Ziele

Nach dem Lesen dieses Kapitels haben Sie Folgendes erreicht:

- Sie wissen, dass für die algorithmische Lösung komplexer Aufgaben analog zur Gestaltung der Ablaufstrukturen von Algorithmen auch entsprechende Konzepte und Konstrukte zur Modellierung der in den Algorithmen benötigten Datenobjekte erforderlich sind.
- Sie wissen, dass wir zwischen elementaren und strukturierten Datenobjekten unterscheiden, und kennen die Unterschiede zwischen diesen beiden Kategorien von Datenobjekten und den Datentypen, die zu ihrer Realisierung benötigt werden.
- Sie wissen, dass wir zudem zwischen Datenobjekten, bei denen im Zuge der Algorithmenausführung nur ihr Wert verändert werden kann (sie werden hier in elementare und strukturierte Datenobjekte unterteilt) und solchen, bei denen sich neben ihrem Wert auch ihre Größe und/oder ihre Struktur ändern kann (vernetzte Datenobjekte), unterscheiden.
- Sie wissen, was man unter Feld, Verbund, verkettete Liste, Baum, Binärbaum und binärer Suchbaum versteht, kennen Konzepte und Konstrukte zu deren Realisierung, wissen wozu man solche Datenobjekte benötigt und kennen wichtige Standardoperationen zu ihrer Erzeugung und Manipulation.
- Sie wissen, was man unter einer abstrakten Datenstruktur, einer Datenkapsel und einem abstrakten Datentyp versteht, wozu man diese benötigt und wie man sie realisieren kann.

3.1 Atomare Datenobjekte und -typen

In Kapitel 1 haben wir (neben Anderem) grundsätzliche Eigenschaften von Algorithmen erörtert und darauf aufbauend elementare Konzepte und Konstrukte zur Gestaltung derselben erläutert. In diesem Zusammenhang haben wir auch bereits einige elementare Konstrukte zur Definition von Datenobjekten eingeführt. Wir haben dabei zunächst unterschieden zwischen

1. *unveränderbaren* Datenobjekten in Form von Literalen (z.B. 17) oder Konstanten (z.B. *pi*) und
2. *veränderbaren* Datenobjekten, also Variablen (z.B. *x*), die während der Ausführung des Algorithmus, dessen Bestandteil sie sind – vor allem durch die Zuweisungsanweisung –, unterschiedliche Werte aus einer bestimmten Wertemenge (die durch ihren im Zuge der Deklaration festgelegten Datentyp bestimmt ist) annehmen können.

Eine weitere, von dieser Unterscheidung unabhängige, also zu ihr orthogonale Unterscheidungsmöglichkeit ergibt sich aufgrund der den Datenobjekten zugeordneten Datentypen, die neben einer Wertemenge auch die Menge der möglichen Operationen definieren.

In dieser Hinsicht haben wir in Abschnitt 1.3.1 bereits Datenobjekte eingeführt, die ganze Zahlen (Datentyp *integer*), reelle Zahlen (Datentyp *real*), Wahrheitswerte (Datentyp *boolean*) und Zeichen (Datentyp *char*) repräsentieren. Diesen Datenobjekten ist gemeinsam, dass sie in unserem Sinne als logisch ganze, nicht mehr weiter zerlegbare Artefakte aufgefasst werden und in diesem Sinne als atomar angesehen und deshalb auch als *atomare*, *elementare*, *einfache* oder *unstrukturierte* Datenobjekte (bzw. Datentypen) bezeichnet werden, wobei sich keine dieser Bezeichnungen gegenüber den anderen klar durchgesetzt hat.

Wir haben zwar auch schon Texte bzw. Zeichenketten (Datentyp *string*) kurz behandelt, diese zählen wir aber nicht mehr zu den elementaren Datenobjekten. Wir widmen uns den Zeichenketten weiter hinten bei der Behandlung von Feldern noch einmal ausführlicher.

Analog zur Gestaltung der Ablaufstruktur von Algorithmen benötigt man bei der algorithmischen Lösung komplexer Aufgaben auch Konzepte und Konstrukte zur Gestaltung (wir sagen dazu auch *Modellierung*) der zu manipulierenden Datenobjekte. Neben den elementaren Datenobjekten sind dafür vor allem die so genannten *strukturierten* und *dynamischen* oder *vernetzten* Datenobjekte notwendig. In den folgenden Abschnitten behandeln wir Konzepte und Konstrukte für solche Datenobjekte und -typen mit dem Ziel, möglichst adäquate Mittel für eine realitätskonforme Modellierung der in Algorithmen benötigten Daten bereitzustellen.

3.2 Strukturierte Datenobjekte und -typen

In vielen Algorithmen ist es sinnvoll, wenn nicht gar notwendig, eine Menge einzelner Datenobjekte in einer Variablen zusammenzufassen, weil sie z.B. logisch zusammen gehören und daher eine logische (wenn auch strukturierte) Einheit bilden. Bei derartigen Datenobjekten unterscheiden wir zwischen solchen, deren einzelne Elemente alle denselben Datentyp haben – wir nennen diese ***Felder*** (*arrays*) – und solchen, deren Elemente unterschiedliche Datentypen haben können – wir nennen diese ***Verbunde*** (*compounds*).

3.2.1 Felder

Bevor wir unterschiedliche Ausprägungen von Feldern im Detail behandeln, wollen wir eine Definition für dieses Konzept geben.

Definition: *Feld*

Ein *Feld* (*array*) ist ein Datenobjekt, das eine Sammlung einer fixen Anzahl von Elementen repräsentiert, die alle denselben Datentyp haben und auf die über einen *Index* zugegriffen werden kann.

