

Datenorganisation und Datenstrukturen

Reihungen, Verbunde, Referenzen,
abstrakte Datentypen

Foliensatz von A. Weber zur Vorlesung Informatik I, Bonn, 2002/03
Überarbeitet von W. Küchlin zu Informatik I, Tübingen 2003/04

- **Datenstrukturen** sind höheres Organisationskonzept
 - Vgl. die bislang behandelten Binärcodierungen elementarer Datentypen
- Neben Algorithmen ein fundamentaler Bestandteil der Informatik
 - **Algorithmen und Datenstrukturen** Titel mancher einführender Vorlesungen in die Informatik
- In **diesem Abschnitt Grundbausteine von Datenstrukturen**
 - Einige höhere Datenstrukturen werden später behandelt werden

- Ein „klassisches Buch“



Algorithmen und Datenstrukturen

Niklaus Wirth

Pascal Version, 5 Auflage, B.G. Teubner Stuttgart

ISBN 3-519-02250-7 (1999)

320 Seiten, 93 Abbildungen, 30 Tabellen

- Eine **eindimensionale Reihung** (*array*) besteht aus einer bestimmten Anzahl von Daten gleicher Art
 - Kann als einzelne Zeile oder Spalte einer Tabelle gedacht werden
 - Auf jedes Element der Reihung kann mit demselben Zeitaufwand zugegriffen werden
 - Z. B. in der Form **a[i]**
 - Auf diese Art werden etwa Werte einer Funktion an den Stellen i gespeichert, wie z. B. die Werte eines Eingangsignals zu den Zeitpunkten $i=1, \dots, k$

Zeitpunkt	1	2	3	4	...	30	31
Signalstärke	10.5	10.5	12.2	9.8	...	13.1	13.3

Reihungen (arrays)

- Sind die Reihungselemente von einem Typ T (z. B. Ganzzahl), so ist die Reihung selbst vom Typ *Reihung von T*
- *Zweidimensionale Reihungen* speichern die Werte mehrerer eindimensionaler Zeilen (sofern alle vom gleichen Typ sind) in Tabellen-(Matrix-)Form
 - $a[i,j]$ ist das Element in der j -ten Spalte der i -ten Zeile
 - Alternative Syntax zu $a[i][j]$
- Entsprechend dreidimensionale (bzw. n -dimensionale) Reihungen

Reihungen (arrays)

- **Arrays** repräsentieren also **Funktionen** vom **Indexbereich** in einen **Wertebereich**, der der Typ der Array-Elemente ist
- Sei etwa $t: \text{IN} \times \text{IN} \rightarrow \text{IR}$ eine Funktion, die einem Koordinatenpaar einen Temperaturwert zuordnet
 - Der Wert der Funktion an der Stelle $(1,1)$, also $t(1,1)$ findet sich dann im Array t an der Stelle $t[1,1]$
- Arrays eignen sich in der Praxis grundsätzlich nur dann zur Speicherung einer Funktion, wenn diese dicht ist, d. h. wenn die Abbildung für die allermeisten Indexwerte definiert ist
 - Sonst würde eine Arraydarstellung viel zuviel Platz beanspruchen
 - Außerdem geht dies nur für endliche Funktionen

Reihungen (arrays): Zeichenreihen (strings)

- Wichtiger Spezialfall
 - Zeichenreihen (strings)
 - Reihung von Zeichen (array of char)
 - Viele Programmiersprachen haben dafür eigene Syntax
 - In Java z.B. Buchstabenfolge in Anführungszeichen
 - “Text“
 - » Ergibt folgendes Speicherbild

T	e	x	t
---	---	---	---
 - Für andere Reihungen Notation mit geschweiften Klammern
 - {10.5, 10.5, 12.2, 9.8,..., 13.1, 13.3}
 - {'T', 'e', 'x', 't'}

Verbunde (records, structs)

- Arrays modellieren also Beziehungen zwischen Elementen gleichen Typs
- Oft bestehen aber auch **Beziehungen zwischen Werten unterschiedlichen Typs**
 - Etwa zwischen Name und Monatsverdienst eines Beschäftigten
- Wir verbinden zusammengehörige Daten unterschiedlichen Typs zu einem **Verbund** (*record*, *structure*, *struct*)
- **Beispiel:** Stammdaten

Name	"Mustermann"
Vorname	"Martin"
GebTag	10
GebMonat	05
GebJahr	1930
Familienstand	"verheiratet"
...	...

