Informatik I
Programmieren und Softwaretechnik

Peter Bastian
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen,
Universität Heidelberg
Im Neuenheimer Feld 368, 69120 Heidelberg,
`Peter.Bastian@iwr.uni-heidelberg.de`

Überarbeitet von:

Nicolas Neuß
Institut für Informatik,
Universität Heidelberg
Im Neuenheimer Feld 348, 69120 Heidelberg,
`Nicolas.Neuss@iwr.uni-heidelberg.de`

Version 2.0

Erstellt: 29. September 2006

URL für die Vorlesung:
`http://www-dbs.informatik.uni-heidelberg.de/teaching/ws2006/info1`

Inhaltsverzeichnis

1 Grundbegriffe **1**
1.1 Formale Systeme: MIU . . . 1
1.2 Beispiel: MIU-System . . . 1
1.3 Systematische Erzeugung aller Worte des MIU-Systems . . . 2
1.4 Lösung des MU-Rätsels . . . 2
1.5 Turingmaschine . . . 3
1.6 Problem, Algorithmus, Programm . . . 6
1.7 Turing-Äquivalenz . . . 7
1.8 Reale Computer . . . 8
1.9 Programmiersprachen . . . 9
1.10 Komplexität von Programmen . . . 10

2 Auswertung von Ausdrücken **11**
2.1 Arithmetische Ausdrücke . . . 11
2.1.1 Wie wertet der Rechner so einen Ausdruck aus? . . . 11
2.1.2 Auswertung eines zusammengesetzten Ausdrucks . . . 12
2.2 Funktionen . . . 12
2.3 Selektion . . . 13

3 Syntaxbeschreibung mit Backus-Naur Form **14**
3.1 EBNF . . . 14
3.2 Kurzschreibweisen . . . 15
3.3 Syntaxbeschreibung für FC++ . . . 15
3.4 Kommentare . . . 17

4 Das Substitutionsmodell **18**

5 Funktionen und Prozesse **19**
5.1 Linear-rekursive Prozesse . . . 19
5.2 Linear-iterative Prozesse . . . 20
5.3 Baumrekursion . . . 21
5.4 Größenordnung . . . 25
5.5 Wechselgeld . . . 26
5.6 Der größte gemeinsame Teiler . . . 28
5.7 Zahlendarstellung im Rechner . . . 29
5.7.1 Gebräuchliche Zahlenbereiche in C++ . . . 30
5.8 Darstellung reeller Zahlen . . . 31
5.9 Festkommazahlen . . . 31
5.10 Fließkommaarithmetik . . . 31
5.10.1 Typische Wortlängen . . . 32
5.10.2 Fehler in der Fließkommaarithmetik . . . 32
5.11 Typkonversion . . . 32

5.12 Wurzelberechnung mit dem Newtonverfahren 33

6 Fortgeschrittene funktionale Programmierung **35**
6.1 Funktionen in der Mathematik 35
6.2 Funktionale Programmiersprachen 35
6.3 Warum funktionale Programmierung? 36

7 Prozedurale Programmierung **37**
7.1 Lokale Variablen und die Zuweisung 37
7.1.1 Konstanten 37
7.1.2 Variablen 38
7.1.3 Problematik der Zuweisung 38
7.1.4 Umgebungsmodell 39
7.2 Syntax von Variablendefinition und Zuweisung 39
7.2.1 Lokale Umgebung 40
7.3 Anweisungsfolgen (Sequenz) 41
7.3.1 Beispiel 41
7.4 Bedingte Anweisung (Selektion) 42
7.5 Schleifen 42
7.5.1 While-Schleife 43
7.5.2 For-Schleife 43
7.5.3 Beispiele 44
7.5.4 Schleifen in funktionalen Sprachen 45

8 Benutzerdefinierte Datentypen **45**
8.1 Aufzählungstyp 46
8.2 Felder 47
8.2.1 Sieb des Eratosthenes 47
8.3 Zeichen und Zeichenketten 49
8.3.1 Datentyp `char` 49
8.3.2 ASCII 49
8.3.3 Zeichenketten 50
8.4 Typedef 51
8.5 Das Acht-Damen-Problem 52

9 Einschub: Wiederholung Aufwand **55**
9.1 Beispiel 1: Telefonbuch 55
9.2 Beispiel 2: Pascal-Dreieck 56
9.3 Zusammengesetzte Datentypen 57
9.3.1 Anwendung: Rationale Zahlen 57
9.3.2 Gemischtzahlige Arithmetik 60

10 Globale Variablen und das Umgebungsmodell **62**
10.1 Globale Variablen . . . 62
10.1.1 Beispiel: Konto . . . 62
10.2 Das Umgebungsmodell . . . 63
10.3 Stapel . . . 66
10.4 Monte-Carlo Methode zur Bestimmung von π . . . 67
10.4.1 Pseudo-Zufallszahlen . . . 67
10.4.2 Monte-Carlo funktional . . . 69

11 Zeiger und dynamische Datenstrukturen **70**
11.1 Zeiger . . . 70
11.2 Zeiger im Umgebungsmodell . . . 71
11.3 Call by reference . . . 73
11.3.1 Referenzen in C++ . . . 74
11.4 Zeiger und Felder . . . 74
11.5 Zeiger auf zusammengesetzte Datentypen . . . 75
11.6 Problematik von Zeigern . . . 76
11.7 Dynamische Speicherverwaltung . . . 77
11.7.1 Probleme bei dynamischen Variablen . . . 78
11.8 Die einfach verkettete Liste . . . 78
11.8.1 Initialisierung . . . 80
11.8.2 Durchsuchen . . . 80
11.8.3 Einfügen . . . 80
11.8.4 Entfernen . . . 81
11.8.5 Kritik am Programmdesign . . . 82
11.8.6 Listenvarianten . . . 83
11.9 Endliche Menge . . . 83
11.9.1 Schnittstelle . . . 83
11.9.2 Datentyp und Initialisierung . . . 84
11.9.3 Test auf Mitgliedschaft . . . 84
11.9.4 Einfügen in eine Menge . . . 85
11.9.5 Ausgabe . . . 85
11.9.6 Entfernen . . . 85
11.9.7 Vollständiges Programm . . . 86

12 Klassen **87**
12.1 Motivation . . . 87
12.2 Klassendefinition . . . 87
12.3 Objektdefinition . . . 88
12.4 Kapselung . . . 88
12.5 Konstruktoren und Destruktoren . . . 90
12.6 Implementierung der Klassenmethoden . . . 91
12.7 Klassen im Umgebungsmodell . . . 92
12.8 Beispiel: Monte-Carlo objektorientiert . . . 94

12.8.1 Zufallsgenerator 94
12.8.2 Klasse für das Experiment 95
12.8.3 Monte-Carlo-Funktion und Hauptprogramm 95
12.9 Initialisierung von Unterobjekten 96
12.10 Selbstreferenz 97
12.11 Überladen von Funktionen und Methoden 98
12.11.1 Automatische Konversion 98
12.11.2 Überladen von Methoden 99
12.12 Objektorientierte und funktionale Programmierung 100
12.13 Operatoren 102
12.14 Anwendung: rationale Zahlen objektorientiert 102
12.15 Beispiel: Turingmaschine 105
12.15.1 Band 105
12.15.2 TM-Programm 105
12.15.3 Turingmaschine 106
12.15.4 Turingmaschinen-Hauptprogramm 107
12.16 Abstrakter Datentyp 110
12.16.1 Beispiel 1: Positive $\mathbf{m}$-Bit-Zahlen im Computer 110
12.16.2 Beispiel 2: ADT Stack 111
12.16.3 Beispiel 3: Das Feld 112

13 Klassen und dynamische Speicherverwaltung 113
13.1 Klassendefinition 113
13.2 Konstruktor 114
13.2.1 Ausnahmen 114
13.3 Indizierter Zugriff 115
13.4 Copy–Konstruktor 116
13.5 Zuweisungsoperator 117
13.6 Hauptprogramm 118
13.7 Default-Methoden 118

14 Vererbung von Schnittstelle und Implementierung 120
14.1 Motivation: Polynome 120
14.2 Implementation 121
14.3 Öffentliche Vererbung 121
14.4 Beispiel zu public/private und öffentlicher Vererbung 122
14.5 Ist-ein-Beziehung 123
14.6 Konstruktoren, Destruktor und Zuweisungsoperatoren 124
14.7 Auswertung 124
14.8 Weitere Methoden 124
14.9 Gleichheit 126
14.10 Benutzung von `Polynomial` 126
14.11 Diskussion 127
14.12 Private Vererbung 128

14.12.1 Eigenschaften der privaten Vererbung 128
14.13 Zusammenfassung 129

15 Methodenauswahl und virtuelle Funktionen 130
15.1 Motivation: Feld mit Bereichsprüfung 130
15.2 Virtuelle Funktionen 131

16 Abstrakte Klassen 133
16.1 Motivation 133
16.2 Schnittstellenbasisklassen 133
16.3 Beispiel: geometrische Formen 135
16.4 Beispiel: Nochmals funktionales Programmieren 137
16.5 Beispiel: Exotische Felder 138
16.5.1 Dynamisches Feld 138
16.5.2 Listenbasiertes Feld 139
16.5.3 Anwendung 142
16.6 Zusammenfassung 144

17 Generische Programmierung 145
17.1 Funktionsschablonen 145
17.1.1 Beispiel: wieder funktionales Programmieren 146
17.2 Klassenschablonen 147
17.2.1 Beispiel: Feld fester Größe 150
17.2.2 Beispiel: Smart Pointer 151

18 Effizienz generischer Programmierung 154
18.1 Beispiel: Bubblesort 154
18.2 Effizienz 155
18.3 RISC 157
18.3.1 Aufbau eines RISC-Chips 158
18.3.2 Befehlszyklus 158
18.3.3 Pipelining 158
18.3.4 Probleme mit Pipelining 159
18.3.5 Funktionsaufrufe 159
18.3.6 Realisierung virtueller Funktionen 160
18.3.7 Inlining 160
18.4 Zusammenfassung 162
18.4.1 Nachteile der generischen Programmierung 162

19 Containerklassen 163
19.1 Motivation 163
19.2 Listenschablone 163
19.3 Iteratoren 165
19.4 Doppelt verkettete Liste 167

19.4.1 Struktur . . . 167
19.4.2 Implementation . . . 168
19.4.3 Verwendung . . . 172
19.4.4 Diskussion . . . 173
19.4.5 Beziehung zur STL-Liste . . . 173
19.5 Feld . . . 174
19.6 Stack . . . 177
19.7 Queue . . . 178
19.8 DeQueue . . . 179
19.9 Prioritätswarteschlangen . . . 180
19.10 Set . . . 182
19.11 Map . . . 184
19.12 Anwendung: Huffman-Kode . . . 186
19.12.1 Trie . . . 187
19.12.2 Konstruktion von Huffmankodes . . . 187
19.12.3 Implementation . . . 188

20 Effiziente Algorithmen und Datenstrukturen **191**
20.1 Heap . . . 191
20.1.1 Einfügen . . . 192
20.1.2 Reheap . . . 193
20.1.3 Entfernen des Wurzelelements . . . 193
20.1.4 Komplexität . . . 193
20.1.5 Datenstruktur . . . 193
20.1.6 Implementation . . . 194

21 Sortieren **197**
21.1 Das Sortierproblem . . . 197
21.2 Sortierverfahren mit quadratischer Komplexität . . . 197
21.2.1 Selectionsort (Sortieren durch Auswahl) . . . 197
21.2.2 Bubblesort . . . 198
21.2.3 Insertionsort (Sortieren durch Einfügen) . . . 199
21.3 Sortierverfahren optimaler Ordnung . . . 199
21.3.1 Mergesort (Sortieren durch Mischen) . . . 199
21.3.2 Heapsort . . . 202
21.3.3 Quicksort . . . 203
21.3.4 Anwendung . . . 204
21.4 Suchen . . . 206
21.4.1 Binäre Suche in einem Feld . . . 206
21.4.2 Binäre Suchbäume . . . 207
21.4.3 Einfügen und Löschen . . . 208
21.4.4 Ausgeglichene Bäume . . . 209
21.4.5 Implementation von (a,b)-Bäumen . . . 210
21.4.6 Literatur . . . 214

1 Grundbegriffe

1.1 Formale Systeme: MIU

Definition: (Wikipedia) Ein formales System ist ein System von Symbolketten und Regeln. Die Regeln sind Vorschriften für die Umwandlung einer Symbolkette in eine andere.
Mathematisch: $F = (\mathcal{A}, \mathcal{B}, \mathcal{X}, \mathcal{R})$, wobei

- $\mathcal{A}$ das Alphabet, eine Menge von Symbolen
- $\mathcal{B}$ die Menge der wohlgebildeten Worte,
- $\mathcal{X} \subset \mathcal{B}$ die Menge der Axiome und
- $\mathcal{R}$ die Menge der Produktionsregeln

sind.

1.2 Beispiel: MIU-System

Das MIU-System handelt von Wörtern (Zeichenketten), die nur aus den drei Buchstaben M, I, und U bestehen.

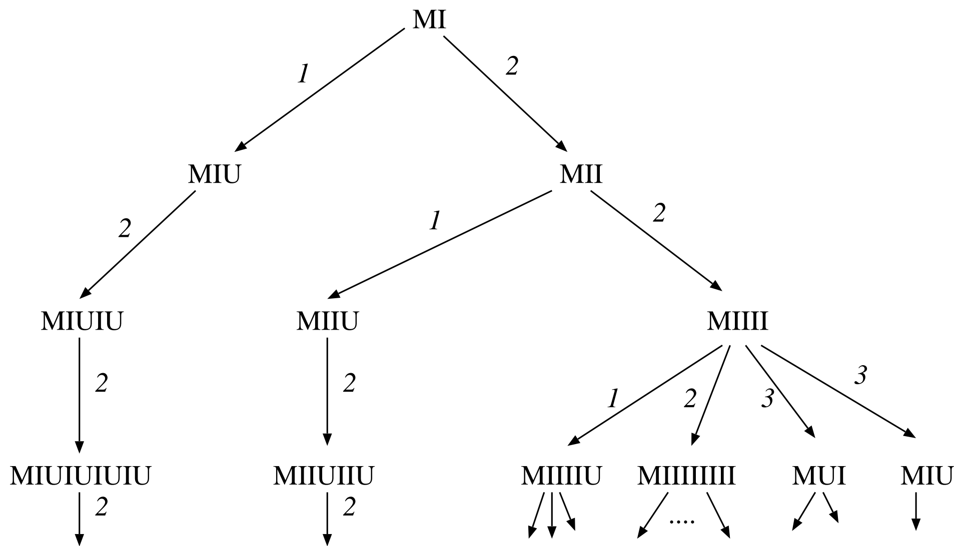
- $\mathcal{A}_{\text{MIU}} = \{\text{M}, \text{I}, \text{U}\}$.
- $\mathcal{X}_{\text{MIU}} = \{\text{MI}\}$.
- $\mathcal{R}_{\text{MIU}}$ enthält die Regeln:
  1. $\text{M}x\text{I} \to \text{M}x\text{IU}$. Hierbei steht x für eine beliebige Zeichenkette.
     Beispiel: MI $\to$ MIU. Man sagt MIU wird aus MI abgeleitet.
  2. $\text{M}x \to \text{M}xx$.
     Beispiele: MI $\to$ MII, MIUUI $\to$ MIUUIIUUI.
  3. $\text{M}x\text{III}y \to \text{M}x\text{U}y$ (x und y sind wieder beliebige Zeichenketten).
     Beispiele: MIII $\to$ MU, UIIIIM $\to$ UUIM, UIIIIM $\to$ UIUM.
  4. $\text{M}x\text{UU}y \to \text{M}xy$.
     Beispiele: UUU $\to$ U, MUUUIII $\to$ MUIII.
- $\mathcal{B}_{\text{MIU}}$ sind dann alle Worte die ausgehend von den Elementen von $\mathcal{X}$ mithilfe der Regeln aus $\mathcal{R}$ erzeugt werden können, also $\mathcal{B} = \{\text{MI}, \text{MIU}, \text{MIUUI}, \ldots\}$.

Beobachtung: B_{MIU} enthält offenbar unendlich viele Worte.

Problem: Ist MU ein Wort des MIU-Systems?
Oder mathematisch: $\text{MU} \in \mathcal{B}_{\text{MIU}}$?

1.3 Systematische Erzeugung aller Worte des MIU-Systems

Dies führt auf folgende Baumstruktur:



Beschreibung: Ganz oben steht das Anfangswort MI. Auf MI sind nur die Regeln 1 und 2 anwendbar. Die damit erzeugten Wörter stehen in der zweiten Zeile. Ein Pfeil bedeutet, dass ein Wort aus dem anderen ableitbar ist. Die Zahl an dem Pfeil ist die Nummer der angewendeten Regel. In der dritten Zeile stehen alle Wörter, die durch Anwendung von zwei Regeln erzeugt werden können, usw.

Bemerkung: Wenn man den Baum in dieser Reihenfolge durchgeht (Breitendurchlauf), so erzeugt man nach und nach alle Wörter des MIU-Systems.

Folgerung: Falls $\mathrm{MU} \in \mathcal{B}_{\mathrm{MIU}}$, wird dieses Verfahren in endlicher Zeit die Antwort liefern. Wenn dagegen $\mathrm{MU} \notin \mathcal{B}_{\mathrm{MIU}}$, so werden wir es mit obigem Verfahren nie erfahren!

Sprechweise: Man sagt: Die Menge $\mathcal{B}_{\mathrm{MIU}}$ ist *rekursiv aufzählbar*.

Frage: Wie löst man nun das MU-Rätsel?

1.4 Lösung des MU-Rätsels

Referenz: Siehe `http://home.arcor.de/rainer.randig/projekte/mathe-kunst-2/mu-raetsel.htm`.

Zur Lösung muss man das Rätsel in einer Metasprache analysieren, die über das einfache Anwenden der Regeln hinausgeht.

Beobachtung: Alle Ketten haben immer M vorne. Auch gibt es nur dieses eine M, das man genausogut hätte weglassen können. Hofstadter wollte aber das Wort MU herausbekommen, das in Zen-Koans eine Rolle spielt:

Ein Mönch fragte einst Meister Chao-chou:
„Hat ein Hund wirklich Buddha-Wesen oder nicht?“
Chao-chou sagte: „Mu.“

Beobachtung: Die Zahl der I in einzelnen Worten ist niemals ein Vielfaches von 3, also auch nicht 0.

Beweis: Ersieht man leicht aus den Regeln.

