Studiengang Software Engineering
Fakultät für Informatik, Kommunikation, Medien
Campus Hagenberg

3 Datentypen

3.1 Überblick und Begriffe

implizit: durch Angabe des Wertes wird der Datentyp automatisch festgelegt

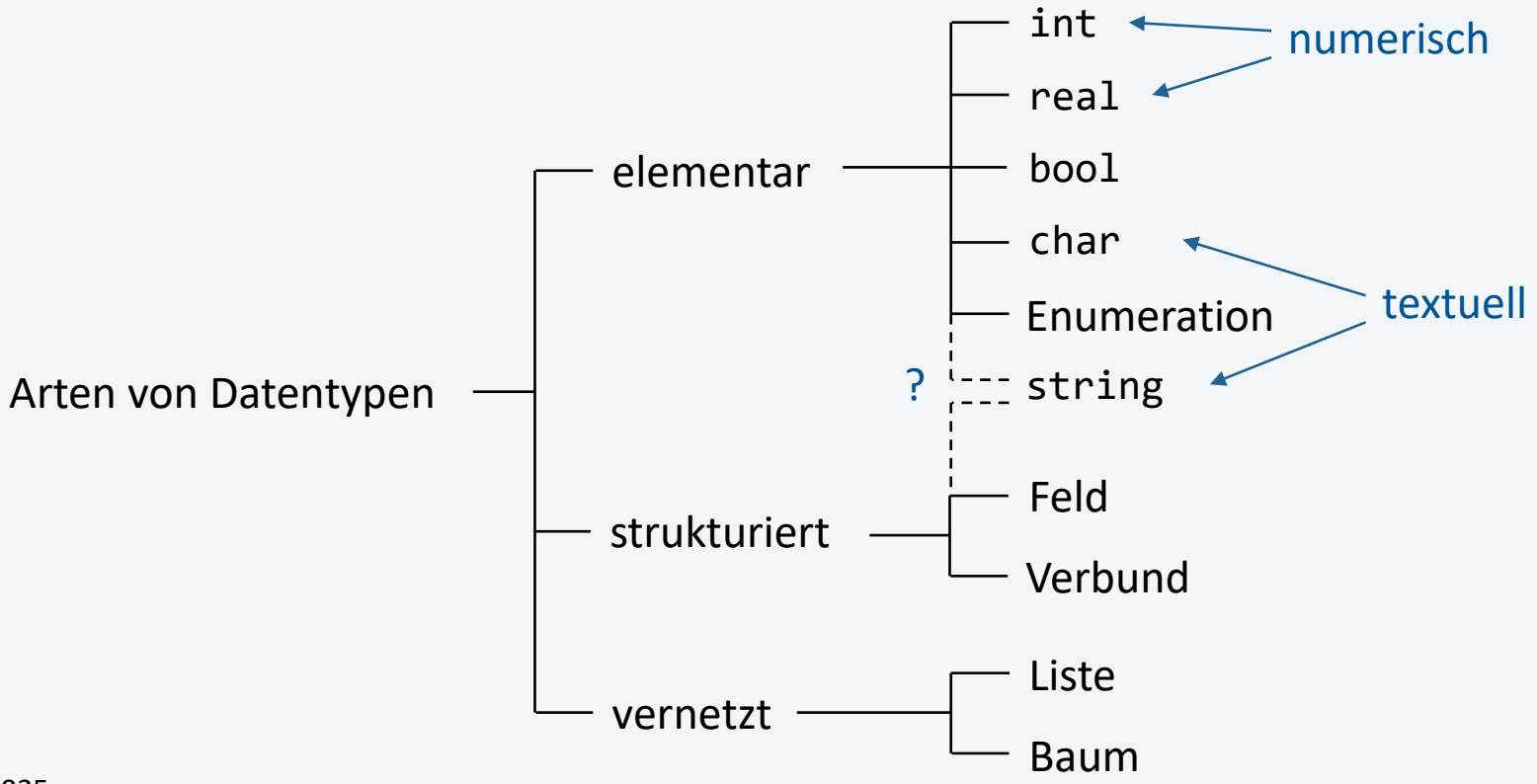
- Jedem Datenobjekt ist (implizit oder explizit) ein Datentyp zugeordnet
- Datentypen definieren
  - den Wertebereich (ggf. die Genauigkeit) von Datenobjekten und
  - die Operationen, die mit den Datenobjekten erlaubt sind
- Wir unterscheiden elementare, strukturierte und vernetzte Datenobjekte
- Wir unterscheiden konstante und variable Datenobjekte (Konstante/Variable)
- Wir legen fest, dass Datenobjekte deklariert (definiert) werden müssen

(= bei uns keine Unterscheidung)

```
var
  i: int
  x, y: real
  isPrime: bool
  name: string
```

Datentypen – Übersicht

- Elementare Datenobjekte sind nicht weiter zerlegbar
- Strukturierte Datenobjekte setzten sich aus elementaren oder strukturierten Objekten zusammen
- Vernetzte, dynamische Datenobjekte können wachsen und schrumpfen



3.2 Elementare Datentypen

Bisher bereits bekannte elementare Datentypen

Integer (`int`)	Real (`real`)	Character (`char`)	Boolean (`bool`)
ganze Zahlen	Gleitkommazahlen	druckbare Zeichen	Wahrheitswerte
`=,<,≠,…`	`=,<,≠, …`	`=,<,≠, …`	`=,<,≠, …`
`+, -, *, div, mod`	`+, -, *, /`	`Int(), Char()`	`and, or, xor, not`

In Pascal z.B.:

- `INTEGER, REAL, CHAR, BOOLEAN`
- `SHORTINT, INTEGER, LONGINT, BYTE, WORD`
  8 Bit (Pascal) (Bei Pascal 16 Bit) 32 Bit

Aufzählungstyp (Enumerationstyp)

Der Wertebereich wird durch Aufzählung der möglichen Werte individuell festgelegt

Dadurch kann eine Variable auch nur diesen Werten zugeordnet werden.