Felder verwenden wir in verschiedenen Ausprägungen, sowohl als *eindimensionale Felder*, wenn wir z.B. Vektoren (im Sinne der Mathematik) benötigen, als auch als *mehrdimensionale Felder*, wenn wir z.B. Matrizen (wieder im Sinne der Mathematik) benötigen.

Eindimensionale Felder

Ein *eindimensionales Feld* (*one-dimensional array*) *a* fasst eine fixe Anzahl von Elementen desselben Datentyps, des so genannten *Elementdatentyps* (*element data type*), so zu einer Einheit – in der diese Elemente durchnummeriert sind – zusammen, dass über einen *Index i* auch auf die einzelnen Elemente zugegriffen werden kann. Dazu wird der *Indexoperator* [] auf das Feld *a* mit dem Index *i* angewandt: *a*[*i*] liefert das Element mit dem Index *i* im Feld *a*. Der Wertebereich, aus dem dieser Index stammen muss, wird als *Indexbereich* (*index range*) bezeichnet. Dabei handelt es sich um einen Unterbereich der ganzen Zahlen, der durch den kleinsten (ersten, *first*) und den größten (letzten, *last*) Indexwert angegeben wird.[1]

In der von uns verwendeten Notation zur Beschreibung von Algorithmen sieht die Deklaration einer eindimensionalen Feldvariable *a* folgendermaßen aus:

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

Dabei müssen *first* und *last* Literale oder Konstanten sein und es muss $first \leq last$ gelten. Das Feldobjekt *a* hat dann $last - first + 1$ Elemente. Die eckigen Klammern, die den Indexbereich in der Deklaration einschließen, dürfen nicht mit dem Indexoperator in Ausdrücken verwechselt werden, sie dienen in der Deklaration nur der Abgrenzung des Indexbereichs von den restlichen Bestandteilen.

Beispiel 3.1 zeigt eine einfache Deklaration einer Feldvariablen.

Beispiel 3.1 **Deklaration einer Feldvariablen**

▶Abbildung 3.1 zeigt, wie eine Feldvariable *a* (also ein Feldobjekt) mit zehn Elementen vom Datentyp *integer* und einem Indexbereich von 1 bis 10 deklariert wird.

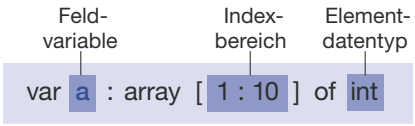

Abbildung 3.1: Deklaration einer Feldvariablen

1 Es gibt zwar Programmiersprachen, Pascal etwa, bei denen neben ganzen Zahlen z.B. auch Zeichen zum Indizieren von Feldern erlaubt sind, da dies in den meisten Sprachen aber nicht möglich ist, wollen wir nur ganze Zahlen als Index zulassen. Das ist ohne Beschränkung der Allgemeinheit möglich, denn jeder Indexdatentyp muss auch in Sprachen wie Pascal auf die ganzen Zahlen abgebildet werden können.

Es gibt Programmiersprachen (Beispiele s. u.), bei denen anstelle eines Indexbereichs die Anzahl der Elemente eines Felds anzugeben ist. Wir wollen das in unserem Pseudocode durch eine Deklaration nach folgendem Muster

```
var a = array [n] of ElementType
```

nachbilden, wobei n ein Literal oder eine Konstante (jeweils ganzzahlig) mit $n > 0$ sein muss. Je nach Programmiersprache wird dann dem ersten Element des Felds der Index 0 (z.B. in C/C++, Java und C#) oder der Index 1 (z.B. in BASIC, MatLab und Smalltalk) zugeordnet. Aufgrund dieser Eigenschaften werden Felder in solchen Programmiersprachen entweder als *null-basierend* (*zero-based*) oder als *eins-basierend* (*one-based*) bezeichnet.

In ▶Abbildung 3.2 ist dargestellt, wie man sich die Repräsentation der Feldvariablen *a* aus Abbildung 3.1 im Hauptspeicher eines Computers vorstellen kann: Wie jede andere Variable hat sie eine Startadresse und ab dieser Adresse liegen die einzelnen Elemente des Felds unmittelbar hintereinander im Speicher. Mit dem Indexoperator und einem Index i (wobei in diesem Fall $1 \le i \le 10$ gelten muss) kann mittels des Ausdrucks $a[i]$ ein beliebiges, in diesem Fall – weil eins-basierend – das i-te Element des Felds adressiert werden.

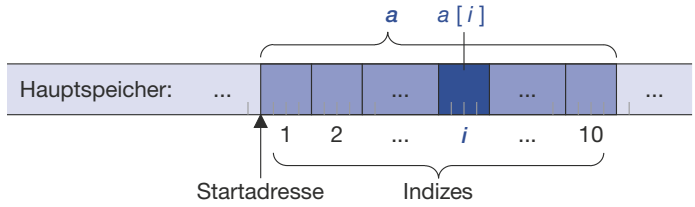

Abbildung 3.2: Anordnung der Elemente einer Feldvariablen aus Beispiel 3.1 im Speicher

Hinweis: Die in Abbildung 3.2 gezeigte, sequenzielle und lückenlose Anordnung der einzelnen Feldelemente im Speicher ermöglicht z.B. einem Compiler die einfache Abbildung der Indexoperation $a[i]$ auf maschinennahe Konzepte (Speicheradressen), denn aus der Startadresse $AddrOf(\downarrow a)$ des Felds, einem Index i, der Untergrenze des Indexbereichs *first* und dem Speicherbedarf eines Elements in Byte $elementSize = SizeOf(\downarrow ElementType)$ kann die Adresse von $a[i]$ durch Anwendung der so genannten *Speicherabbildungsfunktion* (*memory addressing function*) berechnet werden:

$$AddrOf(\downarrow a[i]) = AddrOf(\downarrow a) + (i - first) \cdot elementSize$$

Diese Speicherabbildungsfunktion ermöglicht den effizienten Zugriff – nämlich in konstanter Zeit, also unabhängig vom Wert des Index – auf jedes Element des Felds.