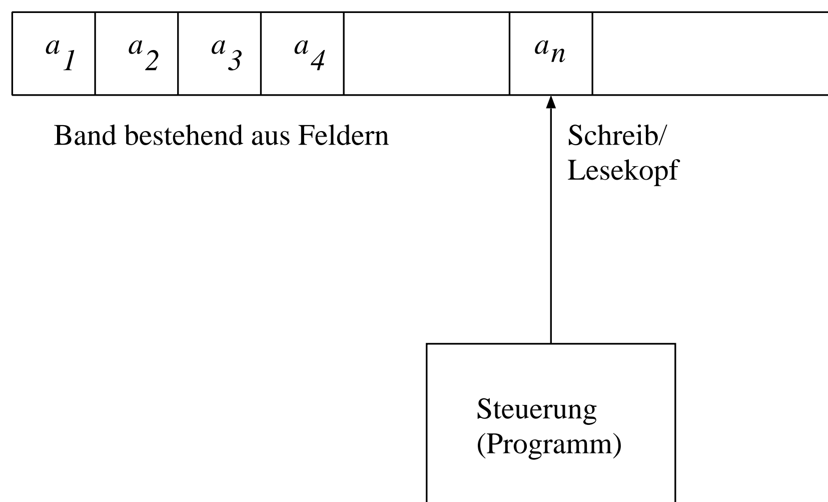
Wissen: Der Logiker Kurt Gödel hat gezeigt, dass es in allen „ausreichend ausdrucksstarken“ Regelsystemen Worte gibt, deren Korrektheit man innerhalb des Systems nicht beweisen kann. Mehr dazu in der mathematischen Logik.

1.5 Turingmaschine

Als weiteres Beispiel für ein „Regelsystem“ betrachten wir die Turingmaschine (TM).
Diese wurde 1936 von Alan Turing (1912-1954) zum theoretischen Studium der Berechenbarkeit eingeführt.

Wissen: Der sogenannte Turing-Preis (Turing Award) ist so etwas wie der „Nobelpreis der Informatik“.

Eine TM besteht aus einem festen Teil („Hardware“) und einem variablen Teil („Software“). TM bezeichnet somit nicht eine Maschine, die genau eine Sache tut, sondern ist ein allgemeines Konzept, welches eine ganze Menge von verschiedenen Maschinen definiert. Alle Maschinen sind aber nach einem festen Schema aufgebaut.
Die Hardware besteht aus einem einseitig unendlich großen Band welches aus einzelnen Feldern besteht, einem Schreib-/Lesekopf und der Steuerung. Jedes Feld des Bandes trägt ein Zeichen aus einem frei wählbaren (aber für eine Maschine festen) Bandalphabet (Menge von Zeichen). Der Schreib-/Lesekopf ist auf ein Feld positioniert, welches dann gelesen oder geschrieben werden kann. Die Steuerung enthält den variablen Teil der Maschine und wird nun beschrieben.



Die Steuerung kann folgende Operationen auf der Hardware ausführen:

- Überschreibe Feld unter dem Schreib-/Lesekopf mit einem Zeichen und gehe ein Feld nach rechts.
- Überschreibe Feld unter dem Schreib-/Lesekopf mit einem Zeichen und gehe ein Feld nach links.

Die Steuerung selbst besteht aus einer Tabelle, die beschreibt wie man von einem Zustand in einen anderen gelangen kann. Diese Tabelle nennt man auch Programm.

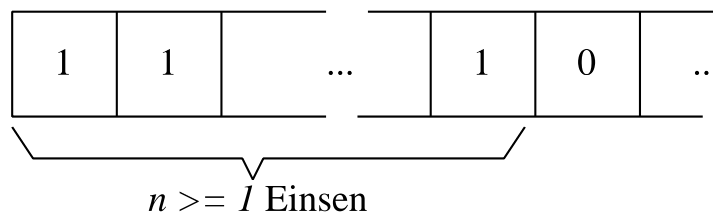
Beispiel:

Zustand	Eingabe	Operation	Folgezustand
1	0	0,links	2
2	1	1,rechts	1

Die Maschine funktioniert nun in einzelnen Schritten. Am Anfang jedes Schrittes ist die Maschine in einem bestimmten Zustand q und unter dem Schreib-/Lesekopf befindet sich ein Zeichen x, die Eingabe. Das Paar (q, x) bestimmt nun die Zeile der Tabelle in der man die auszuführende Operation b und den Folgezustand q' findet. Die Operation b wird nun ausgeführt und die Steuerung in den Zustand q' gesetzt. Damit ist die Maschine bereit für den nächsten Schritt.
Damit die Maschine starten und stoppen kann, gibt es noch zwei ausgezeichnete Zustände:

- Die Verarbeitung beginnt im Anfangszustand.
- Landet die Maschine im Endzustand wird die Bearbeitung gestoppt.

Beispiel: Löschen einer Einserkette. Das Bandalphabet enthalte nur die Zeichen 0 und 1. Zu Beginn der Bearbeitung habe das Band folgende Gestalt:



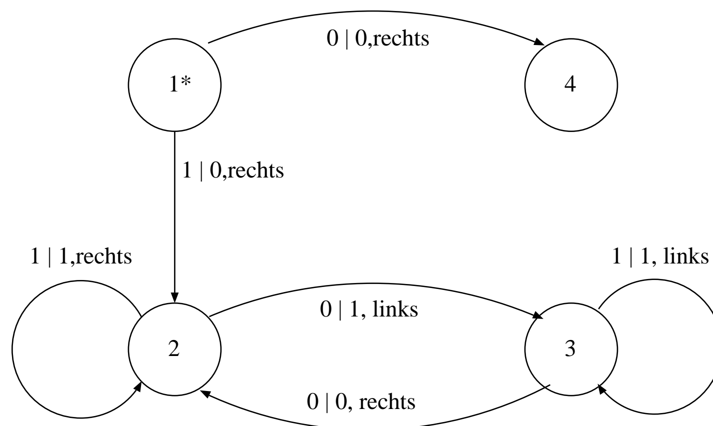
Der Kopf steht zu Beginn auf dem ersten Feld. Folgendes Programm mit zwei Zuständen löscht die Einserkette und stoppt:

Zustand	Eingabe	Operation	Folgezustand	Bemerkung
1	1	0,rechts	1	Anfangszustand
	0	0,rechts	2	
2				Endzustand

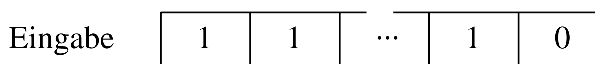
Beispiel: Raten Sie was folgendes Programm macht:

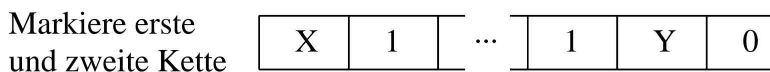
Zustand	Eingabe	Operation	Folgezustand	Bemerkung
1	1	0,rechts	2	Anfangszustand
	0	0,rechts	4	
2	1	1,rechts	2	
	0	1,links	3	
3	1	1,links	3	
	0	0,rechts	2	
4				Endzustand

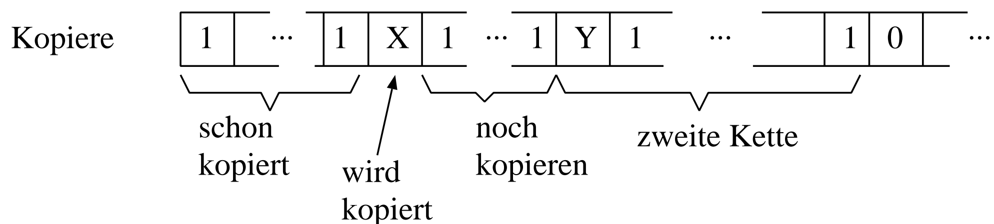
TM-Programme lassen sich übersichtlicher als *Übergangsgraph* darstellen. Jeder Knoten ist ein Zustand. Jeder Pfeil entspricht einer Zeile der Tabelle. Hier das Programm des vorigen Beispiels als Graph:



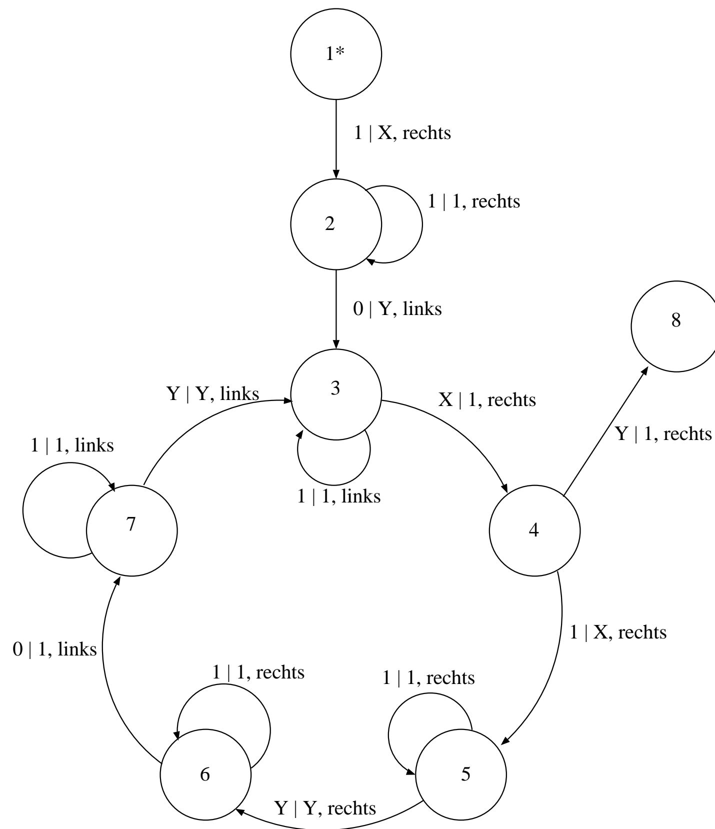
Beispiel: Verdoppeln einer Einserkette. Eingabe: n Einsen wie in Beispiel 1. Am Ende der Berechnung sollen ganz links $2n$ Einsen stehen, sonst nur Nullen.
Wie löst man das mit einer TM? Hier eine Idee:

Eingabe 

Markiere erste und zweite Kette 

Kopiere 

Das komplette Programm ist schon ganz schön kompliziert und sieht so aus:



Bemerkung: Wir erkennen die drei wesentlichen Komponenten von Berechnungsprozessen:

- Grundoperationen
- Selektion
- Wiederholung

1.6 Problem, Algorithmus, Programm

Definition: Ein Problem ist eine zu lösende Aufgabe.

Beispiel: Finde die kleinste von $n \geq 1$ Zahlen $x_1, \ldots, x_n$, $x_i \in \mathbb{N}$.

Definition: Ein Algorithmus beschreibt, eventuell in umgangssprachlicher Form, wie das Problem gelöst werden kann. Beispiele im Alltag sind Kochrezepte, Aufbauanleitung für Abholmöbel, etc.

Beispiel: Das Minimum von n Zahlen könnte man so finden: Setze $\min = x_1$. Falls $n = 1$ ist man fertig. Ansonsten teste der Reihe nach für $i = 2, 3, \ldots, n$ ob $x_i < \min$. Falls ja, setze $\min = x_i$.

Ein Algorithmus muss gewisse Eigenschaften erfüllen:

- Ein Algorithmus beschreibt ein generelles Verfahren zur Lösung einer Schar von Problemen.
- Trotzdem soll die Beschreibung des Algorithmus endlich sein. Nicht erlaubt ist also z. B. eine unendlich lange Liste von Fallunterscheidungen.
- Ein Algorithmus besteht aus einzelnen Elementaroperationen, deren Ausführung bekannt und endlich ist. Als Elementaroperationen sind also keine „Orakel“ erlaubt.

Bemerkung: Spezielle Algorithmen sind:

- Deterministische Algorithmen: In jedem Schritt ist bekannt, welcher Schritt als nächstes ausgeführt wird.
- Terminierende Algorithmen: Der Algorithmus stoppt für jede zulässige Eingabe nach endlicher Zeit.

Definition: Ein Programm ist eine Formalisierung eines Algorithmus. Ein Programm kann auf einer Maschine (z. B. TM) ausgeführt werden.

Beispiel: Das Minimum von n Zahlen kann mit einer TM berechnet werden. Die Zahlen werden dazu in geeigneter Form kodiert (z. B. als Einserketten) auf das Eingabeband geschrieben.
Wir haben also das Schema:

Problem $\Longrightarrow$ Algorithmus $\Longrightarrow$ Programm.

1.7 Turing-Äquivalenz

Auf einem PC mit unendlich viel Speicher könnte man mit Leichtigkeit eine TM simulieren. Das bedeutet, dass man zu jeder TM ein äquivalentes PC-Programm erzeugen kann, welches das Verhalten der TM Schritt für Schritt nachvollzieht. Ein PC (mit unendlich viel Speicher) kann daher alles berechnen, was eine TM berechnen kann.
Interessanter ist aber, dass man zeigen kann, dass die TM trotz ihrer Einfachheit alle Berechnungen durchführen kann, zu denen der PC in der Lage ist. Zu einem PC mit gegebenem Programm kann man also eine TM angeben, die die Berechnung des PCs nachvollzieht! Computer und TM können dieselbe Klasse von Problemen berechnen.

Bemerkung: Im Laufe von Jahrzehnten hat man viele (theoretische und praktische) Berechnungsmodelle erfunden. Die TM ist nur eines davon. Jedes Mal hat sich herausgestellt: Hat eine Maschine gewisse Mindesteigenschaften, so kann sie genausoviel wie eine TM berechnen. Dies nennt man *Turing-Äquivalenz*.
Die Church’sche These (Alonzo Church 1903-1995) lautet daher:

Alles was man für intuitiv berechenbar hält kann man mit einer TM ausrechnen.

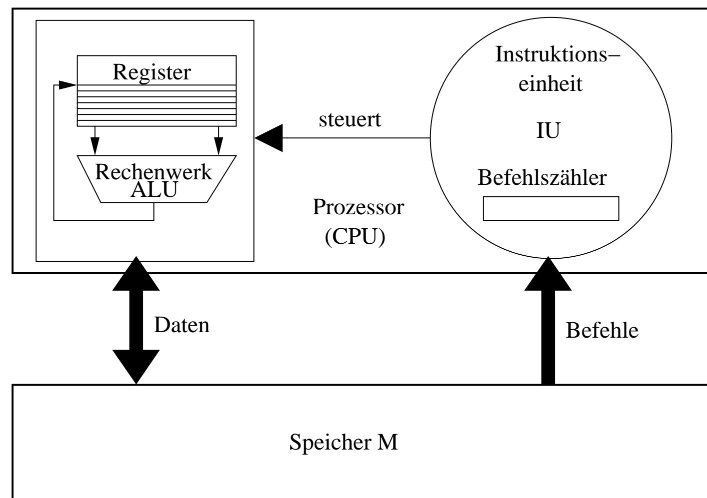
Dabei heißt intuitiv berechenbar, dass man einen Algorithmus dafür angeben kann.
Mehr dazu in Theoretische Informatik.

Folgerung: Berechenbare Probleme kann man mit fast jeder Computersprache lösen. Unterschiede bestehen aber in der Länge und Eleganz der dafür nötigen Programme. (Auch die Effizienz ihrer Ausführung kann sehr unterschiedlich sein, allerdings hängt dieser Punkt sehr von der Compilerimplementation ab.)

1.8 Reale Computer

Praktische Computer basieren meist auf dem von John von Neumann 1945 eingeführten Konzept (ähnliche Ideen wurden allerdings schon von Zuse 1937 veröffentlicht).

Geschichte: John von Neumann (1903-1957) war einer der bedeutendsten Mathematiker. Von ihm stammt die Spieltheorie, die mathematische Begründung der Quantenmechanik, sowie wichtige Beiträge zu Informatik und Numerik.



Der Speicher M besteht aus endlich vielen Feldern, von denen jedes eine Zahl aufnehmen kann. Im Unterschied zur TM kann auf jedes Feld ohne vorherige Positionierung zugegriffen werden (wahlfreier Zugriff, random access).
Der Speicher enthält sowohl Daten (das Band in der TM) als auch Programm (die Tabelle in der TM). Den einzelnen Zeilen der Programmtabelle der TM entsprechen beim von Neumannschen Rechner die Befehle.
Befehle werden von der Instruktionseinheit (instruction unit, IU) gelesen und dekodiert.
Die Instruktionseinheit steuert das Rechenwerk, welches noch zusätzliche Daten aus dem Speicher liest bzw. Ergebnisse zurückschreibt.
Die Maschine arbeitet zyklisch die folgenden Aktionen ab:

- Befehl holen
- Befehl dekodieren

- Befehl ausführen

Dies nennt man Befehlszyklus. Viel mehr über Rechnerhardware erfährt man in der Vorlesung „Technische Informatik“.

Bemerkung: Weder Turing-Maschine noch das von Neumannsche Rechnerkonzept geben die Wirkungsweise moderne Rechner genau wieder. Diese stehen in Wechselwirkung mit der Außenwelt und haben insbesondere die Fähigkeit, auf äußere Einwirkungen hin (etwa Tastendruck) den Programmfluss zu unterbrechen und an anderer Stelle (Turingmaschine: in anderem Zustand) wieder aufzunehmen.

1.9 Programmiersprachen

Die Befehle, die der Prozessor ausführt, nennt man Maschinenbefehle oder auch Maschinensprache. Sie ist relativ umständlich, und es ist sehr mühsam größere Programme darin zu schreiben. Andererseits können ausgefeilte Programme sehr kompakt sein und sehr effizient ausgeführt werden.

Beispiel: Ein Schachprogramm auf einem 6502-Prozessor findet man unter

    http://www.6502.org/source/games/uchess/uchess.pdf

Es benötigt weniger als 1KB an Speicher!

Die weitaus meisten Programme werden heute in sogenannten *höheren Programmiersprachen* erstellt. Sinn einer solchen Sprache ist, dass der Programmierer Programme möglichst

- schnell (in Sinne benötigter Programmiererzeit) und
- korrekt (Programm löst Problem korrekt)

erstellen kann.
Wir lernen in dieser Vorlesung die Sprache C++. C++ ist eine Weiterentwicklung der Sprache C, die Ende der 1960er Jahre entwickelt wurde.
Programme in einer Hochsprache lassen sich *automatisch* in Programme der Maschinensprache übersetzen. Ein Programm, das dies tut, nennt man *Übersetzer* oder Compiler.
Ein Vorteil dieses Vorgehens ist auch, dass Programme der Hochsprache in verschiedene Maschinensprachen (*Portabilität*) übersetzt und andererseits verschiedene Hochsprachen auch in ein und dieselbe Maschinensprache übersetzt werden können (*Flexibilität*).

Frage: Warum gibt es verschiedene Programmiersprachen?

Antwort: Wie bei der Umgangssprache: teils sind Unterschiede historisch gewachsen, teils sind die Sprachen wie Fachsprachen auf verschiedene Problemstellungen hin optimiert.

