

Computational Astrophysics I: Introduction and basic concepts

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

SoSe 2024, 20.6.2024



Programming in Fortran - Part 1

Introduction

Fortran = Formula translation

- first high level language (John Backus¹ 1957) – in contrast to Assembler
- contains a lot of builtin features, like power “**” and a data type for complex numbers
- initially on punch[ed] cards with → 80 characters per line

Advantages of punch cards (K. Ganzhorn 1966)

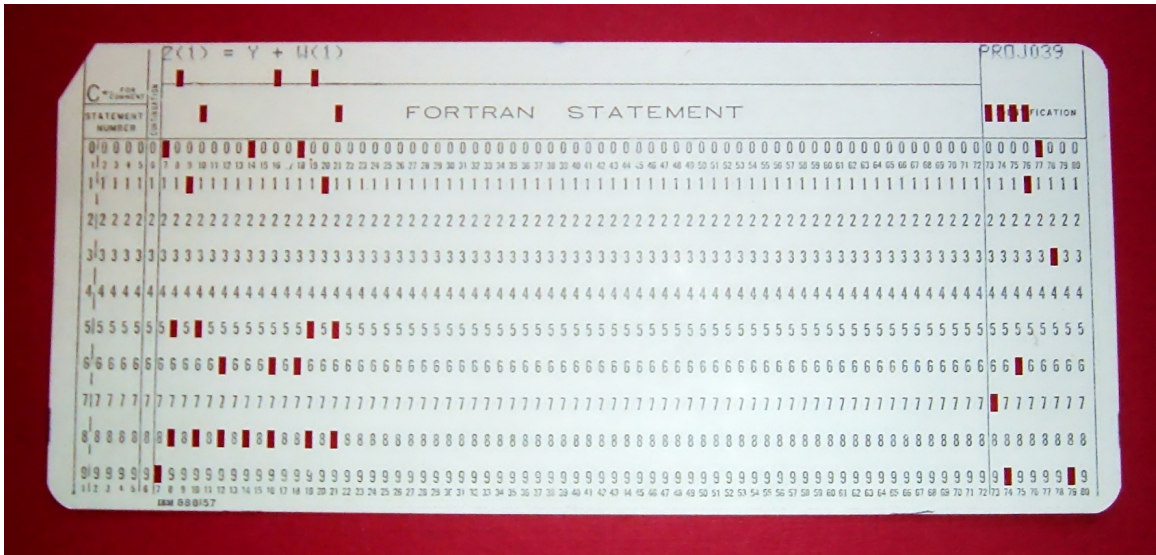
- machine and visual readability
- mechanical dublica-, mixa- and sortability
- outstandingly superior signal-to-noise ratio ($> 10^6$)
- cheap
- universal suitability for machine data input and output



¹1924 - 2007, Turing Award 1977, also ALGOL 58

Punch cards I

Original idea by Herman Hollerith (1860-1929) in 1898 → Hollerith constants in early FORTRAN

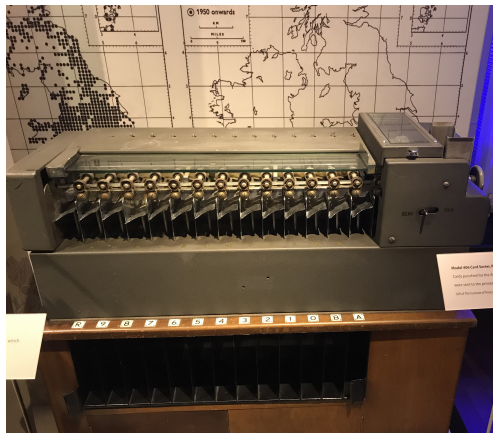


Punch cards II



Computer History Museum (CHM), Mountain View, 2019
Type 31 Alphabetical Duplicating Punch, IBM, USA,
1933

→ keyboard-operated card punch



CHM, 2019

Model 406 Card Sorter, Power-Samas, UK, 1954
→ used for Atlas of British Flora, can sort 300 cards /
min

- FORTRAN I (1954-57)
- FORTRAN II (1958)
- FORTRAN IV (1961)
- FORTRAN 66 (ANSI/ASA)
- FORTRAN 77 (structured, before: GOTO)
- Fortran 90 (free form)
- Fortran 95 (WHERE, FORALL, removed: REAL loops)
- Fortran 2000, Fortran 2003 (polymorphism, inheritance, object-oriented)
- Fortran 2008 (Coarray Fortran)
- Fortran 2018 (e.g., teams for Coarray Fortran)



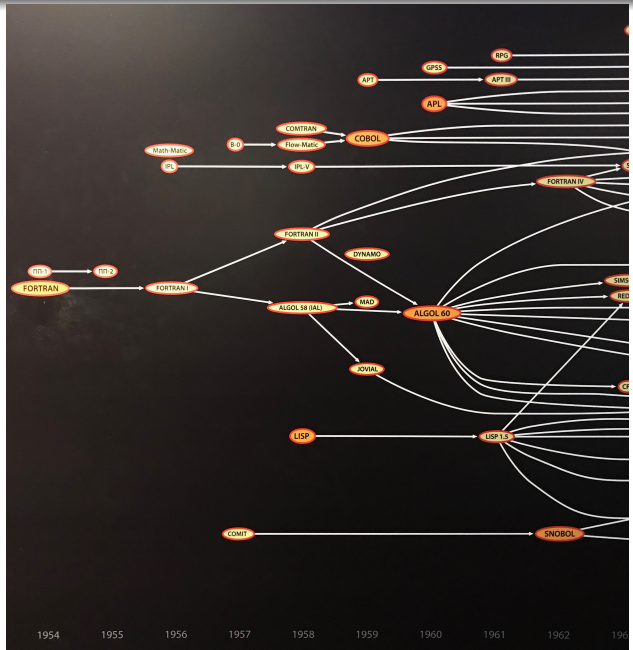
CHM, 2019

Conventions

In the following we will only consider FORTRAN 77 with some elements from Fortran 90/95.

Why Fortran? I

Fortran was the first high level language and is therefore the ancestor of all modern programming languages. Fortran is still used and developed (modern compilers).



Fortran can be easier (and therefore better) optimized by *Compilers* (see Sect. 31) – since Backus – and is also *easy to learn*.

This is amongst others due to:

- The Fortran programming language is more strict (=less flexible), e.g., loop variables (counter) cannot be changed within the loop:

```
DO I = 1 , 10  
  I = 10  
ENDDO
```

not permitted! (Error, e.g., gfortran *Error: Variable 'I' cannot be redefined inside loop*)
→ real loops, (max.) number of iterations determined before loop entry

High level language Fortran vs. Assembly I. → Machine language

Machine language

```
100 077400201750
101 075400000000
102 040000205670
103 200001200102
104 060100002720
```

Assembly code (line by line → machine code)

```
AXT 1000,B
PXD ,0
LOOP ADD MEM+1000,B
TIX LOOP,B,1
STO ISUM
```

Fortran (syntact. analysis → machine code)

```
ISUM = 0
DO 42 I = 1, 1000
42 ISUM = ISUM + MEM(I)
```

- originally: programmers had to write machine instructions (add, shift registers) as numbers
- *Assembly language* uses names instead of numbers for code and memory locations, but still: 1 line code for 1 computer instruction; translation to machine language by *Assembler*
- high level language (Fortran): English-like statements ; translated by *compiler* to machine code

- static data types: data types are known at compile time
- *but*: no explicit declaration (type checking) of arguments of procedures:
SUBROUTINE VEC_X_MATRIX (VEC, MATRIX, FLAG)
- therefore also possible: passing multi-dimensional arrays without giving the dimensions (e.g. re-dimensioning in procedure possible)
- no pointer needed (but they exist, e.g., Cray-pointer), procedures get only the start address of the arrays/variables

```
CALL ADDZ (ARG1,ARG2(2:4,2:5))  
...  
SUBROUTINE ADDZ (ARG1,ARG2)  
DIMENSION ARG2(3,4)
```



Why Fortran? V

- Fortran is a *procedural* language without templates, header files, etc. ²
→ short compile time, highly optimizable by compilers (see Sect. 31)

“Weaknesses” of Fortran

On the other hand: no type checking when passing variables to procedures (exception: optional INTERFACE).

Many Fortran programmers use GOTO instructions, one of the main reasons of “Spaghetti” code

There is no general exception handling, no good graphics library, etc.

However, OS for *Prime* in 1970s completely in Fortran →



CHM, 2019

²but: MODULE, INTERFACE with INCLUDE

Table: Survey among IT people

Skill	% of the respondent	annual income / €
C	16.8	55 304
C++	19.2	53 568
Cobol	1.7	65 813
Fortran	0.6*	67 185
Java	28.8	54 164
SQL	43.0	50 881

3446 respondents, c't 2011, 6

* = 21 respondents

- LINPACK → benchmarking of supercomputers (PFLOPS) & LAPACK
→ systems of linear equations
- astrophysics:
radiative transfer (e.g., CMFGEN, PoWR, FASTWIND, TMAP, MOCASSIN),
CMBFAST (but outdated),
hydrodynamics (ZEUS),
data analysis (e.g., ABSOLUT for absorption lines)
- particle physics: SIMDET (simulation of detectors of colliders), PYTHIA (until V6.4, MC simulation of particle decays)
- but also application software: WRplot (with some C-routines for graphics and file handling)

Current Fortran compilers (June 2024):

- gfortran (GCC) 9.5 (May 2022)
- ifort (Intel) 2024.1.0 (April 2024)
- PGI CUDA Fortran (PGI/NVIDIA) 20.4 (May 2020)
- g95 (Andy) 0.93 (Oct 2012!)
- openf90/openf95 (AMD x86 Open64) 4.5.2 (2014!)
- and many others ...

each of these compilers has specific advantages and disadvantages, some compilers (`gfortran`, `g95`) do not support all features (e.g. `ENCODE`).