Form: `type T = (e1, e2,...,en)`

Beispiele

```
type
  Shape = (circle, oval, rectangle, square)
  Gender = (female, male, divers)
  DayOfWeek = (monday, tuesday, wednesday, thursday, friday, saturday, sunday)
```

Aufzählungstypen

- erweitern das Typsystem (führt zu neuen Typen wie z.B. `Shape`, `Gender` etc.)
- ermöglichen neue Konstantenwerte (wie z.B. `rectangle`, `female`, `monday`)

Ich kann auch
type Num = int

Verwendung von Aufzählungstypen

Deklaration von Variablen

```
var
  s: Shape
  g: Gender
  d: DayOfWeek
```

Zuweisung von Werten

```
s := circle
g := divers
d := friday
```

Vergleich

```
if d = friday then
  Write(↓"Vorlesung")
end
if d > friday then
  Write(↓"Wochenende")
end
```

zum Vergleich wird die Reihenfolge der Typendefinition verwendet.
Deshalb ist Samstag > Freitag

Außer Zuweisung und Vergleich sind keine Operationen erlaubt

Beispiele

Verwendung eines Datenobjekts Schachfigur ohne Aufzählungstyp

```
-- 1..queen, 2..king, 3..rook, 4..bishop, 5..knight, 6..pawn
var
  p: int
```

```
p := 6
```

Bauer

```
p := 42
```

Fehler! Leider möglich.

Konsequenzen

- Keine Typsicherheit ungültige Werte sind zuweisbar - keine Typsicherheit
- Auch die Verwendung von Zeichen oder Zeichenketten statt *Integer* führt zum gleichen Problem
- Auch die Verwendung von Konstanten führt zum gleichen Problem

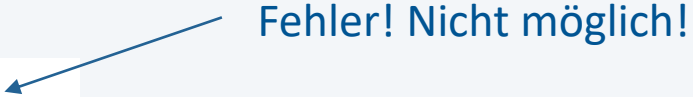
```
const
  queen: int = 1
```

Beispiele

Verwendung eines Datenobjekts Schachfigur mit Aufzählungstyp

```
type
  ChessPiece = (queen, king, rook, bishop, knight, pawn)
var
  p: ChessPiece
```

```
p := pawn
```




```
p := 42
```

Vorteile des Einsatzes von Aufzählungstypen:

- Bessere Lesbarkeit (pawn statt 6)
- Typsicherheit: unerlaubte Zuweisungen und Operationen werden zur Übersetzungszeit erkannt

Typen für strukturierte Datenobjekte

Motivation

- Oft ist es nützlich und sinnvoll, mehrere Datenobjekte zu einem „logischen Ganzen“ zusammenzufassen, so dass solche Datenobjekte sowohl als Ganzes manipuliert (z.B. ihr Inhalt ausgegeben) werden können, als auch, dass auf ihre Komponenten/Elemente zugegriffen und diese einzeln manipuliert werden können.

Beispiel

- „Drucke Name und Geburtsdatum aller Studierenden der LVA“

Die wichtigsten Datentypen für strukturiere Datenobjekte sind:

- Felder (`array`) und
- Verbunde (`compound`)

Strukturiert: 'Wert, der weiter zerlegbar ist'

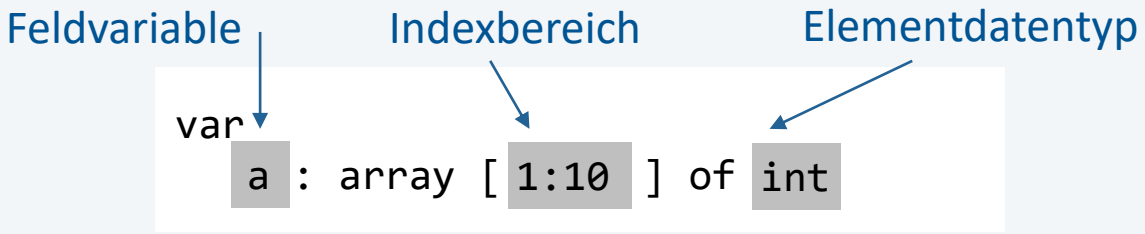
3.3 Felder (*Arrays*)

Ein Feld (`array`) ist eine Sammlung von Elementen, die zusammen ein logisches Ganzes bilden und alle vom gleichen Datentyp sind, ein Index identifiziert die Elemente eindeutig.

```
var
  a: array [first:last] of ElementType
```

Beispiel: Deklaration einer Feldvariablen mit 10 `int`-Elementen:

Vorteil: 1. und letzter Wert definiert



- Indexbereich mit Unter- und Obergrenze (`1:10`)
- Elementdatentyp (`int`)
- geordnete Anordnung der Elemente keine Menge, Position relevant
- der Begriff Feldlänge bezeichnet die Anzahl der Elemente

