

# Abstrakte Datentypen

## Sonderübung

Robin Flemming & Jonas Riedel

PR10

31.01.2023

1. ARRAYS
2. Felder in Unterprogrammen
3. Übungsaufgabe
4. ADT - Abstrakte Datentypen
5. Übungsaufgabe
6. Schnittstellen (INTERFACE-Blöcke)
7. Übungsaufgabe

# ARRAYS (Felder)

- homogene Datenstruktur (alle Elemente haben denselben Typ)
- ein- oder mehrdimensional (Vektor, Matrix, Tensor, ...)
- Feldtyp charakterisiert durch **Elementtyp** (bel. skalarer Typ) und **Rang** (bzw. rank) = Anzahl der Dimensionen ( $\leq 15$ )
- entweder **statisch** oder **dynamisch**

# Deklaration statischer ARRAY-Variablen

```
< Elementtyp >, DIMENSION ([ $l_1$  :]  $u_1$ , ... , [ $l_r$  :]  $u_r$ ) :: my_array
```

$r$  - rank

## Beispiel

```
INTEGER, DIMENSION (5) :: vector
```

! Vektor der Länge 5

```
INTEGER, DIMENSION (1:4, 1:4) :: matrix
```

!  $4 \times 4$  - Matrix

# Zugriff auf Feldeinträge und Feldkonstruktor

```

INTEGER, DIMENSION (5) :: vector           ! Vektor der Länge 5
INTEGER, DIMENSION ( 1 : 4, 1 : 4) :: M      ! 4 × 4 - Matrix
INTEGER, DIMENSION ( 1 : 2, 1 : 2) :: Teilfeld ! 2 × 2 - Matrix
INTEGER :: Feldeintrag

```

**Ausdehnung:**

- Untergrenze (Default-Untergrenze: 1)
- Obergrenze

```

Feldeintrag=M(2,3)      ! Zugriff auf einzelne Feldeinträge
Teilfeld=M(1 : 2, 1 : 2) ! Zugriff auf Teilfelder

```

- In Fortran **beginnen Arrays (per default) immer mit 1!**
- Ein 1-dimensionales Array (entspricht Vektor) kann man mit dem Feldkonstruktor vorbelegen:

```
INTEGER,DIMENSION(1:3):: A
```

```
A=(/1,2,3/)
```

# Vordefinierte Funktionen

INTEGER, DIMENSION (5) :: vector	! Vektor der Länge 5
INTEGER, DIMENSION (1 : 4, 1 : 4) :: M	! 4 × 4 - Matrix

## Standardfunktionen:

LBOUND (M, i)	$=: l_i$	Untergrenze in der i-ten Dimension
UBOUND (M, i)	$=: u_i$	Obergrenze in der i-ten Dimension
LBOUND (M)	$= (/l_1, l_2, \dots, l_r/)$	liefert Vektor
UBOUND (M)	$= (/u_1, u_2, \dots, u_r/)$	
SIZE (M, i)	$= u_i - l_i + 1 =: d_i$	Anz. Indexwerte in der i-ten Dimension
SIZE (M)	$= \prod_{i=1}^r \text{SIZE (M, i)}$	
SHAPE (M)	$= (/d_1, d_2, \dots, d_r/)$	

# Deklaration dynamischer ARRAY-Variablen

```
< Elementtyp >, DIMENSION ( : , : , ... , : ), ALLOCATABLE :: my_array
```

## Beispiel

```
INTEGER, DIMENSION (:), ALLOCATABLE :: dyn_vec  
INTEGER, DIMENSION ( : , : ), ALLOCATABLE :: dyn_mat, A, B
```

Dynamische Felder müssen **allokiert** werden. D. h. die Gestalt des Feldes wird festgelegt und entsprechend Speicherplatz beschafft.

```
ALLOCATE (dyn_mat(4, 4), dyn_vec(0 : 6))
```