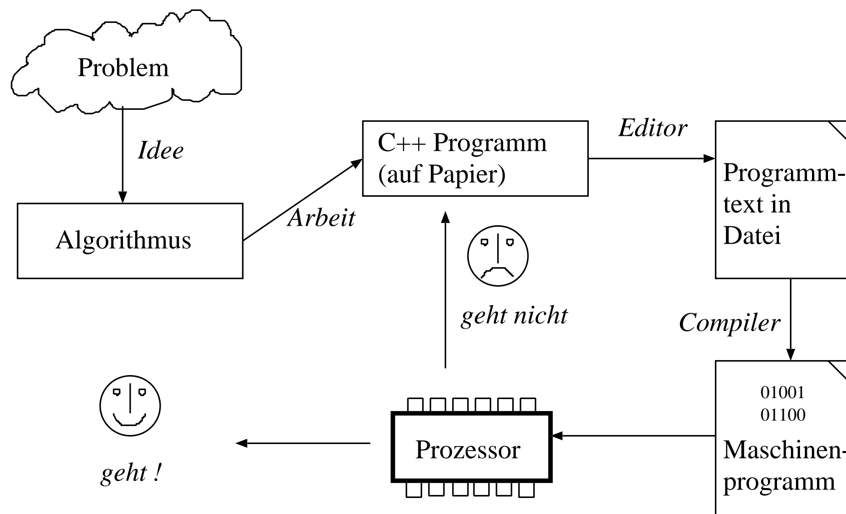


Abbildung 1: Workflow bei der Programmerstellung.

1.10 Komplexität von Programmen

Die Leistungsfähigkeit von Computern wächst schnell.
Moore'sches Gesetz: Die Leistung von Mikroprozessoren verdoppelt sich etwa alle zwei Jahre.

Beispiel: (Linux-Kernel)

Zeit	Proz	Takt	RAM	Disk	Linux Kernel
1982	Z80	6	64KB	800KB	6KB (CPM)
1988	80286	10	1MB	20MB	20KB (DOS)
1992	80486	25	20MB	160MB	140KB (0.95)
1995	PII	100	128MB	2GB	2.4MB (1.3.0)
1999	PII	400	512MB	10GB	13.2MB (2.3.0)
2001	PIII	850	512MB	32GB	23.2MB (2.4.0)

Offensichtlich wächst die Größe der Programme ähnlich Speicher und Rechenleistung!

Problem: Das Erstellen großer Programme skaliert mehr als linear, d. h. zum Erstellen eines doppelt so großen Programmes braucht man mehr als doppelt so lange.

Abhilfe: Verbesserte Programmiertechnik, Sprachen und Softwareentwurfsprozesse. Einen wesentlichen Beitrag leistet hier die objektorientierte Programmierung, die wir in dieser Vorlesung am Beispiel von C++ erlernen werden.

2 Auswertung von Ausdrücken

2.1 Arithmetische Ausdrücke

Beispiel: Auswertung von:

$$5 + 3 \text{ oder } ((3 + (5 * 8)) - (16 * (7 + 9))).$$

Programm:

```
#include <iostream>
using namespace std;

int main ()
{
    cout << ((3+(5*8))-(16*(7+9))) << endl;
}
```

Übersetzen (in Unix-Shell):

```
> g++ -o erstes erstes.cc
```

Ausführung:

```
> erstes
-213
```

Bemerkung:

- Ohne „-o erstes“ wäre der Name „a.out“ verwendet worden.
- Das Programm berechnet den Wert des Ausdrucks zwischen << ... << und druckt ihn auf der Konsole aus.

2.1.1 Wie wertet der Rechner so einen Ausdruck aus?

Die Auswertung eines zusammengesetzten Ausdruckes lässt sich auf die Auswertung der vier elementaren Rechenoperationen $+, -, *$ und $/$ zurückführen.
Dazu fassen wir die Grundoperationen als zweistellige Funktionen auf:

$$+, -, *, / \ : \ \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}.$$

Jeden Ausdruck können wir dann äquivalent umformen:

$$((3 + (5 * 8)) - (16 * (7 + 9))) \ \equiv \ -(+(3, *(5, 8)), *(16, +(7, 9))).$$

Definition: Die linke Schreibweise nennt man Infix-Schreibweise (*infix notation*), die rechte *Präfix-Schreibweise* (*prefix notation*).

Bemerkung: Die Infix-Schreibweise ist für arithmetische Ausdrücke bei Hinzunahme von Präzedenzregeln wie „Punkt vor Strich" und dem Ausnutzen von Kommutativitäts- und Assoziativgesetz kürzer und leichter lesbar als die Präfix-Schreibweise.

Bemerkung: Es gibt auch eine Postfix-Schreibweise, welche zum Beispiel in HP-Taschenrechnern, dem Emacs-Programm „Calc" oder der Computersprache Forth verwendet wird.

Die vier Grundoperationen $+, -, *, /$ betrachten wir als atomar. Im Rechner gibt es entsprechende Baugruppen, die diese atomaren Operationen realisieren.

2.1.2 Auswertung eines zusammengesetzten Ausdrucks

```
  -(+(3,*(5,8)),*(16,+(7,9)))
= -(+(3,  40  ),*(16,+(7,9)))
= -(    43     ,*(16,+(7,9)))
= -(    43     ,*(16,  16  ))
= -(    43     ,     256     )
=            -213
```

Bemerkung: Dies ist nicht die einzig mögliche Reihenfolge der Auswertung der Teiloperationen, alle Reihenfolgen führen jedoch zum gleichen Ergebnis!

2.2 Funktionen

Zu den schon eingebauten Funktionen wie $+, -, *, /$ kann man noch weitere benutzerdefinierte Funktionen hinzuzufügen.

Beispiel:

```
int quadrat (int x)
{
    return x*x;
}
```

Die erste Zeile (Funktionskopf) vereinbart, dass die neue Funktion namens `quadrat` als Argument eine Zahl `x` vom Typ `int` als Eingabe bekommt und einen Wert vom Typ `int` als Ergebnis liefert.
Der anschließende Funktionsrumpf (*body*) sagt, was die Funktion tut.

Bemerkung: C++ ist streng typgebunden (*strongly typed*), d. h. jedem Bezeichner (z. B. `x` oder `quadrat`) ist ein Typ zugeordnet. Diese Typzuordnung kann nicht geändert werden (statische Typbindung, *static typing*).

Bemerkung: Der Typ `int` entspricht dabei (kleinen) ganzen Zahlen. Andere Typen sind `float`, `double`, `char`, `bool`.

Programm: (Verwendung)

```
#include <iostream>
using namespace std;

int quadrat (int x)
{
    return x*x;
}

int main ()
{
    cout << (quadrat(3)+quadrat(4+4)) << endl;
}
```

Bemerkung:

- Neue Funktionen kann man (in C) nur in Präfix-Schreibweise definieren.
- `main` ist eine Funktion ohne Argumente und mit Rückgabetyp `int`.
- Die Programmausführung beginnt mit `main`.

2.3 Selektion

Fehlt noch: Steuerung des Programmverlaufs in Abhängigkeit von Daten.
Beispiel: Betragsfunktion

$$|x| = \begin{cases} -x & x < 0 \\ x & x \geq 0 \end{cases}$$

Um dies ausdrücken zu können, führen wir einen speziellen dreistelligen Operator `cond` ein:
Programm: **(Absolutwert)**

```
#include <iostream>
#include "cond.h"
using namespace std;

int absolut (int x)
{
    return cond( x<=0, -x , x);
}

int main ()
{
    cout << absolut(-3) << endl;
}
```

Der Operator cond erhält drei Argumente: Einen Boolschen Ausdruck und zwei normale Ausdrücke. Ein Boolscher Ausdruck hat einen der beiden Werte „wahr“ oder „falsch“ als Ergebnis. Ist der Wert „wahr“, so ist das Resultat des cond-Operators der Wert des zweiten Arguments, ansonsten der des dritten.

Bemerkung: cond kann keine einfache *Funktion* sein:

- cond kann auf verschiedene Typen angewendet werden, und auch der Typ des Rückgabewerts steht nicht fest.
- Oft wird cond nicht alle Argumente auswerten dürfen, um nicht in Fehler oder Endlosschleifen zu geraten.

3 Syntaxbeschreibung mit Backus-Naur Form

3.1 EBNF

Die Regeln nach denen wohlgeformte Sätze einer Sprache erzeugt werden, nennt man Syntax. Die Syntax von Programmiersprachen ist recht einfach. Zur Definition verwendet man eine spezielle Schreibweise, die erweiterte Backus-Naur Form (EBNF):
Man unterscheidet in der EBNF folgende Zeichen bzw. Zeichenketten:

- Unterstrichene Zeichen oder Zeichenketten sind Teil der zu bildenden, wohlgeformten Zeichenkette. Sie werden nicht mehr durch andere Zeichen ersetzt, deshalb nennt man sie terminale Zeichen.
- Zeichenketten in spitzen Klammern, wie etwa < Z > oder < Ausdruck > oder < Zahl >, sind Symbole für noch zu bildende Zeichenketten. Regeln beschreiben, wie diese Symbole durch weitere Symbole und/oder terminale Zeichen ersetzt werden können. Da diese Symbole immer ersetzt werden, nennt man sie nichtterminale Symbole.
- $< \epsilon >$ bezeichnet das „leere Zeichen“.
- Die normal gesetzten Zeichen(ketten)

  $$::= \quad | \quad \{ \quad \} \quad \}^+ \quad [\quad]$$

  sind Teil der Regelbeschreibung und tauchen nie in abgeleiteten Zeichenketten auf.

Jede Regel hat ein Symbol auf der linken Seite gefolgt von „::=“. Die rechte Seite beschreibt, durch was das Symbol der linken Seite ersetzt werden kann.

Beispiel:

$$<\mathrm{A}> \quad ::= \quad \underline{\mathrm{a}} <\mathrm{A}> \underline{\mathrm{b}}$$
$$<\mathrm{A}> \quad ::= \quad < \epsilon >$$

Ausgehend vom Symbol $<\mathrm{A}>$ kann man somit folgende Zeichenketten erzeugen:

$$<\mathrm{A}> \to \underline{\mathrm{a}}<\mathrm{A}>\underline{\mathrm{b}} \to \underline{\mathrm{aa}}<\mathrm{A}>\underline{\mathrm{bb}} \to \ldots \to \underbrace{\underline{a}\ldots\underline{a}}_{n \text{ mal}}<\mathrm{A}>\underbrace{\underline{b}\ldots\underline{b}}_{n \text{ mal}} \to \underbrace{\underline{a}\ldots\underline{a}}_{n \text{ mal}}\underbrace{\underline{b}\ldots\underline{b}}_{n \text{ mal}}$$

Bemerkung: Offensichtlich kann es für ein Symbol mehrere Ersetzungsregeln geben. Wie im MIU-System ergeben sich die wohlgeformten Zeichenketten durch alle möglichen Regelanwendungen.

3.2 Kurzschreibweisen

Oder:
Das Zeichen „ | “ („oder“) erlaubt die Zusammenfassung mehrer Regeln in einer Zeile.
Beispiel: <A> ::= a <A> b | < ϵ >

Option:
<A> ::= [<B>]   ist identisch zu   <A> ::= <B> | < ϵ >

Wiederholung mit $n \geq 0$:
<A> ::= { <B> }   ist identisch mit   <A> ::= <A> <B> | < ϵ >

Wiederholung mit $n \geq 1$:
<A> ::= { <B> }$^+$ ist identisch zu   <A> ::= <A> <B> | <B>

3.3 Syntaxbeschreibung für FC++

Die bisher behandelte Teilmenge von C++ nennen wir FC++ („funktionales C++“ und wollen die Syntax in EBNF beschreiben.

Syntax: (Zahl)

<Zahl> ::= [+ | -] { <Ziffer> }$^+$

Syntax: (Ausdruck)

<Ausdruck> ::= <Zahl> | [-] <Bezeichner> |
(<Ausdruck> <Operator> <Ausdruck>) |
<Bezeichner> ([<Ausdruck> { , <Ausdruck> }]) |
<Cond>
<Bezeichner> ::= <Buchstabe> { <Buchstabe oder Zahl> }
<Operator> ::= + | - | * | /

Weggelassen: Regeln für <Buchstabe> und <Buchstabe oder Zahl>.
Hier die Syntax einer Funktionsdefinition in EBNF:

Syntax: (Funktionsdefinition)

<Funktion> ::= <Typ> <Name> (<formale Parameter>)
{ <Funktionsrumpf> }
<Typ> ::= <Bezeichner>
<Name> ::= <Bezeichner>
<formale Parameter> ::= < ϵ > |
<Typ> <Name> { , <Typ> <Name> }

Die Argumente einer Funktion in der Funktionsdefinition heissen formale Parameter. Sie bestehen aus einer kommaseparierten Liste von Paaren aus Typ und Name. Damit kann man also n-stellige Funktionen mit $n \geq 0$ erzeugen.

Für den Moment haben wir zwei Regeln für den Funktionsrumpf:

<Funktionsrumpf> ::= return <Ausdruck> ;
<Funktionsrumpf> ::= cout << <Ausdruck> << endl ;

Die erste Variante dient der Rückgabe des Wertes einer Funktion. Die zweite Variante tauchte in der Funktion `main` auf, um ein Ergebnis auszudrucken.
Hier ist noch die Syntax für die Selektion:

Syntax:(**Cond**)

<Cond>	::=	cond (<BoolAusdr> , <Ausdruck> , <Ausdruck>)
<BoolAusdr>	::=	true | false | (<Ausdruck> <VglOp> <Ausdruck>) |
		(<BoolAusdr> <LogOp> <BoolAusdr>) |
		! <BoolAusdr>
<VglOp>	::=	== | != | < | > | <= | >=
<LogOp>	::=	&& | ||

Bemerkung: Beachte dass der Test auf Gleichheit als == geschrieben wird!

Syntax: (FC++ Programm)

<FC++-Programm>	::=	{ <Include> } { <Funktion> }$^+$
<Include>	::=	#include " <Name> "

Bemerkung:(**Leerzeichen**) C++ Programme erlauben das Einfügen von Leerzeichen, Zeilenvorschüben und Tabulatoren („whitespace") um Programme für den Menschen lesbarer zu gestalten. Hierbei gilt folgendes zu beachten:

- Bezeichner, Zahlen, Schlüsselwörter und Operatorzeichen dürfen keinen Whitespace enthalten:
  - `zaehler` statt `zae hler`,
  - 893371 statt 89 3371,
  - `return` statt `re tur n`,
  - `&&` statt `& &`.
- Folgen zwei Bezeichner, Zahlen oder Schlüsselwörter nacheinander so muss ein Whitespace (also mindestens ein Leerzeichen) dazwischen stehen:
  - `int f(int x)` statt `intf(intx)`,
  - `return x;` statt `returnx;`.

Die obige Syntaxbeschreibung mit EBNF ist nicht mächtig genug, um fehlerfrei übersetzbare C++ Programme zu charakterisieren. So enthält die Syntaxbeschreibung üblicherweise nicht solche Regeln wie:

- Kein Funktionsname darf doppelt vorkommen.
- Genau eine Funktion muss `main` heissen.

- Namen müssen an der Stelle bekannt sein wo sie vorkommen.

Bemerkung: Mit Hilfe von EBNF lassen sich sogenannte kontextfreie Sprachen definieren. Entscheidend ist, dass in EBNF-Regeln links immer nur genau ein nichtterminales Symbol steht. Zu jeder kontextfreien Sprache kann man ein Programm (genauer: einen Kellerautomaten) angeben, das für jedes vorgelegte Wort in endlicher Zeit entscheidet, ob es in der Sprache ist oder nicht. Man sagt: kontextfreie Sprachen sind entscheidbar. Die Regel „Kein Funktionsname darf doppelt vorkommen“ lässt sich mit einer kontextfreien Sprache nicht formulieren und wird deshalb extra gestellt.

3.4 Kommentare

Mit Hilfe von Kommentaren kann man in einem Programmtext Hinweise an einen menschlichen Leser einbauen. Hier bietet C++ zwei Möglichkeiten an:

```
// nach // wird der Rest der Zeile ignoriert
/* Alles dazwischen ist Kommentar */
```

4 Das Substitutionsmodell

Selbst wenn ein Programm vom Übersetzer fehlerfrei übersetzt wird, muss es noch lange nicht korrekt funktionieren. Was das Programm tut bezeichnet man als *Semantik* (Bedeutungslehre). Das in diesem Abschnitt vorgestellte Substitutionsmodell kann die Wirkungsweise funktionaler Programme beschreiben.

Definition:(Substitutionsmodell) Die Auswertung von Ausdrücken geschieht wie folgt:

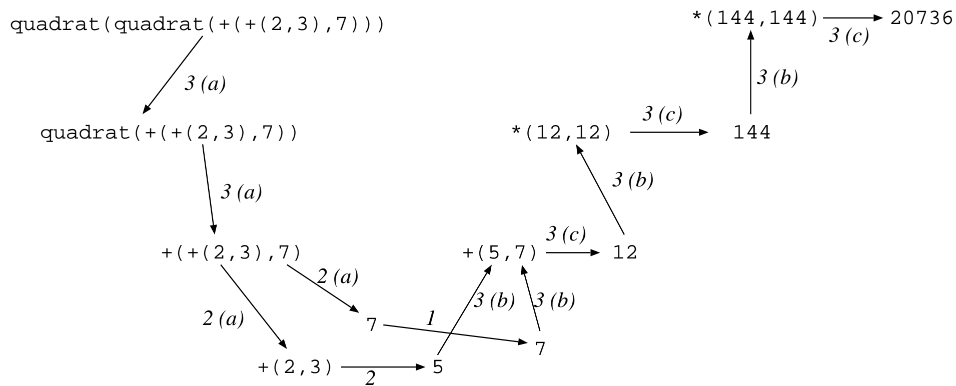
1. <Zahl> wird als die Zahl selbst ausgewertet.
2. <Name> ($<a_1>$, $<a_2>$, ..., $<a_n>$) wird für Elementarfunktionen folgendermaßen ausgewertet:
   a) Werte die Argumente aus. Diese sind wieder Ausdrücke. Unsere Definition ist also rekursiv!
   b) Werte die Elementarfunktion <Name> auf den so berechneten Werten aus.
3. < Name > ($<a_1>$, $<a_2>$, ..., $<a_n>$) wird für benutzerdefinierte Funktionen folgendermaßen ausgewertet:
   a) Werte die Argumente aus.
   b) Werte den Rumpf der Funktion <Name> aus, wobei jeder formalen Parameter durch den berechneten Wert des Arguments ersetzt wird.
4. cond ($<a_1>$, $<a_2>$, $<a_3>$) wird ausgewertet gemäß:
   a) Werte $<a_1>$ aus.
   b) Ist der erhaltene Wert `true`, so erhält man den Wert des cond-Ausdrucks durch Auswertung von $<a_2>$, ansonsten von $<a_3>$. *Wichtig:* nur *eines* der beiden Argumente $<a_2>$ oder $<a_3>$ wird ausgewertet.

Bemerkung: Die Namen der formalen Parameter sind egal, sie entsprechen sogenannten gebundenen Variablen in logischen Ausdrücken.