Moreover, `gfortran` is only a front end for `gcc` and translates Fortran into an abstract syntax tree (AST) → less optimization potential.

The PGI compiler and `ifort` are commercial products (but can be obtained for free under specific conditions), the other compilers are for free and mostly open source.

gfortran

- `gfortran myprogram.f` → creates executable program `a.out` from source file `myprogram.f`
- `gfortran -o myprogram myprogram.f` → creates executable program `myprogram`
- `gfortran -c myprogram.f` → creates object file `myprogram.o`
- `gfortran -o myprogram myprogram.o` → links the object file against the runtime libraries to create the program

ifort

- first step: set compiler path etc., e.g.,
`source /opt/intel/[...]/bin/ifortvars.sh intel64`
- `ifort -o myprogram.exe myprogram.f` → as for `gfortran`

- Intel Fortran Language Reference, about 900 pages, extensive *reference* containing many examples
- Using GNU Fortran, about 200 pages, documentation of gfortran intrinsics and behavior
- Modern Fortran Explained (Metcalf et al. 2018), including Fortran 2018
- Modern Fortran (Curcic 2020), most recent Fortran book, focus on parallelization

... what a developer usually has to know:

- ① Where to put the semicolons?
- ② How to insert a comment?

In Fortran 77 (default source format, format because of → punchcards):

① one line = one instruction

(end of line usually after column 72, continuation lines possible via a character in column 6 or via & in the previous line).

→ A semicolon separates multiple instructions within a single line.

```
1234567          ... 72 column number
```

```
    WRITE (*, '(A,A,A)')
```

```
    > "Hello"
```

```
    & " world!" ; END
```

→ columns 1 - 5 for label (pos. integer number) reserved, e.g.,

```
123456
```

```
    IF (BERROR) GOTO 999
```

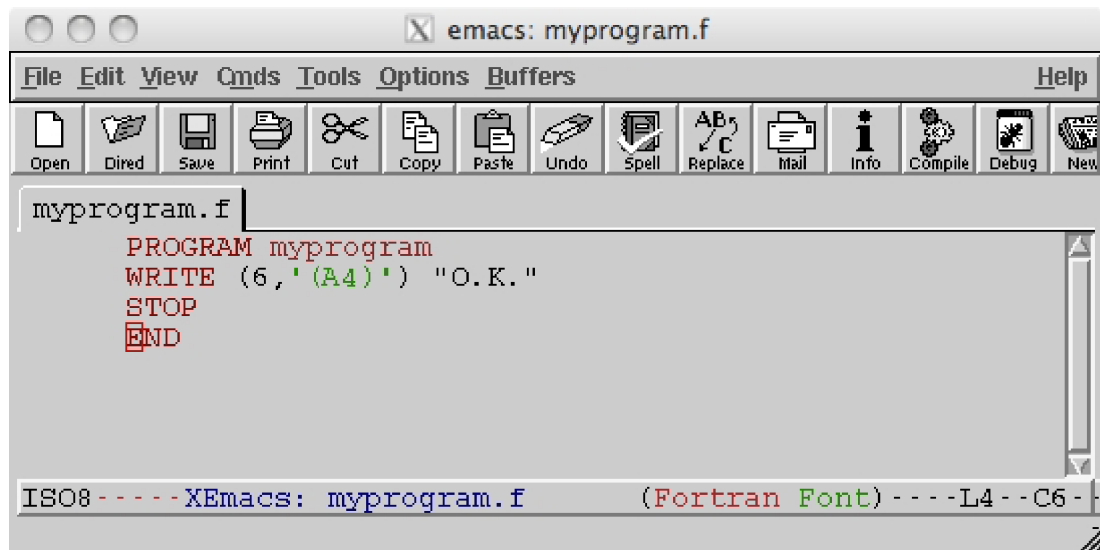
```
    ...
```

```
  999 STOP 'INPUT ERROR'
```

- 2 A **comment line** is indicated by a comment indicator (!, C, *) in the first column, since Fortran 90, also via a preceding exclamation mark "!". Also, everything after column 72 is ignored (comment).

```
0000000001 ... 777777777
1234567890 ... 01234567
C      a comment
      ! comment
      EXP(-PI)    comment
```

A simple program:



The image shows a screenshot of the XEmacs text editor window. The title bar reads "emacs: myprogram.f". The menu bar includes "File", "Edit", "View", "Cmds", "Tools", "Options", "Buffers", and "Help". The toolbar contains icons for Open, Dired, Save, Print, Cut, Copy, Paste, Undo, Spell, Replace, Mail, Info, Compile, Debug, and New. The main text area displays the following Fortran code:

```
myprogram.f  
PROGRAM myprogram  
WRITE (6, '(A4)') "O.K."  
STOP  
END
```

The status bar at the bottom shows "ISO8 - - - - XEmacs: myprogram.f (Fortran Font) - - - - L4 - - C6 -".

- PROGRAM program_name
→ optional, recommended
- STOP
→ optional, recommended
- END [PROGRAM program_name]
→ obligatory

Hints:

- Fortran doesn't distinguish lowercase and uppercase characters for instructions and variable names.
- Fortran can be written without blanks, e.g., PROGRAMmyprogram
- default file extensions for Fortran source files:
.f .for .f90 (and some more)
→ depends on compiler

Labels and GOTO

- lines (= instruction) can/must be labeled with a number in the beginning

```
100 READ(*,101,ERR=102) RADIUS
101 FORMAT (F15.0)
...
STOP
102 PRINT * , 'Wrong input, again please!'
GOTO 100
```

- labels can be targeted by GOTO → immediate jump to this instruction → way to hell ...



the universal output instruction is

WRITE

³ E.g.:

```
WRITE (*,*) "Hello world!" , "Hello!"
```

Meaning of the *:

```
WRITE (UNIT,FORMAT)
```

³there also exists PRINT

The instructions PRINT and WRITE(*,*) write to stdout (terminal, shell). Furthermore, WRITE can write to a “unit” (device):

```
WRITE(J,*)
```

where J is an integer number with $0 \leq J \leq 2\,147\,483\,640$ (ifort)

unit	uninitialized meaning	channel
*	<i>Always:</i> screen / keyboard	STDOUT / STDIN
0	screen (terminal)	STDERR
5	keyboard (terminal)	STDIN
6	screen (terminal)	STDOUT

The import of data is done by

READ

- syntax similar to WRITE, e.g.

```
READ (*,*) RADIUS
```

i.e. READ (UNIT,FORMAT)

- problem: wrong type at input (e.g., string instead of float) causes program crash, therefore always:
- catch wrong input, e.g.:

```
100 READ(*,*,ERR=101) RADIUS
```

```
...
```

```
STOP
```

```
101 PRINT * , 'Wrong input, again please!' ; GOTO 100
```

```
END
```

- instead of * in READ / WRITE better use format string
- e.g., WRITE (*, '(A,F10.2)') 'Radius is ', RADIUS
- enclose string with ' ' or " " and put them in parentheses ()

Example:

- A : text, A26 → 26 characters
- Fn.d : fixed point, F4.1 → one decimal place, two digits left before decimal point
- En.dEz : floating-point, E10.1E2 → 10 digits in total, 1 digit after decimal point, 2 digits for exponent

Hint

Reading any format of numbers (integer, fixed, float) via fixed point format with number of positions after decimal point = zero into a REAL variable, e.g.,

F20.0

FORMAT statement

- the format can also be set via an instruction, e.g.,

```
    READ (5,200) KARTE
200  FORMAT (A132)
```

or use `READ(5,FMT=200)` for better readability

- advantage of the FORMAT statement:
 - re-usability (e.g., multiple READ / WRITE statements with same format)
 - clarity: collecting FORMAT statements between STOP and END PROGRAM
 - can contain text:

```
    WRITE (*,100), SUM
100  FORMAT ("The result is: ",F10.4)
```

Note that the argument of the FORMAT statement is written without quotation marks and that a FORMAT statement must be always labelled.

Alternatively: Format strings can be constructed during runtime! (flexibility)

```
CHARACTER*8 FORM
...
FORM = "(A,F5.1)"
...
FORM(7:7) = "2" ! changes F5.1 to F5.2
WRITE (0,FMT=FORM) "found unknown", X
```

Fortran allows the use of variables without explicit declaration.

→ data type (INTEGER, REAL) is determined by initial letter of the variable name:

- initial letter I, J, K, L, M, N for integer numbers, e.g.,

```
DO I = 1, K → counting loop
```

```
Nparticipants = 8
```

- all others: floating points (REAL), e.g.,

```
PI = 3.141
```

```
XMASS = 2.E33
```

more on that later ...

FORTRAN 77 – fixed form:

```

                                77777
1234567890                      34567
      WRITE (*,*) ... comment
```

Fortran 90 – free form:

```
WRITE (*,*) ... ! comment
```

- default: fixed form
- free form via: file extension `.f90`
or compiler option:

```
gfortran: -ffree-form
ifort:    -free
```

Programming in Fortran - Part 2

Data types, input, output, files, execution control

We already know:

- integer number variables (INTEGER) start with: I, J, K, L, M, N
- floating point number variables (REAL): any other letter

with help of the IMPLICIT instruction it is possible to assign other letters to the variables, e.g.,

- IMPLICIT COMPLEX (c, z)
- IMPLICIT LOGICAL (b)

Advantage of implicit declarations

compact declaration block (only needed for arrays and other data types)

Big disadvantage:

Typos of variable names are not detected at compile time. E.g.,

```
RSUN = 69.57E9
```

```
RADIUS = RADIUS / RSUNN
```

often leads to errors which are difficult to reproduce
(here: division by 0)

Therefore better and always recommended:

```
IMPLICIT NONE
```

as the first statement in each program (or function/subroutine).