Das Feld wird **deallokiert**, d. h. der Speicherplatz wird wieder freigegeben.

```
DEALLOCATE (dyn_mat, dyn_vec)
```

# Vordefinierte Funktionen

INTEGER, DIMENSION ( 3 , 4 ) :: M

! 2 - dim. Feld

SUM (M, i), i = 1, 2

Vektor der Spalten- bzw. Zeilensummen

$$\text{SUM (M)} = \sum_i \sum_j m_{ij}$$

Summe aller Matrixeinträge

PRODUCT (M, i), i = 1, 2

Vektor der Spalten- bzw. Zeilenprodukte

$$\text{PRODUCT (M)} = \prod_i \prod_j m_{ij}$$

Produkt aller Matrixeinträge

MINVAL (M, i)

kleinster Eintrag

MAXVAL (M, i)

größter Eintrag

MINLOC (M, i)

Position des kleinsten Eintrags

MAXLOC (M, i)

Position des größten Eintrags



# Vordefinierte Funktionen

INTEGER, DIMENSION ( 4 , 4 ) :: M, A, B

LOGICAL, DIMENSION ( 2 , 3 ) :: L

INTEGER, DIMENSION ( 4 ) :: v, w

TRANSPOSE (M)                      Liefert transponierte Matrix (für 2-dim. Felder!)

MATMUL (A, B)                      =  $A \cdot B$

MATMUL (A, w)                      =  $A \cdot w$

MATMUL (v, B)                      =  $v^T \cdot B$

DOT\_PRODUCT (v, w)      Skalarprodukt ( $\sum_i v_i \cdot w_i$ )

Für LOGICAL-Felder:

ANY (L [, i ])      Logisches ODER (.TRUE., falls **ein** Eintrag .TRUE.)

ALL (L [, i ])      Logisches UND (.TRUE., falls **alle** Einträge .TRUE.)

# Vordefinierte Funktionen

- alle intrinsischen Operatoren ( $+$ ,  $-$ ,  $*$ ,  $/$ , ...) und die meisten intrinsischen Funktionen (z. B. ABS) auf Felder anwendbar (sofern der Elementtyp stimmt)
- dabei wird **eintragsweise** gerechnet
- ACHTUNG: Operanden (bei binären Operatoren) müssen gestaltkonform sein
- Skalare sind mit bel. Feldern gestaltkonform

z. B.  $3 + v$ ,  $v + w$ ,  $v * w$ ,  $v == w$ , ANY ( $v == w$ ), ...

# Felder in Unterprogrammen

## Felder als UP-Parameter

- statisches Feld oder
- Feld übernommener Gestalt: Gestalt zum Aufrufzeitpunkt des UPs durch a. A. festgelegt

```
FUNCTION mult (A, v)  
  INTEGER, DIMENSION ( : , : ) :: A  
  INTEGER; DIMENSION (0 : ) :: v  
  ...
```

# Felder in Unterprogrammen

## Felder als Funktionsergebnis

Indexgrenzen müssen zum Aufrufzeitpunkt des UPs berechenbar sein. Möglich: Ausdrücke, die von d. UP-Parametern abhängen, z. B.

```
FUNCTION mult (A, v)
  INTEGER, DIMENSION ( : , : ) :: A
  INTEGER, DIMENSION ( 0 : ) :: v
  INTEGER, DIMENSION ( 1 : SIZE (A, 1)) :: mult
```

# Felder in Unterprogrammen

## Automatische Felder

- lokale Variablen des UPs
- Indexgrenzen müssen zum Aufrufzeitpunkt des UPs berechenbar sein, können von d. UP-Parametern abhängen

```

FUNCTION mult (A, v)
  INTEGER, DIMENSION ( : , : ) :: A
  INTEGER, DIMENSION ( 0 : ) :: v
  INTEGER, DIMENSION ( 1 : SIZE (A, 1)) :: mult
  INTEGER, DIMENSION ( SIZE (A, 2), SIZE (A, 1) ) :: T
  ...
  T = TRANSPOSE (A)
  ...