Beispiel:

```
quadrat(3) = *(3,3) = 9
```

```
quadrat(quadrat((2+3)+7))
   = quadrat(quadrat(+(+(2,3),7)))
   = quadrat(quadrat(+(   5   ,7)))
   = quadrat(quadrat(      12     ))
   = quadrat(       *(12,12)       )
   = quadrat(          144         )
   =          *(144,144)
   = 20736
```



5 Funktionen und Prozesse

5.1 Linear-rekursive Prozesse

Beispiel:(Fakultätsfunktion)

$$\begin{aligned} n! &= \prod_{i=1}^{n} i \\ &= 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \end{aligned}$$

Oder rekursiv:

$$n! = \begin{cases} 1 & n = 1 \\ n(n-1)! & n > 1 \end{cases}$$

Programm: **(Rekursive Berechnung der Fakultät)**

```
#include <iostream>
#include "cond.h"
using namespace std;

int fakultaet (int n)
{
    return cond( n<=1, 1, n*fakultaet(n-1) );
}

int main ()
{
    cout << fakultaet(17) << endl;
}
```

Die Auswertung kann mithilfe des Substitutionsprinzips wie folgt geschehen:

```
fakultaet(5) = *(5,fakultaet(4))
             = *(5,*(4,fakultaet(3)))
```

```
= *(5,*(4,*(3,fakultaet(2))))
= *(5,*(4,*(3,*(2,fakultaet(1)))))
= *(5,*(4,*(3,*(2,       1    ))))
= *(5,*(4,*(3,           2     )))
= *(5,*(4,               6      ))
= *(5,                  24       )
= 120
```

Definition: Dies bezeichnen wir als linear rekursiven Prozess (die Zahl der *verzögerten* Operationen wächst linear in n).

5.2 Linear-iterative Prozesse

Interessanterweise lässt sich die Kette verzögerter Operationen bei der Fakultätsberechnung vermeiden. Betrachte dazu folgendes Tableau von Werten von n und $n!$:

$$\begin{array}{cccccccccccccc} n & 1 & & 2 & & 3 & & 4 & & 5 & & 6 & \dots \\ & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ n! & 1 & \rightarrow & 2 & \rightarrow & 6 & \rightarrow & 24 & \rightarrow & 120 & \rightarrow & 720 & \dots \end{array}$$

Idee: Führe das Produkt als zusätzliches Argument mit.

Programm: **(Iterative Fakultätsberechnung)**

```
#include <iostream>
#include "cond.h"
using namespace std;

int fakIter (int produkt, int zaehler, int ende)
{
    return cond( zaehler>ende,
                 produkt,
                 fakIter(produkt*zaehler,zaehler+1,ende) );
}

int fakultaet (int n)
{
    return fakIter(1,1,n);
}

int main ()
{
    cout << fakultaet(5) << endl;
}
```

Die Analyse mit Hilfe des Substitutionsprinzips liefert:

```
fakultaet(5) = fakIter(1,1,5)
             = fakIter(1,2,5)
             = fakIter(2,3,5)
             = fakIter(6,4,5)
             = fakIter(24,5,5)
             = fakIter(120,6,5)
             = 120
```

Sprechweise: Dies nennt man einen linear iterativen Prozess. Der Zustand des Programmes lässt sich durch eine feste Zahl von Zustandsgrößen beschreiben (hier die Werte von `zaehler` und `produkt`). Es gibt eine Regel wie man von einem Zustand zum nächsten kommt, und es gibt den Endzustand.

Bemerkung:

- Von einem Zustand kann man ohne Kenntnis der Vorgeschichte aus weiterrechnen.
- Die Zahl der durchlaufenen Zustände ist proportional zu n.
- Die Informationsmenge zur Darstellung des Zustandes ist konstant.
- Bei geeigneter Implementierung ist der Speicherplatzbedarf konstant.
- Beim Lisp-Dialekt Scheme wird diese Optimierung von am Ende aufgerufenen Funktionen (*tail-call position*) im Sprachstandard verlangt.
- Bei anderen Sprachen (auch C++) ist diese Optimierung oft durch Compilereinstellungen erreichbar (nicht automatisch, weil das Debuggen erschwert wird).
- Beide Arten von Prozessen werden durch rekursive Funktionen beschrieben!
- Der entscheidende Unterschied ist, dass die Auswerteregeln von `cond` andere sind als von +!

5.3 Baumrekursion

Beispiel: (Fibonacci-Zahlen)

$$\mathrm{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & n > 1 \end{cases} .$$

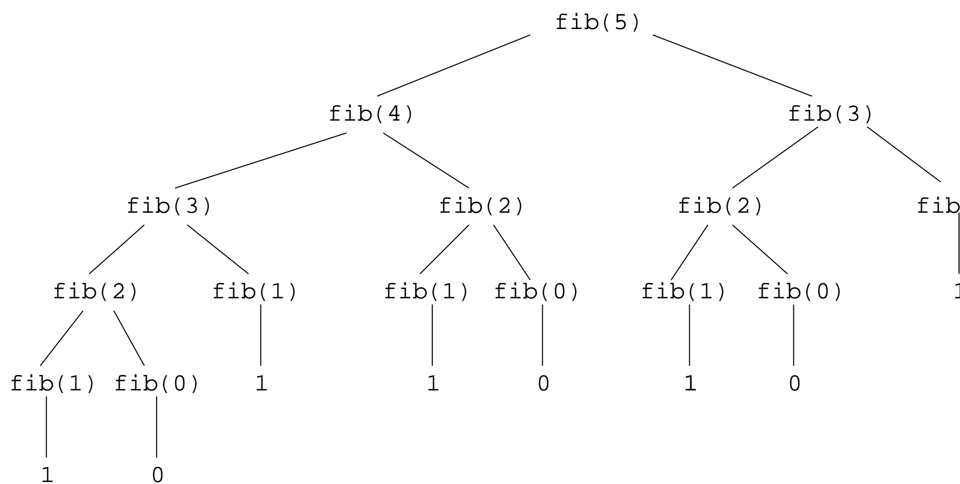
Programm: **(Fibonacci rekursiv)**

```
#include <iostream>
#include "cond.h"

int fib (int n)
{
    return cond( n==0, 0,
                 cond( n==1, 1,
                       fib(n-1)+fib(n-2) ) );
}

int main ()
{
    std::cout << fib(41) << std::endl;
}
```

Auswertung von `fib(5)` nach dem Substitutionsmodell:

```
  fib(5)
= +(fib(4),fib(3))
= +(+(fib(3),fib(2)),+(fib(2),fib(1)))
= +(+(+(fib(2),fib(1)),+(fib(1),fib(0))),+(+(fib(1),fib(0)),fib(1)))
= +(+(+(+(fib(1),fib(0)),fib(1)),+(fib(1),fib(0))),+(+(fib(1),fib(0)),fib(1)))
= +(+(+(+(   1  ,   0  ),   1  ),+(   1  ,   0  )),+(+(   1  ,   0  ),   1 ))
= +(+(+(        1         ,   1  ),         1       ),+(        1        ,   1 ))
= +(+(              2             ,         1       ),            2             )
= +(                           3                    ,             2             )
= 5
```

Baumdarstellung



`fib(5)` baut auf `fib(4)` und `fib(3)`, `fib(4)` baut auf `fib(3)` und `fib(2)`, usw.

Definition: Ein Baum besteht aus

1. einem Knoten und

2. $n \geq 0$ weiteren Bäumen, auf die der Knoten verweist. Diese Bäume heißen *Unter-* oder *Teilbäume*.

3. Wenn jeder Knoten auf höchstens zwei Kinderknoten verweist, so heißt der Baum *Binärbaum*.

Bezeichnung: Der Rekursionsprozess bei der Fibonaccifunktion heißt daher baumrekursiv.

Frage:

- Wie schnell wächst die Anzahl der Operationen bei der rekursiven Auswertung der Fibonaccifunktion?
- Wie schnell wächst die Fibonaccifunktion selbst?

Antwort: (Wachstum von fib)
$F_n :=$ fib(n) erfüllt die lineare 3-Term-Rekursion

$$F_n = F_{n-1} + F_{n-2}$$

Die Lösungen dieser Gleichung sind von der Form $a\lambda_1^n + b\lambda_2^n$, wobei $\lambda_{1/2}$ die Lösungen der quadratischen Gleichung $\lambda^2 = \lambda + 1$ sind, also $\lambda_{1/2} = \frac{1\pm\sqrt{5}}{2}$. Einbeziehen der Anfangsbedingungen $F_0 = 0, F_1 = 1$ liefert

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

für große n.

Antwort: (Aufwand zur rekursiven Berechnung von fib(n))

- Der Gesamtaufwand ist *größer* als eine Konstante multipliziert die Zahl A_n der Blätter im Berechnungsbaum. Diese erfüllt die Rekursion:

$$A_0 = 1\,,\ A_1 = 1\,,\ A_n = A_{n-1} + A_{n-2}\,,\quad n > 1$$

  woraus man $A_n =$ fib$(n+1) \approx \lambda_1^{n+1}$ ersieht.

- Der Gesamtaufwand ist *kleiner* als eine Konstante multipliziert die Anzahl G_n der Knoten im Baum. Diese erfüllt aber:

$$\begin{aligned} G_n &= A_n + A_{n-2} + \ldots + A_0 \quad \text{(oder bis } A_1\text{)} \\ &\approx \lambda_1^{n+1}\left(1 + \frac{1}{\lambda_1^2} + \ldots\right) \\ &\leq C(\lambda_1)\lambda_1^n \end{aligned}$$

Bemerkung:

- Der Rechenaufwand wächst somit exponentiell.
- Der Speicherbedarf wächst hingegen nur linear in n.

Programm: **(Fibonacci iterativ)**

```
#include <iostream>
#include "cond.h"

int fibIter (int letzte, int vorletzte, int zaehler)
{
    return cond( zaehler==0,
                 vorletzte,
                 fibIter(vorletzte+letzte, letzte, zaehler-1) );
}

int fib (int n)
{
    return fibIter(1,0,n);
}

int main ()
{
    std::cout << fib(40) << std::endl;
}
```

Hier liefert das Substitutionsmodell:

```
  fib(2)
= fibIter(1,0,2)
= cond( 2==0, 0, fibiter(1,1,1))
= fibiter(1,1,1)
= cond( 1==0, 1, fibiter(2,1,0))
= fibIter(2,1,0)
= cond( 0==0, 2, fibiter(3,2,-1))
= 2
```

Bemerkung:

- Man braucht hier offenbar drei Zustandsvariablen.
- Der Rechenaufwand des linear iterativen Prozesses ist proportional zu n, also viel schneller als der baumrekursive.

5.4 Größenordnung

Es gibt eine formale Ausdrucksweise für Komplexitätsaussagen wie „der Aufwand zur Berechnung von `fib(n)` wächst exponentiell“.
Sei n ein Parameter der Berechnung, z. B.

- Anzahl gültiger Stellen bei der Berechnung der Quadratwurzel
- Dimension der Matrix in einem Programm für lineare Algebra
- Größe der Eingabe in Bits

Mit $R(n)$ bezeichnen wir den Bedarf an Resourcen für die Berechnung, z. B.

- Rechenzeit
- Anzahl auszuführender Operationen
- Speicherbedarf

Definition:

- $R(n) = \Omega(f(n))$, falls es von n unabhängige Konstanten c_1, n_1 gibt mit
$$R(n) \geq c_1 f(n) \quad \forall n \geq n_1.$$
- $R(n) = O(f(n))$, falls es von n unabhängige Konstanten c_2, n_2 gibt mit
$$R(n) \leq c_2 f(n) \quad \forall n \geq n_2.$$
- $R(n) = \Theta(f(n))$, falls $R(n) = \Omega(f(n)) \wedge R(n) = O(f(n))$.

Beispiel: $R(n)$ bezeichne den Rechenaufwand der rekursiven Fibonacci-Berechnung:

$$R(n) = \Omega(n)\,, \quad R(n) = O(2^n)\,, \quad R(n) = \Theta(\lambda_1^n)$$

Bezeichnung:

$R(n) = \Theta(1)$	konstante Komplexität
$R(n) = \Theta(\log n)$	logarithmische Komplexität
$R(n) = \Theta(n)$	lineare Komplexität
$R(n) = \Theta(n \log n)$	fast optimale Komplexität
$R(n) = \Theta(n^2)$	quadratische Komplexität
$R(n) = \Theta(n^p)$	polynomiale Komplexität
$R(n) = \Theta(a^n)$	exponentielle Komplexität

5.5 Wechselgeld

Aufgabe: Ein gegebener Geldbetrag ist unter Verwendung von Münzen zu 1, 2, 5, 10, 20 und 50 Cent zu wechseln. Wieviele verschiedene Möglichkeiten gibt es dazu?

Idee: Es sei der Betrag a mit n verschiedenen Münzarten zu wechseln. Eine der n Münzarten habe den Nennwert d. Dann gilt:

- Entweder wir verwenden eine Münze mit Wert d, dann bleibt der Rest $a - d$ mit n Münzarten zu wechseln.
- Wir verwenden die Münze mit Wert d *nicht*, dann müssen wir den Betrag a mit den verbleibenden $n - 1$ Münzarten wechseln.

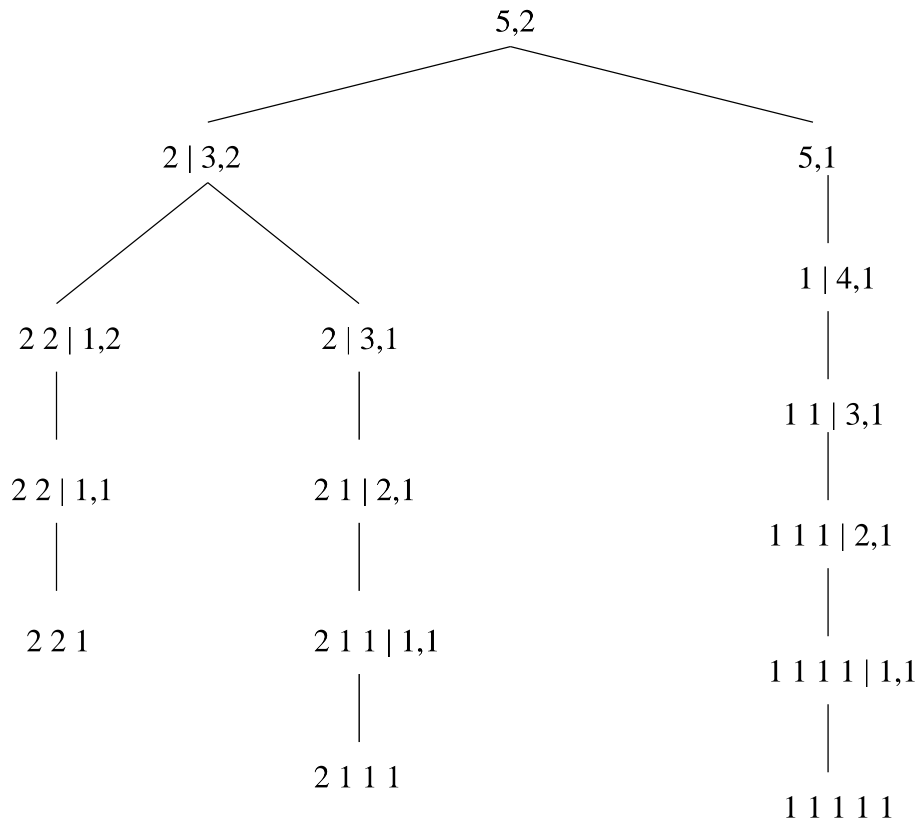
Folgerung: Ist $A(a, n)$ die Anzahl der Möglichkeiten den Betrag a mit n Münzarten zu wechseln, und hat Münzart n den Wert d, so gilt

$$A(a, n) = A(a - d, n) + A(a, n - 1)$$

Bemerkung: Es gilt auch:

- $A(0, n) = 1$ für alle $n \geq 0$.
- $A(a, n) = 0$ falls $a > 0$ and $n = 0$.
- $A(a, n) = 0$ falls $a < 0$.

Beispiel: Wechseln von 5 Cent in 1 und 2 Centstücke:



Bemerkung: Dies ist wieder ein baumrekursiver Prozess.

Programm: **(Wechselgeld zählen)**

```
#include <iostream>
#include "cond.h"

int nennwert (int nr)
{
    // uebersetze Muenzart in Muenzwert
    return cond(nr==1, 1,
                cond(nr==2, 2,
                     cond(nr==3, 5,
                          cond(nr==4, 10,
                               cond(nr==5, 20,
                                    cond(nr==6, 50, 0))))));
}

int wg (int betrag, int muenzarten)
{
    return cond(betrag==0, 1,
                cond(betrag<0 || muenzarten==0, 0,
                     wg(betrag,muenzarten-1) +
                     wg(betrag-nennwert(muenzarten),muenzarten)));
```

```
}

int wechselgeld (int betrag) {
    return wg(betrag,6);
}

int main () {
    std::cout << wechselgeld(500) << std::endl;
}
```

Hier einige Resultate:

```
wechselgeld(50)  = 451
wechselgeld(100) = 4562
wechselgeld(200) = 69118
wechselgeld(300) = 393119
```

Bemerkung: Ein iterativer Lösungsweg ist hier nicht ganz so einfach.

5.6 Der größte gemeinsame Teiler

Definition: Als den *größten gemeinsamen Teiler* (ggT) zweier Zahlen $a, b \in \mathbb{N}_0$ bezeichnen wir die größte natürliche Zahl, die sowohl a als auch b ohne Rest teilt.

Bemerkung: Den ggT braucht man etwa um rationale Zahlen zu kürzen:

$$\frac{91}{287} = \frac{13}{41}, \ \text{ggT}(91, 287) = 7.$$

Idee: Zerlege beide Zahlen in Primfaktoren, der ggT ist dann das Produkt aller gemeinsamer Faktoren. Leider: sehr aufwendig.

Effizienter: Euklidscher Algorithmus. Dieser basiert auf folgenden Überlegungen:

Bezeichnung: Seien $a, b \in \mathbb{N}$. Dann gilt $a = q \cdot b + r$ mit $q \in \mathbb{N}_0$ und $0 \leq r < b$. Wir schreiben $a \bmod b$ für den Rest r. Wenn $r = 0$, so schreiben wir $b|a$.

Bemerkung:

1. Falls $b = 0$ und $a > 0$, so ist $\text{ggT}(a, b) = a$.
2. Aus $\frac{a}{s} = q\frac{b}{s} + \frac{r}{s} \in \mathbb{N}$ ersieht man $\text{ggT}(a, b) = \text{ggT}(b, r)$.

Somit haben wir folgende Rekursion bewiesen:

$$\text{ggT}(a, b) = \begin{cases} a & \text{falls } b = 0 \\ \text{ggT}(b, a \bmod b) & \text{sonst} \end{cases}$$

Programm: **(Größter gemeinsamer Teiler)**

```cpp
#include <iostream>
#include "cond.h"
using namespace std;

int ggT (int a, int b)
{
        return cond( b==0 , a , ggT(b,a%b) );
}


int main ()
{
    cout << ggT(287,91) << endl;
}
```

Hier die Berechnung von `ggt(91,287)`

```
  ggT(91,287)      # 91=0*287+91
= ggT(287,91)      # 287=3*91+14
= ggT(91,14)       # 91=6*14+7
= ggT(14,7)        # 14=2*7+0
= ggT(7,0)
= 7
```

Bemerkung:

- Im ersten Schritt ist $91 = 0 \cdot 287 + 91$, also werden die Argumente gerade vertauscht.
- Der Berechnungsprozess ist iterativ, da nur ein fester Satz von Zuständen mitgeführt werden muss.