This statement switches off implicit declaration and requires the explicit declaration of a variable before use (like, e.g., in *C/C++*)

Example

```
PROGRAM SIMPLE  
IMPLICIT NONE  
REAL RSUN, RADIUS
```

The declaration block

- Explicit declaration of variables is exclusively done in the “declaration block”.
- This block must be at the beginning of the program.
- The initialization of variables via “=” can only be done after this block.

Exceptions:

- Constants (PARAMETER) can/should be initialized in the declaration block:

```
REAL, PARAMETER :: CLIGHT = 2.9979E10
```

- Via the DATA instruction variables can be **initialized** in the declaration block. Note that this is done at **compile time!**

Example - declaration

Correct:

```
REAL R, S, PI, E
DATA PI, E /3.141,2.718/
INTEGER I, NMAX
PARAMETER (NMAX=100)
REAL, PARAMETER :: k_b = 1.38E-23
CHARACTER LINE*132
```

Incorrect:

```
REAL X
X = 3.0 ! not permitted:
        ! initialization via =
        ! in the declaration block
INTEGER Y
```

- completely analogous to `int` in C, e.g.,

```
INTEGER I, M
```

```
I = 0
```

```
M = 1
```

- integers have a sign (*signed*):

```
J = -1000
```

```
K = +200
```

the + is optional

- if not declared otherwise (see below) each `INTEGER` occupies 32 Bit (=4 Byte) in the memory
- therefore: largest integer (32 Bit, two's complement): $2^{31} - 1 = 2\,147\,483\,647$
(smallest: $-(2^{31})$)

- INTEGER of different size than 32 Bit with help of the KIND parameter, e.g.,

```
INTEGER(KIND=8) trillion  
INTEGER*1 c128
```

- KIND means usually the size in Byte (= 8 Bit), so KIND=8 may correspond to 64 Bit
- the default value (if KIND is not used) is KIND=4 (32 Bit)
- all INTEGER of a source code can be set automatically to 64 Bit with the compiler option:

```
gfortran: -fdefault-integer-8  
ifort: -i8
```

- **Warning:** The Fortran standard does not guarantee that KIND corresponds always to Bytes (so KIND=4 for 32 Bit), although most Compilers support this correspondance.

Use of integers:

- for loops: `DO I = 1, 10`
- indices (*subscripts*) of *arrays*: `A(I)`

On the importance of integer arithmetic:

- Usually, integers can be faster processed than floats, as there is, e.g., no *normalization* necessary. Also, operations with integers often require a smaller number of bits.
- Bit patterns of integers are stable. Integer can be represented exactly and can be compared unambiguously.
- With help of integers it is possible to implement a *fixed* point number arithmetic without round-off errors (e.g., GnuCash).

Floating point numbers I

REAL, DOUBLE PRECISION

- Floating point numbers are an *approximative* representation of real numbers.
- Floating point numbers can be declared explicit via

```
REAL radius, pi, euler, x, y  
DOUBLE PRECISION rbb, z
```

(analogously to float or double in C).

- valid assignments are

```
x = 3.0  
y = 1.1E-3  
z = 4.0D-266  
rbb = 2.06798E-300_8
```

The last two assignments refer to DOUBLE PRECISION: D instead of E or subsequent `_8`

Floating point numbers II

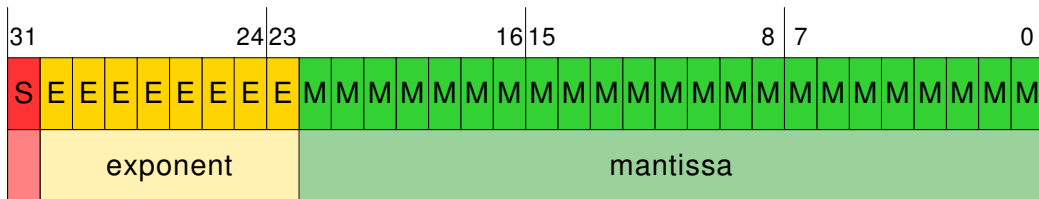
- Representation and usage of floating point numbers is, e.g., regulated by the standard IEEE 754.

$$x = s \cdot m \cdot b^e \quad (1)$$

where $b = 2$ is the basis (IBM Power6: also $b = 10$)

For 32 Bit (single precision):

bits



sign

therefore for single precision (32 Bit):

$$-126 \leq e \leq 127 \rightarrow \max. \approx 10^{38} (= 10^{127 * \log 2})$$

$$\text{decimals: } 7-8 (= \log 2^{23})$$

- analogously for 64 Bit – DOUBLE PRECISION or KIND=8:
exponent: 11 Bit (r), mantissa: 52 Bit
 $-1022 \leq e \leq 1023 \rightarrow \text{max.} \approx 10^{308} (= 10^{1023 \cdot \log 2})$
decimals: 15-16 ($= \log 2^{52}$)

For the representation:

- sign bit: positive = 0, negative = 1
- exponent e results from non negative number E via $e = E - B$ where $B = 2^{r-1} - 1$, e.g.,
 $B = 2^{11-1} - 1 = 1023$
- mantissa is set by *normalization* to the format (example)
 1.0100100×2^4
i.e. with 1 before the point. This 1 is not saved.

Notes about floats (IEEE 754):

- precise saving of integers with 6 up to 9 digits for 32 Bit floats, etc.
- there are positive and negative infinities, e.g., $-1./0. = -\text{Infinity}$
- there exist two zeros: -0 and $+0$, equal for comparison, but, e.g., $1/\pm 0 = \pm\infty$ (different for comparison)
- subnormal (denormal) numbers: fill underflow gap around zero by allowing for leading zeros in mantissa (hence larger exponent)
- NaN: not a number, result of, e.g., $\infty * 0, 0/0, \sqrt{-1}$, indicated by all exponent bits set to 1 and some non-zero number in mantissa

Important: NaNs can propagate through calculations (NaN + 1 = NaN, but NaN⁰ = 1)

value	32 bit pattern (sign, exponent, mantissa)
0.	0 00000000 000000000000000000000000
-0.	1 00000000 000000000000000000000000
inf	0 11111111 000000000000000000000000
-inf	1 11111111 000000000000000000000000
NaN	1 11111111 100000000000000000000000

On the KINDs of floats:

- default for REAL: KIND=4 usually corresponds to 32 Bit
- default for DOUBLE PRECISION: KIND=8 corresponds usually to 64 Bit
- all REAL declarations of a source code can be automatically set to 64 Bit by the compiler option:

```
gfortran: -fdefault-real-8
ifort: -r8
```

this option is usually used together with the analogous integer option (-i8).

Caution: gfortran then sets DOUBLE PRECISION declarations to 16 Byte (=128 Bit)!

- **Warning:** The Fortran standard does not guarantee that KIND corresponds always to Bytes (so KIND=4 for 32 Bit), although most Compilers support this correspondance.

Trade-off between speed and accuracy

Often, accuracy and speed are conflictive goals.

E.g., the use of 128 Bit floats slows down the code significantly, while switching back from 64 Bit to 32 Bit especially for vector operations (SIMD^a, SSE^b, AVX^c) can increase the computation speed.

^aSingle Instruction Multiple Data

^bSIMD Streaming Extension, 128 Bit

^cAdvanced Vector Extensions, 256 Bit

Importance of floating point numbers:

- essential for most scientific calculations, often 64 Bit precision is required

Problems:

- not all rational numbers can be represented exactly (as for, e.g., $1/3$ in decimal system)
- bit patterns not “stable”, e.g. because of denormalization for small numbers
- accuracy of representation depends on the KIND, e.g.
KIND=10 for FPU (floating point unit), but only KIND=8 when saved and for SSE instructions

Warning

Under no circumstances, it is OK to test blindly two floats on identity, instead one should always use a range of accuracy:

`ABS(X - Y) .LE. EPS` *instead of* `X .EQ. Y`

COMPLEX Z, C

- intrinsic data type(!), always occupies two REAL memory cells
- access and assignment of complex variables:

```
REAL REALPART, IMAGINARYPART
COMPLEX Z
Z = ( 2.0 , 1.0 )
Z = CMPLX( 2.0 , 1.0 )
REALPART = REAL( Z )
IMAGINARYPART = AIMAG( Z )
```

- **Trick:** The access on the components of complex variables is also possible with help of EQUIVALENCE:

```
COMPLEX Z
REAL ZR(2), REALPART, IMAGINARYPART
EQUIVALENCE (ZR,Z)
...
REALPART = ZR(1)
IMAGINARYPART = ZR(2)
```


LOGICAL BTEST

- possible values: `.TRUE.` or `.FALSE.`
- example: `BTEST = .FALSE.`
- although only 1 Bit required, 32 Bit are used by default for storage
- default value of most compilers if not initialized:
`.FALSE.`
- recommendation: LOGICAL variable names should start with a B (for boolean)

Computing with LOGICAL

- Variables of type LOGICAL can be combined such that the result is again of type LOGICAL (take care of the ordering, if necessary use parentheses):

```
BBOTH = BONE .AND. BTWO
```

```
BALSO = ( .NOT. BONE ) .OR. BTWO
```

operator	meaning	C/C++
.AND.	logical AND (\wedge)	&&
.OR.	logical OR (\vee)	
.NOT.	logical NOT (\bar{B})	!
.XOR., .NEQV.	exclusive OR ($\dot{\vee}$)	!=
.EQV.	logical EQUIVALENCE (true if both true or both false)	==

- LOGICALS are also the result of comparisons:

```
BTEST = X .GT. Y
```

```
BCHECK = .NOT. ( I .EQ. J )
```

Fortran	meaning	C/C++/Fortran
.LT.	< strictly lesser than	<
.LE.	≤ lesser/equal	<=
.EQ.	= equal	==
.NE.	≠ not equal	/= (in C: !=)
.GE.	≥ greater/equal	>=
.GT.	> strictly greater than	>

Precedence of operators

category	operator	precedence
numeric	**	highest
numeric	*, /	.
numeric	+, -	.
relational	.EQ., .NE., .LT., .LE., .GT., .GE.	.
logical	.NOT.	.
logical	.AND.	.
logical	.OR.	.
logical	.XOR., .EQV., .NEQV.	lowest

→ a higher priority can be always enforced via parentheses ()

→ the evaluation of composed expressions can be optimized by the compiler, e.g., as in

IF (X .NE. 0. .AND. 1./X .GT. 2.) → no further evaluation if $X = 0$.