Deklaration

```
const
  n: int = ...
```

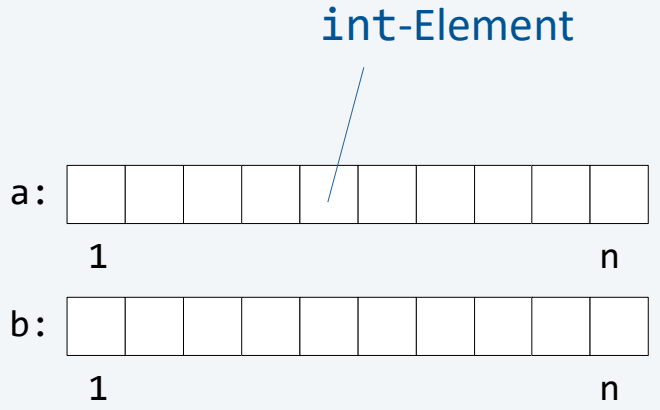
Beispiel (mit expliziter Typdeklaration)

gut für Parameterlisten von Algorithmen

```
type
  IntArray = array [1:n] of int
var
  a, b: IntArray
```

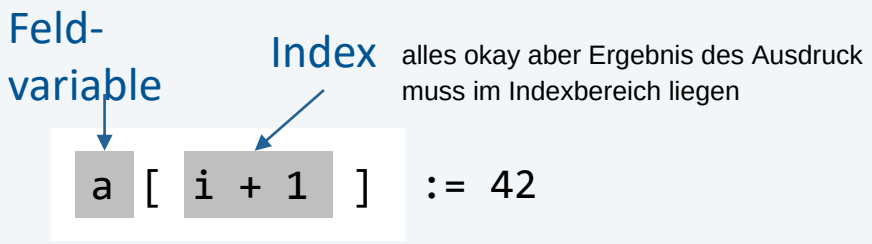
Beispiel (ohne expliziten Typ)

```
var
  a, b: array [1:n] of int
```



Eindimensionale Felder

```
var
   a, b: array [1:10] of int
   i: int
```

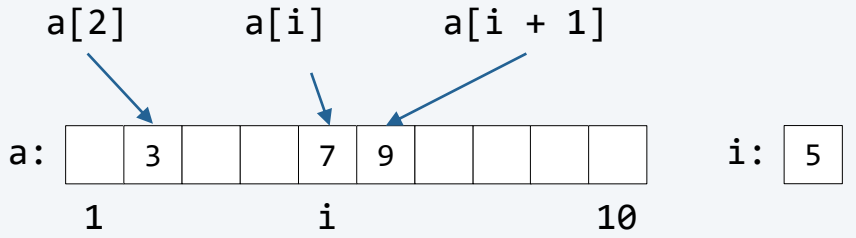


Zugriff auf ein Datenelement

- Index kann durch ein Literal, eine Variable oder einen Ausdruck gebildet werden

```
a[2] := 3
i := 5
a[i] := 7
a[i + 1] := 9
a[11] := 3 -- error
```

Fehler erst zur Laufzeit!

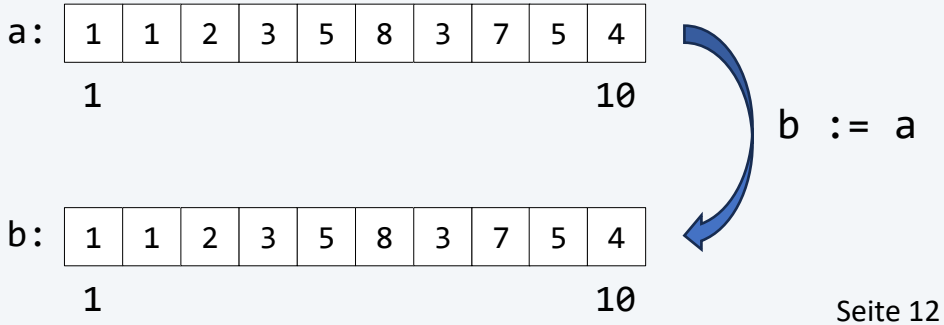


Wertzuweisung bei Feldern

```
b := a
```

kopiert Feldinhalt von a nach b!

nur bei gleichen Elementtypen und gleichen Feldlängen möglich!



Typische Anweisungen für eindimensionale Felder

Deklarationen

immer const verwenden

```
const
  n: int = 100
var
  a, b: array [1:n] of int
  i: int
```

Summe der Elemente a[1], …, a[n]

```
sum := 0
for i := 1 to n do
  sum := sum + a[i]
end
```

Initialisierung der Elemente a[1], …, a[n]

```
for i := 1 to n do
  a[i] := 0
end
```

Initialisieren mit Fibonacci-Folge

```
a[1] := 1          n > 1
a[2] := 1
for i := 3 to n do
  a[i] := a[i - 1] + a[i – 2]
end
```

Ausgabe der Elemente a[1], …, a[n]

```
for i := 1 to n do
  Write(↓a[i])
end
```

Kopieren der Elemente von a nach b in umgekehrter Reihenfolge

```
for i := 1 to n do
  b[i] := a[n – i + 1]
end
```

