Goals for today

- Understand how real numbers are stored in the computer
- Adjust the precision of real (and integer) numbers
- Get to know the CHARACTER data type
- Format text output using format descriptors
- Read and write text files
- Make user input more flexible with namelists

Addition to procedures: SAVE attribute

When local variables of procedures are declared with the SAVE attribute, their values are saved between calls to the procedure.

Local variables that are initialized in the declaration have an implicit SAVE attribute. This means that they are not re-initialized if the procedure is called multiple times!

```
PROGRAM save_me
  IMPLICIT NONE
  INTEGER :: i

  DO i = 1, 6
    CALL fibi()
  END DO
END PROGRAM save_me
```

```
SUBROUTINE fibi()
  IMPLICIT NONE
  INTEGER :: a=1, b=0, c

  c = a + b
  a = b; b = c
  PRINT*, c
END SUBROUTINE fibi
```

```
$ gfortran save_me.f90
$ ./a.out
           1
           1
           2
           3
           5
           8
```

Equivalent to

```
INTEGER, SAVE :: a=1, b=0, c
```

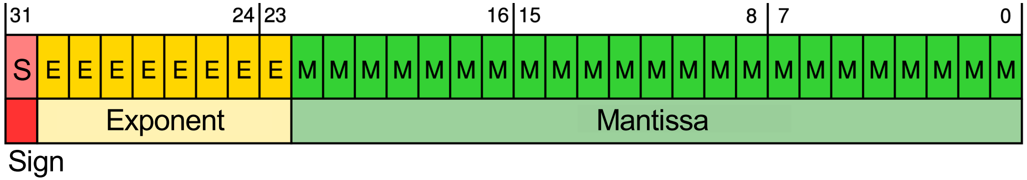
Precision of real numbers

$$x = s \cdot m \cdot b^e$$

Base

Real numbers are stored as floating point numbers. They are composed of the **sign**, the **exponent** and the **mantissa.**

Arrangement of bits for single precision REAL



Machine precision:
Smallest positive number for which the condition $1+\varepsilon > 1$ is fulfilled.

Type	Size	Exponent	Mantissa	ε	Smallest number	Largest number
single	32 bit	8 bit	23 bit	$2^{-23} \approx 1.2 \cdot 10^{-7}$	$\sim 1.4 \cdot 10^{-45}$	$\sim 3.4 \cdot 10^{38}$
double	64 bit	11 bit	52 bit	$2^{-52} \approx 2.2 \cdot 10^{-16}$	$\sim 4.9 \cdot 10^{-324}$	$\sim 1.8 \cdot 10^{308}$
double ext.	80 bit	16 bit	63 bit	$2^{-63} \approx 1.1 \cdot 10^{-19}$	$\sim 3.7 \cdot 10^{-4951}$	$\sim 1.2 \cdot 10^{4932}$
quad(ruple)	128 bit	15 bit	112 bit	$2^{-112} \approx 1.9 \cdot 10^{-34}$	$\sim 6.5 \cdot 10^{-4966}$	$\sim 1.2 \cdot 10^{4932}$

Specifying precision: 2 options

1. In code
   - Syntax on the next slide

2. When compiling
   - For example 8-byte REAL:
     - gfortran -fdefault-real-8 program.f90
     - ifort -r8 program.f90

   - Also works for INTEGERS
     - INTEGERS and REALS can have different sizes
     - ifort -r8 -i4 program.f90

Different REAL kinds

```fortran
PROGRAM reals
USE iso_Fortran_env
IMPLICIT NONE
INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(15,307)
INTEGER, PARAMETER :: QUAD   = SELECTED_REAL_KIND(33,4931)
REAL :: r1                 !default (32 oder 64 bit)
DOUBLE PRECISION :: r2     !twice as many bits
REAL*8 :: r3               !Fortran 77
REAL(KIND=8) :: r4         !Fortran 90 (processor dependent)
REAL(KIND=DOUBLE) :: r5 !Fortran 90 (processor independent)
REAL(KIND=QUAD)   :: r6 !"
REAL(KIND=REAL64) :: r7 !Fortran 2003

PRINT*, REAL_KINDS         !lists available KIND parameters
PRINT*, DOUBLE, QUAD
PRINT*, KIND(r1), KIND(r2), KIND(r3), KIND(r4)
PRINT*, KIND(r5), KIND(r6), KIND(r7)
END PROGRAM reals
```

```
$ gfortran reals.f90
$ ./a.out
           4           8          10          16
           8          16
           4           8           8           8
           8          16           8
```

SELECTED_REAL_KIND(P,R) returns KIND parameter with minimum range $-10^R < n < 10^R$ and minimum precision P.