Verbunde (records, structs)

- Übliche Syntax zur Auswahl: Punkt-Notation
 - Beispiel:

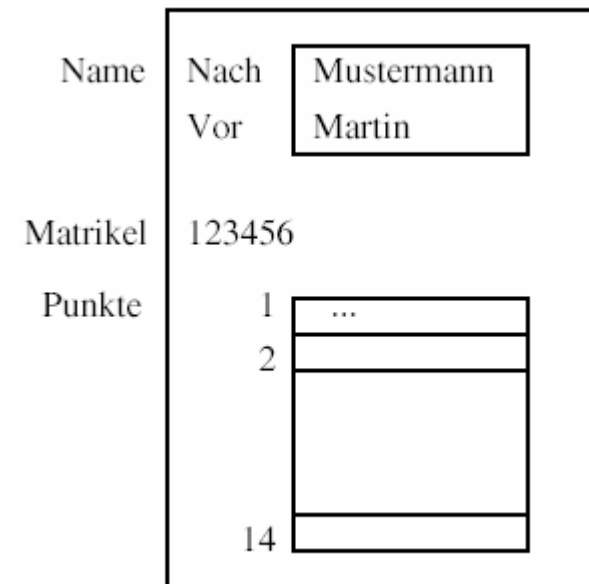
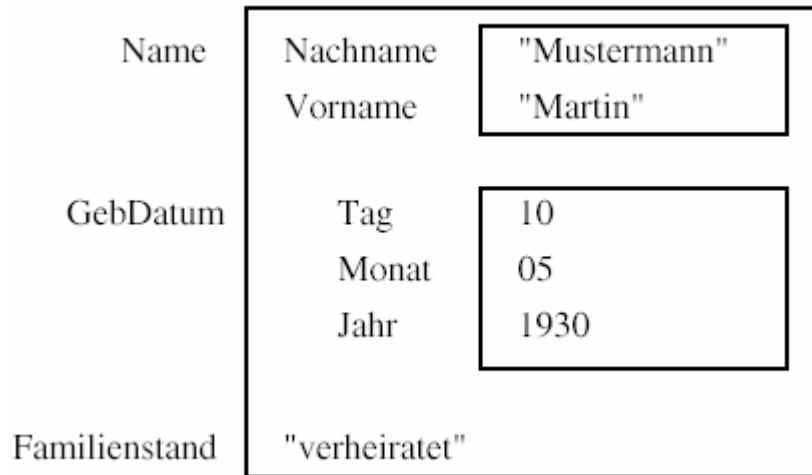
Sei ein konkretes Stammdatenblatt s gegeben. Dann ist

$s.Name = \text{"Mustermann"}$

der Wert der Komponente Name von s . Entsprechend gilt $s.GebTag = 10$ usw.

- Komponenten eines Verbunds können von beliebigem Typ sein
 - Also auch wieder Verbunde, Reihungen, etc.

- **Beispiele komplexer Verbunde:**



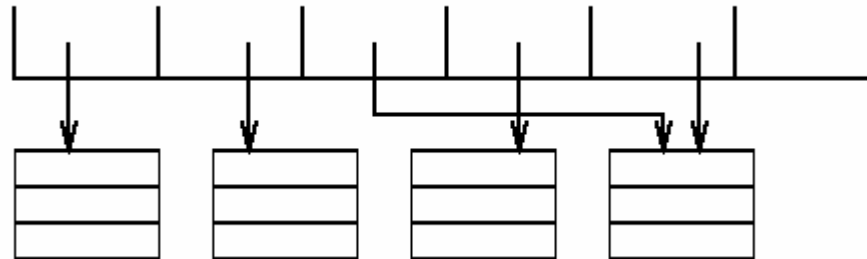
Modellierung des Enthaltenseins - Referenzen

- Ein Verbund kann in einem anderen enthalten sein
 - Vgl. Beispiel von Datum und Stammdatenblatt
- Diese Beziehung des Enthaltenseins (*containment*) kann auf zweierlei Arten modelliert werden
 - Als Enthaltensein durch Wert (*by value*)
 - Als Enthaltensein durch Referenz (*by reference*)

Modellierung des Enthaltenseins - Referenzen

- **Beispiel:**

- Studentin **Musterfrau** belegt zwei verschiedene Übungen
 - Übung durch Verbund realisiert
 - Enthält u.a. String **Übungsleiter**, Tabelle (Reihung) mit Stammdaten des Studierenden sowie einer Tabelle mit Punkten
 - Stammdatenblatt **Musterfrau** also in zwei verschiedenen Verbunden realisiert
 - Wenn dies durch *Enthaltensein durch Wert* modelliert wird, dann existieren zwei *separate Exemplare* des Stammdatenblatts. Wenn dies durch *Enthaltensein durch Referenz* modelliert wird, dann existiert nur **ein Exemplar** des Stammdatenblatts **Musterfrau**, auf das beide Verbunden *verweisen*