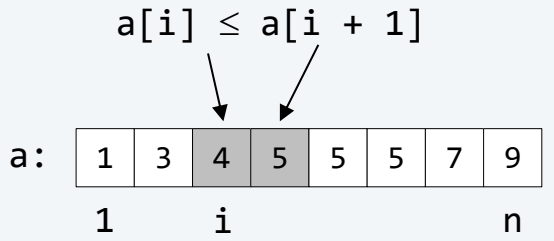
Beispiel

Gesucht ist ein Algorithmus, der prüft, ob die Elemente eines Felds a aufsteigend sortiert sind

```
const
  n: int = ... -- mit 1 ≤ n
```

```
IsSorted(↓a: array[1:n] of int): bool
  var
    i: int
begin
  i := 1
  while (i < n) and (a[i] ≤ a[i + 1]) do
    i := i + 1
  end
  return i = n
end IsSorted
```

bei n=1 kein Problem, weil 1<1= false, sofern der rechte Operand nicht mehr ausgewertet wird, weil es das Element a[i+1] nicht geben würde in den meisten Programmiersprachen ist das so.



würde auch mit if Verzweigung gehen, aber höhere Strukturkomplexität

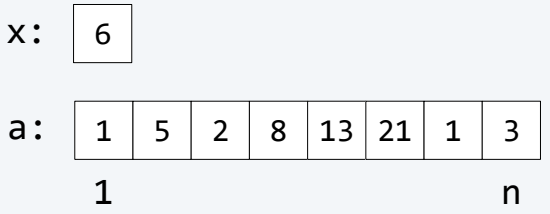
+ man springt nicht aus einer for-Schleife mit return raus!
---wegen Lesbarkeit

Beispiel

Gesucht ist ein Algorithmus, der kleinsten Index `i` sucht, so dass `a[i] = x`

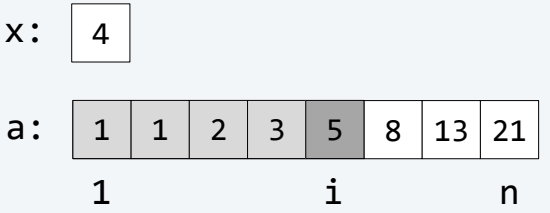
- Feld durchlaufen bis Element x gefunden oder Feldende erreicht ist

```
Search(↓a: array[1:n] of int ↓x: int ↑i: int)
begin
  i := 1
  while (i ≤ n) and (a[i] ≠ x) do
    i := i + 1
  end
  if i > n then
    i := -1 -- not found
  end
end Search
```



- Wenn Feld aufst. sortiert ist, kann Suche enden, wenn a[i] > x ist:

```
while (i ≤ n) and (a[i] < x) do
    …
if (i > n) or (a[i] ≠ x) then
  i := -1
end
```



Beispiel: Datenstruktur für Kartendeck mit 4×13 Karten

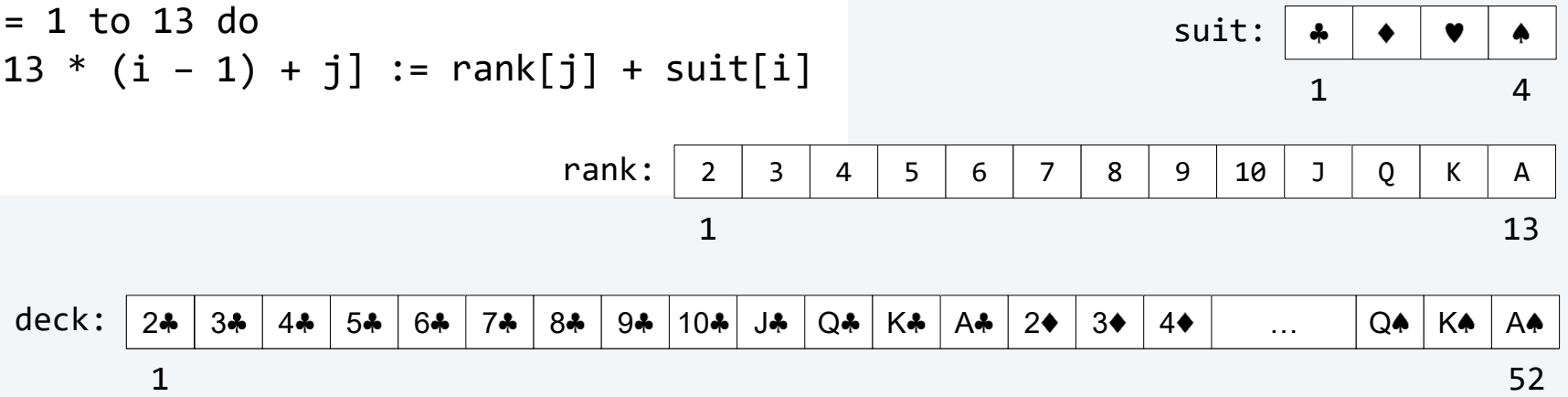
verwende drei Felder

```
var
  rank: array [1:13] of string
  suit: array [1:4] of string
  deck: array [1:52] of string
```

```
rank := ("2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A")
suit := ("♣", "♦", "♥", "♠")
```

initialisiere Datenstruktur für Kartendeck

```
for i := 1 to 4 do
  for j := 1 to 13 do
    deck[13 * (i – 1) + j] := rank[j] + suit[i]
  end
end
```