-1: P not supported

-2: R not supported

-3: neither P nor R supported

KIND parameters for real numbers

Compiler	REAL32	REAL64	REAL128
gfortran	4*	8	16
ifort	4*	8	16
nagfor	1*	2	n/a

* mostly default value

Precision test

```
PROGRAM how_precise
IMPLICIT NONE
REAL*4  :: r1
REAL*8  :: r2
REAL*10 :: r3
REAL*16 :: r4
CHARACTER(LEN=20), PARAMETER :: fmtstr = '(A, 2I12, 3ES15.2E4)'

WRITE(*,fmtstr) ' 4 bytes:', PRECISION(r1), RANGE(r1), EPSILON(r1), TINY(r1), HUGE(r1)
WRITE(*,fmtstr) ' 8 bytes:', PRECISION(r2), RANGE(r2), EPSILON(r2), TINY(r2), HUGE(r2)
WRITE(*,fmtstr) '10 bytes:', PRECISION(r3), RANGE(r3), EPSILON(r3), TINY(r3), HUGE(r3)
WRITE(*,fmtstr) '16 bytes:', PRECISION(r4), RANGE(r4), EPSILON(r4), TINY(r4), HUGE(r4)
END PROGRAM how_precise
```

intrinsic functions

```
$ gfortran how_precise.f90
$ ./a.out
 4 bytes:           6          37      1.19E-0007     1.18E-0038     3.40E+0038
 8 bytes:          15         307      2.22E-0016     2.23E-0308     1.80E+0308
10 bytes:          18        4931      1.08E-0019     3.36E-4932     1.19E+4932
16 bytes:          33        4931      1.93E-0034     3.36E-4932     1.19E+4932
```

Different REAL kinds: constants

- Unnamed constants, which are used directly in the code, are normally also 32bit by default, and rarely 64bit.

- In the code, the precision of constants can be specified in 2 different ways:

  - With an underscore

`1.234_8`	works only if 8 is a valid KIND parameter
`1.234_DOUBLE`	works only if DOUBLE is an INTEGER named constant (corresponding to a KIND parameter)

  - With E, D and Q

`1.234E5`	$1.234 \cdot 10^5$ in single precision
`1.234D5`	$1.234 \cdot 10^5$ in double precision
`1.234Q5`	$1.234 \cdot 10^5$ in quadruple precision

Real numbers in binary system

- Since computers store numbers in binary, they cannot store certain finite decimal numbers exactly, e.g. $0.1_{10} = 0.00011001100110011..._2$
- Only real numbers that have a power of two in the denominator can be represented exactly, e.g. $0.5_{10} = 0.1_2$

```
PROGRAM binary_decimals
WRITE(*,'(F20.18)') 0.1
WRITE(*,'(F20.18)') 0.5
END PROGRAM binary_decimals
```

```
$ gfortran binary_decimals.f90
$ ./a.out
0.100000001490116119
0.500000000000000000
```

Real numbers as counters in counting loops (should be avoided)

- Until Fortran 90 it was possible to use REALS or INTEGERS as counters in counting loops.
- As of Fortran 95 only INTEGERS are allowed (in theory).
  - gfortran warns for REALS
  - ifort accepts REALS

REAL

```
0.000000
0.100000
0.200000
0.300000
0.400000
0.500000
...
4.699998
4.799998
4.899998
4.999998
```

becomes inaccurate

INTEGER

```
0.000000
0.100000
0.200000
0.300000
0.400000
0.500000
...
4.700000
4.800000
4.900000
5.000000
```

```
PROGRAM looptest
IMPLICIT NONE
REAL :: a
INTEGER :: i

DO a=0.,5.,0.1 ! REAL
  WRITE(*,'(F8.6)') a
END DO
DO i=0,50 ! INTEGER
  a = i/10.0
  WRITE(*,'(F8.6)') a
END DO
END PROGRAM looptest
```

Which precision should my program have?

- If possible use 32 bit.
  - Runs faster
  - Needs less memory
  - Output uses less space

- If accuracy of >6 digits is needed, use 64 bit, e.g.
  - When adding very large ($>10^6$) or very small ($<10^{-6}$) numbers
  - For direct solvers of systems of equations

- If in doubt, run the program with 32 bit and 64 bit and compare the result.

- Different precisions can be used in the same code.

CHARACTER data type

CHARACTER variables represent strings.