Werden mehrere Feldobjekte mit den gleichen Eigenschaften, also mit gleichem Elementdatentyp und identischem Indexbereich, benötigt, ist es ratsam, dafür einen eigenen Datentyp vorzusehen. Beispiel 3.2 zeigt, wie wir – hier in Pseudocode, aber bei vielen Programmiersprachen ist das in ähnlicher Weise möglich – mit dem *type*-Konstrukt einen neuen, so genannten *benutzerdefinierten Datentyp* (*user-defined data type*) einführen können, der dann genauso verwendet werden kann wie die elementaren (Standard-)Datentypen, mit denen wir bisher gearbeitet haben (z.B. *integer* und *real*).

Beispiel 3.2

Deklaration und Verwendung eines benutzerdefinierten Felddatentyps

Wenn für die Lösung einer Aufgabe mit Koordinaten aus dem dreidimensionalen Raum gerechnet werden muss, ist es sinnvoll, dafür beispielsweise einen Datentyp *Position3D* gemäß folgender Typdeklaration vorzusehen:

```
type Position3D = array [1:3] of real
```

Dieser benutzerdefinierte Felddatentyp kann nun zur Deklaration von Variablen und in Formalparameterlisten verwendet werden, z.B.:

```
var p1, p2: Position3D
Distance(↓a: Position3D  ↓b: Position3D): real
```

Benutzerdefinierte Datentypen wie *Position3D* in Beispiel 3.2 sind selbstverständlich nicht auf die Verwendung für Felder beschränkt, sondern können auch zur Definition anderer Datenobjekte herangezogen werden.

Felder haben eine fixe, durch ihre Deklaration definierte Anzahl von Feldelementen. Beispielsweise wird durch die Deklaration des Datentyps *Position3D* in Beispiel 3.2 festgelegt, dass alle Datenobjekte dieses Typs Felder sind, die drei Elemente vom Typ *real* enthalten.

Oft tritt die Situation ein, dass in einem konkreten Anwendungsfall nur ein Teil der (durch die Deklaration festgelegten) Elemente eines Felds tatsächlich benötigt wird. In solchen Fällen muss man Vorsorge treffen, dass der aktuelle „Füllstand" des betreffenden Feldobjekts jederzeit festgestellt werden kann.

Beispiel 3.3 zeigt nicht nur, wie man solche *teilweise gefüllten Felder* (*partially filled arrays*) deklariert, sondern auch wie man mit ihnen und mit Feldern generell arbeitet.

Beispiel 3.3

Deklaration und Verwendung eines teilweise gefüllten Felds

Für die Speicherung aller Noten (*marks* oder *grades*), die ein Student während seines Studiums erzielt hat, kann man (um daraus z.B. das Studienergebnis entsprechend der Aufgabenstellung in Abschnitt 2.5 zu berechnen) ein Feld *marks* heranziehen, in dem höchstens *max* Noten gespeichert werden können. Die aktuelle Anzahl der eingetragenen Noten muss dann in einer zusätzlichen Variablen *n* verwaltet werden, wobei zu jedem Zeitpunkt $0 \leq n \leq max$ gelten muss. Das führt zu folgenden Deklarationen:

```
const max = ...  -- maximum number of marks
var marks: array [1:max] of int
    n: int  -- mark counter, marks[1:n] filled
```

Das Einlesen der Noten von einem Datenstrom, der von einem Eingabegerät (*input device*) zur Verfügung gestellt wird, kann dann wie folgt formuliert werden (wobei wir davon ausgehen, dass nicht mehr als *max* Noten im Datenstrom enthalten sind):

```
n := 0  -- no marks up to now
while not end of file do
  n := n + 1
  Read(↑marks[n])  -- assume reading a value in the range from 1 to 5
end -- while
```

Nach der Ausführung des obigen Codestücks ergibt sich der in ▶Abbildung 3.3 dargestellte Zustand eines teilweise gefüllten Felds: der dunklere Bereich des Felds enthält bereits Noten, der hellere Bereich ist (noch) frei.

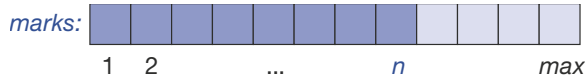

Abbildung 3.3: Teilweise gefülltes Notenfeld *marks*

Zur Berechnung des Notendurchschnitts (*average* vom Datentyp *real*) kann dann das Feld *marks* und der aktuelle Füllstand *n* wie folgt herangezogen werden (wobei wir davon ausgehen, dass $n > 0$ gilt):

```
sum := 0
for i := 1 to n do
  sum := sum + marks[n]
end -- for
average := Real(↓sum) / Real(↓n)
```

Unter der Annahme, dass es einen Algorithmus *Sort* gibt, der die Noten im Feld *marks* aufsteigend sortiert (siehe Kapitel 7), können wir beispielsweise auch den *Median* (*median*), also den Wert in der Mitte des sortierten Bereichs, auf einfache Weise ermitteln:

```
Sort(↕marks  ↓1  ↓n)
if n mod 2 = 1 then  -- odd number of marks:
                     -- there is an element in the middle
  median := marks[(n + 1) / 2]
else  -- even number of marks:
      --   use average of elements left and right of the middle
  median := ( marks[n / 2] + marks[(n / 2) + 1] ) / 2
end -- if
```