Beispiel: Erstelle Kartendeck mit 4×13 Karten

Ausgabe

```
for i := 1 to 52 do
  Write(↓deck[i] ↓' ')
end
```

```
2♣ 3♣ 4♣ 5♣ 6♣ 7♣ 8♣ 9♣ 10♣ J♣ Q♣ K♣ A♣
2♦ 3♦ 4♦ 5♦ 6♦ 7♦ 8♦ 9♦ 10♦ J♦ Q♦ K♦ A♦
2♥ 3♥ 4♥ 5♥ 6♥ 7♥ 8♥ 9♥ 10♥ J♥ Q♥ K♥ A♥
2♠ 3♠ 4♠ 5♠ 6♠ 7♠ 8♠ 9♠ 10♠ J♠ Q♠ K♠ A♠
```

Was passiert, wenn die Reihenfolge der `for`-Schleifen vertauscht wird?

```
for i := 1 to 4 do
  for j := 1 to 13 do
    deck[13*(i - 1) + j] := …
  end
end
```

```
for j := 1 to 13 do
  for i := 1 to 4 do
    deck[13*(i - 1) + j] := rank[j] + suit[i]
  end
end
```

Die Feldelemente werden in einer anderen Reihenfolge initialisiert, das Ergebnis ist aber das gleiche.

Beispiel: Anzahl der Tage im Monat

Gesucht ist ein Algorithmus, der die Anzahl der Tage im Monat (1 – 12) ermittelt (ohne Berücksichtigung von Schaltjahren)

Algorithmus:

```
DaysOfMonth(↓month: int): int
  var d: int
begin
  case month of
     1: d := 31
     2: d := 28
     3: d := 31
     4: d := 30
     5: d := 31
     6: d := 30
     7: d := 31
     8: d := 31
     9: d := 30
    10: d := 31
    11: d := 30
    12: d := 31
    otherwise: d := 0
  end
  return d
end DaysOfMonth
```

Verwendung:

```
var
  i: int
for i := 1 to 13 do
  WriteLn(↓i ↓' ' ↓DaysOfMonth(↓i))
end
```

Ausgabe:

```
1 31
2 28
3 31
4 30
5 31
6 30
7 31
8 31
9 30
10 31
11 30
12 31
13 0
```

Beispiel: Anzahl der Tage im Monat

Gesucht ist ein Algorithmus, der die Anzahl der Tage im Monat (1 – 12) ermittelt (ohne Berücksichtigung von Schaltjahren)

Lösung mit Feld

```
DaysOfMonth(↓month: int): int
  var
    days: array [1:12] of int
begin
  days := (31, 28, 31, 30, 31, 30,
           31, 31, 30, 31, 30, 31)
  if (1 ≤ month) and (month ≤ 12) then
    return days[month]
  else
    return 0
  end
end DaysOfMonth
```

array befüllen, mit schleife z.B.

Verwendung:

```
var
  i: int
for i := 1 to 13 do
  WriteLn(↓i ↓' ' ↓DaysOfMonth(↓i))
end
```

Ausgabe:

```
1 31
2 28
3 31
4 30
5 31
6 30
7 31
8 31
9 30
10 31
11 30
12 31
13 0
```

Algorithmen mit nicht vollständig gefüllten Feldern

- Anzahl der belegten Elemente `n` kann verändert werden
- Nur die Elemente `a[1]`, `a[2]`,..., `a[n]` werden betrachtet
- Typische Schnittstelle

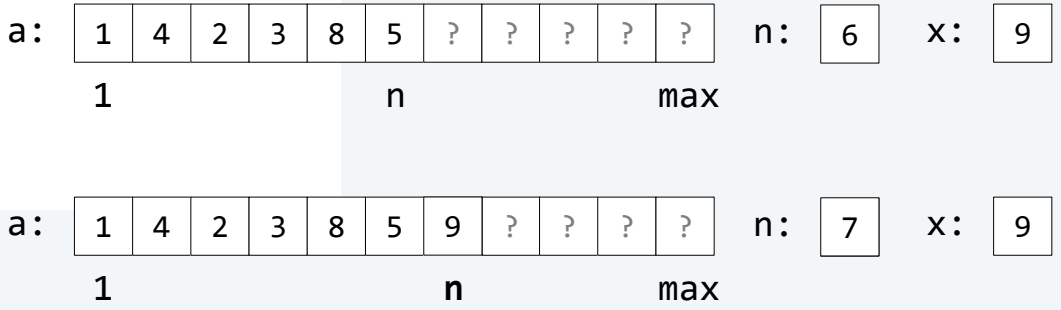
```
F(↓↑a: array[1:max] of int  ↓↑n: int ...)
```

```
const
  max =
```

- Beispiel: Element `x` hinzufügen (Vorbedingung: `1 ≤ n < max`)

n<max --> es hat noch 1 Element Platz

```
Append(↓↑a: array[1:max] of int ↓↑n: int ↓x: int)
begin
  if (1 ≤ n) and (n < max) then
    a[n + 1] := x
    n := n + 1
  end
end Append
```