CHARACTER A, LINE*80

- Variables of type CHARACTER contain a single *ASCII* character (default), therefore KIND=1 (8 Bit)
- With the following *modifier* it is possible to create strings, i.e. “character arrays” (here: 80 characters long):

```
CHARACTER LINE*80
CHARACTER*80 LINE2
CHARACTER (LEN=80) CARD
CHARACTER(80) TEXT
```

- assignment to CHARACTER constants via quotation marks:

```
LINE="Hello world!"
TEXT='Good bye!'
```

- The access on substrings is possible with help of round parentheses ():

```
TEXT='Good bye!'
```

```
WRITE (*,*) TEXT(:2), TEXT(3:6) , TEXT(7:)
```

- important for the comparison of test strings:
upper and lower case are distinguished,
trailing blanks are ignored

```
BTEXT = TEXT .EQ. 'Good bye!   '
```

Comparison: Intrinsic data types in C/C++ and Fortran

Fortran	C/C++	comment
INTEGER	int	32 bit
REAL	float	32 bit
DOUBLE PRECISION	double	64 bit
COMPLEX	—	32 bit
LOGICAL	bool	
CHARACTER	char	
CHARACTER*80	char[81]	Fortran string not null terminated

We already know:

- automatic formatting:

```
(* , *)
```

- text formatting:

```
(* , '(A) ' ) , (* , '(A80) ' )
```

- fixed point formatting:

```
(* , "(F20.0 , F8.2)" )
```

- exponential formatting:

```
(* , "(E5.2 , E10.1E2)" )
```


FORMAT statement: Integer, Logical, and Complex

INTEGER

- formatting via (In), n = number of digits, e.g., I3

LOGICAL

- format indicator: L or Lw
e.g., WRITE (*,"(L)") BTEST
or WRITE (*,"(L3)") BCHECK
- output: T or F (depends on value), or $\Delta\Delta T$ (if (L3))
- for input: only the first (non-blank) character is required: .T, T, .t, t or .F, F, .f, f
(the format with the dot should be avoided in the shell)

COMPLEX

- output as for REAL, components separated by line break, e.g, for $z = 1 + 2i$:

```
PRINT *, Z (1.000000,2.000000)
PRINT "(F4.1)", Z 1.0
                  2.0
```

More output formatting:

- formatting of arbitrary number data types via G, analogously to E: (G10.2E3)
- output of bit patterns: e.g., (B64) for 64 Bit

Format modifiers:

- repetition by preceding number of repetitions, hence: (4F5.2)
- blanks via X, e.g.: (5X,A,X)
- scaling – moving the decimal point per P: (2PE8.2E1) moves the decimal point by two digits to the right
- no line break via \$: (A,\$)

There is i.g. no type checking (at compile time) in Fortran and data types are often not automatically converted. Therefore, this must be done by the developer, e.g., the conversion from INTEGER to REAL

```
X = REAL(N) ! argument can be INTEGER, REAL, COMPLEX  
Y = FLOAT(M) ! argument is INTEGER
```

Moreover, we have already learnt a method to convert even CHARACTER (strings) to INTEGER or REAL:

```
READ (*,"(F20.0)") X
```

reads from STDIN ("string") and converts to REAL

Hint: the asterisk * can be replaced by a variable name:

```
READ (LINE,"(I10)") M
```

The other way around (INTEGER, REAL \rightarrow CHARACTER) is done by WRITE.

```
WRITE (LINE, "(I5)") NUMBER
```

Already known.

The one-liner:

```
IF (EXPRESSION) instruction
```

- EXPRESSION is a logical expression (LOGICAL) with the values `.TRUE.` or `.FALSE.`
- often: comparison (operators) and combinations (sentential connectives), e.g.,

```
IF ( XL .GT. 1. .AND. XL .LT. 200.) GOTO 100
```

The extensive standard form:

```
IF (EXPR) THEN  
    ... instruction ...  
  
ELSE IF (EXPR2) THEN  
    ... other instruction ...  
  
ELSE  
    ... alternative ...  
  
ENDIF
```

Instead of lengthy IF-THEN-ELSEIF blocks it may be more convenient to use:

```
[name:] SELECT CASE (variable)
  CASE (value1)
    ...
  CASE (value2)
    ...
  CASE DEFAULT
    ...
END SELECT name
```

where variable must be of type

- INTEGER
- LOGICAL
- CHARACTER

and value can also be a range:

- min:max
- min:
- :max

Loops are among the most powerful and useful structures in programs.
In Fortran loops begin with the statement:

DO

In principle there are the following types of (DO-) loops:

- “count” loops
- WHILE loops
- “infinite” loops



infinite loops have an empty loop header (no break condition):

```
DO  
  
    ... instruction block ...  
  
END DO
```

Leaving the loop is therefore only possible by an instruction like
IF (BEND) EXIT

EXIT

immediately terminates the current iteration and exits the current(!) loop

Example

```
DO
  K = K + 1
  IF ( K .EQ. 10 ) EXIT
ENDDO
```

We remember:

Blanks

Blanks are optional in Fortran. So they are also optional in END DO, DOUBLE PRECISION, ELSE IF, GO TO, ...

When entering the count loop the loop header is evaluated:

```
DO K = 1 , NMAX , 1  
  ...  
ENDDO
```

- the first argument initializes the loop counter (always!)
- the second argument defines the maximum value of the loop variable

Note: If the loop is exited normally, the loop variable K has the value $NMAX + 1$.

- the third (optional) argument is the increment (or decrement)

Fortran 90:

- loop variables can be *also* of type REAL.

Fortran 95:

- loop variables can only be of type INTEGER.

General loop control

- EXIT terminates the current(!) loop immediately
- CYCLE skips the rest of the current iteration(!) and jumps to the beginning of the next iteration
- CONTINUE does nothing, useful for loops with labels

Loop with label (there was no ENDDO in FORTRAN77):

```
      DO 101 , K = 3 , 10
          ...
101  CONTINUE
```

The WHILE loop checks *before* each iteration for a logical expression:

WHILE

```
DO WHILE (condition)
  ...
END DO
```

- bad: the condition is changed *in* the loop body, but the condition is checked not before the next iteration
- because of performance issues it is recommended to avoid (extensive) WHILE loops

Branching

Fortran

```
IF (X .GT. Y) THEN
```

```
  ...
```

```
END IF
```

C/C++

```
if (x > y) {
```

```
  ...
```

```
}
```

For loop

Fortran

```
DO I = 1, 10
```

```
  ...
```

```
END DO
```

C/C++

```
for (int i = 0; i < 10; ++i) {
```

```
  ...
```

```
}
```

Open a file

```
OPEN (UNIT, FILE="name", ACTION="access", ERR=label)
```

- opens a file with name “name” (similar syntax as in IDL),
- assigns a device number `UNIT` of type `INTEGER` to the file; the only obligatory argument,
- defines the way of access: `READ`, `WRITE`, or `READWRITE` (default),
- and jumps in the case of an error during opening to the label `label` (e.g., file not found).

Example

```
OPEN (22, FILE="output.dat", ERR=999)
```

→ opens the file `output.dat`, jumps to label 999 in the case of an error

The counterpart:

Close a file

```
CLOSE (UNIT,ERR=label)
```

where ERR=label is optional

Example

```
CLOSE (22)
```

Note: For performance issues it may happen that write actions on the file will be postponed until the CLOSE statement (buffering).

Read from an opened file *per line*

```
CHARACTER line*132  
READ (22,"(A132)") line
```

→ read in a text string

or formatted reading into corresponding variables:

```
      READ (22,"(I10,1X,F10.2)", END=21) n , X  
      ...  
21    CLOSE(22)
```

→ if the last line of the file is reached: jump to label 21

Or unformatted reading:

```
READ (22,*,END=11) n , X
```

Write to an opened file, e.g.,

```
WRITE (42,*) "file header:"
```

Or formatted:

```
WRITE (42,"(A,G12.3)") " x = " , x
```

Programming in Fortran - Part 3

Arrays

There are many ways to allocate arrays, i.e., to reserve memory.

Most efficient and stable, the *static* allocation (on stack):

- Size and shape of the array must be determined at/before compile time in the program (but shape can be changed in procedures)
- The process of allocation is done when the program is loaded.
- Allocated memory is only freed at the end of the program.

Recommended and unambiguous is the use of

`DIMENSION` array (array specification)

This instruction can stand alone or as a so-called attribute.

Example: allocation of a “vector”

```
DIMENSION POSITION(3)
```

- The name of the array, POSITION, implies the data type REAL.
- The array consists of 3 elements and has the dimension (rank) 1

Example: allocation of a “matrix”

```
DIMENSION ROTATIONALMATRIX(3,3)
```

- The array has the dimension (rank) 2, the shape (3,3) and the size $3 \times 3 = 9$.

