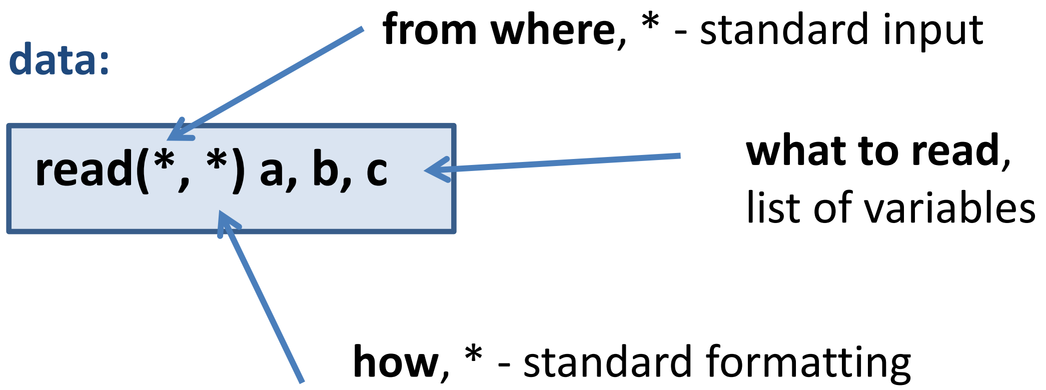
Printing data:



```
write(*, *) a, b, c
```

where, * - standard output
what to print,
list of variables, text strings
how, * - standard formatting

Reading data:



```
read(*, *) a, b, c
```

from where, * - standard input
what to read,
list of variables
how, * - standard formatting

Files are opened with the command **open**. They are closed with a command **close**.

I/O operations - formatting

Formatted output:

```
write(*,10) a, b, c

10 format('Value a = ', F10.6,' value b = ', F10.6,' value c = ', F10.6)
```

- format can be specified before or after the command write or read
- formatting types:
  - F - real number in fixed format
  - E - real number in scientific format
  - I - integer
  - A - string

Write data without end of line character:

```
write(*,10,ADVANCE = 'NO') a, b, c
```

format must be specified

Other language features

1. pointer support
2. structures
3. object-oriented programming

Homework

Optional.

Exercise 2

1. Write a program that calculates definite integral below. Use the rectangular method for integration.

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

2. What is an integral equal to? Justify the findings.

Literature

- http://www.root.cz/serialy/fortran-pro-vsechny/
- http://gcc.gnu.org/onlinedocs/gfortran/
- Compiler documentation ifort
- Clerman, NS Modern Fortran: style and usage; Cambridge University Press: New York, 2012.