Strukturierte Datentypen

Eigenschaften strukturierter Datentypen

Strukturierte Datentypen

- sind auf anderen, einfacheren Datentypen aufgebaut.
- ermöglichen Aggregation von Einzelelementen.

Beispiele für strukturierte Datentypen in Modula-2 sind

- das **Feld** (Array)
- die **Menge** (Set)
- der **Verbund** oder die Struktur (Record)

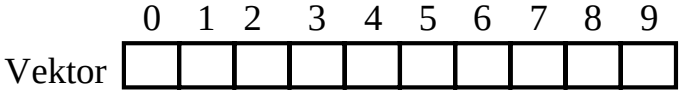
Felder (Arrays)

Das Feld (Array)

- Ein Feld dient zur Aufnahme einer **festen** Anzahl von Informationen des **gleichen** Typs.

Beispiel: Ein Vektor ist ein Feld von Integerzahlen.

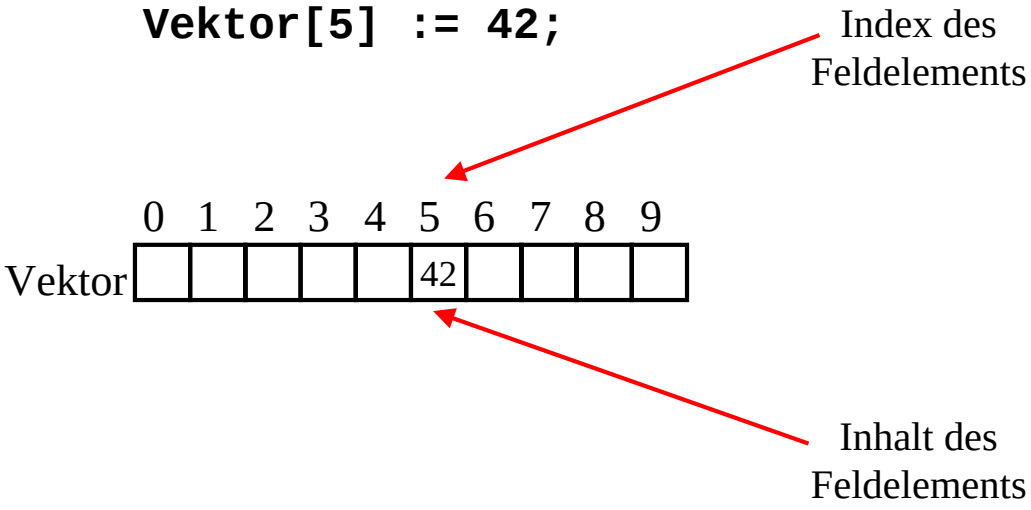
```
TYPE VektorTyp : ARRAY [1..10] OF INTEGER;
...
VARVektor : Vektortyp;
```



Felder (Arrays)

Der Zugriff auf ein bestimmtes Elemente eines Arrays erfolgt über seinen **Index**.

```
Vektor[5] := 42;
```

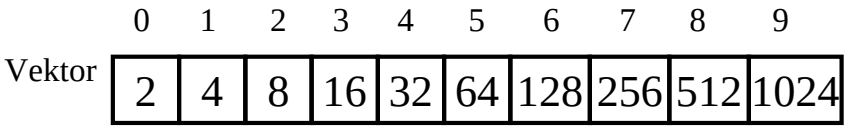


Felder (Arrays)

Beispiel:

Füllen des Vektors mit den 2er-Potenzen von 2^1 bis 2^{10}:

```
FOR Index := 1 TO 10 DO
  Vektor[Index-1] := Pot( 2,Index);
END;
```



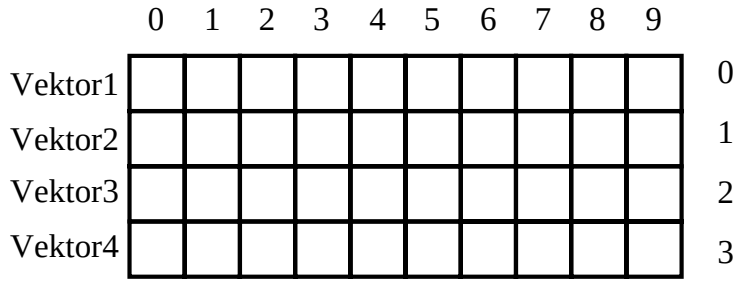
Felder (Arrays)

Ein Array kann über **einem** beliebigen Typ definiert werden.