DIMENSION can also be used as an attribute:

```
REAL(8), DIMENSION(3) :: x, v
```

Data type and attribute(s) are separated from the list of variables by *double colons*.
Moreover:

- The argument of DIMENSION must be a (integer) constant, e.g.,

```
INTEGER, PARAMETER :: NMAX=100  
REAL, DIMENSION(NMAX,NMAX) :: NLINE
```

- The maximum dimension is 7. E.g.,

```
INTEGER, DIMENSION (1,2,3,4,5,6,7) :: rank7
```

```
DIMENSION M(100,100)
```

- arrays can be accessed as whole or per element:

```
M = 0
```

```
M(1,1) = 1
```

- arrays can also be accessed with help of ranges:

```
M(2:100,100) = 1
```

- the use of ranges (bounds) offers the interesting possibility to set negative indices:

```
DIMENSION M(-50:49,100)
```

Since Fortran 90 exists the possibility to set the size (but not the rank) of an array at runtime. Such arrays must be marked as allocatable:

Dynamic Allocation

```
INTEGER, ALLOCATABLE, DIMENSION(:, :) :: M  
...  
READ (*, "(I1000)") I  
ALLOCATE (M(I, I), STAT=ERR)
```

The `ALLOCATE` statement reserves the corresponding amount of memory (dynamic memory, heap), but does not initialize it. If a `STAT` variable is given, this variable is set to 0 in case of successful allocation. If no `STAT` variable is given, the program stops in case of error.

Deallocation – Freeing dynamically allocated arrays

```
DEALLOCATE (M, STAT=ERR)
```


Functions are a type of *procedures* in Fortran. Functions always return a result of a distinct data type:

FUNCTION call

```
Result = NAME (arg1, ... , argN)
```

The definition of a the function is either after the END statement of the PROGRAM or in a separate source file:

FUNCTION definition

```
FUNCTION NAME (arg1, ... , argN)  
  ...  
  NAME = ...  
END
```

Usually the result (returned value) is assigned to the variable of the same name as the function within the function body.

A simple example:

```
PROGRAM cylinder_volume
  IMPLICIT NONE
  REAL r, h, volume
  READ (*,*) r, h
  WRITE (*,*) volume(r,h)
  STOP
  END

  FUNCTION volume(radius,height)
  IMPLICIT NONE
  REAL radius, height, volume
  volume = 3.141 * radius**2 * height
  END
```

The type of the result must be declared if `IMPLICIT NONE` is used.
Arguments are identified via their order in the list of arguments.

Attributes for the declaration of functions:

- Fortran compilers know different builtin (*intrinsic*) functions (e.g. SIN(X)). If these intrinsics shall be overwritten (defining functions of the same name), the function must be declared in the calling procedure as EXTERNAL:

```
REAL, EXTERNAL :: COS
```

- Functions can be *directly* called recursively. In this case a different name for the result must be declared:

```
RECURSIVE FUNCTION FAC(N) RESULT(L)
```

```
...
```

```
L = FAC(N-3)
```

- The type of the result can also be given in the function declaration instead of the declaration block:

```
REAL FUNCTION volume(radius, height)
  IMPLICIT NONE
  REAL radius, height
```

- The arguments of functions are passed by address. Hence they can be manipulated within the function!

```
FUNCTION volume(r,h)
  ...
  r = 42.
```

Different from passing arguments by value in *C/C++* the altered argument is also changed in the calling procedure.

Note

In Fortran the calling and called procedure work on the same memory area, which is specified in the list of arguments.

To avoid side effects like this, a function can be declared as PURE or ELEMENTAL, this requires a declaration of arguments to be INTENT(IN):

```
PURE REAL FUNCTION volumen(r,h)
  INTENT(IN) :: r, h
  ...
  r = 42. ! forbidden
```

In addition to the `FUNCTION` the `SUBROUTINE` is another important procedure. It is some kind of a “subprogram” and has, different from functions, no result ($\hat{=}$ `void` in C/C++).

SUBROUTINE call

```
CALL rname (arg1, ... , argN)
```

Like for the `FUNCTION`, the `SUBROUTINE` is defined outside the calling procedure.

SUBROUTINE definition

```
SUBROUTINE rnam (arg1,...,argN)  
...  
END
```

The `SUBROUTINE` and the calling procedure share the same memory area, which is defined by the list of arguments.

Although the SUBROUTINE has no result, it is possible to distinguish between input and output arguments:

INTENT

```
SUBROUTINE rname (arg1,arg2,arg3,arg4)
  INTEGER, INTENT(IN) :: arg3, arg4
  REAL, INTENT(OUT) :: arg1
  INTEGER, INTENT(INOUT) :: arg2
  ...
END SUBROUTINE
```

Arguments that are declared as `INTENT(IN)` cannot be changed within the subroutine. `INTENT(OUT)` arguments shouldn't be defined when entering the subroutine. These restrictions do not apply to variables that are `INTENT(INOUT)`.

The attribute PURE

```
PURE FUNCTION myfunc(...)  
PURE SUBROUTINE mysubr(...)
```

allows to use procedures without side effects. Only arguments marked as `INTENT(OUT)` can be changed; arguments of `PURE FUNCTION`s cannot be changed.

Intrinsic functions, e.g., `ABS()` are always `PURE`.

- Note: While FUNCTIONS are rather used for short procedures, SUBROUTINES often contain complex subprograms.
- Both types of procedures can also be called via an ENTRY point:

ENTRY

```
SUBROUTINE tue123 (arg1,arg2)
  ...
  ENTRY tue23 (arg2)
```

ENTRY statements cannot stay in IF or DO blocks.

Style

Avoid ENTRY points as they reduce the readability of the source code.

Arguments of procedures:

- literal constants:

```
Result = MYFUNC(3.0, .FALSE.)
```

- scalar variables: $z = x * \text{COS}(\text{alpha})$
- procedures, especially functions (later more ...)
- arrays: as a whole, single elements or ranges

```
CALL MYROUTINE ( A, B(1,1), C(1:2) )
```

Arrays must be dimensioned (again) in the called procedure:

```
DIMENSION A(3,2) , C(2)
```

- especially for CHARACTER arrays the asterisk is recommended (assumed length):

```
CHARACTER(LEN=*) TEXT
```

The program and other procedures can be assembled in a single source file.

- This is recommended for short programs and functions (e.g., many of our exercises).

For more complex programs:

- one procedure per file, which has the same name as the procedure it contains
- compiling and linking:

```
gfortran -c myfunc.f  
gfortran -o myprogram.exe myprogram.f myfunc.o
```

- option `-c` generates a (not linked) object file `myfunc.o`.
- object files can be assembled in libraries:

The archive tools `ar` & `ranlib`

```
ar rv libmylibrary.a myfunc.o
ranlib libmylibrary.a
```

`r` replace or append a file;
 if archive does not exist, it is created

`v` verbose

`ranlib` renew the “table of contents”

libraries which have been created in this way can be linked in at compile time:

```
gfortran -L$MY_PATH -lmylibrary myprogram.f
```

`-L` specifies the path for the library, `-l` the name of the library, which always starts with `lib`. This prefix is omitted for the call.

There are the following restrictions in Fortran for names of variables and procedures:

- the name must start with an ASCII-*letter* (ifort: also \$)
- the name must not be longer than 31 characters (ifort: 63)
- except for letters also allowed are: numbers, underscores `_`, and the dollar symbol \$ (gfortran: `-fdollar-ok` required)

Supported names

```
Radius123
```

```
Begin$_
```

Invalid names

```
3pel      ! does not start with a letter or $
```

```
boundary-condition.2 ! invalid characters - .
```

Supported characters

Fortran supports the ASCII character set:

- letters (uppercase/lowercase), numbers
- + the usual special characters:

	blank/TAB	:	colon
=	equality sign	!	exclamation mark
+	plus	"	double quotation mark
-	minus	%	per cent
*	asterisk	&	ampersand
	slash	;	semicolon
()	parentheses	<	lesser than
.	point	>	greater than
,	comma	?	question mark
'	single quotation mark	\$	Dollar symbol

+ some symbols which are only printable (e.g., curly braces)

- Fortran 90/95 allows up to 132 characters per line (arbitrary in free form)
- continuation lines: up to 19 (ifort: 511)

gfortran

option	usual arguments
<code>-ffixed-line-length-n</code>	0, 72 (default), 80, 132, none (=0)
<code>-ffree-line-length-n</code>	0, 132 (default), none (=0)

ifort

option	usual arguments	alt. options
<code>-extend-source [n]</code>	72, 80, 132 (default)	-72, -80, -132
<code>-free</code>	– (no line limit)	

Some intrinsic functions and subroutines I

- random numbers

```
result = RAND(I)
```

returns uniformly distributed random numbers of a sequence ($I = 0$); Restart if argument different from 0



- date and time

```
CALL ITIME (ITARRAY)  
CALL IDATE (IDARRAY)
```

fills ITARRAY(3) with hour, minute, second;
fills IDARRAY(3) with day, month, year

Some intrinsic functions and subroutines II

- representable numbers

```
Result = HUGE(I) ! also for integers
```

```
Result = TINY(X)
```

```
Result = EPSILON(X)
```

returns the largest or smallest representable number, respectively, of the same type as the argument X . `EPSILON` gives the smallest representable number E , such that $1 + E > 1$ (minimal step width for floats).