Wollen wir auch noch die *Häufigkeiten* (*frequencies*) der einzelnen Noten berechnen, können wir dies durch Verwendung eines zweiten Felds *freqs*, das Platz für die Speicherung der Häufigkeiten der fünf Noten bietet und die jeweilige Note als Index in *freqs* benutzt (siehe ▶Abbildung 3.4, die zeigt, wie sich beispielsweise die Häufigkeit der Note 1 ergibt):

```
  var freqs: array [1:5] of int
begin
  -- initialize frequencies to zero
  for i := 1 to 5 do
    freqs[i] := 0
  end -- for
  -- count each mark in frequencies array
  for i := 1 to n do
    mark := marks[i]
    freqs[mark] := freqs[mark] + 1
  end -- for
```

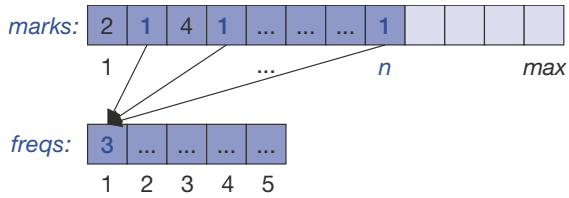

Abbildung 3.4: Noten und ihre Häufigkeiten

Zeichenketten

Bei der Behandlung der Datentypen in Abschnitt 1.3.2 haben wir für Datenobjekte, die Texte oder Zeichenketten repräsentieren, zwar schon den Datentyp *string* eingeführt, die Details seiner Realisierung aber noch offen gelassen. Dies wollen wir im Folgenden nachholen und damit gleichzeitig weitere Möglichkeiten des Einsatzes eindimensionaler Felder aufzeigen.

Zeichenketten sind Folgen von einzelnen Zeichen. Deshalb bietet es sich an, Zeichenketten als Felder mit dem Elementtyp Zeichen (*char*) zu realisieren. Dazu benötigt man die maximal erlaubte Anzahl der Zeichen, für die das Feld Platz bieten soll, und kann dann den Datentyp *string* folgendermaßen definieren:

```
const max = …
type string = array [1:max] of char
```

Nachdem man auch unterschiedlich lange Zeichenketten darstellen möchte, muss ihre aktuelle Länge separat verwaltet werden, was eine zusätzliche Variable vom Datentyp *integer* erforderlich macht. Für die Repräsentation einer Zeichenkette wären daher – analog zu den teilweise gefüllten Feldern in Beispiel 3.3 – zwei Variablen erforderlich, z.B.:

```
var s: string
    sLen: int  -- holds current length of s, 0 ≤ sLen ≤ max
```

Das ist umständlich und fehleranfällig, da bei Operationen auf Zeichenketten immer beide Variablen (z.B. *s* und *sLen*) betroffen wären und konsistent gehalten werden müssten.

Um das zu vermeiden, wurden bei der Entwicklung von Programmiersprachen spezielle Zeichenketten-Repräsentationen entwickelt. Zwei Arten davon, nämlich Pascal- und C-Zeichenketten, werden im Folgenden erörtert, um dem Leser einen Eindruck über typische Repräsentationsformen zu vermitteln.

Pascal-Zeichenketten *Pascal-Zeichenketten* werden in der Programmiersprache Pascal verwendet (vgl. [Wirth 1971a]) und sind durch folgende Typdeklaration definiert:

```
type PascalString = array [0:255] of char
```

Dabei ist festgelegt, dass das Element mit dem Index 0 nicht Teil der Zeichenkette ist, sondern ihre aktuelle Länge repräsentiert. Es wird daher auch als *Längenbyte* bezeichnet. Durch das Längenbyte können nur Werte zwischen 0 und 255 repräsentiert werden, die maximale Länge von Pascal-Zeichenketten beträgt daher 255 Zeichen.

Beispiel 3.4 zeigt die Deklaration und die Verwendung einer Pascal-Zeichenketten-Variablen sowie ihre Repräsentation im Hauptspeicher eines Computers.

Beispiel 3.4 **Pascal-Zeichenkette und ihre Repräsentation im Speicher**

Das folgende Pseudocode-Fragment deklariert eine Pascal-Zeichenketten-Variable *ps* und weist ihr einen Wert zu:

```
  var ps: PascalString
begin
  ps := "Hello"
```

▶Abbildung 3.5 zeigt die Repräsentation des Zeichenkettenobjekts *ps* im Hauptspeicher eines Computers, wobei die Fragezeichen beliebige Zeichen bedeuten, die nicht mehr zur aktuellen Zeichenkette gehören.

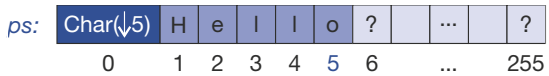

Abbildung 3.5: Repräsentation der Zeichenkette "Hello" als Pascal-Zeichenkette im Speicher

Der Vorteil von Pascal-Zeichenketten liegt darin, dass die aktuelle Länge einfach und effizient mit der Funktion

```
Length(↓ps: PascalString): int
begin
  return Int(↓ps[0])
end Length
```

zu ermitteln ist. Der wesentliche Nachteil von Pascal-Zeichenketten liegt (wegen des Längenbytes) in der Beschränkung auf die maximale Länge von 255 Zeichen; ein weiterer in der Verschwendung von Speicherplatz für die Repräsentation kurzer Zeichenketten.

