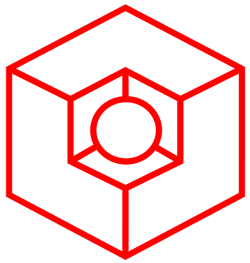


BremHLR

Kompetenzzentrum für Höchstleistungsrechnen Bremen

Introduction to Fortran

Lars Nerger (BremHLR & Alfred Wegener Institute)

Universität Bremen

CONSTRUCTOR UNIVERSITY

Hochschule Bremerhaven

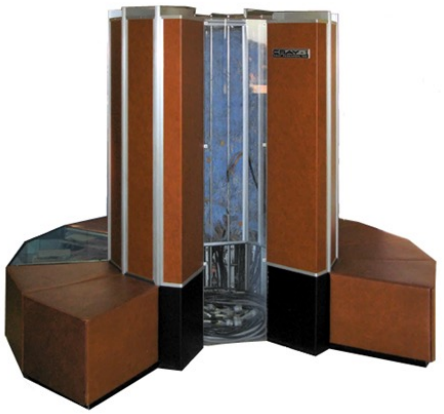
NATIONALES HOCHLEISTUNGS RECHNEN

Fortran?

- Abbreviated from **FOR**mula **TRAN**slation
- Aimed at numerical computations
- First version in 1957
- Widely used was FORTRAN77, standardized 1978
- New standards Fortran 2003 and 2008 added object-oriented features (but not often used by now)
- Fortran 2018 and 2023 improved the interoperability with C and added some ‘native’ features for parallelization
- Currently, most used features were introduced with Fortran90/95 more and more feature of Fortran 2003 and later are being used (but there are still programs coded in FORTRAN77)
- Today:
  - A procedural and also object-oriented programming language

Why Fortran?

- Today, most programs are **not** coded in Fortran!
- But: Many programs for large-scale numerical computations are coded in Fortran
  - Often their development was started when Fortran was essential to obtain good performance (e.g. on old CRAY computers)
  - Also:
    - Handling matrices and higher-dimensional arrays is easy
    - It's easy to get very good compute performance



Cray-1, year 1976
The first "supercomputer"

- Situation Today
  - Still, the most widely spread simulation programs, e.g. for earth sciences (ocean, atmosphere, land surface, climate), are coded in Fortran
  - Switching to C would be a huge effort – without obvious performance gain (Perhaps, finding programmers would be easier)
  - Performance is important!
    (it's easier to mess up performance in with C – and particularly C++)

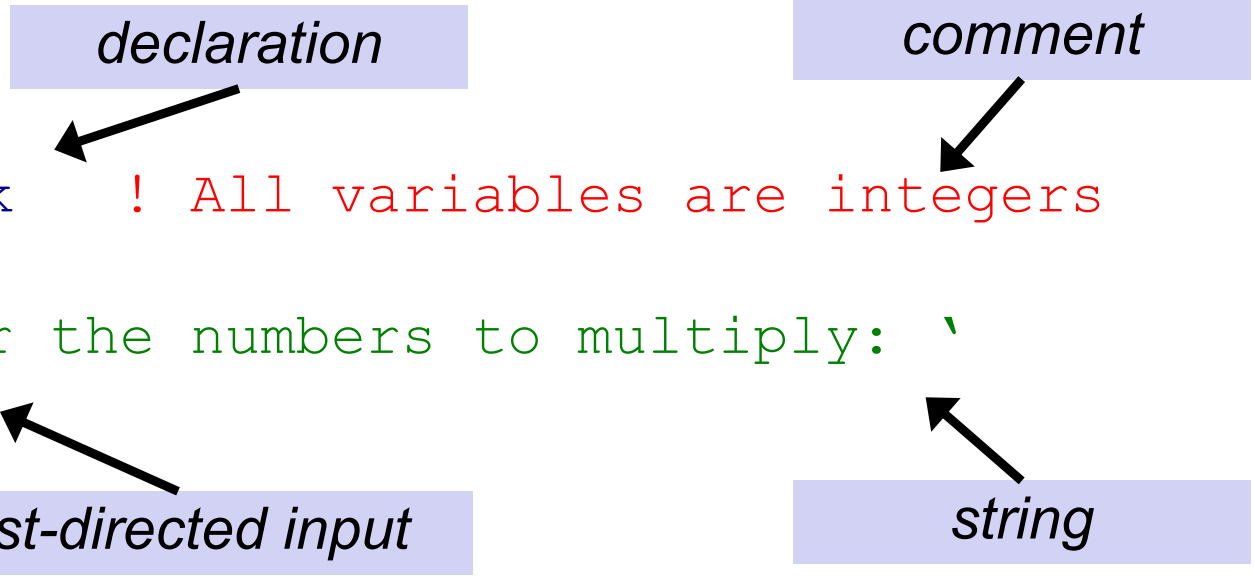
Structure of a Fortran program

```
PROGRAM program_name

IMPLICIT NONE
    [declaration section]
    [execution section]
END PROGRAM program_name
```