Einfügen in sortierten Feldern

sinnvoll, weil danach das array besser durchsucht werden kann

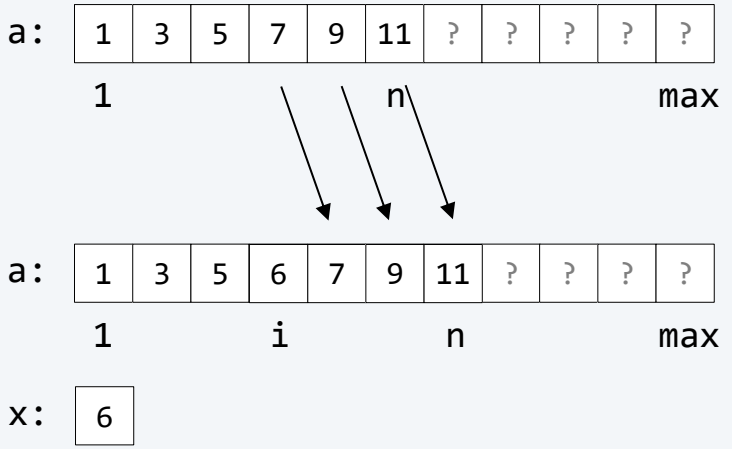
Lösungsidee

- Zuerst muss die Position gefunden werden, an der das neue Element x eingefügt werden soll
- Dann müssen alle der Einfügeposition nachfolgenden Elemente um eine Position nach rechts verschoben werden

Algorithmus (Grobstruktur):

```
Insert(↓↑a: array[1:max] of int ↓↑n: int ↓x: int)
  var
    i: int
begin
  i := search for insert position
  move elements a[i], ... a[n]
  a[i] := x
  n : = n + 1
end Insert
```

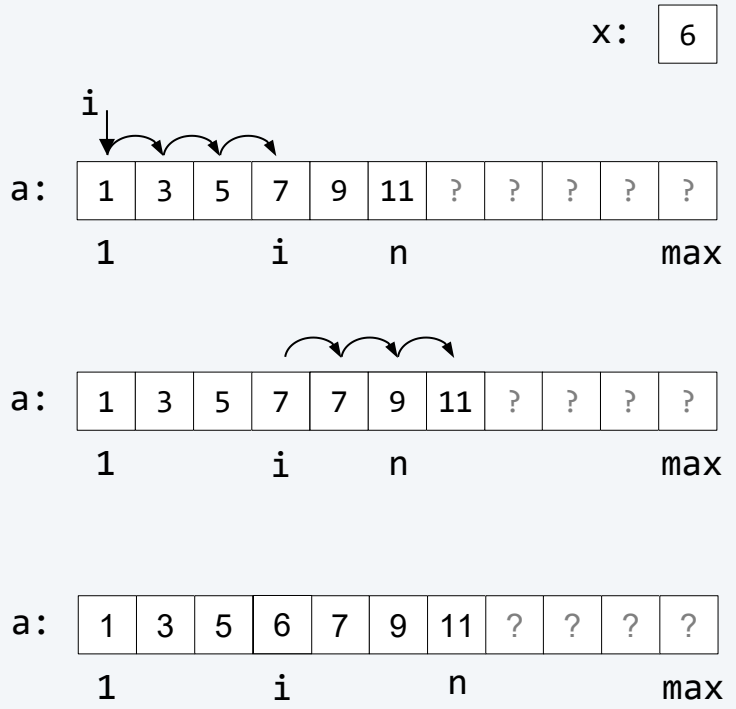
Pseudo-Anweisungen



Einfügen in sortierten Feldern

Algorithmus

```
Insert(↓↑a: array[1:max] of int ↓↑n: int ↓x: int)
  var i: int
begin
  -- search for insert position
  i := 1
  while (i ≤ n) and (a[i] ≤ x) do
    i := i + 1
  end
  -- move elements a[i], ... a[n]
  for k := n + 1 downto i + 1 do
    a[k] := a[k - 1]
  end
  -- x einfügen
  a[i] := x
  n : = n + 1
end Insert
```

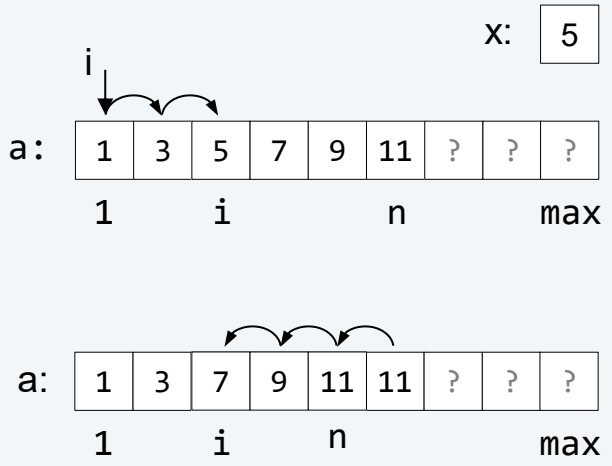


Entfernen eines Elements

Zuerst muss die Position gefunden werden, an der das zu entfernende Element x (erstmals) vorkommt

Dann müssen alle rechts von dieser Position liegenden Elemente um eine Stelle nach links verschoben (Anmerkung: letztes Element bleibt bestehen)

```
Remove(↓↑a: array[1:max] of int ↓↑n: int ↓x: int)
  var i: int
begin
  i := 1
  while (i ≤ n) and (a[i] ≠ x) do
    i := i + 1
  end
  for k := i to n - 1 do
    a[k] := a[k + 1]
  end
  if i ≤ n then
    n := n – 1
  end
end Remove
```