C-Zeichenketten *C-Zeichenketten* werden in der Programmiersprache C verwendet (vgl. [Kernighan u. Ritchie 1988]) und sind durch folgende Deklarationen definiert:

```
const max = …
type CString = array [0:max] of char
```

Dabei ist festgelegt, dass eine C-Zeichenkette durch ein spezielles Zeichen, das sogenannte *Terminierungszeichen* mit dem Code 0, also *Char*(↓0), abgeschlossen und in C mit '\0' notiert wird. Deshalb werden C-Zeichenketten im Engl. auch als *null-terminated strings* bezeichnet. Das Terminierungszeichen kann deshalb innerhalb der Zeichenkette nicht vorkommen.

Beispiel 3.5 zeigt die Deklaration und die Verwendung einer C-Zeichenketten-Variablen sowie ihre Repräsentation im Hauptspeicher eines Computers.

Beispiel 3.5

C-Zeichenkette und ihre Repräsentation im Speicher

Das folgende Pseudocode-Fragment deklariert eine C-Zeichenketten-Variable *cs* und weist ihr einen Wert zu:

```
  var cs: CString
begin
  cs := "Hello"
```

▶Abbildung 3.6 zeigt die Repräsentation des Zeichenkettenobjekts *cs* im Hauptspeicher eines Computers, wobei die Fragezeichen beliebige Zeichen bedeuten, die nicht mehr zur aktuellen Zeichenkette gehören.

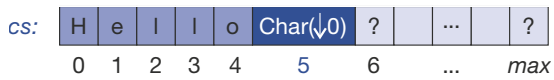

Abbildung 3.6: Repräsentation der Zeichenkette "Hello" als C-Zeichenkette im Speicher

Der Vorteil von C-Zeichenketten liegt darin, dass ihre Länge nicht beschränkt ist, denn die maximale Länge (*max* in der obigen Deklaration) kann beliebig groß gewählt werden. Der wesentliche Nachteil ist, dass die Ermittlung der Länge aufwändig ist, denn dafür muss das Terminierungszeichen (beispielsweise auf folgende Weise) gesucht werden:

```
Length(↓cs: CString): int
  var i: int
begin
  i := 0
  while cs[i] ≠ Char(↓0) do
    i := i + 1
  end -- while
  return i
end Length
```

Hinweis: In objektorientierten Programmiersprachen (z.B. in C++, Java und C#) werden Zeichenketten durch „echte“ Objekte[2] spezieller Zeichenketten-Klassen (z.B. der Klasse *java.lang.String* in Java) dargestellt. (Kapitel 13 behandelt die Grundlagen der objektorientierten Programmierung.) Die interne Repräsentation dieser Zeichenkettenobjekte bleibt dem Benutzer zwar verborgen, es werden aber auch Zeichenfelder verwendet, nur ist das für den Benutzer nicht mehr relevant. Diese Zeichenkettenobjekte beseitigen durch geschickte Realisierung beide Nachteile der oben behandelten Zeichenketten-Repräsentationen: die Zeichenketten können somit beliebig lang werden (wie bei C-Zeichenketten) *und* ihre Länge kann effizient ermittelt werden (wie bei Pascal-Zeichenketten).

Zwei- und mehrdimensionale Felder

Die Nützlichkeit bzw. Notwendigkeit zwei- und mehrdimensionaler Felder (*two-* und *multi-dimensional arrays*) braucht nicht weiter argumentiert werden; bei vielen Aufgabenstellungen finden wir z.B. Matrizenobjekte, etwa für eine Entscheidungstabelle oder ein Schachbrett.

Beispiel 3.6 zeigt die Verwendung eines zweidimensionalen Felds.

Beispiel 3.6 **Zweidimensionales Feld (Matrix) und Repräsentation im Speicher**

Für die Lösung einer Mathematikaufgabe benötigt man eine Matrix mit vier Zeilen und drei Spalten, eine 4x3-Matrix. Den zweidimensionalen Felddatentyp *Matrix* und eine Variable *m* von diesem Typ können wir wie folgt festlegen:

```
type Matrix = array [1:4, 1:3] of real
var m: Matrix
```

▶Abbildung 3.7 (a) zeigt eine schematische Darstellung der Matrix *m*, wie man sie z.B. in Mathematikbüchern findet, während in (b) illustriert wird, wie man sich die Repräsentation der Variablen *m* im Hauptspeicher eines Computers vorstellen kann. Nachdem der Hauptspeicher konzeptuell eine lineare, also eindimensionale, Anordnung von Bytes ist, muss die Matrix, also das zweidimensionale Feld, linearisiert (in eine Dimension transformiert) werden. Da es in den meisten Programmiersprachen so gehandhabt wird, gehen wir davon aus, dass die Zeilen (*rows*) der Matrix nacheinander im Speicher angeordnet werden, die Matrix wird also *zeilenweise* im Speicher abgelegt (*row major order*).

2 Damit sind Objekte im Sinne der objektorientierten Programmierung (OOP) gemeint, deren Datentyp eine Klasse ist und die über Datenkomponenten und Methoden verfügen (siehe dazu Kapitel 13).