- Upper case for Fortran statements is not required (Fortran is case-insensitive)

A simple Fortran program

```
PROGRAM first_example

IMPLICIT NONE

INTEGER :: i, j, k    ! All variables are integers

WRITE (*,*) 'Enter the numbers to multiply: '
READ (*,*) i, j

k = i * j

WRITE (*,*) 'Result = ', k

END PROGRAM program-name
```



Compile and execute

Compile with

```
$ gfortran myprogram.f90 -o myprogram
```

This generates the program ‘myprogram’

Then execute the program

```
$ ./myprogram
```

```
 Enter the numbers to multiply:
4 6
 Result =            24
```

Numbers can also be entered

- as 4, 6
- in two lines
- from a file with ‘<‘

Declaring variables

- All variables have a be declared specifying their type:

```
INTEGER :: i                          integer variable
REAL :: time                          real-valued variable
CHARACTER(len=10) :: string           string of length 10
LOGICAL :: test1                      logical variable
```

- Variables can be assigned a value upon declaration

```
INTEGER :: i = 10
```

(valid until the value is changed the first time by the program)

- Named constants should be declared as 'PARAMETER'

```
REAL, PARAMETER :: pi = 3.141593
```

- `LOGICAL` has values `.TRUE.` or `.FALSE.` (2 dots!)

Arithmetic operators and functions

- Basic duty of Fortran programs: Do arithmetics
- Operators:

Type	Operator		Associativity
Arithmetic	** (exponentiation)		right to left
	*	/	left to right
	+	-	left to right

high priority

low priority

- Intrinsic functions, for example:

Function	Explanation
ABS(x)	Absolute value
SQRT(x)	Square root
EXP(x)	Exponential function
SIN(x), COS(x), ASIN(x),…	Trigonometric functions
MAX(x1, x2, …xn), MIN(…)	Maximum/minimum

Arithmetic calculations

Distinguish REAL and INTEGER arithmetics:

```
REAL :: f
INTEGER :: i
f = 3.0/2.0          ! Result f = 1.5    (up to numerical precision)
i = 3/2              ! Result i = 1      (truncated to integer)
```

Mixed-mode arithmetic

For each single operation

- Integer is converted to real
- Computations are performed in floating-point arithmetic
- For integer result, truncation is performed

What is: `1.0 + 1/4 = 1.0` (Be careful!)

Control constructs: Branches

- Execute some block of code depending on some condition:

```
IF (i < 0) THEN
    … program statements executed in case that i<0 …
END IF
```

- "`i < 0`" can be any logical expression or logical variable

Branches with alternatives

- Allow for alternative branches (REAL r)

```
IF (r < 0.0) THEN
    WRITE (*,*) 'i less than 0.0'
ELSE IF (r == 0.0) THEN
    WRITE (*,*) 'i equal to 0.0'
ELSE IF (r > 0.0) THEN
    WRITE (*,*) 'i more than 0.0'
ELSE
    WRITE (*,*) 'Something is really wrong!'
END IF
```

Relational and logical operators

- Operators used for logical expressions

Type	Operator						Associativity
Relational	<	<=	>	>=	==	/=	none
Logical	.NOT.						right to left
	.AND.						left to right
	.OR.						left to right
	.EQV.		.NEQV.				left to right

- `.EQV.` is true when two logical expressions yield the same result

Control constructs: Loops

The ‘while’ loop:

- Repeat indefinitely until some condition is satisfied

```
DO
    …
    IF (logical_expr) EXIT
    …
END DO
```

- ‘`logical_expr`’ can be any logical expression or logical variable

  Example: `k<13.5`, with k being computed within the loop

The iterative DO loop

A counting loop:

- Repeat according to loop counter, e.g. 10 times

```
DO i = 1, 10
        …
    Statements
    …
END DO
```

General form:

```
INTEGER :: i, istart, iend, incr

DO i = istart, iend, incr
```

- `'incr'` defines increment of `i`
- `i` must not be changed inside the loop
- `i` is not guaranteed to have a specific value after completion of the loop

Arrays

- Arrays can be declared with fixed size:

```
REAL :: field(10,20)
```

  - the array has ‘rank’ 2
  - Any rank >= 1 is possible

- In a loop one can access the array elements

```
DO i = 1, 20
   DO j = 1, 10
        WRITE (*,*) field(j, i)
   END DO
END DO
```

  - Array indices range from 1 to upper limit
  - The first index is contiguous in memory

Dynamical allocation of arrays

- We don’t need to set the dimensions at compile time
- Dynamical allocation allows for arrays of varying size:

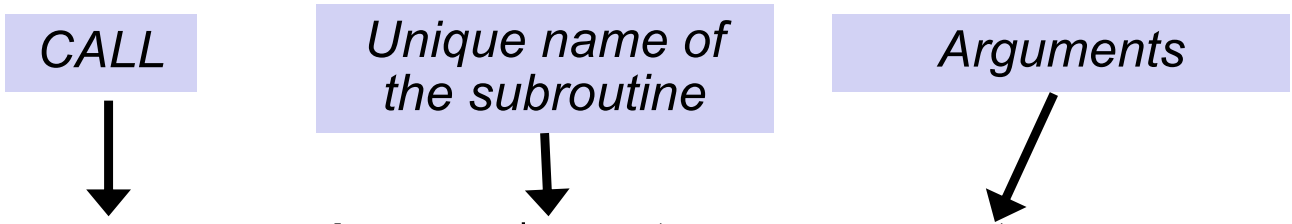
```
REAL, ALLOCATABLE :: field(:,:) ! Declare array of rank 2
INTEGER :: i, j
…
i = 10
j = 20
ALLOCATE(field(i, j))
…
… do some work with field
…
DEALLOCATE(field)
```

Procedures: Subroutines

- Good programming practice: Structure tasks into smaller procedures

- Example: A matrix-matrix multiplication

```
PROGRAM matmul

IMPLICIT NONE

REAL :: matA(10,10), matB(10,10), result(10,10)

CALL read_matrices(matA, matB)

CALL multiply_matrices(matA, matB, result)

CALL write_result_matrix(result)

END PROGRAM matmul
```



Structure of Subroutines

- Good programming practice: Structure tasks into smaller procedures

- Example: A matrix-matrix multiplication

```
SUBROUTINE multiply_matrices(matA, matB, matRes)
IMPLICIT NONE
REAL, INTENT(in)  :: matA(10,10)
REAL, INTENT(in)  :: matB(10,10)
REAL, INTENT(out) :: matRes(10,10)
```

Input arguments (must not be changed)

Output argument (has to be set in routine)

… Compute multiplication and store result in `matRes` …

```
END SUBROUTINE multiply_matrices
```

- Size of the matrices can also be given as integer arguments

Fortran Modules

- Declare variables for global use (similar for C header files)

- Declaration

```
MODULE mymod
     INTEGER :: ivar1
     REAL :: rvar2
END MODULE mymod
```

Static variables, e.g.: `INTEGER, SAVE :: ivar`

- Use in a program or subroutine

```
PROGRAM example
     USE mymod
     IMPLICIT NONE
     WRITE (*,*) 'ivar1', ivar1
END PROGRAM example
```

Fortran constants in include files

- A situation with MPI:

- Declare and define contants in separate ‘header’ file (`mpif.h`):

```
INTEGER MPI_STATUS_SIZE
PARAMETER (MPI_STATUS_SIZE=4)
```

- Use in a program or subroutine

```
PROGRAM example
    IMPLICIT NONE
    INCLUDE mpif.h
    INTEGER :: status(MPI_STATUS_SIZE)
    …
END PROGRAM example
```

- For own programs rather use a module

Fortran structures: TYPE

- A use-defined data type
- Example

```
PROGRAM example
     TYPE mytype                    ! Define the type
        INTEGER :: ivar
        REAL    :: rvar
     END TYPE mytype

     TYPE(mytype) :: myvar          ! Declare a typed variable

     myvar%ivar = 1                    ! Access 1st type member
     myvar%rvar = 2.0               ! Access 2nd type member
END PROGRAM example
```

- It can be handy to include the type-definition in a module

Good programming practice

- Comment your code
- Use meaningful variable names
- Use indenting to mark code blocks
- Use structured programming with subroutines
- Implement stepwise; test separate steps

Compiling Fortran programs

- Without MPI

```
gfortran -o example example.f90
```

- With MPI

```
mpif90 -o example example.f90
```

  - Option –o sets name of the executable program (without name will be `a.out`)

Libraries

- Precompiled (binary) collections of subroutines
- Can be written in C, Fortran, etc. (MPI is C)
- Combine with your code in linker stage of compilation
- Specification of library and library path can be necessary, e.g.

```
gfortran -o example example.f90 -LPATH_TO_LIBRARY -lmpi
```

- Syntax '`-lmpi`' for a library binary file `libmpi.so`

Some reading material

- An online Fortran tutorial:

http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html

- Books:

my favorite:

Stephen J. Chapman

Fortran 95/2003 for Scientists and Engineers

McGraw-Hill

but there are many others…

Exercise

- Write a program that writes ‘Hello world’ to the screen.

  (You can use Fortran or C)

- Compile

- Run the program to check whether it’s working correctly