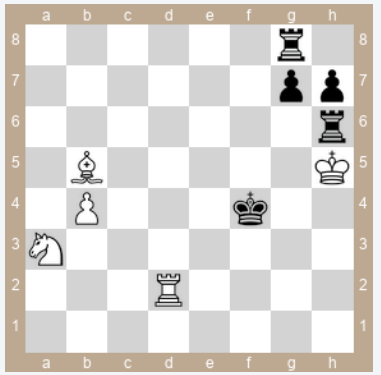


Zwei- und mehrdimensionale Felder

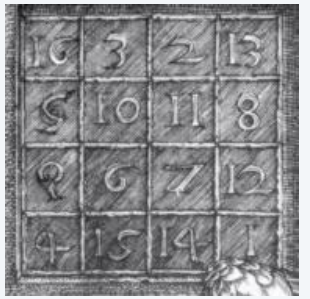
- In vielen Aufgabenstellungen finden wir zweidimensionale Daten (Tabellen)

Beispiele

- Schachbrett
- Matrizen
- Entscheidungstabellen
- …



$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Lösung: der Elementtyp eines Felds kann wiederum ein Feldtyp sein

```
type
  Chessboard = array [1:n] of array [1:n] of ChessPiece
```

oder einfacher:

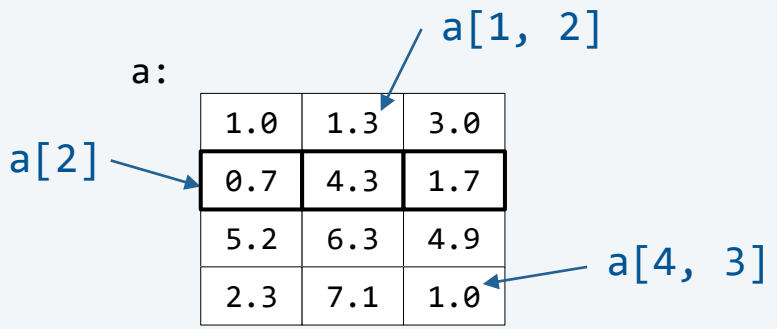
```
type
  Chessboard = array [1:n, 1:n] of ChessPiece
```

Zugriff auf Feldelemente

```
var
  a: array [1:4, 1:3] of real
```

Zugriff auf Feldelemente mittels Literal

```
a[1, 2] := 1.3
```



Zugriff auf Feldelemente mittels Indexvariable

```
for i := 1 to 4 do
  for j := 1 to 3 do
    a[i, j] := 0.0
  end
end
```

a:

0.0	0.0	0.0
0.0	0.0	0.0
0.0	0.0	0.0
0.0	0.0	0.0

Zugriff auf inneres Feld

```
var
  b: array [1:3] of real
```

```
b := a[2]
```

Beispiel: Magische Quadrate

Ein magisches Quadrat ist eine quadratische Matrix, für die die Summe jeder Zeile, jeder Spalte und der beiden Diagonalen denselben Wert ergibt.

Beispiel: 4x4-Quadrat

```
const
  n: int = 4
var
  sq: array [1:n, 1:n] of int
```

Einlesen der Werte

```
for i := 1 to n do
  for j := 1 to n do
    Read(↑sq[i, j])
  end
end
```

Summe der Diagonale

```
sumD := 0
for i := 1 to n do
  sumD := sumD + sq[i, i]
end
```

Prüfe Summen über Zeilen

```
for i := 1 to n do
  sum := 0
  for j := 1 to n do
    sum := sum + sq[i, j]
  end
  if sum ≠ sumD then
    WriteLn(↓"Fehler")
  end
end
```

3.4 Verbundobjekte (Records)

- Ein Verbund (compound, record oder structure) ist ein Datenobjekt, das eine fixe Anzahl von Komponenten, deren Datentyp beliebig ist, so zusammenfasst, dass man auf das Datenobjekt als Ganzes und über entsprechende Bezeichner auch auf die einzelnen Komponenten zugreifen kann

Beispiele:

```
type
  T = compound
    s1, s2: T1
    s3: T2
    ...
    sn: Tn
  end -- T
```

```
type
  Point = compound
    x, y: int
  end -- Point
```

```
type
  Color = compound
    red, green, blue: int
  end -- Color
```

... zufällig homogen

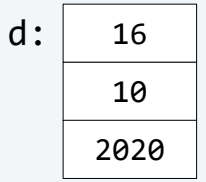
Beispiel

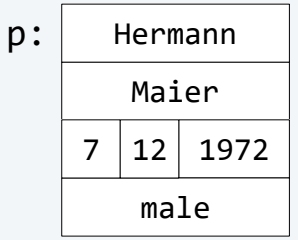
Datenobjekt zur Speicherung von personenbezogenen Daten

```
type
  Date = compound
    day, month, year: int
  end -- Date
```

```
type
  Person = compound
    firstName: string
    surname: string
    dob: Date
    gender: (female, male, divers)
  end -- Person
```

```
var
  d: Date
  p: Person
```



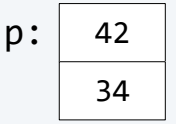


Zugriff auf Datenelemente

Beispiel

```
type
  Point = compound
    x, y: int
  end -- Point
var
  p: Point
```

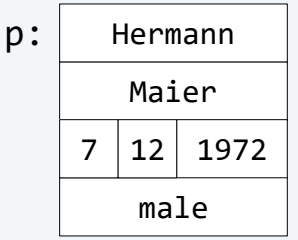
```
p.x := 42
p.y := 34
```



Beispiel

```
var
  p: Person
```

```
p.firstName := "Hermann"
p.surname := "Maier"
p.dob.day := 7
p.dob.month := 12
p.dob.year := 1972
p.gender := male
```