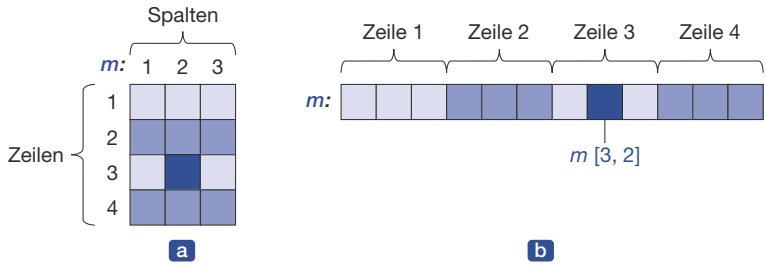

Abbildung 3.7: Anordnung der Elemente eines zweidimensionalen Felds (a) im Speicher (b)

Wenn eine Bearbeitung aller Elemente einer Matrix erforderlich ist, erfolgt dies typischerweise mittels zweier ineinander geschachtelter Zählschleifen. Das folgende Codestück zeigt, wie die Matrix *m* mit dem Wert 0.0 initialisiert werden könnte:

```
for i := 1 to 4 do     -- for all rows
  for j := 1 to 3 do   -- for all columns
    m[i, j] := 0.0
  end -- for
end -- for
```

Da es für den Elementtyp keinerlei Einschränkungen gibt, kann ein zweidimensionales Feld – was Abbildung 3.7 (b) nahe legt – auch als eindimensionales Feld interpretiert werden, dessen Elemente eindimensionale Felder sind, also ein „Feld von Feldern".

Hinweis: Wie bei eindimensionalen Feldern muss auch bei zweidimensionalen für den Zugriff auf ein Element durch Anwendung der *Speicherabbildungsfunktion* (*memory addressing function*) die Speicheradresse dieses Elements berechnet werden.

Für eine Matrix *m* mit der Deklaration

```
var m: array [first1:last1, first2:last2] of ElementType
```

kann aus der Startadresse *AddrOf*(↓*m*), den beiden Indizes *i* und *j*, den beiden Untergrenzen der Indexbereiche *first1* und *first2*, dem Speicherbedarf eines Elements in Byte *elementSize* = *SizeOf*(↓*ElementType*) sowie dem Speicherbedarf einer ganzen Zeile *rowSize* = (*last1* – *first1* + 1) · *elementSize* die Adresse von *m*[*i, j*] wie folgt berechnet werden:

$$AddrOf(\downarrow m[i, j]) = AddrOf(\downarrow m) + (i - first1) \cdot rowSize + (j - first2) \cdot elementSize$$

Diese Speicherabbildungsfunktion gewährleistet – wie bei eindimensionalen Feldern – einen effizienten Zugriff (in konstanter Zeit) auf alle Elemente der Matrix, unabhängig von den Werten der beiden Indizes.

Die Anzahl der Dimensionen eines Feldobjekts ist in der Regel nicht begrenzt, so dass neben den ein- und zweidimensionalen Feldobjekten, die wir bisher betrachtet haben, in den meisten Programmiersprachen auch Feldobjekte mit drei und mehr Dimensionen möglich sind. Da sich weder die Deklaration noch die Verwendung solcher mehrdimensionalen Felder grundsätzlich von den ein- und zweidimensionalen unterscheidet, brauchen wir darauf nicht näher einzugehen.

3.2.2 Verbunde

In der Praxis finden wir häufig Aufgabenstellungen, bei deren algorithmischer Lösung wir, wenn der Algorithmus möglichst realitätsnah gestaltet werden soll, auf Datenobjekte stoßen, die mehrere Elemente unterschiedlichen Datentyps zu einem logischen Ganzen zusammenfassen, wie in Beispiel 3.7 gezeigt. Deshalb ist es naheliegend, auch dafür ein Konzept und ein Konstrukt zur Umsetzung vorzusehen.

Beispiel 3.7 **Getrennte Speicherung von Studierendendaten**

Für ein Softwaresystem zur Verwaltung der Daten von Studierenden müssen deren Daten wie z.B. die Identifikationsnummer (*id*), der Name (*name*) und – vorerst – das Alter (*age*) verfügbar sein. Dazu könnte man (mit den bisher eingeführten Möglichkeiten) z.B. folgende drei Felder vorsehen:

```
const max = …
var studentIds:   array [1:max] of int
    studentNames: array [1:max] of string
    studentAges:  array [1:max] of int
```

Durch die in Beispiel 3.7 gezeigte Lösungsvariante werden zusammengehörende Eigenschaften auf mehrere Datenobjekte verteilt. Die Zusammengehörigkeit geht dabei verloren und ist aus der dazugehörigen Algorithmenbeschreibung nicht mehr so ohne Weiteres erkennbar.

Um diesen Nachteil zu vermeiden, führen wir das Konzept des Verbunds ein, für das wir eine Definition angeben, bevor wir uns mit den Details und den dafür notwendigen Konstrukten beschäftigen.

Definition: *Verbund*

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 die entsprechenden Bezeichner auch auf die einzelnen Komponenten zugreifen kann.

Verbunde verwenden wir immer dann, wenn wir für eine algorithmische Lösung ein Objekt aus der Problemdomäne modellieren müssen, das mehrere für den Algorithmus relevante Eigenschaften aufweist, die deshalb zusammengefasst werden sollen.

Ein Verbund (*compound*) *c* fasst benannte Komponenten so zu einer Einheit zusammen, dass auf ihn sowohl als Ganzes, als auch auf die einzelnen Komponenten separat zugegriffen werden kann. Für den Zugriff auf eine Verbundkomponente wird der *Komponentenselektionsoperator* . (Punkt) auf den Verbund *c* mit einem Komponentennamen *n* angewandt: *c.n* bezeichnet die Komponente mit dem Namen *n* des Verbunds *c*.

