Goals for today

- Understand how characters are stored in the computer
- Use new functions and statements for strings and arrays
- Get to know the COMPLEX data type and do calculations with complex numbers

ASCII character set

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Letter Number Punctuation Symbol Other Undefined Character changed from 1963 version and/or 1965 draft

https://en.wikipedia.org/wiki/ASCII

String functions

- TRIM(string) truncates trailing spaces
- // concatenates strings
- string(m:n) returns a substring from m to n (always needs m and n)
- LEN(string) returns the length of a string
- LEN_TRIM(string) returns the length of a string without trailing spaces
- INDEX(string,substring) returns the location of a substring in a string
- CHAR(n) and ACHAR(n) convert an INTEGER into a CHARACTER*.
- ICHAR(n) and IACHAR(n) convert a CHARACTER into an INTEGER*.
- WRITE(string,...) and READ(string,...) write and read a string (=internal file)

*ACHAR and IACHAR use ASCII character set, CHAR and ICHAR are system dependent

Example

```
PROGRAM strings
IMPLICIT NONE
CHARACTER(LEN=20) :: a='eins zwei drei', b='file', c, d
INTEGER :: n

PRINT*, LEN(a), LEN_TRIM(a)
PRINT*, a(1:4)
PRINT*, INDEX(a,'zwei')
PRINT*, (CHAR(n), n=0,127)
PRINT*, ICHAR('a')
n=99
WRITE(c,'(i2)') n
d = TRIM(b) // TRIM(c)
PRINT*, d

END PROGRAM strings
```

```
$ locale charmap
UTF-8
```

UTF-8 is an extension of ASCII

```
$ gfortran strings.f90
$ ./a.out
          20          14
 eins
           6
```

Control characters

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[
\]^_`abcdefghijklmnopqrstuvwxyz{|}~
          97
 file99
```

Strings can also be compared

The result of the comparison depends on the positions of the characters in the character set.

Relational operators are system dependent:
==, /=, <, <=, >, >=

Lexical functions are always based on the ASCII character set:
LLT (less than), LLE (less than or equal to),
LGT (greater than), LGE (greater than or equal to)

For strings with multiple characters, the characters are compared one after the other.

```fortran
PROGRAM string_comp
IMPLICIT NONE
LOGICAL :: l1, l2, l3, l4, l5

l1 = 'A' < 'a'
l2 = LLT('A', 'a')
l3 = 'AAAAA' < 'AAAAB'
l4 = 'AB' < 'AAAA'
l5 = 'AAAAA' < 'AAAA'

PRINT*, l1, l2, l3, l4, l5

END PROGRAM string_comp
```

```
$ gfortran string_comp.f90
$ ./a.out
 T T T F F
```

Strings as arguments in functions / subroutines

If the length of a CHARACTER argument is unknown, it can be defined as variable length (LEN=*), or determined with an input argument.

```
$ gfortran character_arguments.f90
$ ./a.out
 Enter string length:
7
 The string is: abcdefg!
```

```
PROGRAM character_arguments
IMPLICIT NONE
INTEGER :: n

WRITE(*,*) 'Enter string length:'
READ (*,*) n
CALL print_str(abc(n))

CONTAINS

FUNCTION abc(n)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
CHARACTER(LEN=n) abc
CHARACTER(LEN=26) :: alphabet = 'abcdefghijklmnopqrstuvwxyz'
abc = alphabet(1:n)
END FUNCTION abc

SUBROUTINE print_str(abc)
IMPLICIT NONE
CHARACTER(LEN=*), INTENT(IN) :: abc
WRITE (*,*) 'The string is: ', abc, '!'
END SUBROUTINE print_str

END PROGRAM character_arguments
```

Length n

Variable length

Read and write 2D arrays

Fortran stores arrays column by column and reads/writes them row by row

→ Use (implied) DO loops to read/write 2D arrays

```
PROGRAM read_array
IMPLICIT NONE
INTEGER :: i, j
INTEGER :: array2d(3,5)

OPEN(2,FILE='numbers.txt',STATUS='old')
READ(2,*) array2d
CLOSE(2)