The length of the variable is set with the LEN parameter (or earlier *).

Strings are padded or truncated such that their length matches the declared length (except for LEN=*).

With TRIM trailing spaces are truncated.

With // strings are concatenated.

```
PROGRAM characters
  IMPLICIT NONE
  CHARACTER :: a !one character
  CHARACTER*10 :: b !a string of length 10 (Fortran 77)
  CHARACTER(LEN=20) :: c, d !two strings of length 20 (Fortran 90)
  CHARACTER(LEN=*), PARAMETER :: word='Yes' ! automatic length

  a = 'A'
  b = "Hello"
  c = "'ABCDEFGHIJKLMNOPQRSTUVWXYZ'"
  d = 'Everything clear'

  PRINT*, a, '!'
  PRINT*, b // c, '!'
  PRINT*, TRIM(d), '?'
  PRINT*, word, '!'
END PROGRAM characters
```

No difference between " and '

```
$ gfortran characters.f90
$ ./a.out
 A!
 Hello     'ABCDEFGHIJKLMNOPQRS!
 Everything clear?
 Yes!
```

Format descriptors

More format descriptors: http://fortranwiki.org/fortran/show/Edit+descriptors

The format of input and output can be specified with format descriptors:

Formatted instead of list-directed input and output

Data type		Format descriptor	Example
Strings		rAw	A10
Integers		rIw.m	3I5
Real numbers	Decimal notation	rFw.d	F10.4
	Scientific notation, 0 as first digit	rEw.dEe	5E14.4
	Scientific notation, 1-9 as first digit	rESw.dEe	ES15.2E4
	Decimal or scientific notation, depending on size	rGw.dEe	G13.3
Logicals		rLw	L10
Space		wX	3X
“Tab”		T[R,L]c	T51

r: repeat count, w: field width, m: minimum number of characters, d: number of decimal places, e: length of exponent, c: column number

Formatted WRITE statements: 3 options

```
INTEGER :: i=123456
REAL :: x=3.14159
```

1. Directly specify the format descriptor in the WRITE statement

```
WRITE(*,'(1X,I6,F10.2)') i, x
```

2. Specify the label of an associated FORMAT statement

```
WRITE(*,100) i, x
100 FORMAT(1X,I6,F10.2)
```

3. Define a string containing the format descriptor

```
CHARACTER(LEN=16) :: fmtstr

fmtstr='(1X,I6,F10.2)'
WRITE(*,fmtstr) i, x
```

Example

```fortran
PROGRAM format_write
IMPLICIT NONE
INTEGER :: i=1, j=-2
REAL :: a(5), b
CHARACTER(LEN=6) :: nm='Marina'

CALL RANDOM_NUMBER(a)
b=3.14E-14

WRITE(*,'(5F7.3)') a
WRITE(*,'(A,2X,I4,T25,I4)') nm, i, j

WRITE(*,10) b, a
10 FORMAT(6(ES12.3))
END PROGRAM format_write
```

```
$ gfortran format_write.f90
$ ./a.out
  0.637  0.641  0.197  0.944  0.081
Marina     1               -2
   3.140E-14   6.369E-01   6.410E-01   1.966E-01
   9.443E-01   8.140E-02
```

- Format descriptor strings are delimited with parentheses. Repeat counts before the parentheses repeat the whole expression inside the parentheses.
- Decimal places are rounded mathematically
- Sign and exponent count towards the field width.
- If the field width is too small or the type of an output variable does not match the format descriptor, asterisks are displayed.

Formatted READ statements

Work similarly to formatted WRITE statements, but can be complicated:

- Field widths have to agree
- Real numbers are only interpreted correctly if they have a decimal point

```
PROGRAM format_read
IMPLICIT NONE
CHARACTER(LEN=10) :: message
REAL :: a, b, c

WRITE(*,'(A,$)') 'Please enter a message and three reals: '
READ(*,'(A,3F10.4)') message, a, b, c

WRITE(*,'(2A,3F10.4)') 'This is what you entered: ', message, a, b, c

END PROGRAM format_read
```

```
$ gfortran format_read.f90
$ ./a.out
Please enter a message and three reals:
Hello you   1.5       1.5E-2    1500
This is what you entered:
Hello you     1.5000    0.0150    0.1500
```

In general, list-directed input is easier to use, except that strings need quotes if they contain spaces, commas, or slashes.