- command line arguments for program call

```
Result = IARGC()
```

```
CALL GETARG(N,TEXT)
```

returns the number of command line arguments; fills the CHARACTER variable `TEXT` with the N th command line argument

- `CALL SYSTEM (TEXT)`: starts a *shell* in which the command(s) stored in `TEXT` are executed, e.g.,

```
CALL SYSTEM("cp scratch.ps wrplot1.ps")
```

Note that one cannot read from `STDOUT` with `SYSTEM`, i.e. `CALL SYSTEM ("ls")` is useless

- `LEN (TEXT)` returns the length of the string `TEXT`
- `TRIM (TEXT)` removes trailing blanks from string `TEXT`

The graphical library PGPLOT can be called from C or Fortran programs. This must be declared at link time:

Linker call

```
-lX11 -L$PGPLOT_DIR -lpgplot
```

where the environment variable `$PGPLOT_DIR` must contain the (absolute) path
The following environment variables should therefore be set:

PGPLOT environment (bash)

```
export PGPLOT_DIR=${HOME}/PGPLOT/  
export LD_LIBRARY_PATH=${PGPLOT_DIR}  
export PATH=${PATH}:${PGPLOT_DIR}  
export PGPLOT_DEV="/XWIN"
```

The most important procedures of PGPLOT:

- PGBEG

```
IER = PGBEG(0,"/XWIN",1,1)  
IF (IER .NE. 1) STOP "PGPLOT failed"
```

a function(!) that starts PGPLOT (server). The second argument is for the output device; this can be, e.g., `"/XWIN"`, also `"?"` which implies that the user is asked for a device at run time.

In the case of success the function returns '1'.

- PGENV

```
CALL PGENV (XMIN,XMAX,YMIN,YMAX,JUST,AXIS)
```

defines the drawing area; for `JUST = 1` the x- and y-axis are scaled equally; `AXIS=1` also plots the coordinate axes and labels

- PGLAB

```
CALL PGLAB("x","y","title")
```

defines the labels of the x- and y-axis and of the plot

- PGPT

```
CALL PGPT(1,X,Y,-1)
```

draws a single point (1st argument) with given X- and Y-coordinate with the size given in pixel (last argument)

- PGEND

```
CALL PGEND
```

Closes the plot correctly.

- PGSCI

```
CALL PGSCI(INDEX)
```

Set color index for the following drawing actions.

- PGSCR

```
CALL PGSCR(INDEX,R,G,B)
```

Set the color for the given color index:

INDEX=0 is background, INDEX=1 the foreground (default: white on black). R, G, B are the color fractions red, green, blue, each in the range 0 to 1

Programming in Fortran - Part 4

Variables and passing variables

We already know:

- functions and subroutines get a list of arguments (usually names of variables) when called

```
CALL MY_SUB (arg1, arg2)
```

- internally: only start addresses are passed, i.e. correct dimensioning in calling and called procedures necessary:

```
SUBROUTINE calcE (velo, pos)
  REAL :: velo(2), pos(2)
```


- the list of arguments for a declared/defined procedure and for its call must (normally) be matching

```
SUBROUTINE my_sub (arg1, arg2, arg3)
```

```
...
```

```
CALL my_sub (x, y, z)
```

however, the last argument(s) can be omitted (not checked by compiler):

```
SUBROUTINE my_sub (arg1, arg2, arg3)
```

```
...
```

```
CALL my_sub (x)
```

→ not recommended, bad style!

- there is (usually) no type checking (exception: intrinsics, procedures in same source file) and no automatic type conversion, e.g.,

```
CALL PGPT(1.,0_D,0,1.)
```

might not give the expected result ...

- as a preceding declaration of procedures is not necessary (cf. include files in *C/C++*) the compiler can usually not check the matching of argument lists

Solution:

- To make the argument list verifiable by the compiler an explicit `INTERFACE` block can be included in the *calling* procedure.
- This contains exact copies of the declaration block of the called procedure, including the header, declaration of arguments, and the `END` statement.

INTERFACE

```
...  
INTERFACE  
  
  SUBROUTINE my_sub(arg1, arg2)  
    REAL :: arg1  
    INTEGER :: arg2  
  END SUBROUTINE my_sub  
  
  FUNCTION my_func(N)  
    INTEGER :: N  
  RETURN  
  
END INTERFACE  
...
```

Rules for (explicit) INTERFACE blocks:

- the INTERFACE block is part of the declaration section must therefore appear *before* any instruction
- each procedure can only have one interface per calling procedure
- an explicit interface is required, if one or more arguments have the following attributes:
 - ALLOCATABLE
 - OPTIONAL (see below)
 - reference: POINTER
 - object attribute: TARGET, VOLATILE
- an explicit interface is also required for functions whose result is an array or a pointer

The OPTIONAL attribute

- arguments which are marked with the OPTIONAL attribute can be omitted for the call of the procedure, if they:
 - are the last ones in the list of arguments
 - or all following arguments of this list are passed by a *keyword*

```
INTERFACE
  SUBROUTINE calcT (a, x, h)
    OPTIONAL x,h
  END
END INTERFACE
...
CALL calcT(a, h=r)
```

- the *keyword* must be identical to the name of the dummy argument in the interface

The PRESENT function:

- the presence of *optional* arguments while passing them can be determined with help of the function

```
PRESENT(arg)
```

E.g.,

```
SUBROUTINE calcE (velo, pos)
  REAL, OPTIONAL, DIMENSION(2) :: pos
  ...
  IF ( PRESENT(pos) ) THEN
    r = SQRT( pos(1) * pos(1) + pos(2) * pos(2) )
    Eg = - GM / r
  ENDIF
```

obvious: PRESENT has result of type LOGICAL

Volatility of variables:

```
SUBROUTINE output(pos,t,fpos)
  REAL :: pos(2), fpos(2), t
  LOGICAL bset
  ...
  bset = .TRUE.
```

the used LOGICAL is a *local* variable, which is by default *automatic*, i.e. it is removed from memory when leaving the the procedure

→ when re-entering the procedure output the value of bset is (usually) deleted

Solution:

SAVE

variables can be made non-automatic by the SAVE attribute or by the SAVE declaration statement (corresponds to static in C/C++). Then, the content of the variable is saved for the re-entry of the procedure.

e.g.

```
SUBROUTINE output(pos,t,fpos)
  REAL :: pos(2), fpos(2), t
  LOGICAL, SAVE :: bset
  SAVE fpos
```

- Moreover, the compiler can convert any automatic (local) variable to a static (SAVE) variable (different from C++):

```
gfortran: -fno-automatic  
ifort: -save
```

But, the use of local variables without SAVE attribute for buffering may cause segmentations faults which are hard to track (randomness).

- Fortran procedures usually communicate by passing variables as arguments at call
- Alternatively, a common memory area can be created, the so called COMMON block(s), corresponding to global variables in C/C++:

COMMON

```
COMMON /velpar/ vfinal, vpar2  
COMMON // buffer(1000)
```

where the name of the block (in //) is optional → *blank* Common block

- all procedures that declare a common block with its variables have shared(!) access to the corresponding memory area

- the partitioning of the shared memory does not need to be the same for all procedures, e.g.,

```
COMMON /coords/ x, y, z, i(10)
```

and an alternative declaration

```
COMMON /coords/ r, p, k(11)
```

where $i(1)$ then is the same as $k(2)$ and so on.

The use of COMMON blocks is generally not recommended.

Programming in Fortran - Part 5

Overloading, Modules

Overloading

- if the interface is given a name, the corresponding procedure can be overloaded, i.e. the compiler chooses the matching procedure by the type of arguments:

```
INTERFACE mysub
  SUBROUTINE mysub1 (n,...)
    INTEGER :: n
  END SUBROUTINE mysub1
  SUBROUTINE mysub2 (x,...)
    REAL :: x
  END SUBROUTINE mysub2
END INTERFACE
```

- and call via

```
CALL mysub(arg,...)
```

Overloading/extending operators

- an interface can also overwrite an operator:

```
INTERFACE OPERATOR(op)
  FUNCTION myop (arg1,arg2)
    type, INTENT(IN) :: arg1, arg2
  END INTERFACE
```

- in this case only *functions* with one (unary) or two (binary) *non optional* arguments with INTENT(IN) can appear
- op is, e.g.,: + or - or .myop.
- if an intrinsic operator (e.g., .LE.) is extended, also the alternative notation (here: <=) is affected and the number of arguments must be the same as for the intrinsic form

Extending assignments:

- there is only one assignment operator in Fortran: =
- this operator can only be extended by a SUBROUTINE with two arguments in the following way:

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE myassign (arg1,arg2)
    type, INTENT(OUT) :: arg1
    type, INTENT(IN)  :: arg2
  END SUBROUTINE
END INTERFACE
```

- e.g., $x = n \rightarrow$ *type casting*

Unambiguousness:

- When overloading it must be clear from the list of arguments (type, number of arguments, name) what to choose:

Wrong:

```
INTERFACE f
  FUNCTION fxi (x,i)
  ...
  FUNCTION fix (i,x)
  ...
END INTERFACE f
```

- In the case of ambiguities while extending intrinsic procedures the non-intrinsic procedure is chosen.

Header files

- with help of an explicit `INTERFACE` the compiler gets informed about the data types of the called procedures in the calling procedure
- more convenient: put `INTERFACE` in an extra file, e.g., `file.h` and include it via

```
INCLUDE "file.h"
```

- `INCLUDE` can appear anywhere in the source code
- `INCLUDE` requires as argument a text string which contains the name of a Fortran source file, which is included at this very position
- i.e. *header* files can be created similar to `C++`

Include a self-made header file:

Example

calcE.h :

```
INTERFACE
  SUBROUTINE calcE (vel, pos, E)
    REAL :: vel(2), pos(2), E
  END SUBROUTINE calcE
END INTERFACE
```

kepler.f :