In der von uns verwendeten Notation zur Beschreibung von Algorithmen sieht die Deklaration einer Verbundvariable folgendermaßen aus:

```
var c = compound
      a: AnyType
      b: SameOrAnyOtherType
      …
      z: SameOrYetAnotherType
    end -- compound
```

▶Abbildung 3.8 zeigt, wie man sich die Repräsentation der Komponenten des oben deklarierten Verbunds *c* im Hauptspeicher eines Computers vorstellen kann. Wie die Elemente eines Feldobjekts liegen die Komponenten eines Verbundobjekts lückenlos hintereinander im Speicher, können aber – je nach Datentyp – unterschiedlichen Speicherbedarf aufweisen.

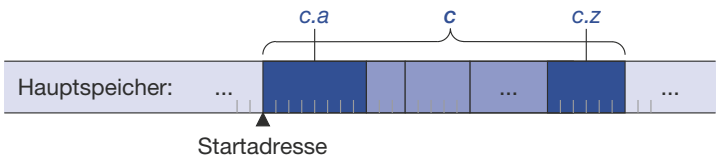

Abbildung 3.8: Anordnung der Komponenten eines Verbunds im Speicher

Hinweis: Wie ein Feld hat auch ein Verbund, wie in Abbildung 3.8 gezeigt, eine Startadresse. Da die einzelnen Komponenten aber unterschiedliche Datentypen haben und somit unterschiedlichen Speicherplatz beanspruchen können, existiert keine Speicherabbildungsfunktion zur effizienten Berechnung der Adresse der *i*-ten Komponente von Verbunden analog zu jener von Feldern; der Zugriff auf die einzelnen Komponenten ist nur über ihre Namen möglich.

Beispiel 3.8 zeigt eine Anwendung des Verbundkonzepts für die in Beispiel 3.7 noch separiert voneinander gespeicherten Daten von Studierenden.

Beispiel 3.8 **Speicherung von Studierendendaten in einem Verbund**

Für das in Beispiel 3.7 erwähnte Softwaresystem zur Verwaltung der Daten von Studierenden ist es sinnvoll, einen benutzerdefinierten Verbundtyp *Student* wie folgt einzuführen:

```
type Student = compound
       id: int
       name: string
       age: int
     end -- compound
```

Diesen Datentyp können wir nun verwenden, um Variablen für die einzelnen Studierenden zu deklarieren und ihre Komponenten mit entsprechenden Werten zu füllen:

```
  var s1, s2, … : Student  -- two students
begin
  s1.id := 2006099
  s1.name := "Max Mustermann"
  s1.age := 19
  s2.id := …
```

Da es hinsichtlich des Datentyps einer Komponente eines Verbunds keinerlei Einschränkungen gibt, kann (in Analogie zu Feldern) der Datentyp einer Verbundkomponente selbst wieder ein Verbund sein, so dass „Verbunde von Verbunden" möglich sind.

Beispiel 3.9 zeigt, wie man damit die Daten von Studierenden praxisgerechter speichern kann.

Beispiel 3.9 **Verbund in Verbund**

Wir können nun die nicht praxisgerechte Definition des in Beispiel 3.8 eingeführten Datentyps *Student* verbessern, denn es war keine gute Idee, wie in dieser Typdeklaration festgelegt, das Alter eines Studierenden zu speichern, da sich dieses permanent ändert. Es ist vielmehr naheliegend, eine Komponente für das Geburtsdatum vorzusehen.

Da Kalenderdaten öfter benötigt werden, ist es sinnvoll, dafür wiederum einen eigenen (Verbund-)Datentyp *Date* einzuführen und diesen bei der Definition des Datentyps *Student* zu verwenden, um statt der Alterskomponente eine Komponente für das Geburtsdatum (*dateOfBirth*) vorzusehen:

```
type Date = compound
       day, month, year: int
     end -- compound
     Student = compound
       …
       dateOfBirth: Date
     end -- compound
var s: Student
```

Die folgenden drei Beispiele für komponentenbezogene Wertzuweisungen zeigen, wie der Komponentenselektionsoperator – analog zum Indizierungsoperator – mehrfach angewendet werden kann, um auf „ineinander geschachtelte“ Komponenten zugreifen zu können:

```
s.dateOfBirth.day   :=    1
s.dateOfBirth.month :=    4
s.dateOfBirth.year  := 1987
```

3.2.3 Gegenüberstellung und Kombination von Feldern und Verbunden

Feld- und Verbundtypen gestatten es, strukturierte Datenobjekte zu bilden. Gemeinsam ist solchen Datenobjekten, dass sie mehrere Elemente/Komponenten so zu einer Einheit zusammenfassen, dass auf diese sowohl als Ganzes als auch auf die einzelnen Bestandteile zugegriffen werden kann.

Tabelle 3.1 stellt die beiden Konzepte einander gegenüber und zeigt wesentliche Unterschiede zwischen Feldern und Verbunden.

Sowohl Feld- als auch Verbundobjekte – jeweils für sich alleine betrachtet – ermöglichen es, bestimmte Sachverhalte in Algorithmen besser zu modellieren, als dies mit elementaren Datenobjekten alleine möglich wäre.

Tabelle 3.1

Unterschiede von Feldern und Verbunden

Unterscheidungsmerkmal	Felder	Verbunde
Datentyp der Bestandteile	*Elemente* müssen gleichen Datentyp haben	*Komponenten* können unterschiedliche Datentypen haben
Zugriff auf die Bestandteile	Mit Index i und Indexoperator: $a[i]$	Mit Name n und Komponentenselektionsoperator: $c.n$

Das Potenzial der Verfügbarkeit strukturierter Datenobjekte entfaltet sich aber erst, wenn Felder und Verbunde miteinander kombiniert werden. Das ist möglich, da es weder bezüglich des Elementtyps eines Felds noch bezüglich der Komponententypen in einem Verbund Einschränkungen gibt. Somit sind Felder von Verbunden und Verbunde mit Feldkomponenten möglich – und das in beliebig tiefer Schachtelung.