Da auch benutzerdefinierte Typen "Typen" sind, können rekursive Schachtelung definiert werden, z.B. ein Array von Vektoren.

Beispiel: Struktur aus 4 Vektoren mit je 10 Integern (4×10-Matrix)

```
TYPE Matrix4x10 = ARRAY[1..4] OF VektorTyp;
```



Felder (Arrays)

Alternative Definition einer 4×10-Matrix :

```
TYPE MatrixType = ARRAY [1..4] OF
                      ARRAY [1..10] OF INTEGER;
```

kürzere Schreibweise:

```
TYPE MatrixType = ARRAY [1..4],[1..10] OF INTEGER;
```

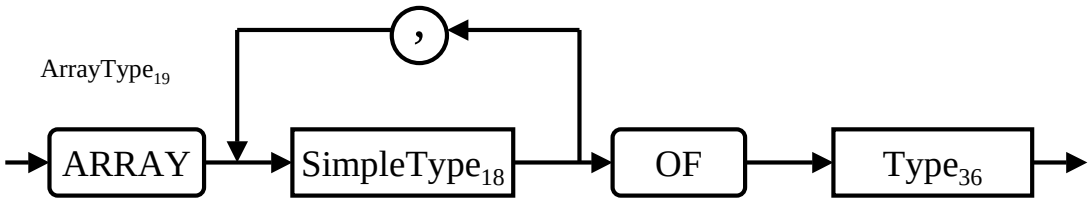
Felder (Arrays)

Zugriff auf die Elemente einer Matrix

```
VAR      Matrix :  MatrixType;
         Index   : INTEGER;
...
Matrix[0,2] := 1;
FOR Index := 0 TO 9 DO
  Matrix[1,Index] := Index;
END;
```

	0	1	2	3	4	5	6	7	8	9	
Vektor1			1								0
Vektor2	0	1	2	3	4	5	6	7	8	9	1
Vektor3											2
Vektor4											3

Felder (Arrays)



- "SimpleType" gibt den **Index-Typ.**
- "Type" gibt den **Element-Typ** an.
- Auch **mehrdimensionale** Arrays sind möglich.

Felder (Arrays)

Beispiel: Indizes von Arrays - **ganze Zahlen (INTEGER)**

```
MODULE array1;
FROM InOut IMPORT WriteInt, WriteLn;
CONST      min=-3;
           max=3;
TYPE feldtype = ARRAY [min..max] OF INTEGER;
VAR i   :  INTEGER;
    feld:  feldtype;
BEGIN
   FOR i:=min TO max DO
      feld[i]:=2*i;
      WriteInt(feld[i]);
   END;
   WriteLn;
END array1.
```

Ausgabe: -6 -4 -2 0 2 4 6

Felder (Arrays)

Beispiel: Indizes von Arrays – **ASCII-Zeichen (CHAR)**

```
MODULE array2;
FROM InOut IMPORT Write, WriteLn;
CONST min="a";
      max="e";
TYPE  feldtype = ARRAY [min..max] OF CHAR;
VAR   feld:feldtype;
      i:CHAR;
BEGIN
   FOR i:= min TO max DO
      feld[i]:=i;
      Write(feld[i]);
   END;
   WriteLn;
END array2.
```

Ausgabe: a b c d e

Felder (Arrays)

Beispiel: Indizes von Arrays - **Enumeration**

```
MODULE array3;
FROM InOut IMPORT WriteInt, WriteLn;
TYPE fische = (goldfisch,hai,karpfen,fugo,hecht);
     feldtype = ARRAY fische OF INTEGER;
VAR i    : fische;
    zahl : INTEGER;
    feld : feldtype;
BEGIN
    zahl:=1;
    FOR i:=goldfisch TO hecht DO
        feld[i]:=zahl*2;
        zahl:=zahl+1;
        WriteInt(feld[i]);WriteLn;
    END;
END array3.
```

Ausgabe: 2 4 6 8 10

Felder (Arrays)

Zusammenfassung:

Die **Indizes** eines Feldes (Arrays) müssen von einem **einfacher Typ** sein.