WRITE(*,'(5I3)') array2d(1,:)

OPEN(2,FILE='numbers.txt',STATUS='old')
READ(2,*) ((array2d(i,j), j=1,5), i=1,3)
CLOSE(2)

WRITE(*,'(5I3)') array2d(1,:)
END PROGRAM read_array
```

```
PROGRAM write_array
IMPLICIT NONE
INTEGER :: i, j
INTEGER :: array2d(3,5)

DO j = 1,5
  array2d(:,j) = j
END DO

OPEN(2,FILE='numbers_new1.txt')
WRITE(2,'(5I2)') array2d
CLOSE(2)

OPEN(2,FILE='numbers_new2.txt')
WRITE(2,'(5I2)') (array2d(i,:), i=1,3)
CLOSE(2)
END PROGRAM write_array
```

```
$ cat numbers.txt
 1 2 3 4 5
 1 2 3 4 5
 1 2 3 4 5
```

```
$ cat numbers_new1.txt
 1 1 1 2 2
 2 3 3 3 4
 4 4 5 5 5
```

```
$ cat numbers_new2.txt
 1 2 3 4 5
 1 2 3 4 5
 1 2 3 4 5
```

```
$ gfortran read_array.f90
$ ./a.out
  1  4  2  5  3
  1  2  3  4  5
```

Intrinsic array functions

Query properties

ALLOCATED: Is the array allocated?

SIZE: How big is the array?

SHAPE: What is the shape of the array?

LBOUND, UBOUND: What are the bounds of the array?

Numerical and logical calculations

SUM: Sum of the array elements

PRODUCT: Product of the array elements

MINVAL, MAXVAL: Minimum and maximum of the array

COUNT: Number of elements that fulfill a condition

ANY: True if condition is fulfilled somewhere

ALL: True if condition is fulfilled everywhere

Determine position of array elements

MINLOC, MAXLOC: Determine the position of the minimum and maximum of the array.

Matrix operations

MATMUL: Matrix multiplication

DOT_PRODUCT: Scalar product

Array manipulation

TRANSPOSE: Transpose array

CSHIFT, EOSHIFT: Move array elements

Create new arrays

MERGE: Mix two arrays

SPREAD: Copy array along a new dimension

RESHAPE: Change the shape of the array

PACK: Create vector with elements of the array that satisfy a condition

UNPACK: Similar to MERGE but with vector instead of array

Example

```
$ gfortran array_functions.f90
$ ./a.out
           7 T F F F T
  1  4  7
  2  5  8      <- array1
  3  6  9

  1  2  3
  4  5  6      <- TRANSPOSE(array1)
  7  8  9

  2  5  8
  3  6  9      <- CSHIFT(array1,1)
  1  4  7

  4  7  0
  5  8  0      <- EOSHIFT(array1,1,DIM=2)
  6  9  0

  5  4  7
  5  5  8      <- MERGE(array1,array2,mask)
  3  6  9
```

```fortran
PROGRAM array_functions
IMPLICIT NONE
INTEGER, DIMENSION(3,3) :: array1, array2
LOGICAL, DIMENSION(3,3) :: mask

array1 = RESHAPE((/1,2,3,4,5,6,7,8,9/),SHAPE(array1))
array2 = 5
mask = array1 > 2

WRITE(*,*) COUNT(mask), ANY(mask), ALL(mask), ALL(mask,DIM=2)
CALL write_array(array1)
CALL write_array(TRANSPOSE(array1))
CALL write_array(CSHIFT(array1,1))
CALL write_array(EOSHIFT(array1,1,DIM=2))
CALL write_array(MERGE(array1,array2,mask))

CONTAINS
  SUBROUTINE write_array(array)
  INTEGER, INTENT(IN) :: array(:,:)
  INTEGER :: i
  CHARACTER :: n

  WRITE(n,'(I1)') SIZE(array,DIM=2)
  DO i = LBOUND(array,DIM=1), UBOUND(array,DIM=1)
    WRITE(*,'('//n//'I3)') array(i,:)
  END DO
  WRITE(*,*)
  END SUBROUTINE write_array
END PROGRAM array_functions
```

DIM as second argument

internal file

Subscript triplets and vector subscripts

- In order to access parts of an array, we can specify the lower and upper limit as well as the stride (→ subscript triplets).