Satz: Der Aufwand von `ggT(a,b)` ist $O(\log n)$, wobei $n = \min(a, b)$.

Beweisskizze: Nach zwei Schritten des EA gilt auf jeden Fall $a_2 \leq n$. Falls nun $b_2 \leq a_2/2$, so gilt $a_3 = b_2 \leq a_2/2$, ansonsten gilt $b_3 = a_2 - b_2 < a_2/2$ und somit $a_4 < a_2/2$, ebenso $a_6 < a_4/2$, usw. Insgesamt erhalten wir daher

$$\text{EA-Schritte} \leq 2 + 2\log_2 a_2 \leq 2 + 2\log_2 n = O(\log n)$$

5.7 Zahlendarstellung im Rechner

In der Mathematik gibt es verschiedene Zahlenmengen:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}.$$

Diese Mengen enthalten alle unendlich viele Elemente, im Computer entsprechen die diversen Datentypen jedoch nur endlichen Mengen.
Um Zahlen aus $\mathbb{N}$ darzustellen, benutzt man ein Stellenwertsystem zu einer Basis β und Ziffern $a_i \in \{0, \dots, \beta - 1\}$
Dann bedeutet

$$(a_{n-1}a_{n-2}\dots a_1 a_0)_\beta \equiv \sum_{i=0}^{n-1} a_i \beta^i$$

Dabei ist n die Wortlänge. Es sind somit die folgenden Zahlen aus $\mathbb{N}$ darstellbar:

$$0, 1, \dots, \beta^n - 1$$

Am häufigsten wird $\beta = 2$, das *Binärsystem*, verwendet.
Zur Darstellung vorzeichenbehafteter Zahlen gibt es verschiedene Möglichkeiten.

1. Zusätzliches Bit für das Vorzeichen.
2. Zweierkomplement ($\beta = 2$)

Beispiel: (Zweierkomplement) Für $n = 3$ setze

0	=	000	-1	=	111
1	=	001	-2	=	110
2	=	010	-3	=	101
3	=	011	-4	=	100

Solange der Zahlenbereich nicht verlassen wird, klappt die normale Arithmetik ohne Beachtung des Vorzeichens:

3	$\rightarrow$	011
-1	$\rightarrow$	111
2	$\rightarrow$	[1]010

5.7.1 Gebräuchliche Zahlenbereiche in C++

$\beta = 2$ und $n = 8, 16, 32$:

`char`	-128...127
`unsigned char`	0...255
`short`	-32768...32767
`unsigned short`	0...65535
`int`	-2147483648...2147483647
`unsigned int`	0...4294967295

5.8 Darstellung reeller Zahlen

Neben den Zahlen aus $\mathbb{N}$ und $\mathbb{Z}$ sind in vielen Anwendungen auch reelle Zahlen $\mathbb{R}$ von Interesse. Wie werden diese im Computer realisiert?

5.9 Festkommazahlen

Eine erste Idee ist die Festkommazahl. Hier interpretiert man eine gewisse Zahl von Stellen als *nach dem Komma*, d. h.

$$(a_{n-1}a_{n-2}\ldots a_q.a_{q-1}\ldots a_0)_\beta \equiv \sum_{i=0}^{n-1} a_i \beta^{i-q}$$

Beispiel: Bei $\beta = 2, q = 3$ hat man drei Nachkommastellen, kann also in Schritten von $1/8$ auflösen.

Bemerkung:

- Jede Festkommazahl ist rational, somit können irrationale Zahlen nicht exakt dargestellt werden.
- Selbst einfache rationale Zahlen können je nach Basis nicht exakt dargestellt werden. So kann $0.1 = 1/10$ mit einer Festkommazahl zur Basis $\beta = 2$ für kein n exakt dargestellt werden.
- Das Ergebnis elementarer Rechenoperationen $+, -, *, /$ muss nicht mehr darstellbar sein.
- Festkommazahlen werden nur in Spezialfällen verwendet, etwa um mit Geldbeträgen zu rechnen. In vielen anderen Fällen ist die im nächsten Abschnitt dargestellte Fließkommaarithmetik brauchbarer.

5.10 Fließkommaarithmetik

Vor allem in den Naturwissenschaften wird die Fließkommaarithmetik (Gleitkommaarithmetik) angewendet. Eine Zahl wird dabei repräsentiert als

$$\pm \left(a_0 + a_1\beta^{-1} + \ldots + a_{n-1}\beta^{-(n-1)}\right) \times \beta^e$$

Die Ziffern a_i bilden die Mantisse und e ist der Exponent (eine ganze Zahl gegebener Länge). Wieder wird $\beta = 2$ am häufigsten verwendet. Das Vorzeichen ist ein zusätzliches Bit.

5.10.1 Typische Wortlängen

`float`: 23 Bit Mantisse, 8 Bit Exponent, 1 Bit Vorzeichen entsprechen

$$23 \cdot \log_{10} 2 = 23 \cdot \frac{\log 2}{\log 10} \approx 23 \cdot 0.3 \approx 7$$

dezimalen Nachkommastellen in der Mantisse.
`double`: 52 Bit Mantisse, 11 Bit Exponent, 1 Bit Vorzeichen entsprechen $52 \cdot 0.3 \approx 16$ dezimalen Nachkommastellen in der Mantisse.

Referenz: Genaueres findet man im IEEE-Standard 754 (floating point numbers).

5.10.2 Fehler in der Fließkommaarithmetik

Darstellungsfehler $\beta = 10$, $n = 3$: Die reelle Zahl 3.14159 wird auf 3.14×10^0 gerundet. Der Fehler beträgt maximal 0.005, man sagt $0.5ulp$, ulp heißt *units last place*.

Bemerkung:

- Wenn solche fehlerbehafteten Daten als Anfangswerte für Berechnungen verwendet werden, können die Anfangsfehler erheblich vergrößert werden.
- Durch Rundung können weitere Fehler hinzukommen.
- Vor allem bei der Subtraktion kann es zum Problem der *Auslöschung* kommen, wenn beinahe gleichgroße Zahlen voneinander abgezogen werden.

Beispiel: Berechne $b^2 - 4ac$ in $\beta = 10$, $n = 3$ für $b = 3.34, a = 1.22, c = 2.28$. Eine exakte Rechnung liefert

$$3.34 \cdot 3.34 - 4 \cdot 1.22 \cdot 2.28 = 11.1556 - 11.1264 = 0.0292$$

Mit Rundung der Zwischenergebnisse ergibt sich dagegen

$$\ldots 11.2 - 11.1 = 0.1$$

Der absolute Fehler ist somit $0.1 - 0.0292 = 0.0708$. Damit ist der relative Fehler $\frac{0.0708}{0.0292} = 240\%$! Nicht einmal eine Stelle des Ergebnisses $1.00 \cdot 10^{-1}$ ist korrekt!

5.11 Typkonversion

Im Ausdruck `5/3` ist „`/`“ die ganzzahlige Division ohne Rest, in `5.0/3.0` wird eine Fließkommadivision durchgeführt.
Will man eine gewisse Operation erzwingen, kann man eine explizite Typkonversion einbauen:

`((double) x)/3`	Fließkommadivision
`((int) y)/((int) 3)`	Ganzzahldivision

5.12 Wurzelberechnung mit dem Newtonverfahren

Problem: $f : \mathbb{R} \to \mathbb{R}$ sei eine „glatte" Funktion, $a \in \mathbb{R}$. Wir wollen die Gleichung

$$f(x) = a$$

lösen.

Beispiel: $f(x) = x^2 \rightsquigarrow$ Berechnung von Quadratwurzeln.
Mathematik: $\sqrt{a}$ ist die positive Lösung von $x^2 = a$.
Informatik: Will Algorithmus zur Berechnung des Zahlenwerts von $\sqrt{a}$.

Ziel: Konstruiere ein Iterationsverfahren mit folgender Eigenschaft: zu einem Startwert $x_0 \approx x$ finde $x_1, x_2, \ldots$, welche die Lösung x immer besser approximieren.

Definition: (Taylorreihe)

$$f(x_n + h) = f(x_n) + hf'(x_n) + \frac{h^2}{2} f''(x_n) + \ldots$$

Wir vernachlässigen nun den $O(h^2)$-Term ($|f''(x)| \leq C$, kleines h) und verlangen $f(x_n+h) \stackrel{!}{=} a$. Dies führt zu

$$h = \frac{a - f(x_n)}{f'(x_n)}$$

und somit zur Iterationsvorschrift

$$x_{n+1} = x_n + \frac{a - f(x_n)}{f'(x_n)}.$$

Beispiel: Für die Quadratwurzel erhalten wir mit $f(x) = x^2$ und $f'(x) = 2x$ die Vorschrift

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Abbruchkriterium: $|f(x_n) - a| < \varepsilon$ für eine vorgegebene (kleine) Zahl ε.

Programm: **(Quadratwurzelberechnung)**

```
#include <iostream>
#include "cond.h"
using namespace std;

bool gut_genug (double xn, double a) {
    return abs(xn*xn-a)<=1E-15;
}

double wurzelIter (double xn, double a) {
    return cond( gut_genug(xn,a),
```

```
                    xn,
                    wurzelIter(0.5*(xn+a/xn),a) );
}

double wurzel (double a)
{
    return wurzelIter(1,a);
}

int main ()
{
    cout.precision(20);
    cout << wurzel(2) << endl;
}
```

Hier ist die Auswertung der Wurzelfunktion im Substitutionsmodell (nur die Aufrufe von `wurzelIter` sind dargestellt):

```
  wurzel(2)
= wurzelIter(1,2)
= wurzelIter(1.5,2)
= wurzelIter(1.4166666666666667407,2)
= wurzelIter(1.4142156862745098866,2)
= wurzelIter(1.4142135623746898698,2)
= wurzelIter(1.4142135623730951455,2)
= 1.4142135623730951455
```

Bemerkung:

- `cout.precision(20)` bewirkt, dass 20 Stellen nach dem Komma bei Fließkommazahlen ausgegeben werden.
- Dieses Programm entspricht nicht genau unseren Regeln. Es enthält schon eine „Sequenz von Anweisungen“ in der Funktion `gut_genug` sowie in `main`.
- Unter gewissen Voraussetzungen an f kann man zeigen, dass sich die Zahl der gültigen Ziffern mit jedem Schritt verdoppelt.

6 Fortgeschrittene funktionale Programmierung

6.1 Funktionen in der Mathematik

Definition: Eine Funktion $f : X \to Y$ ordnet jedem Element einer Menge X genau ein Element der Menge Y zu.

In der Mathematik ist es nun durchaus üblich, nicht nur einfache Beispiele wie etwa $f : X \to Y$ mit $X = Y = \mathbb{R}$ zu betrachten. Im Gegenteil: in wichtigen Fällen sind X und/oder Y Funktionenräume.

Beispiele:

- Ableitung: Funktionen $\to$ Funktionen
- Stammfunktion: Funktionen $\to$ Funktionen
- Mittelwert: Funktionen $\to$ Zahlen

6.2 Funktionale Programmiersprachen

Funktionale Programmiersprachen wie Scheme, ML oder Haskell haben ein wesentliches Merkmal, welches FC++ (oder auch C++) nicht bietet, nämlich die Behandlung von Funktionen als Objekte erster Klasse. Das heißt, dass Funktionen (so wie Zahlen in FC++) *lokal erzeugt* werden können und als *Argumente* und *Rückgabewerte* von Funktionen auftreten können.

Beispiel: In Scheme erhält man eine (anonyme) Funktion durch

```
(lambda <parameter-liste> <Funktionsrumpf>)
```

So etwa:

```
(lambda (x) 2)          ; konstante Funktion
(lambda (x) x)          ; Identität
(lambda (x) (* x x))    ; Quadratfunktion
```

Anwendung: In Scheme kann man schreiben:

```
(define (inkrementierer n)
   (lambda (x) (+ x n)))
(map (inkrementierer 5) '(1 2 3))  => (6 7 8)
```

Die Übertragung der `inkrementierer`-Definition in C++ wäre etwas wie:

```
int inkrementierer (int n)
{
    int lokale_funktion (int k)
    {
        return k+n;
    }
    return lokale_funktion;
}
```

Leider ist dieses Konstrukt nicht erlaubt.

Bemerkung:

- Das Schlüsselwort `lambda` deutet auf den von Alonzo Church in den 1930ern entwickelten *Lambda-Kalkül* hin, in dem etwa die Identität als $(\lambda x.x)$ geschrieben wird. Der Lambda-Kalkül ist eine äußerst kleine, mathematisch sehr elegante Programmiersprache, von der man Turing-Äquivalenz zeigen kann. Scheme, Haskell, etc basieren auf diesem Kalkül.

- Die Behandlung von Funktionen als Objekte erster Klasse impliziert eine aufwendigere Art der Speicherverwaltung, die mit einem gewissen Effizienzverlust einhergeht. Aus diesem Grund wird diese Technik zum Beispiel in C++ nicht erlaubt.

- Auch FC++ wäre Turing-äquivalent, wenn man beliebig lange `int`-Zahlen hätte. Der Lambda-Kalkül kommt sogar ohne Zahlen aus.

6.3 Warum funktionale Programmierung?

- Mathematisch am besten verstanden

- $\Rightarrow$ relativ oft sind Korrektheitsbeweise von funktionalen Programmen möglich

- Wenn man zusätzlich Syntaxtransformationen erlaubt, lassen sich viele bekannte Merkmale von Programmiersprachen (z.B. lokale Variablen, Schleifen, Objekte) sehr einfach erhalten

Aber: Funktionales Programmieren ist nicht für alle Situationen die beste Wahl! Zum Beispiel legt die Interaktion mit der Außenwelt oder ihre *effiziente* Nachbildung oft andere Paradigmen nahe. Funktionale Sprachen haben deshalb auch oft nicht-funktionale Sprachelemente.

7 Prozedurale Programmierung

7.1 Lokale Variablen und die Zuweisung

Erinnerung: Bis jetzt haben wir Namen nur als Funktionssymbole und im Zusammenhang mit formalen Parametern einer Funktion kennengelernt.
Innerhalb des Funktionsrumpfes steht der Name des formalen Parameters für einen Wert (der zum Zeitpunkt der Funktionsdefinition unbekannt ist).

7.1.1 Konstanten

In C++ kann man konstante Werte wie folgt mit Namen versehen:

```
float umfang (float r)
{
    const double pi = 3.14159265;
    return 2*r*pi;
}

int hochacht (int x)
{
    const int x2 = x*x;        // jetzt gibt es ein x2
    const int x4 = x2*x2;      // nun ein x4
    return x4*x4;
}
```

Bemerkung:

- Einer solchen Konstanten kann nur *einmal* ein Wert zugeordnet werden.
- Die Auswertung solcher Funktionsrümpfe erfordert eine Erweiterung des Substitutionsmodells:
    - Ersetze formale Parameter durch aktuelle Parameter.
    - Erzeuge (der Reihe nach) die durch die Zuweisungen gegebenen Name-Wert Zuordnungen und ersetze neue Namen im Rest des Rumpfes durch den Wert.
- Diese Einführung neuer Namen ist in funktionalen Sprachen sehr einfach durch lokale Funktionen zu erreichen. Wir haben den Bereich funktionaler Programmierung damit eigentlich noch nicht verlassen.

7.1.2 Variablen

C++ erlaubt aber auch, die Zuordnung von Werten zu Namen zu ändern:

Beispiel: (Variablen)

```
int hochacht (int x)
{
    int y = x*x;     // Zeile 1: Definition/Initialisierung
    y = y*y;         // Zeile 2: Zuweisung
    return y*y;
}
```

Bemerkung:

- Zeile 1 definiert eine Variable `y`, die Werte vom Typ `int` annehmen kann.
- Zeile 2 nennt man eine Zuweisung. Die links des = stehende Variable erhält den Wert des rechts stehenden Ausdrucks als neuen Wert.
- Der Typ einer Variablen kann aber nicht geändert werden!

7.1.3 Problematik der Zuweisung

Beispiel:

```
int bla (int x)
{
    int y = 3;           // Zeile 1
    const int x1 = y*x;  // Zeile 2
    y = 5;               // Zeile 3
    const int x2 = y*x;  // Zeile 4
    return x1*x2;        // Zeile 5
}
```

Bemerkung:

- Obwohl `x1` und `x2` durch denselben Ausdruck `y*x` definiert werden, haben sie im allgemeinen verschiedene Werte.
- Dies bedeutet das Versagen des Substitutionsmodell, bei dem ein Name im ganzen Funktionsrumpf durch seinen Wert ersetzt werden kann.
- Die Namensdefinitionen und Zuweisungen werden *der Reihe nach* abgearbeitet. Das Ergebnis hängt auch von dieser Reihenfolge ab. Dagegen war die Reihenfolge der Auswertung von Ausdrücken im Substitutionsmodell egal.

7.1.4 Umgebungsmodell

Wir können uns die Belegung der Variablen als Abbildung bzw. Tabelle vorstellen, die jedem Namen einen Wert zuordnet:

$$w : \{ \text{ Menge der gültigen Namen } \} \rightarrow \{ \text{ Menge der Werte } \}.$$

Beispiel: Abbildung w bei Aufruf von `bla(4)` nach Zeile 4

Name	Typ	Wert
`x`	`int`	4
`y`	`int`	5
`x1`	`int`	20
`x2`	`int`	15

Definition: Der Ort, an dem diese Abbildung im System gespeichert wird, heißt Umgebung. Die Abbildung w heißt auch Bindungstabelle. Man sagt, w bindet einen Namen an einen Wert.

Bemerkung:

- Ein Ausdruck wird in Zukunft immer relativ zu einer Umgebung ausgewertet, d.h. nur Ausdruck und Umgebung zusammen erlauben die Berechnung des Wertes eines Ausdruckes.
- Die Zuweisung können wir nun als Modifikation der Bindungstabelle verstehen: nach der Ausführung von `y=5` gilt $w(\mathrm{y}) = 5$.

7.2 Syntax von Variablendefinition und Zuweisung

Syntax:

<Def> ::= <ConstDef> | <VarDef>
<ConstDef> ::= const <Typ> <Name> = <Ausdruck>
<VarDef> ::= <Typ> <Name> [= <Ausdruck>]

Syntax:

<Zuweisung> ::= <Name> = <Ausdruck>

Bemerkung:

- Wir erlauben zunächst Variablendefinitionen nur innerhalb von Funktionsdefinitionen. Diese Variablen bezeichnet man als lokale Variablen.
- Bei der Definition von Variablen *kann* die Initialisierung weggelassen werden. In diesem Fall ist der Wert der Variablen bis zur ersten Zuweisung unbestimmt.
  Aber: Fast immer ist es empfehlenswert, auch Variablen gleich bei der Definition zu initialisieren!

7.2.1 Lokale Umgebung

Wie sieht die Umgebung im Kontext mehrerer Funktionen aus?