Reading and writing text files

- To open / close:
  OPEN() and CLOSE() with specification of a file number and a file name
  - The file number can be any number except 5 (corresponds to stdin) and 6 (corresponds to stdout)
  - The STATUS specifier in the OPEN statement can be used to set the status of the file:
    - OLD: The file already exists and is opened for input or output.
    - NEW: The file does not exist and is created.
    - REPLACE: The file already exists and any previous contents are deleted.
    - SCRATCH: A temporary file is created for use within the program and will be deleted when the program exits.

- To read / write:
  READ() and WRITE(), where the first * is replaced by the file number.
  - Reading is easy if the number of values is known, if not it is a bit more complicated (examples follow)

Examples: Write… ...and read a text file

```
PROGRAM filewrite
IMPLICIT NONE
INTEGER :: n, i
REAL, ALLOCATABLE :: a(:)

PRINT*, 'How many students?'
READ*, n
ALLOCATE(a(n))

CALL RANDOM_NUMBER(a)
a = 1+4*a

OPEN(2, FILE='grades.txt', STATUS='new')
WRITE(2,*) n ! First line is n
DO i = 1,n ! Then the rest
  WRITE(2,*) NINT(a(i))
END DO
CLOSE(2)
END PROGRAM filewrite
```

File does not exist yet

```
PROGRAM fileread
IMPLICIT NONE
INTEGER :: n, i
REAL, ALLOCATABLE :: a(:)

OPEN(1, FILE='grades.txt', STATUS='old')
READ(1,*) n ! This is the number of values
ALLOCATE(a(n))
DO i=1,n
  READ(1,*) a(i)
END DO
CLOSE(1)

WRITE(*,10) 'The average grade is:', SUM(a)/n
10 FORMAT(A,1X,F4.2)
END PROGRAM fileread
```

File exists

```
$ gfortran fileread.f90
$ ./a.out
The average grade is: 2.42
```

Read text file with an unknown number of values

IOSTAT as a third argument:

- 0: Line was read successfully
- <0: Reached end of file
- >0: An error has occurred

REWIND moves the pointer back to the beginning of the file

```
PROGRAM fileread
IMPLICIT NONE
INTEGER :: n, i, ios
REAL, ALLOCATABLE :: a(:)
REAL :: b

OPEN(1, FILE='grades.txt', STATUS='old')
n = 0
DO ! Loop for counting the values
  READ(1,*,IOSTAT=ios) b
  IF (ios<0) EXIT ! Reached end of file
  IF (ios/=0) STOP 'I cannot read this line.'
  n = n+1
END DO

PRINT*, 'I found', n, 'grades.'
ALLOCATE(a(n))

REWIND(1) ! Back to the start
DO i=1,n
  READ(1,*) a(i)
END DO
CLOSE(1)

WRITE(*,10) 'The average grade is:', SUM(a)/n
10 FORMAT(A,1X,F4.2)
END PROGRAM fileread
```

Namelists

- Simple and flexible way to read input parameters from a text file
- All variables are declared in the program
- In the text file they have the same names but (possibly) a different order
- Not all variables must appear in the namelist → Set default values
- One text file can contain several namelists

In the program

```
NAMELIST / groupname / var1 [, var2, ...]
```

In the text file

```
&groupname [var1=value1 [, var2=value2, ...]] /
```

Example

```
&INPUTS
  L=100.
  Nx=128
  total_time=60.
  outputfilename='outfile'
/
```

starts with &

ends with /

blank line

```
$ gfortran diffusion.f90
$ ./a.out
&INPUTS
 L=  100.000000    ,
 NX=128          ,
 TOTAL_TIME=  60.0000000     ,
 OUTPUTFILENAME="outfile                  ",
 /
```

```
PROGRAM diffusion

USE findiff

IMPLICIT NONE
REAL :: L=1. ! Length of the model domain
INTEGER :: nx=1 ! Number of grid points
REAL :: total_time=0. ! Simulation time
CHARACTER(LEN=25) :: outputfilename='xyz' ! File name

NAMELIST /inputs/ L, nx, total_time, outputfilename

OPEN(7,FILE='namelist',STATUS='old')
READ(7,NML=inputs)
CLOSE(7)

WRITE(*,NML=inputs)

! Rest of the program

END PROGRAM diffusion
```

NML= is optional if the name comes second.
Note: No quotation marks

Summary

- Many real numbers cannot be stored exactly in the computer.
- The precision determines how many significant digits a real number has after the decimal point (in floating point representation). It can be specified with a compiler flag or in the code.
- Formats of text inputs and outputs can be specified with format descriptors.
- Text files are opened and closed with OPEN() and CLOSE(), and read and written with READ() and WRITE().