```
r(4:1:-1,3:1:-1)
```

- Or we can use a one-dimensional integer array (→ vector subscript).

```
z((/1,2,3/),(/2,1,4,3/))
```

Equivalent

```
REAL :: a(9,5,3), b(9,3), c(3,3)
```

```
! Element by element
DO j = 1,3
  DO i = 1,3
    c(i,j) = a(2*i-1,2,j) + b(2*i,j)
  END DO
END DO

! With subscript triplets
c = a(1:5:2,2,:) + b(2:6:2,:)
```

Not equivalent (data dependency)

```
REAL, DIMENSION(n) :: a, b, c
```

```
! Element by element
DO i = 2,n
  a(i) = a(i-1) * b(i) + c(i)
END DO

! With subscript triplets
a(2:n) = a(1:n-1) * b(2:n) + c(2:n)
```

WHERE

The WHERE statement allows masked assignment (i.e. only for certain elements of an array).

Example:
LOG assignment results in NaN where a is negative (or SIGFPE).

```
b = LOG(a)
```

DO loop is tedious (and slow)

```
DO j = 1,5
  DO i = 1,5
    IF (a(i,j) > 0.) THEN
      b(i,j) = LOG(a(i,j))
    ELSE
      b(i,j) = -999.
    END IF
  END DO
END DO
```

WHERE as block

```
WHERE (a > 0.)
  b = LOG(a)
ELSEWHERE
  b = -999.
END WHERE
```

Single-line WHERE

```
b = -999.
WHERE (a > 0.) b = LOG(a)
```

FORALL (Fortran ≥95) and DO CONCURRENT (Fortran ≥2008).

- DO loops specify the order in which arrays are processed.
- FORALL and DO CONCURRENT choose the optimal order and allow parallelization → faster
- FORALL allows only direct assignments, DO CONCURRENT is more flexible (allows also temporary variables, READ and WRITE).
- As of Fortran 2018, FORALL is officially deprecated and should not be used in new programs.

FORALL

```
FORALL (i=1:5,j=1:5,a(i,j)>0)
  b(i,j) = LOG(a(i,j))
END FORALL
```

```
FORALL (i=1:5,j=1:5,a(i,j)>0) b(i,j) = LOG(a(i,j))
```

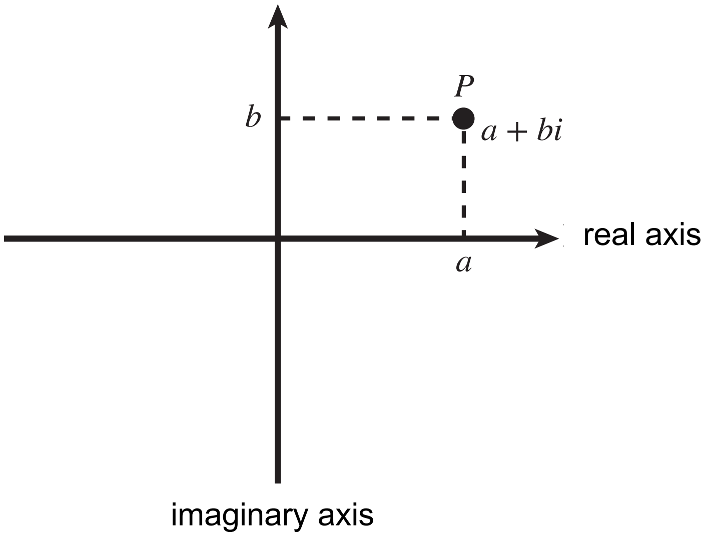
DO CONCURRENT

```
DO CONCURRENT (i=1:5,j=1:5,a(i,j)>0)
  b(i,j) = LOG(a(i,j))
END DO
```

optional condition

COMPLEX data type

Complex variables and constants are stored in Cartesian coordinates and consist of two REALS:




```
PROGRAM complex
IMPLICIT NONE
INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(15,307)
REAL(KIND=DOUBLE), PARAMETER :: PI = 3.141592653589793
COMPLEX :: a = (1.1E6, -0.5E2)
COMPLEX(KIND=DOUBLE) :: b = (PI, 1.)
COMPLEX :: c, d
COMPLEX, DIMENSION(256) :: array1

array1 = (0.,0.)