Wertzuweisungsaktion

Verbunde werden bei Wertzuweisung vollständig kopiert

Deklaration

```
var
  p, q: Point
```


```
p.x := 42
p.y := 34
```

Wertzuweisung

```
q := p
```


```
p.y := 12
```

Beispiel: Rechteck und Punkt

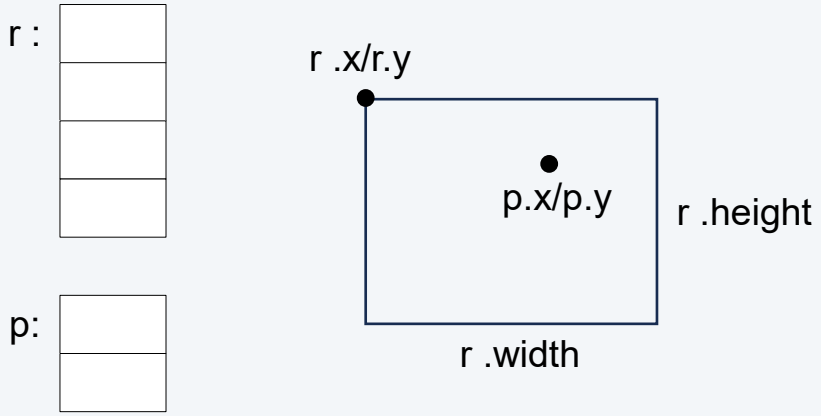
Deklarationen

```
type
  Rectangle = compound
    x, y, width, height: int
  end -- Rectangle
```

```
type
  Point = compound
    x, y: int
  end -- Point
```

Algorithmus

```
Contains(↓r: Rectangle ↓p: Point): bool
begin
  return (r.x ≤ p.x)
         and (p.x ≤ r.x + r.width)
         and (r.y ≤ p.y)
         and (p.y ≤ r.y+r.height)
end Contains
```



Beispiel: Polygon

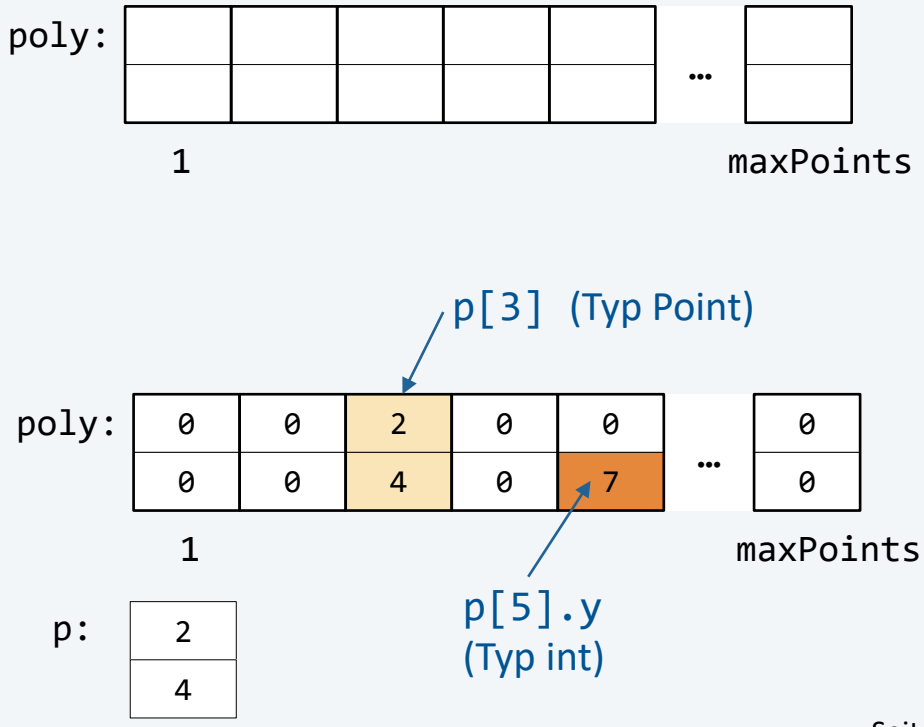
Deklarationen

```
const
  maxPoints: int = 100
type
  Point = compound
    x, y: int
  end -- Point
  Polygon = array [1:maxPoints] of Point
var
  poly: Polygon
  p: Point
```

```
for i := 1 to maxPoints do
  poly[i].x := 0
  poly[i].y := 0
end
```

```
poly[5].y := 7
```

```
p.x := 2
p.y := 4
poly[3] := p
```



Beispiel: Stichwortverzeichnis

Deklarationen

```
const
  maxPages: int = 10
  maxEntries: int = 100
type
  Entry = compound
    word: string
    pages: array[1:maxPages] of int
  end -- Entry
var
  index: array [1:maxEntries] of Entry
```

Verwendung

```
var
  i, j: int
```

```
for i := 1 to maxEntries do
  index[i].word := ""
  for j := 1 to maxPages do
    index[i].pages[j] := 0
  end
end
```

```
index[1].word := "Hermann Maier"
Index[1].pages[1] := 4
```

Nachteile dieser Realisierungsform:

- Begrenzte Seitennummern-Einträge wegen fixer Feldlänge
- Durch Vergrößerung der Feldlänge verschwendet man ggf. Speicher

Lösung durch vernetzte dynamische Datenstrukturen (s. Kapitel 6)