```

# Aufgabe: Museumsausstellung

Vom 14. Januar (Montag) bis zum 27. Januar (Sonntag) hat eine neue Ausstellung im Museum geöffnet. Die Besucherzahlen pro Tag sollen zusammen mit dem Datum (ohne Monat) gespeichert werden. Öffne die Datei 'Besucherzahlen\_Januar.txt' und lies die Anzahl der Besucher ein.

- Das Personal wurde ausversehen mitgezählt, ziehe deswegen an jedem Tag 5 von der Besucheranzahl ab.
- Gib den Tag mit den meisten bzw. den wenigsten Besuchern kommentiert mit Datum und Anzahl aus.
- Speichere in einem 2x2-Array die Besucheranzahl von den zwei Wochenenden ab.
- Es lohnt sich erst ab 20 Besuchern zu öffnen, gib aus, ob am Wochenende immer mehr als 20 Personen da waren.

# Abstrakte Datentypen (auch: benutzerdefinierte Typen)

- definiert heterogene Datenstruktur
- heterogen bedeutet: Komponenten können unterschiedliche Typen haben

## Allgemeine Syntax

```
TYPE < Typname >
```

```
  [PRIVATE]
```

```
    < Typvereinbarung für Komponenten >
```

```
END TYPE
```

# Abstrakte Datentypen (auch: benutzerdefinierte Typen)

## Beispiel

```
TYPE fraction
  INTEGER :: nominator
  INTEGER :: denominator
END TYPE
```

Die Komponententypen können verschieden sein:

```
TYPE bike
  CHARACTER(15) :: brand
  INTEGER :: size
  ...
END TYPE
```



# Nutzen von ADT

- möglichst viel von der inneren Struktur verstecken
- Informationen, die zusammen gehören, aber unterschiedliche Datentypen haben, an einem Ort speichern (Zum Beispiel wird jeder Student über Name, s-Nummer und Matrikelnummer gekennzeichnet. Diese Infos sollten daher zusammen gespeichert werden.)
- erstellen von Datentypen, die in der Praxis gebraucht werden, aber nicht in F95 vorhanden sind (z.B. Rationale Zahlen, Intervallarithmetik, ...)

# Komponentenzugriff

**Problem:** Es gibt fast keine intrinsischen (d.h. vordefinierte) Funktionen dazu  
→ Alle gewünschten Funktionalitäten müsst ihr selbst programmieren!

Dazu benötigt ihr den **Komponentenzugriff**:

- die einzelnen Komponenten besitzen intrinsische Datentypen (d.h. Real, Integer, ...) und für diese gibt es intrinsische Funktionen  
→ ihr könnt damit arbeiten
- Allgemeine Syntax: **<NameDesADT> % <NameDerKomponenten>**

# Beispiel für Komponentenzugriff

## Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER :: Vorname, Name
END TYPE
```

...

```
FUNCTION aelter (a,b)
  TYPE(Student) :: a,b
  LOGICAL :: aelter

  aelter = a%Alter > b%Alter
END FUNCTION
```

# Typkonstruktor

Um im Programmablauf Variablen eures ADTs zu definieren, ohne sie einlesen zu müssen, eine selbstgeschriebene Funktion zu verwenden oder sie komponentenweise belegen zu müssen, könnt ihr den Typkonstruktor verwenden.

Allgemeine Syntax:  $\langle \text{NameDesADT} \rangle (\langle \text{Komponente1} \rangle, \langle \text{Komponente2} \rangle, \dots)$

## Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER(8) :: Vorname, Name
END TYPE
```

```
TYPE(Student) :: MaxMuster
```

```
MaxMuster = Student(20,'Max','Muster')
```

# Aufgabe: Tierheim

Von einem Tierheim sollen die Tiere mit folgenden Daten gespeichert werden: Name (max.20 Zeichen), ganzzahliges Alter, Größe/Länge in cm (reellwertig) und ein logischer Wert, ob bereits ein neuer Besitzer gefunden wurde. Legt ein dynamisches Array 'tierheim' an, was diese Daten speichern kann. Aus der Datei "Tierheim.txt" sollen dann die Informationen eingelesen werden.