WRITE(*, '(a,$)') 'Enter a complex number: '
READ (*,*) c
WRITE(*, '(a,$)') 'Enter another complex number: '
READ (*,'(2F10.2)') d

PRINT*, a
PRINT*, b
PRINT*, c
PRINT*, d
PRINT*, array1(1)
END PROGRAM complex
```

Intrinsic functions for complex numbers

- ABS(c) and CABS(c) calculate the magnitude of a complex number: $\sqrt{a^2 + b^2}$
- CMPLX(a,b,*kind*) combines a and b to a complex number: $a + bi$
  If kind is present the function value has the specified kind parameter, if not the default kind parameter is used (not that of a and/or b!).
- CONJG(c) calculates the conjugate complex number: $c = a + bi \rightarrow c^* = a - bi$
- INT(c) converts the real part of c into an INTEGER
- REAL(c) converts the real part of c into a REAL
- AIMAG(c) converts the imaginary part of c into a REAL

- Complex numbers can be added, subtracted, multiplied, and divided, or used as arguments in mathematical functions such as SIN, COS, LOG10, SQRT.
- However, they are not ordered and therefore cannot be compared with >, >=, <, <=.

```
SQRT(-1.)
```
gives error or NaN

```
SQRT((-1.,0))
```
gives (0.,1.)

Example: Finding the roots of a quadratic function

With COMPLEX

```
PROGRAM roots
IMPLICIT NONE
REAL :: a, b, c, discriminant
COMPLEX :: x1, x2

PRINT*, 'Enter the coefficients a, b, and c:'
READ*, a, b, c

discriminant = b**2 - 4. * a * c
x1 = (-b + SQRT(CMPLX(discriminant,0.))) / (2*a)
x2 = (-b - SQRT(CMPLX(discriminant,0.))) / (2*a)

PRINT*, 'The roots are: '
PRINT*, 'x1 = ', REAL(x1), ' +i ', AIMAG(x1)
PRINT*, 'x2 = ', REAL(x2), ' +i ', AIMAG(x2)

END PROGRAM roots
```

Without COMPLEX

```
PROGRAM roots
IMPLICIT NONE
REAL :: a, b, c, discriminant, imag_part, real_part, x1, x2

PRINT*, 'Enter the coefficients a, b, and c:'
READ*, a, b, c

discriminant = b**2 - 4. * a * c
IF ( discriminant < 0. ) THEN ! there are complex roots
  real_part = ( -b ) / ( 2. * a )
  imag_part = sqrt ( abs ( discriminant ) ) / ( 2. * a )
  PRINT*, 'This function has complex roots:'
  PRINT*, 'x1 = ', real_part, ' +i ', imag_part
  PRINT*, 'x2 = ', real_part, ' -i ', imag_part
ELSE IF ( discriminant == 0. ) THEN ! there is one repeated root
  x1 = ( -b ) / ( 2. * a )
  PRINT*, 'This function has two identical real roots:'
  PRINT*, 'x1 = x2 = ', x1
ELSE ! there are two real roots
  x1 = ( -b + sqrt(discriminant) ) / ( 2. * a )
  x2 = ( -b - sqrt(discriminant) ) / ( 2. * a )
  PRINT*, 'This function has two distinct real roots:'
  PRINT*, 'x1 = ', x1
  PRINT*, 'x2 = ', x2
END IF
END PROGRAM roots
```

Summary

- Characters are encoded with a character set and stored in the computer as binary numbers. The UTF-8 encoding with the ASCII character set is used most commonly.

- Besides simple indices, parts of an array can also be accessed with subscript triplets, vector subscripts, or with WHERE, FORALL and DO CONCURRENT.

- Complex numbers are declared with the COMPLEX statement and consist of two REALS corresponding to the real and imaginary parts.