```
PROGRAM kepler
  INCLUDE "calcE.h"
  ...
```

Better than separated files with interfaces and procedures:

→ [modules](#)

- contain variables and procedures, which can be used by other procedures
- procedures that shall access a module **must** import this module through

```
USE modulename
```

Structure of a module

```
MODULE name  
  ... declarations ...  
CONTAINS  
  ... procedures ...  
END MODULE name
```

Example

```
MODULE energies
  REAL, PARAMETER :: PI = 3.141519
CONTAINS
  SUBROUTINE calcE (E, vel, pos)
    ...
  END SUBROUTINE calcE
  FUNCTION Egrav (pos)
    ...
  RETURN
END MODULE energies
```

CONTAINS

- the CONTAINS statement allows the definition of functions or subroutines within another procedure (e.g., in PROGRAM) and appears at the end of the definition of the procedure before END
- the procedures that follow a CONTAINS statement are referred to as *internal subprograms* and must *not* contain another CONTAINS (no nesting)
- ENTRY points must appear before CONTAINS, internal subprograms must not contain an ENTRY point
- internal subprograms have access to all names declared in the *host* procedure (e.g., variables), the internal subprogram has an explicit interface
- the host procedure can call the internal subprogram, as well as the internal subprogram can call itself

Compiling modules:

- the module is compiled via

```
gfortan -c module.f
```

the result are two(!) files:

```
module.o  module.mod
```

- .o object file: the usual machine-readable instructions
- .mod file: ASCII file with interface instructions for the compiler, i.e., the functionality is described that is provide to the calling procedure via

```
USE module
```

(explicit interface, see below)

→ correspond to header files in *C++*

- compiled modules are in general not compatible for different compilers, but must exist as source code and re-compiled for each compiler
- as MODULEs are already a kind of an explicit interface, the procedures in them cannot be overloaded by a named interface (see above)
→ instead – preferably in the MODULE:

```
INTERFACE mysub  
  MODULE PROCEDURE mysub1, mysub2  
END INTERFACE mysub
```


Examples:

The Intel Fortran compiler offers a variety of procedures encapsulated in modules:

- USE IFPORT: e.g., `len = FULLPATHQQ (file,output)` returns the full path of a file ;
`CALL GETENV(variable,content)` returns the content of an environment variable
→ intrinsic in gfortran
- USE IFPOSIX: e.g., `CALL PXMKDIR (name, len, mask, result)` makes a directory
- USE IFCORE: e.g., `bpessed = PEEKCHARQQ ()` detects, if a key is pressed (without pausing program)
- USE OMP_LIB: OpenMP library
- Windows specific: e.g., USE IFLOGM for dialog boxes, etc.

Defining structure, i.e. a collection of variables:

```
TYPE typename
  ... declaration of components ...
END TYPE typename
```

declares a (*derived*) data type with its components, e.g.,

```
TYPE star
  REAL :: radius, mass_i, mass_c
  CHARACTER :: spectraltype*2
END TYPE star
```

- declaration of structures created by that:

```
TYPE (star) :: WR144
TYPE (star) :: sun = star (1.,1.,1.,"G2")
```

- access to the components possible via the % symbol (cf. dot . in C++):

```
xinitialmass = sun%mass_i
```

- for TYPE data types an operation can be defined with help of
INTERFACE OPERATOR (op)
e.g., addition component-by-component

Programming in Fortran - Part 6

Name spaces, scopes, pointer, C

We already know:

```
PROGRAM myexe
  INTEGER :: init, k
  init = 4 ; k = 2
  CALL mysub (k)
  ...
SUBROUTINE mysub (m)
  IMPLICIT NONE
  INTEGER :: m, j
  j = init ! does not work
```

Value of variable `init` in subroutine `mysub` not available, as the variable `init` neither

- has been passed by argument,
- was made visible globally by a `COMMON` block,
- nor is automatically globally visible (e.g., `CONTAINS`, `MODULE`)

In Fortran

- variables, hence “instances” of data types
- program units, like functions, subroutines, modules
- certain structures, e.g., named interfaces

are identified via a *name* by the compiler/linker

This name is only visible within a scoping unit:

- TYPE definition
- INTERFACE block
- program unit (e.g., SUBROUTINE, FUNCTION)

(except for scoping units that are contained in these units) and therefore needs to be unambiguous only within this unit:

Example for scopes

```
MODULE mod1           ! 1
  INTEGER hello1     ! 1
  CONTAINS           ! 1
    SUBROUTINE sub2  ! 2
      TYPE mytype3   ! 3
        REAL :: r,t  ! 3
        INTEGER hello1 ! 3 ok, as not in scope1
      END TYPE mytype3 ! 3
      ...            ! 2
    END SUBROUTINE sub2 ! 2
  END MODULE mod1    ! 1
```

These declared names are only visible within the units where they are declared.

If more than one module is used:

```
USE std_lib  
USE math_lib
```

the problem of *name clashes* can occur, i.e. identical names (variables, procedures) in both modules
Instead of parallel name spaces, as in *C++*, there are two methods to circumvent such problems:
Method I:

```
USE module, rename-list
```

where *rename-list* has the form:

```
USE module, name_in_module => aliasName
```

i.e. the name declared in the module is → replaced by another name

Method II: the export of names from a module can be restricted

```
USE std_module, ONLY: name1, name2
```


PRIVATE and PUBLIC

We already know:

```
MODULE graph_ps
  TYPE point
    REAL :: x, y
  END TYPE point
  REAL :: scale = 400.0
  INTEGER :: graphics_unit = 20
```

- procedures that use module `graph_ps` have access to `point`, `scale`, `graphics_unit`
- useful for usage of data type `point`
- sometimes it is required that names (e.g., variables like `graphics_unit`) are not visible from outside the module

for the *encapsulation* of data the attribute

```
PRIVATE :: var1, ...  
REAL, PRIVATE :: x
```

can be used.

The opposite is

```
PUBLIC :: var1, ...  
INTEGER, PUBLIC :: k
```

These attributes can, used as an instruction, also define a default.

```
MODULE mod1  
PUBLIC
```

Example 1

```
MODULE graph_ps
PUBLIC
  TYPE point
    PRIVATE
      REAL :: x
      REAL, PUBLIC :: y
    END TYPE
  REAL, PRIVATE :: scale
```

- procedures that use this module have access to `point` and `point%y`, but *no access* to `scale` and `point%x`
- members (procedures) of this module have complete access to all variables

Example 2

```
MODULE example
  PRIVATE only_int, only_real
  INTERFACE general
    MODULE PROCEDURE only_int, only_real
  END INTERFACE general
  CONTAINS
    SUBROUTINE only_int (i)
      ...
    SUBROUTINE only_real (x)
      ...
  END MODULE bsp
```

- procedures that use example can only use the named interface general

- with help of `PRIVATE` instruction or attribute it is possible to hide *names* in a module from access by external program units
- → cf. also (same effect)
`USE mod1, ONLY name1`
- in the same way the components of a `TYPE` definition can be hidden from access by external units, so that → only procedures of the `MODULE` have access

The consequent use of `PRIVATE` helps to ensure data integrity (cf. global vs. local variables).

Attributes

- for the declaration of data types (e.g., REAL) or functions (result) attributes (modifiers) can be given, e.g.:

Attributes

ALLOCATABLE, AUTOMATIC, DIMENSION, EXTERNAL, INTENT, OPTIONAL, PARAMETER, PRIVATE, PUBLIC, PURE, SAVE

Example

```
INTEGER, SAVE, DIMENSION(2,2) :: sigma
REAL, EXTERNAL :: COS
...
FUNCTION COS (X)
```

- most attributes can also be given as an *instruction*, defining some default for the data types and procedures declared in a procedure

Example

```
SUBROUTINE mysub (x, y, z)
  SAVE
  REAL :: x, y, z, r
  INTEGER :: k
```

→ all local variables are put to the static memory (instead of dynamic memory/heap), cf. compiler option `-fnoautomatic` (gfortran) or `-save` (ifort)

We already know:

Assignment: memory ↔ variable

principle: compiler allocates (reserves) memory (accessible via memory address) for variables following a certain scheme, e.g., INTEGER occupies 4 byte (32 bit), beginning at the *start address*

variable within the source code accessible via *name*

Pointer:

- stores *addresses* of something, e.g., variables, arrays, functions
- in e.g. C pointers are the only realization of a *reference* (regarding the call behavior)

Pointer in Fortran:

- only restricted pointer methods
- pointer are *always* associated with an “object”:
- either by *allocation* → ALLOCATE
- or with help of an *assignment* => i.e. association with an already existing “object”
- no pointer arithmetics

Pointer allocation

```
REAL, POINTER :: p ! declared, but not allocated
                   ! -> not "existing"
ALLOCATE (p)       ! p allocated, from now on usable
p = 2.7182         ! access as for normal REAL variable
DEALLOCATE (p)    ! free memory
```

not very useful example, clearer for arrays → only number of dimensions (shape) not size for declaration required:

Pointer allocation for arrays

```
REAL, DIMENSION (:,:,), POINTER :: cube
...
N = 42
ALLOCATE (cube(N, N, N))
....
ALLOCATE (cube(1,2,3)) ! allocation w/o previous DEALLOCATE
                        ! possible only for POINTER
                        ! note: entries are not kept
...
DEALLOCATE (cube)
```

Pointer assignment

```
REAL, TARGET :: x, y      ! attr. TARGET required
REAL, POINTER :: p, q
y = 0.75
p => x                    ! associates p with x
p = y                    ! normal assignment (values) of y to p
WRITE (*,*) x           ! gives 0.75
q => p                    ! x, p, q are now the same
```

Note: For `p => x` the pointer `p` is assigned to the target `x`, while `q => p` lets the pointer `q` reference the same object as `p`

Meaning of the TARGET attribute:

To enable optimization the compiler needs to know which variables are referenced, i.e. whether there is an alternative method of access.

TARGET and the corresponding POINTER must be of the same type (REAL, INTEGER, ...), moreover, in the case of arrays they must agree shape (rank, dimension):