Jetzt soll folgendes gemacht werden:

- Hund Bello hat Geburtstag, erhöhe sein Alter
- lies von der Tastatur eine Zahl ein, und gib dann die zugehörigen Daten des Tieres mit der ID aus
- gib alle Tiere, die älter als 5 und jünger als 15 Jahre sind, kommentiert mit Namen und Alter aus
- Pippi Langstrumpf möchte sich ein Tier aussuchen, gib eine Liste mit den Namen der Tiere aus, die noch keinen Besitzer haben

- Pippi Langstrumpf hat sich für Herr Nilsson(ID=18) entschieden, ändere seinen Status
- die Tiere sind im letzten Jahr im Schnitt um 10 Prozent gewachsen, erhöhe dementsprechend die Größe
- lies von der Tastatur einen neuen Bewohner des Tierheims ein und speichere ihn mit in das Array (zwischenspeichern und neu allokalieren)
- gib noch einmal alle Daten zur Überprüfung aus

# Schnittstellen (INTERFACE-Blöcke)

## Generische Schnittstellen und Überladung

- Überladung = mehrfache Verwendung von Prozedurnamen bzw. Operatornamen/ -symbolen (+, −, \*, ..., auch: =) für verschiedene Prozeduren
- jede Überladung muss eindeutige **Schnittstelle** haben
- Bei der Überladung eines Operators müssen die Funktionsargumente immer das Attribut Intent(In) haben!



# Generische Schnittstellen und Überladung

```
INTERFACE < generi_name >
    MODULE PROCEDURE <function_1> [, ... , <function_i>]
                        bzw. <subroutine_1> [, ... , <subroutine_i>]
END INTERFACE
```

```
INTERFACE OPERATOR (+)                oder: −, *, /, <, .NOT., ... , .MYOPE.
    MODULE PROCEDURE <function_1> [, ... , <function_i>]

END INTERFACE
```

```
INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE <subroutine_1> [, ... , <subroutine_i>]
END INTERFACE
```

# Generische Schnittstellen und Überladung - ein Beispiel

```

MODULE shifted
  IMPLICIT NONE
  ...
  INTERFACE OPERATOR (.S.)
    MODULE PROCEDURE SHIFT
  END INTERFACE
  ...

CONTAINS

  FUNCTION shift (wort, anzahl)
    CHARACTER(*), INTENT(IN) :: wort
    INTEGER, INTENT(IN) :: anzahl
    CHARACTER(LEN(wort)) :: shift

    shift = ' '
    shift (anzahl + 1 : len(wort)) = wort(1 : len(wort) - anzahl)
    shift (1 : anzahl) = wort(len(wort) - anzahl + 1 : len(wort))
  END FUNCTION

  ...
END MODULE

```

# Aufgabe: Fliegen

Schreibe ein Modul mit einem Datentyp, der für einen Flug den Startflughafen, den Zielflughafen, die Dauer und die Distanz speichert. Überlade dann den Operator '+', der einen Flug mit der Gesamtzeit und Gesamtstrecke, sowie dem Start- und Zielflughafen von den beiden Flügen zurückgibt. Hierbei soll der Start und das Ziel entsprechend gespeichert werden, wenn direktes Weiterfliegen möglich ist. Ansonsten soll der Start zwar übernommen werden, aber im Ziel soll 'noch gesucht' gespeichert werden. Gib in diesem Fall noch eine Warnung aus. Teste das ganze im Hauptprogramm für zwei Eingaben.