- Idee: **Datenstrukturen + Algorithmen** gehören zusammen
- Ein **abstrakter Datentyp** (*abstract data type*) bündelt die Definition einer **Trägermenge** (*carrier set*) von **Objekten** mit einer **Spezifikation** der zugehörigen Operationen (**Methoden**)
 - **algebraischer abstrakter Datentyp**:
 - Spezifikation algebraisch durch Gleichungen
- Datenrepräsentation und Implementierung der Methoden bleiben hinter einer abstrakten **Aufrufsstelle** (*call interface*) verborgen
 - Daten in den Objekten können nur über die Methoden manipuliert werden
 - **Geheimnisprinzip** (*principle of information hiding*)
 - Dadurch muss nur noch der **Hersteller (Programmierer)** des abstrakten Datentyps die Datenrepräsentation und die **Programmierung der Funktionen verstehen**
 - **Benutzer** braucht nur die Funktionsweise der **Schnittstelle zu verstehen**
- Beispiele:
<N; 0, 1, +, -, *>

< B; 0, 1, \wedge , \vee , \neg >

- Der abstrakte Datentyp **spezifiziert** erschöpfend die **Operationen**, die auf einem **Objekt ausgeführt** werden können
- Er kümmert sich nicht um die Realisierung
- Wegen dieser **Abstraktionsebene** heißt der Datentyp **abstrakt**
- Ein abstrakter Datentyp wird durch eine (Objekt-) **Klasse** implementiert (realisiert).
 - Danach sind die **Operationen sehr konkret ausführbar**
 - Z. B. Sortieren der Übungsliste
- Ein **Objekt** in einem abstrakten Datentyp ist im allgemeinen ein Verbund-Objekt, das **Daten** als **Objekt-Zustand** speichert
- Auf den Objekten operieren **Methoden**, die im Datentyp spezifiziert und in der **Klasse** implementiert sind und über das Objekt aufgerufen werden können.

- Ein **Objekt** ist ein (Daten-)Verbund zusammen mit den Algorithmen (Funktionen, Methoden), die auf Verbunden dieses Typs operieren können
 - Der Verbund speichert den Objekt-**Zustand**
 - Die Algorithmen (Methoden) definieren das mögliche Objekt-**Verhalten**
 - Manche Methoden können von außen auf dem Objekt aufgerufen werden
 - indem man dem Objekt eine (Aufruf-)Nachricht schickt
 - Andere Methoden sind von außen unsichtbar
 - Sichtbare Methoden definieren die **Schnittstelle** (*interface*) des Objekts
- Eine (Objekt-)**Klasse** fasst gleichartige Objekte zusammen
 - mit gleichen Methoden und Daten gleichen Typs
 - **Klassendeklaration** definiert den Typ des Verbundes und listet die Implementierung der Methoden
 - Eine Klasse **implementiert** einen abstrakten Datentyp
- Ein **Objekt** ist eine konkrete **Instanz** einer Klasse
 - ein neues Exemplar eines Verbundes mit separaten Daten
 - gebündelt mit den für die Klasse definierten Methoden
 - jede Methode wird auf einem konkreten Objekt aktiviert
- `Rational r = new Rational(2,4); r.normalize(); bool b = r.isZero();`

- Eine **objektorientierte Sprache** erlaubt es, diese **Kopplung** mit **Sprachkonstrukten** zu **fixieren**
 - statt sie - wie etwa in C oder Pascal - nur der Selbstdisziplin des Programmierers zu überlassen
 - Die oft unter hartem Projektdruck leiden ☺
- Außerdem unterstützt eine objektorientierte Sprache den Programmierer durch die **höheren objektorientierten Konzepte** von
 - **Vererbung** (inheritance)
 - **Virtuellen Funktionen** (virtual function)
 - mit **dynamischem Binden** (dynamic binding) und
 - **generischem Programmieren** (generic programming)
- Dadurch können **Gemeinsamkeiten** in der entstehenden **Vielfalt** von abstrakten Datentypen **leichter herausgearbeitet** werden
 - und diese in **Hierarchien strukturiert** werden
- Hierdurch wird die **Komplexität** der Interaktion abstrakter Datentypen wesentlich **reduziert** und bleibt auch bei großen Anwendungen beherrschbar