```
REAL, TARGET :: cube (16, 16, 1000)
REAL, POINTER :: image (:,:)
...
image => cube(:, :, 42) ! Sub-Array
...
NULLIFY (image)
```

It is not necessary, but safer to disassociate a pointer after usage with help of NULLIFY or `image => NULL ()`

Typical application: linked lists

```
TYPE entry
  REAL :: value
  INTEGER :: index
  TYPE (entry), POINTER :: next
END TYPE entry

TYPE (entry), POINTER :: first, current
! with first%index and first%next%index
ALLOCATE (first)
first = entry (1.,10,null())
ALLOCATE (current)
current = entry (2.,20, first)
! = means current%next => first
first => current
! first now points to new entry w/o deleting
! the first entry
```

So, from a list of two elements (each has three components):

```
first : (2.0, 20, associated)
first%next : (1.0, 10, null)
```

it is possible to create a list of three with help of current:

```
allocate (current)
current = entry (3., 30, first)
first => current
print *, first%value ! gives 3.0
```

```
first :          (3.0, 30, associated)
first%next :     (2.0, 20, associated)
first%next%next : (1.0, 10, null)
```

Apart from pointers there is another way to address the same memory area by a different name:

EQUIVALENCE

```
COMPLEX    :: z, c      ! complex has two real entries
REAL       :: zr(2)     ! real array size 2
EQUIVALENCE (zr, z, c) ! all three variables access now
                        ! same memory
```

- this assignment is fixed in the declaration block and cannot be changed any more

- the access to the same *memory area* does not require that the involved variables have to be of the same type:

```
INTEGER    :: i(100)
REAL       :: x(100)
EQUIVALENCE (i,x)
```

→ if memory is short, but very dangerous

- for character variables, the sizes do not need to be the same:

```
CHARACTER :: a*4, b(2)*3
EQUIVALENCE (a, b(1)(3:))
```

character variable “a” corresponds to the last four characters of the array “b”

In Fortran, data are passed by “reference”, i.e. the calling procedure

```
CALL SUBROUTINE calcE (velo, pos, E)
```

- passes only *start addresses*. This enables data exchange with C functions, if they use pointers as arguments:

```
c_function_(int *i){ ... }
```

- data exchange is done by the argument list, i.e. use of subroutine calls and void functions in C:

```
CALL c_func (x,i) in Fortran  
void c_func_ (float *x, int *i){ ... }
```

Note:

- the Fortran compiler appends an underscore “_” to the name of the subroutine, therefore the name of the called C function must end with an underscore in the C source code
- C always expects null terminated strings, so in Fortran:

```
text = text // CHAR (0) ! appending null
```

Compiling and linking of the Fortran program:

```
gcc      -c csource.c
gfortran -c fprogram.f
gfortran -o fprogram.exe fprogram.o csource.o
```

As Fortran uses column-(major)-order arrays (first index runs first), while C/C++ uses row-(major)-order arrays (last index runs first) this must be taken into account for procedure calls. So the correspondance is, e.g.

Fortran	C/C++
INTEGER m(2,3)	int m[3][2] ;
CHARACTER*6 text[4]	char text[4][6] ;

However, note that C/C++ strings must be terminated by null, i.e. text element has effectively only a length of 5.

Access to Fortran from C I

From above mentioned differences in argument passing between C and Fortran, access from C to Fortran procedures:

- pointers for all arguments in your C code
- especially, no literal (=constant) arguments, e.g. `x = fort_func_(1.,y)` ;
- most probably, appending underscores in the call to the Fortran procedures is required, e.g., Fortran: `REAL MYFUNC(Z,Y)` → C: `x = myfunc_(a,b)` ;
- compilation of C and Fortran procedures separately, linking `.o` files together
- Fortran expects arrays in row-major order

Call to LINPACK

```
double a[ndim][ndim], help[ndim*ndim] ;  
...  
for (int i=0; i<ndim; ++i)  
  for (int j=0, j<ndim; ++j) help[j+ndim*i] = a[j][i];  
...  
dgefa_(help, &ndim, &ndim, &ipvt, &info) ;
```

Programming in Fortran - Part 7

Parallelization, Optimization

Arrays with WHERE and FORALL I

WHERE specifies a *mask* for an array and works like an IF, i.e.,

```
WHERE (mask) Instruction
```

or

```
WHERE (mask)  
  instruction(s)  
ELSE WHERE (mask2)  
  instruction(s)  
ELSE WHERE  
  instruction(s)  
END WHERE
```

mask is a so-called logical array expression that restricts the *instructions* (array instruction) correspondingly

Example: vectors (1d arrays)

```
REAL :: A(5), B(5), C(5)
DATA A /0.,1.,1.,1.,0./
DATA B /10.,11.,12.,13.,14./
C = -1.
WHERE (A .NE. 0.) C = B / A
WHERE (C .GT. 0.)
  C = ALOG (C)
ELSE WHERE
  C = OURFUNC (C)
END WHERE
```

Important note: difference between execution of a function (right hand side) and assignment regarding (intrinsic) ELEMENTAL FUNCTIONS , e.g.,

ALOG: execution only for the masked elements of C (in parallel),

OURFUNC: execution for *all* elements of C, but assignment only to masked elements of C

A more general form of WHERE can be performed with FORALL:

```
FORALL (ind-spec, . . . , mask) instruction
```

or

```
FORALL (ind-spec, . . . , mask)  
  instruction  
END FORALL
```

where *ind-spec* corresponds to index triplets for arrays:

L:U:S, e.g., 2:21:2

L lower bound; U upper bound; S stepsize, optional, e.g., also -1

Example: FORALL

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0)  
  B(I, J) = 1.0 / A(I, J)  
END FORALL
```

is equivalent to

```
WHERE (A .NE. 0) B = 1.0 / A
```

→ WHERE and FORALL instructions can be executed in “parallel” (usually vectorization)

Coarray Fortran

- since Fortran 2008 also in the standard, supported by gfortran (option `-fcoarray=lib`, external library required) and ifort (option `-coarray=distributed`, using commercial library)
- generates multiple *images* of the same program and executes them on different processes (shared or distributed) → cf. Message Passing Interface (MPI)
- therefore: new syntax
→ arrays with square brackets `[]`, i.e.,
`variable` is a local image, `variable[2]` is remote image

Code snippet: Hello! with Coarray

```
INTEGER :: I                ! local variable
CHARACTER(LEN=20) :: NAME[*] ! Coarray
! local image for user interaction
IF (THIS_IMAGE() == 1) THEN
  WRITE(*,"(A)") 'Enter your name: '
  READ(*,"(A)") NAME

  ! distribute information
  DO I = 2, NUM_IMAGES()
    NAME[I] = NAME
  END DO
END IF
SYNC ALL ! barriere for assuring data synchronized
! output from all processes
WRITE(*,"(3A,I0)") "Hello ", NAME," from Image ", THIS_IMAGE()
```

Different levels (up to “agressive”) of automatic optimization for *speed* by compiler option `-O[n]` with $n = 0, 1, 2, 3$ (gfortran, ifort), includes, e.g.

- loop unrolling: replacing loop completely or partially by multiple copies of loop body
→ saves time for loop control; pipelining, vectorization (SSE, AVX!) of loop body operations → increases code size

Example: loop unrolling

Original loop

```
DO I = 1, 100
  X(I) = X(I) + 5.
ENDDO
```

Unrolled by factor 4

```
DO I = 1, 100, 4
  X(I)   = X(I)   + 5.
  X(I+1) = X(I+1) + 5.
  X(I+2) = X(I+2) + 5.
  X(I+3) = X(I+3) + 5.
ENDDO
```

- loop collapsing: replace nested loops by one loop → improves chances for, e.g., loop unrolling

Example: loop collapsing

Original loop

```
DO I = 1, 100
  DO J = 1, 200
    X(I,J) = 5.
  ENDDO
ENDDO
```

Loop collapsed

```
DIMENSION X(100,200), Y(20000)
EQUIVALENCE (X,Y)
DO I = 1, 20000
  Y(I) = 5.
ENDDO
```

- constant propagation: (literal) constants assigned to a variable are propagated through the code → saves computation time at runtime

Example: constant propagation

Original

```
X = 3.  
Y = X + 2.
```

Constant propagation

```
X = 3.  
Y = 5.
```

- constant folding: expand operations with constants at compiletime → saves computation time at runtime

Example: constant folding

Original

```
X = 1. / 2.
```

Constant propagation

```
X = 0.5
```

- function inlining: calls to simple functions are replaced by the function body → saves time for call and returning → increases code size

Example: Function inlining

Original

```
SUM = ADD(X, Y)
```

```
...
```

```
FUNCTION ADD (X, Y)
```

```
  ADD = X + Y
```

```
END
```

Function eliminated

```
SUM = X + Y
```

- removal of, e.g., unreference variables → dangerous, in -O3 (most aggressive) this might have unwanted effects
- + many more (see compileroptimization.com)

Due to many restrictions (declaration block, restricted DO-loops) Fortran can be easier and faster optimized!

I don't know what the language of the year 2000 will look like, but I know it will be called Fortran.

Tony Hoare
(quick-sort algorithm, Turing Award 1980)
on a conference in 1982

Curcic, M. 2020, Modern Fortran (MANNING PUBN)

Metcalf, M., Reid, J., & Cohen, M. 2018, Modern Fortran Explained (Oxford University Press)