- **`INTEGER`**
- **`CARDINAL`**
- **`CHAR`**
- **`BOOLEAN`**
- **`BITSET`**
- Aufzählungstypen (Enumeration), z.B. **`fische`**
- Unterbereiche (Subrange), z.B. **`[1..3]`** oder **`[hai..fugo]`**

...

NICHT möglich sind

- **`REAL`**
- Strukturierte Datentypen (Arrays, Records)

Verbunde (Records)

Verbund (Record)

Der Verbund dient der Zusammenfassung von Elementen beliebigen, auch verschiedenen Datentyps:

Beispiele:

- Personendaten: Name, Adresse, Geb.-Datum, Geschlecht
- Datum: Tag, Monat, Jahr
- Schein: Name, Matr.-Nr., KlausurPunkte

Verbunde (Records)

Beispiele: Datum und Schein

```
TYPE  Datum = RECORD
          Tag   : INTEGER;
          Monat : INTEGER;
          Jahr  : INTEGER;
      END;

      Schein = RECORD
          Name          : ARRAY[1..10] OF CHAR;
          MatrNr        : INTEGER;
          KlausurPunkte : INTEGER;
      END;
```

Verbunde (Records)

Der **Zugriff** auf Komponenten von Strukturen erfolgt durch den **Namen des RECORDS**, an den **ein Punkt** und der **Name der Komponente** angehängt wird (**dot notation**):

Beispiel:

```
TYPE   Datum = RECORD
                 Tag    : INTEGER;
                 Monat  : INTEGER;
                 Jahr   : INTEGER;
               END;
VAR Silvester : Datum;
:
Silvester.Tag   := 31;
Silvester.Monat := 12;
Silvester.Jahr  := 2002;
```


Verbunde (Records)

Durch die **Wiederverwendung (Reuse)** der definierten Typen ist der Aufbau **komplexer Datentypen** möglich:

```
TYPE  AnredeTyp = (Frau,Herr);
      String    = ARRAY [0..40] OF CHAR;
      NameTyp   = RECORD
                    Vorname,Nachname : String;
                    Anrede           : AnredeTyp;
                  END;
      Person    = RECORD
                    Name      : NameTyp;
                    GebDatum  : Datum;
                  END;
      Personal  = ARRAY[1..20] OF Person;
```

Verbunde (Records)

Zugriff auf komplexere Komponenten

Beispiel:

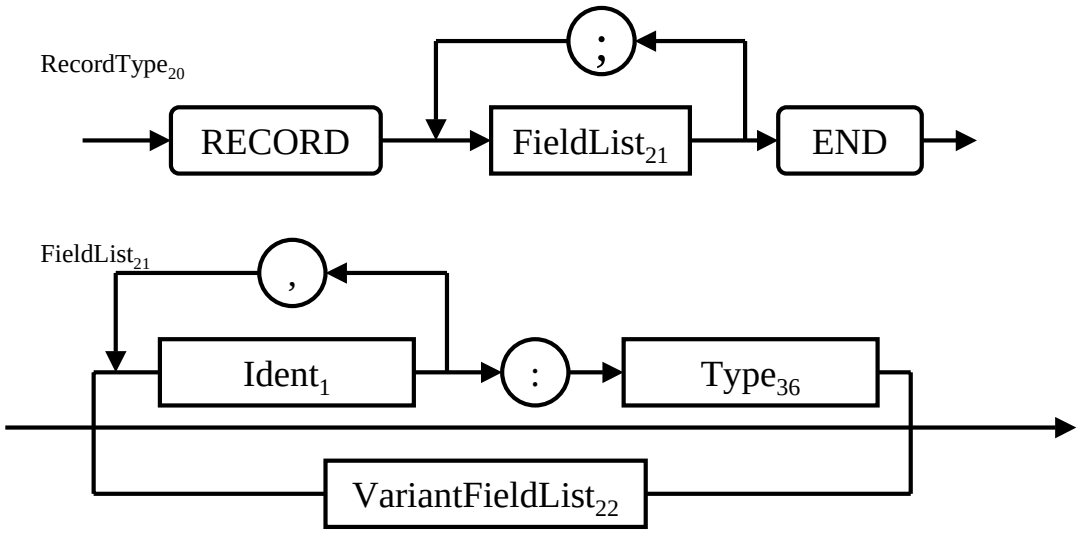
```
Personal[1].Name.Vorname   := 'Otto';
Personal[1].Name.Nachname  := 'Müller';
Personal[1].Name.Anrede    :=  Herr;
Personal[1].GebDatum.Tag   :=  1;
Personal[1].GebDatum.Monat :=  4;
Personal[1].GebDatum.Jahr  :=  1960;
```