Beispiel 3.10 zeigt die Möglichkeit der Schachtelung von Feldern und Verbunden anhand der Speicherung mehrerer Studenten mit ihren Noten.

Beispiel 3.10 **Mehrere Studenten mit ihren Noten**

Für das in Beispiel 3.8 und Beispiel 3.9 erwähnte Softwaresystem zur Verwaltung der Daten von Studierenden ist es notwendig, auch die Noten der Studierenden zu speichern. Dazu kann beispielsweise – wie das Codestück unten zeigt – eine Komponente *marks*, die ein Feldobjekt repräsentiert, vorgesehen werden (vgl. auch Beispiel 3.3). Um eine einfache Verarbeitung der Daten aller Studierenden zu ermöglichen, ist es außerdem sinnvoll, ein Feld *students* mit dem Elementdatentyp *Student* vorzusehen.

```
const maxMarks = …
      maxStudents = …

type Student = compound
       id: int
       name: string
       dateOfBirth: Date
       marks: array [1:maxMarks] of int
       nMarks: int -- number of marks
     end -- compound

var students: array [1:maxStudents] of Student
    nStudents: int -- number of Students
```

Das folgende Codestück zeigt anhand einer Zuweisung, wie man Index- und Komponentenselektionsoperator kombinieren kann, um ausgehend von der Variablen *students* auf alle Bestandteile dieses strukturierten Datenobjekts zuzugreifen, beispielsweise auf die zweite Note des ersten Studenten.

```
students[1].marks[2] := 3
```

▶Abbildung 3.9 zeigt den hierarchischen Aufbau der Variablen *students*. Daraus ist ersichtlich, dass es sich um ein Feld von Verbunden des Datentyps *Student* handelt, wobei in jedem dieser Verbunde auch Felder (z.B. in Form des Datentyps *string*) als Komponenten vorkommen. Auf den untersten Ebenen der Hierarchie kommen nur mehr elementare Datentypen vor (hier z.B. *char* als Elemente von *string* und *int*).

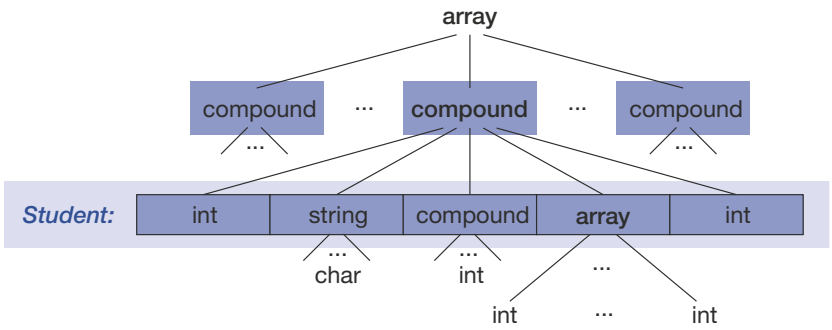

Abbildung 3.9: Hierarchischer Aufbau der Variablen *students*

3.3 Vernetzte oder dynamische Datenobjekte und -typen

Den elementaren und strukturierten Datentypen, die wir bisher behandelt haben, ist gemeinsam, dass Variablen dieser Datentypen während ihrer Lebensdauer ihren Wert, aber nicht ihre Struktur ändern können. Folglich bleibt die Größe des Speicherplatzes, den sie benötigen, konstant.

In der Praxis begegnen wir Aufgabenstellungen, in denen Datenobjekte vorkommen, die charakterisiert sind durch die Veränderung ihres Werts *und* ihrer Struktur, also Datenobjekte, die aus Komponenten bestehen, die miteinander in Beziehung stehen, also vernetzt sind. Wir nennen diese deshalb *vernetzte Datenstrukturen*, um hervorzuheben, dass der Struktur des Datenobjekts besondere Bedeutung zukommt. Solche Datenobjekte können zur Laufzeit eines Algorithmus, also dynamisch, wachsen und schrumpfen und/oder es kann sich die Beziehungsstruktur zwischen ihren Komponenten ändern, weshalb sie auch als *dynamische Datenstrukturen* bezeichnet werden. Das heißt, dass sowohl die Größe als auch die Struktur und der Inhalt der Datenstruktur veränderbar sind. Wichtige Repräsentanten solcher Datenobjekte mit Strukturcharakter sind *verkettete Listen* und *Bäume*, vor allem *Binärbäume*; sie werden später in eigenen Abschnitten ausführlich behandelt.

Bevor wir uns den verschiedenen Ausprägungen vernetzter Datenobjekte bzw. ihrer Datentypen widmen, müssen die dafür notwendigen Konzepte eingeführt werden: *Zeiger*(*datentypen*) und die dynamische *Speicherallokation* sowie *-freigabe*.

3.3.1 Zeiger und Zeigerdatentypen

Wie in Abschnitt 1.3 erläutert, werden alle Variablen, die in einem Algorithmus vorkommen (nach der Transformation des Algorithmus in ein Programm einer bestimmten Programmiersprache und seiner Übersetzung in eine ausführbare Version bei der Ausführung), im Speicher eines Computers repräsentiert: Für jede Variable wird dazu