Vereinfachung durch **`WITH ... DO`**

```
WITH Personal[1] DO
    WITH Name DO
          Vorname = 'Otto';
          Nachname = 'Müller';
          Anrede = Herr;
    END;
END;
```


Verbunde (Records)

Syntax zur Definition eines RECORD:



- "Type" kann ein beliebiger Typ (Elementartyp, Aufzählungstyp, Bereichstyp oder Strukturtyp) sein.

Mengen (Sets)

Mengen (Sets)

Mengen sind ungewöhnliche Typen für Programmiersprachen:

- beliebig viele Elemente
- beliebige Art der Elemente
- rekursive Mengenbildung (Mengen von Mengen)
- keine Ordnung zwischen den Elementen in der Menge
- keine Duplikate

Diese nützliche Eigenschaften sind in Programmiersprachen sehr schwierig zu implementieren. In PASCAL und Modula-2 sind SETS nur magere Imitationen der mathematischen Mengen.

Es gibt nur wenig verbreitete Programmiersprachen mit Mengen als Grundelementen (SETL, Set-Language).

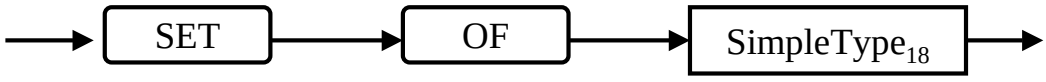
Mengen (Sets)

Mengen in Modula-2

- Mengen an sich sind ungewöhnliche Typen für Programmiersprachen.
- In Modula-2 werden Mengen über einem bestimmten Basistyp definiert.
- Der Basistyp hat nur einen begrenzten Wertvorrat und muss ein Unterbereichstyp oder Aufzählungstyp sein.
- Die Größe der Mengen ist abhängig von der Implementierung (je nach Wortbreite des Rechners auf 16 bzw. 32 Werte beschränkt).
  Mathematische Mengen erlauben beliebig viele Elemente beliebigen Typs. Zusätzlich sind rekursive Mengenbildungen erlaubt.

Mengen (Sets)

SetType$_{17}$



"SimpleType" gibt den Basistyp an (Aufzählungen, Bereichstypen).

Beispiel einer Mengendefinition in Modula-2:

```
TYPE   Farben = (rot,gruen,blau,gelb);
       Farbmenge = SET OF Farben;
```

Hierbei ist **`Farben` der Basistyp** der Menge **`Farbmenge`**.

Mengen (Sets)

Modula-2 stellt Operatoren zur Aufnahme (Inklusion) und zum Ausschluss (Exklusion) von Elementen zur Verfügung:

- **`INCL(Set,Element)`**
- **`EXCL(Set,Element)`**

Außerdem kann mit Hilfe des Operators **`IN`** geprüft werden, ob ein Element in einer Menge enthalten ist:

- **`Element IN Set`**

Weitere Operatoren sind: **`+`** (Vereinigung ∪)
`*` (Schnitt ∩)
`-` (Differenz /)
`<=` (Untermenge ⊆)
`>=` (Obermenge ⊇)