Programm:

```
int g (int x)
{
    int y = x*x;
    y = y*y;
    return h(y*(x+y));
}

int h (int x)
{
    return cond(x<1000, g(x), 88);
}
```

Es gilt folgendes:

- Jede Auswertung einer Funktion erzeugt eine eigene lokale Umgebung. Mit Beendigung der Funktion wird diese Umgebung wieder vernichtet!
- Zu jedem Zeitpunkt der Berechnung gibt es eine aktuelle Umgebung. Diese enthält die Bindungen der Variablen der Funktion, die gerade ausgewertet wird.
- In Funktion `h` gibt es keine Bindung für `y`, auch wenn `h` von `g` aufgerufen wurde.
- Wird eine Funktion n mal rekursiv aufgerufen, so existieren n verschiedene Umgebungen für diese Funktion.

Bemerkung: Man beachte auch, dass eine Funktion kein Gedächtnis hat: wird sie mehrmals mit gleichen Argumenten aufgerufen, so sind auch die Ergebnisse gleich. Diese fundamentale Eigenschaft funktionaler Programmierung ist also (bisher) noch erhalten.

Bemerkung: Tatsächlich wäre obiges Konstrukt auch nach Einführung einer `main`-Funktion nicht kompilierbar, weil die Funktion `h` beim Übersetzen von `g` noch nicht bekannt ist. Um dieses Problem zu umgehen, erlaubt C++ die vorherige Deklaration von Funktionen. In obigem Beispiel könnte dies geschehen durch Einfügen der Zeile

```
int h (int x);
```

vor die Funktion g.

7.3 Anweisungsfolgen (Sequenz)

- Funktionale Programmierung bestand in der Auswertung von Ausdrücken.
- Jede Funktion hatte nur eine einzige Anweisung (`return`).
- Mit Einführung von Zuweisung (oder allgemeiner Nebeneffekten) macht es Sinn, die *Ausführung mehrer Anweisungen* innerhalb von Funktionen zu erlauben. Diesen Programmierstil nennt man auch *imperative Programmierung*.

Erinnerung: Wir kennen schon eine Reihe wichtiger Anweisungen:

- Variablendefinition (ist in C++ eine Anweisung, nicht aber in C),
- Zuweisung,
- Ausgabeanweisung `cout << ...`,
- return-Anweisung in Funktionen.

Bemerkung:

- Jede Anweisung endet mit einem Semikolon.
- Überall wo eine Anweisung stehen darf, kann auch eine durch geschweifte Klammern eingerahmte Folge (*Sequenz*) von Anweisungen stehen.
- Anweisungen werden der Reihe nach abgearbeitet.

Syntax:(Anweisung)

<Anweisung>	::=	<EinfacheAnw> | { { <EinfacheAnw> }$^+$ }
<EinfacheAnw>	::=	<VarDef> ; | <Zuweisung> ; |
		<Selektion> | ...

7.3.1 Beispiel

Die folgende Funktion berechnet `fib(4)`. `b` enthält die letzte und `a` die vorletzte Fibonaccizahl.

```
int f4 ()
{
    int a=0;     // a=fib(0)
    int b=1;     // b=fib(1)
    int t;

    t = a+b; a = b; b = t;     // b=fib(2)
    t = a+b; a = b; b = t;     // b=fib(3)
    t = a+b; a = b; b = t;     // b=fib(4)
    return b;
}
```

Bemerkung: Die Variable `t` wird benötigt, da die beiden Zuweisungen

$$\left\{\begin{array}{lcl} \texttt{b} & \leftarrow & \texttt{a+b} \\ \texttt{a} & \leftarrow & \texttt{b} \end{array}\right\}$$

nicht gleichzeitig durchgeführt werden können.

Bemerkung: Man beachte, dass die Reihenfolge in

```
t = a+b;
a = b;
b = t;
```

nicht vertauscht werden darf. In der funktionalen Programmierung mussten wir hingegen weder auf die Reihenfolge achten noch irgendwelche „Hilfsvariablen“ einführen.

7.4 Bedingte Anweisung (Selektion)

Anstelle des `cond`-Operators wird in der imperativen Programmierung die bedingte Anweisung verwendet.

Syntax: **(Bedingte Anweisung, Selektion)**

< Selektion > ::= if (< BoolAusdr >) < Anweisung >
[else < Anweisung >]

Ist die Bedingung in runden Klammern wahr, so wird die erste Anweisung ausgeführt, ansonsten die zweite Anweisung nach dem `else` (falls vorhanden).

Beispiel: Die funktionale Form

```
int absolut (int x)
{
    return cond( x<=0, -x , x);
}
```

ist äquivalent zu

```
int absolut (int x)
{
    if (x<=0)
        return -x;
    else
        return x;
}
```

7.5 Schleifen

Iterative Prozesse kommen so häufig vor, dass man hierfür eine Abstraktion schaffen muss. In C++ gibt es dafür verschiedene imperative Konstrukte, die wir jetzt kennenlernen.

7.5.1 While-Schleife

Programm: **(Fakultät mit While-Schleife)**

```
int fak (int n)
{
    int ergebnis=1;
    int zaehler=2;

    while (zaehler<=n)
    {
        ergebnis = zaehler*ergebnis;
        zaehler  = zaehler+1;
    }
    return ergebnis;
}
```

Bemerkung: Warum funktioniert dieses Programm? Man überlegt sich, dass *vor* und *nach* jedem Durchlauf der `while`-Schleife folgende Bedingung gilt:

$$(2 \leq \texttt{zaehler} \leq \texttt{n} + 1) \wedge (\texttt{ergebnis} = (\texttt{zaehler} - 1)!)$$

Diese Bedingung nennt man Schleifeninvariante. Ist die `return`-Anweisung erreicht, so muss zusätzlich $\texttt{zaehler} = \texttt{n} + 1$ gelten, und daher ist $\texttt{ergebnis} = \texttt{n}!$.

Bemerkung: Man beachte das Zusammenspiel von

- Initialisierung von `ergebnis` und `zaehler`,
- *Ausführungsbedingung* der `while`-Schleife,
- Reihenfolge der Anweisungen im Schleifenkörper.

Syntax: **(While-Schleife)**

<WhileSchleife> ::= <u>while</u> <u>(</u> <BoolAusdr> <u>)</u> <Anweisung>

Die Anweisung wird solange ausgeführt wie die Bedingung erfüllt ist.

7.5.2 For-Schleife

Die obige Anwendung der `while`-Schleife ist ein Spezialfall, der so häufig vorkommt, dass es dafür eine Abkürzung gibt:

Syntax: (For-Schleife)

<ForSchleife> ::= <u>for</u> <u>(</u> <Init> <u>;</u> <BoolAusdr> <u>;</u> <Increment><u>)</u>
<Anweisung>

<Init> ::= <VarDef> | <Zuweisung>

<Increment> ::= <Zuweisung>

Init entspricht der Initialisierung des Zählers, BoolAusdr der Ausführungsbedingung und Increment der Inkrementierung des Zählers.

Programm: **(Fakultät mit For-Schleife)**

```
int fak (int n)
{
    int ergebnis=1;

    for (int zaehler=2; zaehler<=n; zaehler = zaehler+1)
        ergebnis = zaehler*ergebnis;

    return ergebnis;
}
```

Bemerkung:

- Der *Gültigkeitsbereich* von `zaehler` erstreckt sich nur über die `for`-Schleife (ansonsten hätte man es wie `ergebnis` außerhalb der Schleife definieren müssen).
- Die Initialisierungsanweisung enthält Variablendefinition und Initialisierung.
- Wie beim Fakultätsprogramm mit `while` wird die Inkrementanweisung am Ende des Schleifendurchlaufes ausgeführt.

7.5.3 Beispiele

Wir benutzen nun die neuen Konstruktionselemente um die iterativen Prozesse zur Berechnung der Fibonaccizahlen und der Wurzelberechnung nochmal zu formulieren.

Programm: **(Fibonacci mit For-Schleife)**

```
int fib (int n)
{
    int a=0;
    int b=1;
    for (int i=0; i<n; i=i+1)
    {
        int t=a+b; a = b; b = t;
    }
    return a;
}
```

Bemerkung: Hier lautet die Schleifeninvariante $(0 \leq \mathtt{i} \leq \mathtt{n}) \wedge (\mathsf{a} = (\mathsf{Fib}(\mathtt{i}))$. Die Schleife wird n-mal durchlaufen, der Aufwand ist $O(n)$.

Programm: **(Newton mit While-Schleife)**

```
#include <iostream>
#include "cond.h"
using namespace std;

double wurzel (double a)
  {
       double x=1.0;

       while (abs(x*x-a)>1E-12)
           x = 0.5*(x+a/x);
       return x;
  }

int main ()
{
     cout.precision(20);
     cout << wurzel(2) << endl;
}
```

7.5.4 Schleifen in funktionalen Sprachen

Auch in funktionalen Sprachen gibt es Abkürzungen (sog. syntaktischen Zucker) für Schleifen. So könnte man die iterative Berechnung der Fibonacci-Zahlen in Scheme wie folgt formulieren:

Programm: (Fibonacci iterativ in Scheme)

```
(define (fib n)
  (do ((a 0 b)
       (b 1 (+ a b))
       (k 0 (+ k 1)))
     ((= k n) a)))
```

Bemerkung: Diese Schleife wird vor dem Ausführen in das Konstrukt mit (lokaler) Hilfsfunktion transformiert. Daher braucht man hier keine Hilfsvariable `t`.

8 Benutzerdefinierte Datentypen

Die bisherigen Programme haben nur mit Zahlen (unterschiedlichen Typs) gearbeitet. „Richtige“ Programme bearbeiten allgemeinere Daten, z.B.

- Zuteilung der Studenten auf Übungsgruppen,
- Flugreservierungssystem,
- Textverarbeitungsprogramm, Zeichenprogramm, ...

Bemerkung: Im Sinne der Berechenbarkeit ist das keine Einschränkung, denn auf beliebig großen Bändern (Turing-Maschine), in beliebig tief verschachtelten Funktionen (Lambda-Kalkül) oder in beliebig großen Zahlen (FC++ mit langen Zahlen) lassen sich beliebige Daten kodieren.
Da dies aber sehr umständlich und ineffizient ist, erlauben praktisch alle Programmiersprachen dem Programmierer die Definition neuer Datentypen.

8.1 Aufzählungstyp

Dies ist ein Datentyp, der aus endlich vielen Werten besteht. Jedem Wert ist ein Name zugeordnet.

Beispiel:

```
enum color {white, black, red, green, blue, yellow};
...
color bgcolor = white;
color fgcolor = black;
```

Syntax: (Aufzählungstyp)

<Enum> ::= enum <Identifikator>
{ <Identifikator> [, <Identifikator>] } ;

Programm: (Vollständiges Beispiel)

```
#include <iostream>
using namespace std;

enum Zustand { neu, gebraucht, alt, kaputt };

int druckeZustand (Zustand x) {
    if (x==neu)        { cout << "neu";        return 1; }
    if (x==gebraucht)  { cout << "gebraucht";  return 1; }
    if (x==alt)        { cout << "alt";        return 1; }
    if (x==kaputt)     { cout << "kaputt";     return 1; }
    return 0; // unbekannter Zustand ?!
}

int main () {druckeZustand(alt);}
```

8.2 Felder

Wir lernen nun einen ersten Mechanismus kennen, um aus einem bestehenden Datentyp, wie `int` oder `float`, einen neuen Datentyp zu erschaffen: das Feld.

Definition: Ein Feld besteht aus einer *festen Anzahl* von Elementen eines Grundtyps. Die Elemente sind angeordnet, d. h. mit einer Numerierung versehen. Die Numerierung ist fortlaufend und beginnt bei 0.

Bemerkung: In der Mathematik entspricht dies dem (karthesischen) Produkt von Mengen.

Beispiel: Das mathematische Objekt eines Vektors $x = (x_0, x_1, x_2)^T \in \mathbb{R}^3$ wird in C++ durch

```
double x[3];
```

dargestellt. Auf die Komponenten greift man wie folgt zu:

```
x[0] = 1.0;  // Zugriff auf das erste Feldelement
x[1] = x[0]; // das zweite
x[2] = x[1]; // und das letzte
```

D.h. die Größen `x[k]` verhalten sich wie jede andere Variable vom Typ `double`.

Syntax: **(Felddefinition)**

$$\langle \text{FeldDef} \rangle \quad ::= \quad \langle \text{Typ} \rangle \; \langle \text{Name: } \rangle \; \underline{[} \; \langle \text{Anzahl} \rangle \; \underline{]}$$

Erzeugt ein Feld mit dem Namen <Name: >, das <Anzahl> Elemente des Typs <Typ> enthält.

Bemerkung: Eine Felddefinition darf wie eine Variablendefinition verwendet werden.

8.2.1 Sieb des Eratosthenes

Als Anwendung des Feldes betrachten wir eine Methode zur Erzeugung einer Liste von Primzahlen, die Sieb des Eratosthenes genannt wird.

Idee: Wir nehmen eine Liste der natürlichen Zahlen größer 1 und streichen alle Vielfachen von $2, 3, 4, \ldots$. Alle Zahlen, die durch diesen Prozess *nicht* erreicht werden, sind die gesuchten Primzahlen.

Bemerkung:

- Es genügt nur die Vielfachen der Primzahlen zu nehmen (Primfaktorzerlegung).
- Um nachzuweisen, dass $N \in \mathbb{N}$ prim ist, reicht es, $k \nmid N$ für alle Zahlen $k \in \mathbb{N}$ mit $k \leq \sqrt{N}$ zu testen.

Programm: **(Sieb des Eratosthenes)**

```cpp
#include <iostream>
using namespace std;

int main ()
{
    const int n = 50000;
    bool prim[n];

    // Initialisierung
    prim[0] = false;
    prim[1] = false;
    for (int i=2; i<n; i=i+1)
        prim[i] = true;

    // Sieb
    for (int i=2; i<sqrt((double) n); i=i+1)
        if (prim[i])
            for (int j=2*i; j<n; j=j+i)
                prim[j] = false;

    // Ausgabe
    for (int i=0; i<n; i=i+1)
        if (prim[i])
            cout << i << " ";

    return 0;
}
```

Bemerkung: Der Aufwand des Algorithmus lässt sich wie folgt abschätzen:

1. Der Aufwand der Initialisierung ist $\Theta(n)$.
2. Unter der Annahme einer „konstanten Primzahldichte“ erhalten wir

$$\text{Aufwand}(n) \leq C \sum_{k=2}^{\sqrt{n}} \frac{n}{k} = Cn \sum_{k=2}^{\sqrt{n}} \frac{1}{k} \leq Cn \int_1^{\sqrt{n}} \frac{dx}{x} = Cn \log \sqrt{n} = \frac{C}{2} n \log n$$

3. $O(n \log n)$ ist bereits eine fast optimale Abschätzung, da der Aufwand ja auch $\Omega(n)$ ist. Man kann die Ordnungsabschätzung daher nicht wesentlich verbessern, selbst wenn man zahlentheoretisches Wissen über die Primzahldichte hinzuziehen würde.

Bemerkung: Die Beziehung zur funktionalen Programmierung ist etwa folgende:

- Mit großen Feldern operierende Algorithmen kann man nur schlecht rein funktional darstellen.

- Dies ist vor allem eine Effizienzfrage, weil oft kleine Veränderungen großer Felder verlangt werden. Bei funktionaler Programmierung müsste man ein neues Feld erzeugen und als Rückgabewert verwenden.
- Algorithmen wie das Sieb des Eratosthenes formuliert man daher funktional auf andere Weise (Datenströme, Streams), was interessant ist, allerdings manchmal auch recht komplex wird.

8.3 Zeichen und Zeichenketten

8.3.1 Datentyp char

- Zur Verarbeitung von einzelnen Zeichen gibt es den Datentyp char, der genau ein Zeichen aufnehmen kann:

```
char c = '%';
```

- Die Initialisierung benutzt die einfachen Anführungsstriche.
- Der Datentyp char ist kompatibel mit int (Zeichen entsprechen Zahlen im Bereich $-128 \ldots 127$). Man kann daher sogar mit ihm rechnen:

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1+c2;
```

Normalerweise sollte man diese Eigenschaft aber nicht brauchen!

8.3.2 ASCII

Die den Zahlen $0 \ldots 127$ zugeordneten Zeichen nennt man den American Standard Code for Information Interchange oder kurz ASCII. Den druckbaren Zeichen entsprechen die Werte $32 \ldots 127$.

Programm: (ASCII)

```
#include <iostream>
using namespace std;

int main ()
{
    for (int i=32; i<=127; i=i+1)
        cout << i << " entspricht '" << (char) i << "'" << endl;
}
```

Bemerkung:

- Die Zeichen $0, \dots, 31$ dienen Steuerzwecken wie Zeilenende, Papiervorschub, Piepston, etc.
- Für die negativen Werte $-128, \dots, -1$ (entspricht $128, \dots, 255$ bei vorzeichenlosen Zahlen) gibt es verschiedene Belegungstabellen (ISO 8859-n), mit denen man Zeichensätze und Sonderzeichen anderer Sprachen abdeckt.
- Noch komplizierter wird die Situation, wenn man *Zeichensätze* für Sprachen mit sehr vielen Zeichen (Chinesisch, Japanisch, etc) benötigt, oder wenn man mehrere Sprachen gleichzeitig behandeln will.
  Stichwort: Unicode.

8.3.3 Zeichenketten

Zeichenketten realisiert man in C am einfachsten mittels einem `char`-Feld. Konstante Zeichenketten kann man mit doppelten Anführungsstrichen auch direkt im Programm eingeben.

Beispiel: Initialisierung eines `char`-Felds:

```
char c[10] = "Hallo";
```

Bemerkung: Das Feld muss groß genug sein, um die Zeichenkette samt einem Endezeichen (in C das Zeichen mit ASCII-Code 0) aufnehmen zu können. Diese feste Größe ist oft *sehr unhandlich*, und viele Sicherheitsprobleme entstehen aus der Verwendung von zu kurzen `char`-Feldern von unachtsamen C-Programmierern!

Programm: **(Zeichenketten im C-Stil)**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    char name[32] = "Peter Bastian";

    for (int i=0; name[i]!=0; i=i+1)    // Ausgabe der einzelnen
        cout << name[i];                // Zeichen
    cout << endl;                       // neue Zeile

    cout << name << endl;               // geht natürlich auch !
}
```

In C++ gibt es einen Datentyp `string`, der sich besser zur Verarbeitung von Zeichenketten eignet als bloße `char`-Felder:

Programm: **(Zeichenketten im C++-Stil)**

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string vorname = "Peter";
    string nachname = "Bastian";
    string name = vorname + " " + nachname;
    cout << name << endl;
}
```

8.4 Typedef

Mittels der typedef-Anweisung kann man einem bestehenden Datentyp einen neuen Namen geben.

Beispiel:

```
typedef int MyInteger;
```

Damit hat der Datentyp int auch den Namen MyInteger erhalten.

Bemerkung: MyInteger ist kein neuer Datentyp. Er darf synonym mit int verwendet werden:

```
MyInteger x=4;  // ein MyInteger
int y=3;        // ein int

x = y;          // Zuweisung OK, Typen identisch
```

Anwendung: Verschiedene Computerarchitekturen (Rechner/Compiler) verwenden unterschiedliche Größen etwa von int-Zahlen. Soll nun ein Programm portabel auf verschiedenen Architekturen laufen, so kann man es an kritischen Stellen mit MyInteger schreiben. MyInteger kann dann an einer Stelle *architekturabhängig* definiert werden.

Beispiel: Auch Feldtypen kann man einen neuen Namen geben:

```
typedef double Punkt3d[3];
```

Dann kann man bequem schreiben:

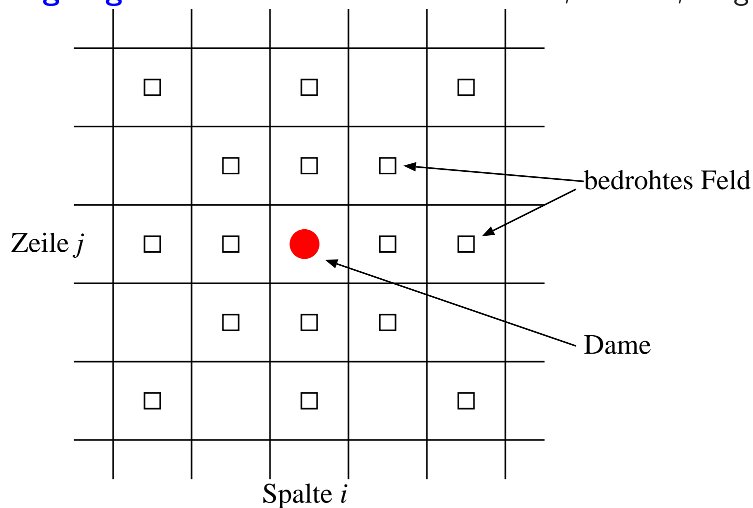
```
Punkt3d a,b;
a[0] = 0.0; a[1] = 1.0; a[2] = 2.0;
b[0] = 0.0; b[1] = 1.0; b[2] = 2.0;
```

Bemerkung: Ein Tipp zur Syntax: Man stelle sich eine Felddefinition vor und schreibt typedef davor.

8.5 Das Acht-Damen-Problem

Problem: Wie kann man acht Damen so auf einem Schachbrett positionieren, dass sie sich nicht gegenseitig schlagen können?

Zugmöglichkeiten der Dame: horizontal, vertikal, diagonal



Bemerkung:

- Ist daher die Dame an der Stelle (i, j), so bedroht sie alle (i', j') mit
  - $i = i'$ oder $j = j'$
  - $(i - i') = (j - j')$ oder $(i - i') = -(j - j')$
- Bei jeder Lösung steht in jeder Zeile/Spalte des Bretts genau eine Dame.

Idee:

- Man baut die Lösungen sukzessive auf, indem man erst in der ersten Zeile eine Dame platziert, dann in der zweiten, usw.
- Die Platzierung der ersten n Damen kann man durch ein `int`-Feld der Länge n beschreiben, wobei jede Komponente die Spaltenposition der Dame enthält.

Programm: (**Acht-Damen-Problem**)

```
#include <iostream>
using namespace std;

const int board_size = 8;          // globale Konstante
typedef int columns[board_size];   // neuer Datentyp "columns"

bool good_position (int new_row, columns queen_cols, int new_col)
{
    for (int row=0; row<new_row; row=row+1)
```

```
        if ((queen_cols[row] == new_col) ||
            (new_row-row == abs(queen_cols[row]-new_col)))
            return false;
    return true;
}

void display_board (columns queen_cols)
{
    cout << endl << endl;
    for (int i=0; i<board_size; i=i+1)
    {
        for (int j=0; j<board_size; j=j+1)
            if (j!= queen_cols[i])
                cout << '.';
            else
                cout << 'D';
        cout << endl;
    }
}


int queen_configs (int row, columns queen_cols)
{
    if (row == board_size) {
        display_board (queen_cols);
        return 1;
    }
    else {
        int nr_configs = 0;
        for (int col=0; col<board_size; col=col+1)
            if (good_position (row, queen_cols, col))
            {
                queen_cols[row] = col;
                nr_configs = nr_configs +
                    queen_configs (row+1, queen_cols);
            }
        return nr_configs;
    }
}

int main ()
{
    columns queen_cols;
    cout << "Es wurden " << queen_configs (0, queen_cols)
         << " Loesungen gefunden." << endl;
    return 0;
```

```
}
```

Bemerkung: Dieses Programm benutzt zwei neue Elemente:

- Funktionen ohne Rückgabewert kann man mit `void` kennzeichnen.
- Es wurde eine globale Konstante `board_size` verwendet.

Bemerkung:

- Dieses Verfahren des Ausprobierens verschiedener Möglichkeiten durch eine sogenannte Tiefensuche in einem Baum ist als Backtracking bekannt.
- Für $n = 8$ gibt es 92 Lösungen. Eine davon ist

```
. . . . . . . D
. . . D . . . .
D . . . . . . .
. . D . . . . .
. . . . . D . .
. D . . . . . .
. . . . . . D .
. . . . D . . .
```

9 Einschub: Wiederholung Aufwand

Definition: Mit Aufwand bezeichnet man allgemein die Menge an Ressourcen, die für die Lösung einer bestimmten Aufgabe benötigt werden. Dies kann Rechenzeit, Speicher, Entwicklungszeit, etc sein. Wir interessieren uns vor allem für Rechenzeit und Speicher.

Bemerkung: Man ist vor allem an der Lösung großer Probleme interessiert. Daher interessiert der Aufwand $A(n)$ für große n, wobei n in geeigneter Weise die Problemgröße misst.

9.1 Beispiel 1: Telefonbuch

Wir betrachten den Aufwand für das Finden eines Namens in einem Telefonbuch der Seitenzahl n.

Algorithmus: (A1) Blättere das Buch von Anfang bis Ende durch.

Satz: Sei $C_1 > 0$ die (maximale) Zeit, die das Durchsuchen einer Seite benötigt. Der maximale Zeitaufwand $A_1 = A_1(n)$ für Algorithmus A1 ist dann abschätzbar durch

$$A_1(n) = C_1 n$$

Algorithmus: (A2) Rekursives Halbieren.

Satz: Sei $C_1 > 0$ die (maximale) Zeit, die das Durchsuchen einer Seite benötigt, und $C_2 > 0$ die (maximale) Zeit, die das Prüfen einer Seite und anschließende Umblättern benötigt. Der maximale Zeitaufwand $A_2 = A_2(n)$ für Algorithmus A2 ist dann abschätzbar durch

$$A_2(n) = C_1 + C_2 \log_2 n$$

Satz: Für große Telefonbücher ist Algorithmus 2 „besser", d.h. der maximale Zeitaufwand ist kleiner.

Beweis:

$$\frac{A_1(n)}{A_2(n)} = \frac{C_1 n}{C_1 + C_2 \log_2 n} \to +\infty$$

Beobachtung:

- Die genauen Werte von C_1, C_2 sind für diese Aussage unwichtig.
- Für spezielle Eingaben (z.B. Andreas Aalbert) kann auch Algorithmus 1 besser sein.

Definition: Man sagt $A(n) = O(f(n))$, wenn es $C > 0$ und $N > 0$ gibt mit

$$A(n) \leq C f(n)\,, \quad \forall n \geq N$$

Bemerkung: Um „Algorithmus 2 ist für große Telefonbücher besser" zu schließen, reichen die Informationen $A_1(n) = O(n)$ und $A_2(n) = O(\log n)$ aus. Man beachte auch, dass wegen $\log_2 n = \frac{\log n}{\log 2}$ gilt $O(\log_2 n) = O(\log n)$.

9.2 Beispiel 2: Pascal-Dreieck

Wir bestimmen die Binomialkoeffizienten $B_{n,k}$ für $0 \leq k \leq n$ gemäß der Rekursionsformel

$$B_{n,0} = B_{n,n} = 1$$
$$B_{n,k} = B_{n-1,k-1} + B_{n-1,k}\,, \quad 0 < k < n$$

Die Rechenzeit für die Berechnung von $B_{n,k}$ bezeichnen wir mit $A_{n,k}$.

Satz: Es gebe $C_1 > 0$ und $C_2 \geq 0$ mit

- $0 < A_{n,0} \leq C_1$ und $0 < A_{n,n} \leq C_1$.
- $A_{n,k} \leq C_2 + A_{n-1,k-1} + A_{n-1,k}$ für $0 < k < n$.

Dann gilt $A_{n,k} = O(B_{n,k})$.

Beweis: Setze $\tilde{A}_{n,k} := A_{n,k} + C_2$. Dann gilt

$$\tilde{A}_{n,k} = C_2 + A_{n,k} \leq 2C_2 + A_{n-1,k-1} + A_{n-1,k} = \tilde{A}_{n-1,k-1} + \tilde{A}_{n-1,k}$$

sowie

$$\tilde{A}_{n,0} \leq C_1 + C_2\,, \quad \tilde{A}_{n,n} \leq C_1 + C_2$$
$$\tilde{A}_{n,k} \leq (C_1 + C_2)B_{n,k} \Leftrightarrow A_{n,k} \leq (C_1 + C_2)B_{n,k} - C_2 \leq (C_1 + C_2)B_{n,k}$$

Bemerkung:

- Die Aussage des Satzes hängt also überhaupt nicht von den genauen Werten C_1 oder C_2 ab! Sogar ein Extremfall wie $C_2 = 0$ würde zum selben Ergebnis führen.
- Meist sind nur kleine Teile von Programmen für die Laufzeit wirklich wichtig. Um diese Bereiche zu finden, gibt es geeignete Werkzeuge (Stichwort: *Profiling*).
- Die entscheidende Verbesserung ist sehr oft die Wahl eines besseren Algorithmus!

9.3 Zusammengesetzte Datentypen

Bei zusammengesetzten Datentypen kann man eine beliebige Anzahl möglicherweise verschiedener (sogar zusammengesetzte) Datentypen zu einem neuen Datentyp kombinieren. Diese Art Datentypen nennt man Strukturen.

Beispiel: Aus zwei `int`-Zahlen erhalten wir die Struktur `Rational`:

```
struct Rational { // Schluesselwort struct
    int zaehler;  // eine Liste von
    int nenner;   // Variablendefinitionen
} ;               // Semikolon nicht vergessen
```

Dieser Datentyp kann nun wie folgt verwendet werden

```
Rational p;       // Definition einer Variablen
p.zaehler = 3;    // Initialisierung der Komponenten
p.nenner = 4;
```

Syntax: (Zusammengesetzter Datentyp)

<StructDef> ::= struct <Name: > { { <Komponente> ; }$^{+}$ } ;
<Komponente> ::= <VarDef> | <FeldDef> | ...

Eine Komponente ist entweder eine Variablendefinition ohne Initialisierung oder eine Felddefinition. Dabei kann der Typ der Komponente selbst zusammengesetzt sein.

Bemerkung: Strukturen sind ein Spezialfall von sehr viel mächtigeren Klassen, die später im OO-Teil behandelt werden.

Bemerkung: Im Gegensatz zu Feldern kann man mit Strukturen (zusammengesetzten Daten) gut funktional arbeiten, siehe etwa Abelson&Sussman: *Structure and Interpretation of Computer Programs*.

9.3.1 Anwendung: Rationale Zahlen

Programm: (Rationale Zahlen, die erste)

```
#include <iostream>
using namespace std;

struct Rational {
    int zaehler;
    int nenner;
} ;

// Abstraktion: Konstruktor und Selektoren

Rational erzeuge_rat (int z, int n)
{
```

```
    Rational t;
    t.zaehler = z;
    t.nenner = n;
    return t;
}
int zaehler (Rational q)
{
    return q.zaehler;
}
int nenner (Rational q)
{
    return q.nenner;
}

// Arithmetische Operationen

Rational add_rat (Rational p, Rational q)
{
    return erzeuge_rat(zaehler(p)*nenner(q)+zaehler(q)*nenner(p),
                       nenner(p)*nenner(q));
}

Rational sub_rat (Rational p, Rational q)
{
    return erzeuge_rat(zaehler(p)*nenner(q)-zaehler(q)*nenner(p),
                       nenner(p)*nenner(q));
}

Rational mul_rat (Rational p, Rational q)
{
    return erzeuge_rat(zaehler(p)*zaehler(q),
                       nenner(p)*nenner(q));
}

Rational div_rat (Rational p, Rational q)
{
    return erzeuge_rat(zaehler(p)*nenner(q),
                       nenner(p)*zaehler(q));
}

void drucke_rat (Rational p)
{
    cout << zaehler(p) << "/" << nenner(p) << endl;
}

int main ()
```

```
{
    Rational p = erzeuge_rat(3,4);
    Rational q = erzeuge_rat(5,3);
    drucke_rat(p); drucke_rat(q);

    // p*q+p-p*p
    Rational r = sub_rat(add_rat(mul_rat(p,q), p),
                         mul_rat(p,p));
    drucke_rat(r);
        return 0;
}
```

Bemerkung: Man beachte die Abstraktionsschicht, die wir durch den Konstruktor `erzeuge_rat` und die Selektoren `zaehler` und `nenner` eingeführt haben. Diese Schicht stellt die sogenannte Schnittstelle dar, über die unser Datentyp verwendet werden soll.

Problem: Noch ist keine Kürzung im Programm eingebaut. So liefert das obige Programm $1104/768$ anstatt $23/16$.

Abhilfe: Normalisierung im Konstruktor:

```
Rational erzeuge_rat (int z, int n)
{
    int g;
    Rational t;

    if (n<0) { n = -n; z = -z; }
    g = ggT(abs(z),n);
    t.zaehler = z/g;
    t.nenner = n/g;
    return t;
}
```

Bemerkung: Ohne die Verwendung des Konstruktors hätten wir in allen arithmetischen Funktionen Änderungen durchführen müssen, um das Ergebnis in Normalform zu bringen.

Komplexe Zahlen

Analog lassen sich komplexe Zahlen einführen:

Programm: **(Komplexe Zahlen, Version 1)**

```
#include <iostream>
using namespace std;

struct Complex {
    float real;
    float imag;
} ;
```

```
Complex erzeuge_complex (float re, float im)
{
    Complex t;
    t.real = re; t.imag = im;
    return t;
}
float real (Complex q) {return q.real;}
float imag (Complex q) {return q.imag;}

Complex add_complex (Complex p, Complex q)
{
    return erzeuge_complex(real(p) + real(q),
                           imag(p) + imag(q));
}

// etc

void drucke_complex (Complex p)
{
    cout << real(p) << "+i*" << imag(p) << endl;
}

int main ()
{
    Complex p = erzeuge_complex(3.0,4.0);
    Complex q = erzeuge_complex(5.0,3.0);
    drucke_complex(p);
    drucke_complex(q);
    drucke_complex(add_complex(p,q));
}
```

Bemerkung: Hier wäre bei Verwendung der Funktionen `real` und `imag` zum Beispiel auch eine Änderung der internen Darstellung zu Betrag/Argument ohne Änderung der Schnittstelle möglich.

9.3.2 Gemischtzahlige Arithmetik

Problem: Was ist, wenn man mit komplexen und rationalen Zahlen gleichzeitig rechnen will?

Antwort: Im Bastian-Skript ist an dieser Stelle folgende Lösung ausgeführt:

1. Führe eine sogenannte variante Struktur (Schlüsselwort `union`) ein, die entweder eine rationale oder eine komplexe Zahl enthalten kann ⇝ neuer Datentyp `Combination`

2. Füge auch eine Kennzeichnung hinzu, um was für eine Zahl (rational/komplex) es sich tatsächlich handelt ⇝ neuer Datentyp `Mixed`.

3. Funktionen wie `add_mixed` prüfen die Kennzeichnung, konvertieren bei Bedarf und rufen dann `add_rat` bzw. `add_complex` auf.

Bemerkung: Diese Lösung hat etliche Probleme:

- Umständlich und unsicher
- Das Hinzufügen weiterer Zahlentypen macht eine Änderung von bestehenden Funktionen nötig
- Typprüfungen zur Laufzeit → keine optimale Effizienz
- Hätten gerne: Infix-Notation und `cout` mit unseren Zahlen

Bemerkung: Einige dieser Probleme werden wir mit den objektorientierten Erweiterungen von C++ vermeiden. Man sollte sich allerdings auch klar machen, dass das Problem von Arithmetik mit verschiedenen Zahltypen und eventuell auch verschiedenen Genauigkeiten tatsächlich extrem komplex ist. Eine vollkommene Lösung darf man daher nicht erwarten.

10 Globale Variablen und das Umgebungsmodell

10.1 Globale Variablen

Bisher: *Funktionen haben kein Gedächtnis!* Ruft man eine Funktion zweimal mit den selben Argumenten auf, so liefert sie auch dasselbe Ergebnis.

Grund:

- Funktionen hingen nur von ihren Parametern ab.
- Die lokale Umgebung bleibt zwischen Funktionsaufrufen nicht erhalten.

Das werden wir jetzt ändern!

10.1.1 Beispiel: Konto

Ein Konto kann man einrichten (mit einem Anfangskapital versehen), man kann abheben (mit negativem Betrag auch einzahlen), und man kann den Kontostand abfragen.

Programm: **(Konto)**

```
#include <iostream>
using namespace std;

int konto; // die GLOBALE Variable

void einrichten (int betrag)
{
    konto = betrag;
}

int kontostand ()
{
    return konto;
}

int abheben (int betrag)
{
    konto = konto-betrag;
    return konto;
}

int main ()
{
    einrichten(100);
    cout << abheben(25) << endl;
    cout << abheben(25) << endl;
    cout << abheben(25) << endl;
}
```

Bemerkung:

- Die Variable `konto` ist außerhalb jeder Funktion definiert.
- Die Variable `konto` wird zu Beginn des Programmes erzeugt und *nie* mehr zerstört.
- Alle Funktionen können auf die Variable `konto` zugreifen. Man nennt sie daher eine globale Variable.

Frage: Oben haben wir eingeführt, dass Ausdrücke relativ zu einer Umgebung ausgeführt werden. In welcher Umgebung liegt `konto`?

10.2 Das Umgebungsmodell

Die Auswertung von Funktionen und Ausdrücken mit Hilfe von Umgebungen nennt man *Umgebungsmodell* (im Gegensatz zum Substitutionsmodell).

Definition: **(Umgebung)**

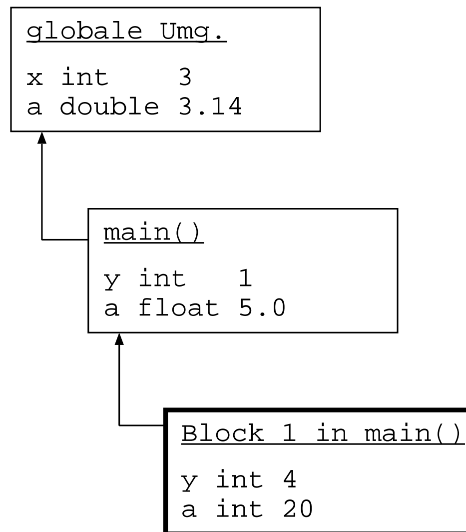
- Eine Umgebung enthält eine Bindungstabelle, d. h. eine Zuordnung von Namen zu Werten.
- Es kann beliebig viele Umgebungen geben. Umgebungen werden während des Programmlaufes implizit (automatisch) oder explizit (bewusst) erzeugt bzw. zerstört.
- Die Menge der Umgebungen bildet eine Baumstruktur. Die Wurzel dieses Baumes heißt globale Umgebung.
- Zu jedem Zeitpunkt des Programmablaufes gibt es eine aktuelle Umgebung. Die Auswertung von Ausdrücken erfolgt relativ zur aktuellen Umgebung.
- Die Auswertung relativ zur aktuellen Umgebung versucht den Wert eines Namens in dieser Umgebung zu ermitteln, schlägt dies fehl wird rekursiv in der nächst höheren („umschließenden“) Umgebung gesucht.

Eine Umgebung ist also relativ kompliziert. Das Umgebungsmodell beschreibt, wann Umgebungen erzeugt/zerstört werden und wann die Umgebung gewechselt wird.

Beispiel:

```
int x=3;
double a=4.3;       // 1
void main ()
{
    int y=1;
    float a=5.0;    // 2
    {
        int y=4;
        int a=8;    // 3

        a = 5*y;    // 4
        ::a = 3.14;// 5
    }
}                   // 6
```

Nach Zeile 5:



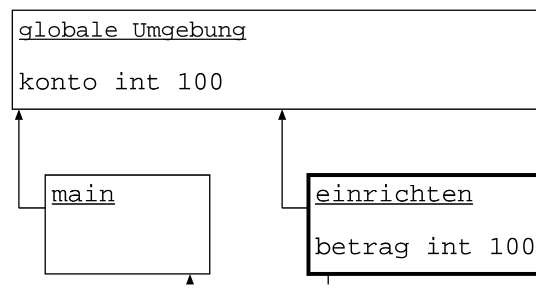
Eigenschaften:

- In einer Umgebung kann ein Name nur höchstens einmal vorkommen. In verschiedenen Umgebungen kann ein Name mehrmals vorkommen.
- Kommt auf dem Pfad von der aktuellen Umgebung zur Wurzel ein Name mehrmals vor, so verdeckt das erste Vorkommen die weiteren.
- Eine Zuweisung wirkt immer auf den sichtbaren Namen. Mit vorangestelltem `::` erreicht man die Namen der globalen Umgebung.
- Eine Anweisungsfolge in geschweiften Klammern bildet einen sogenannten Block, der eine eigene Umgebung besitzt.
- Eine Schleife `for (int i=0; ...` wird in einer eigenen Umgebung ausgeführt. Diese Variable `i` gibt es im Rest der Funktion nicht.

Beispiel: (Funktionsaufruf)

```
int konto;

void einrichten (int betrag)
{
    konto = betrag;   // 2
}

void main ()
{
    einrichten(100); // 1
}
```

Nach Marke 2:



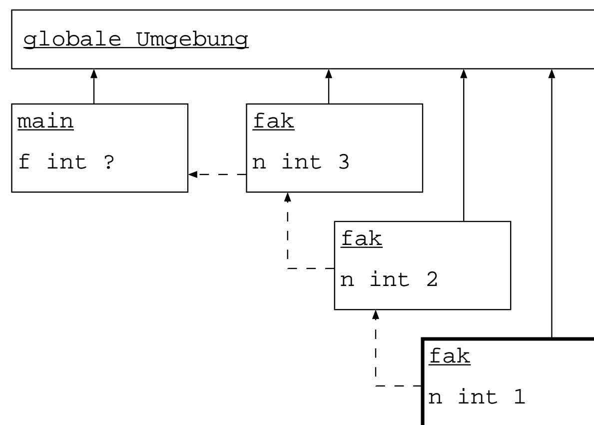
Bemerkung:

- Jeder Funktionsaufruf startet eine neue Umgebung unterhalb der globalen Umgebung. Dies ist dann die aktuelle Umgebung.
- Am Ende einer Funktion wird ihre Umgebung vernichtet und die aktuelle Umgebung wird die, in der der Aufruf stattfand (gestrichelte Linie).
- Formale Parameter sind ganz normale Variable, die mit dem Wert des aktuellen Parameters initialisiert sind.
- Sichtbarkeit von Namen ist in C++ am Programmtext abzulesen (statisch) und somit zur Übersetzungszeit bekannt. Sichtbar sind: Namen im aktuellen Block, nicht verdeckte Namen in umschließenden Blöcken und Namen in der globalen Umgebung.

Beispiel: (Rekursiver Aufruf)

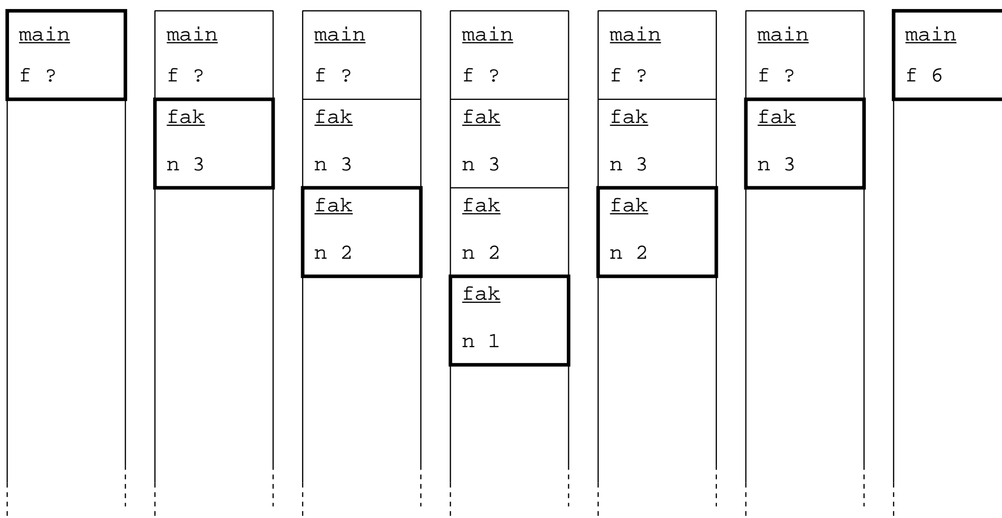
```
int fak (int n)
{
    if (n==1)
        return 1;
    else
        return n*fak(n-1);
}

void main ()
{
    int f=fak(3); // 1
}
```

Während Auswertung von `fak(3)`:



Bemerkung: Im obigen Beispiel gibt es zusätzlich noch eine „versteckte“ Variable für den Rückgabewert einer Funktion.

Beispiel: Die Berechnung von `fak(3)` führt zu:



10.3 Stapel

Bezeichnung: Die Datenstruktur, mit der das Umgebungsmodell normalerweise implementiert wird, nennt man Stapel, Stack oder LIFO (last in, first out). Im Deutschen ist, historisch bedingt, auch die Bezeichnung Keller gebräuchlich.

Definition: Ein Stapel ist eine Datenstruktur, welche folgende Operationen zur Verfügung stellt:

- Erzeugen eines leeren Stapels (*create*)
- Ablegen eines neuen Elements auf dem Stapel (*push*)
- Test, ob Stapel leer (*empty*)
- Holen des zuletzt abgelegten Elements vom Stapel (*pop*)

Programm: (Stapel)

```
#include <iostream>
using namespace std;

typedef int element_type;   // Integer-Stack

// START stack-library ...

const int stack_size = 1000;

element_type stack[stack_size];
int top = 0;  // Stapelzeiger

// Stack-Operationen

void push (element_type e)
{
    stack[top] = e;
    top = top+1;
}
bool empty ()
{
    return top==0;
}
element_type pop ()
{
    top = top-1;
    return stack[top];
}
```

```
int main ()
{
    push(4);
    push(5);
    while (!empty())
        cout << pop() << endl;
    return 0;
}
```

Bemerkung: Die Stapel-Struktur kann man verwenden, um rekursive in nicht rekursive Programme zu transformieren (siehe Bastian-Skript). Dies ist aber normalerweise nicht von Vorteil, da der für Rekursion verwendete Stapel höchstwahrscheinlich effizienter verwaltet wird.

10.4 Monte-Carlo Methode zur Bestimmung von π

Folgender Satz soll zur (näherungsweisen) Bestimmung von π herangezogen werden:

Satz: *Die Wahrscheinlichkeit q, dass zwei Zahlen $u, v \in \mathbb{N}$ keinen gemeinsamen Teiler haben, ist $\frac{6}{\pi^2}$. Zu dieser Aussage siehe [Knuth, Vol. 2, Theorem D].*

Um π zu approximieren, gehen wir daher wie folgt vor:

- Führe N „Experimente" durch:
  - Ziehe „zufällig" zwei Zahlen $1 \leq u_i, v_i \leq n$.
  - Berechne ggT(u_i, v_i).
  - Setze
$$e_i = \begin{cases} 1 & \text{falls ggT}(u_i, v_i) = 1 \\ 0 & \text{sonst} \end{cases}$$
- Berechne relative Häufigkeit $p(N) = \frac{\sum_{i=1}^{N} e_i}{N}$. Wir erwarten $\lim_{N \to \infty} p(N) = \frac{6}{\pi^2}$.
- Also gilt $\pi \approx \sqrt{6/p(N)}$ für große N.

10.4.1 Pseudo-Zufallszahlen

Um Zufallszahlen zu erhalten, könnte man physikalische Phänomene heranziehen, von denen man überzeugt ist, dass sie „zufällig" ablaufen (z.B. radioaktiver Zerfall). Solche Zufallszahl-Generatoren gibt es tatsächlich, sie sind allerdings recht teuer.

Daher begnügt man sich stattdessen mit Zahlenfolgen $x_k \in \mathbb{N}$, $0 \leq x_k < n$, welche deterministisch sind, aber zufällig „aussehen". Für die „Zufälligkeit" gibt es verschiedene Kriterien. Beispielsweise sollte jede Zahl gleich oft vorkommen, wenn man die Folge genügend lang macht:

$$\lim_{m \to \infty} \frac{|\{i | 1 \leq i \leq m \wedge x_i = k\}|}{m} = \frac{1}{n}, \quad \forall k = 0, \ldots, n-1.$$

Einfachste Methode: Ausgehend von einem x_0 verlangt man für $x_1, x_2, \ldots$ die Iterationsvorschrift

$$x_{n+1} = (ax_n + c) \bmod m.$$

Damit die Folge zufällig aussieht, müssen $a, c, m \in \mathbb{N}$ gewisse Bedingungen erfüllen, die man in [Knuth, Vol. 2, Kapitel 3] nachlesen kann.

Programm: (**π mit Monte Carlo Methode**)

```
#include<iostream>
using namespace std;

unsigned int x = 93267;

unsigned int zufall ()
{
    unsigned int ia = 16807,  im = 2147483647;
    unsigned int iq = 127773, ir = 2836;
    unsigned int k;

    k = x/iq;                // LCG xneu = (a*xalt) mod m
    x = ia*(x-k*iq)-ir*k;    // a = 7^5, m = 2^31-1
    if (x<0) x = x+im;       // Implementierung ohne lange Arithmetik
    return x;                // siehe Numerical Recipes in C, Kap. 7.
}

unsigned int ggT (unsigned int a, unsigned int b)
{
    if (b==0) return a;
    else      return ggT(b,a%b);
}

int experiment ()
{
    unsigned int x1,x2;

    x1 = zufall(); x2 = zufall();
    if (ggT(x1,x2)==1)
        return 1;
    else
        return 0;
}

double montecarlo (int N)
{
    int erfolgreich=0;

    for (int i=0; i<N; i=i+1)
```

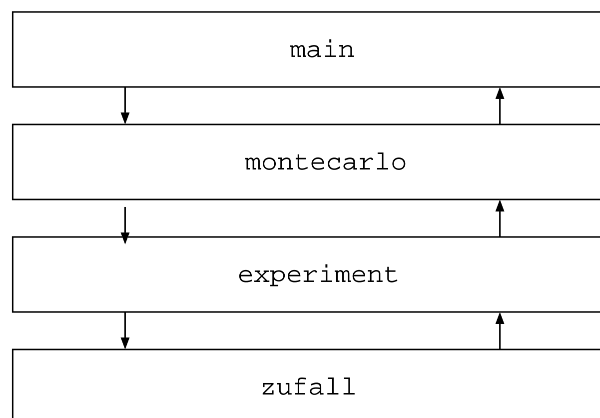
```
        erfolgreich = erfolgreich+experiment();

    return ((double)erfolgreich)/((double)N);
}

int main ()
{
    cout.precision(20);
    cout << sqrt(6.0/montecarlo(10000)) << endl;
}
```

10.4.2 Monte-Carlo funktional

Die Funktion `zufall` widerspricht offenbar dem funktionalen Paradigma (sonst müsste sie ja immer denselben Wert zurückliefern!). Stattdessen hat sie „Gedächtnis" durch die globale Variable `x`.

Frage: Wie würde eine funktionale(re) Version des Programms ohne globale Variable aussehen?

Antwort: Eine Möglichkeit wäre es, `zufall` den Parameter x zu übergeben, woraus dann ein neuer Wert berechnet wird. Dieser Parameter müsste aber von `main` aus durch alle Funktionen hindurchgetunnelt werden:



Für `experiment`→`montecarlo` ist obendrein die Verwendung eines zusammengesetzten Datentyps als Rückgabewert nötig.

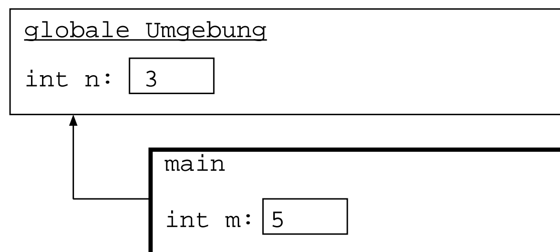
Beobachtung: In dieser Situation entstünde durch Beharren auf einem funktionalen Stil zwar kein Effizienzproblem, die Struktur des Programms würde aber deutlich komplizierter.

11 Zeiger und dynamische Datenstrukturen

11.1 Zeiger

Wir können uns eine Umgebung als Sammlung von Schubladen (Orten) vorstellen, die Werte aufnehmen können. Jede Schublade hat einen Namen, einen Typ und einen Wert:

```
int n=3;

void main () {
    int m=5;
}
```



Idee: Es wäre nun praktisch, wenn man so etwas wie „Erhöhe den Wert in *dieser* Schublade (Variable) um eins“ ausdrücken könnte.

Anwendung: Im Konto-Beispiel möchten wir nicht nur ein Konto sondern viele Konten verwenden. Hierzu benötigt man einen Mechanismus, der einem auszudrücken erlaubt, *welches* Konto verändert werden soll.

Idee: Man führt einen Datentyp Zeiger ein, der auf Variablen (Schubladen) zeigen kann. Variablen, die Zeiger als Werte haben können, heißen Zeigervariablen.

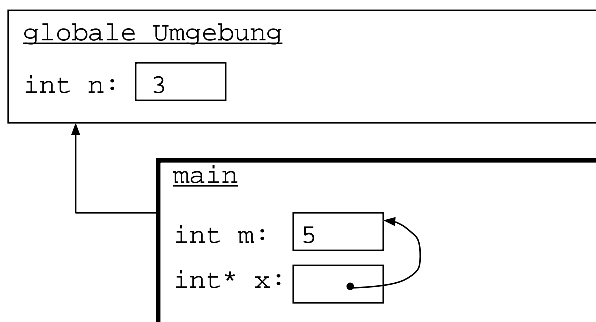
Bemerkung: Intern entspricht ein Zeiger der Adresse im physikalischen Speicher, an dem der Wert einer Variablen steht.

Notation: Die Definition „`int *x;`“ vereinbart, dass `x` auf Variablen (Schubladen) vom Typ `int` zeigen kann. Man sagt `x` habe den Typ `int*`.
Die Zuweisung „`x = &n;`“ lässt `x` auf den Ort zeigen, an dem der Wert von `n` steht.
Die Zuweisung „`*x = 4;`“ verändert den Wert der Schublade, „auf die `x` zeigt“.

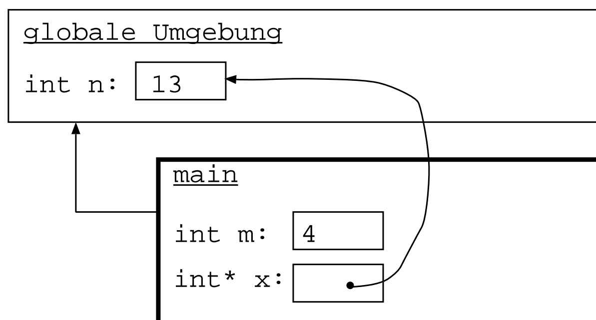
Beispiel:

```
int n=3;

int main ()
{
    int  m=5;  // 1
    int* x=&m; // 2
    *x = 4;    // 3
    x  = &n;   // 4
    *x = 13;   // 5
    return 0;
}
```

Nach (2)



Nach (5)



11.2 Zeiger im Umgebungsmodell

Im Umgebungsmodell gibt es eine Bindungstabelle, mittels derer jedem Namen ein Wert (und ein Typ) zugeordnet wird, etwa:

Name	Wert	Typ
n	3	int
m	5	int

Mathematisch entspricht das einer Abbildung w, die die *Symbole* $\mathtt{n}, \mathtt{m}$ auf die Wertemenge abbildet:

$$w : \{\mathtt{n}, \mathtt{m}\} \to \mathbb{Z}.$$

Die Zuweisung „`n = 3;`“ („=“ *ist Zuweisung*) manipuliert die Bindungstabelle so, dass nach der Zuweisung $w(\mathtt{n}) = 3$ („=“ *ist Gleichheit*) gilt.
Wenn auf der rechten Seite der Zuweisung auch ein Name steht, etwa „`n=m+1;`“, dann gilt nach der Zuweisung $w(\mathtt{n}) = w(\mathtt{m}) + 1$. Auf beide Namen wird also w angewandt.

Problem: Wir haben mehrere verschiedene Konten und möchten eine Funktion schreiben, die Beträge von Konten abhebt. In einer Variable soll dabei angegeben werden, von *welchem* Konto abgehoben wird.

Idee: Wir lassen Namen selbst wieder als Werte von (anderen) Namen zu, z.B.

Name	Wert	Typ
n	3	int
m	5	int
x	n	int*

Verwirklichung:

1. & ist der (einstellige) Adressoperator: „x=&n“ ändert die Bindungstabelle so, dass $w(\mathtt{x}) = \mathtt{n}$.
2. * ist der (einstellige) Dereferenzierungsoperator: Wenn x=&n gilt, so weist „*x=4;“ dem Wert von n die Zahl 4 zu.
3. Den Typ eines Zeigers auf einen Datentyp X bezeichnet man mit X*.

Bemerkung:

- Auf der rechten Seite einer Zuweisung kann auf einen Namen der &-Operator genau einmal angewandt werden. Dieser *verhindert* die Anwendung von w.
- Der $*$-Operator wendet die Abbildung w einmal auf das Argument rechts von ihm an. Der $*$-Operator kann mehrmals und sowohl auf der linken als auch auf der rechten Seite der Zuweisung angewandt werden.

Bemerkung: Auch eine Zeigervariable x kann wieder von einer anderen Zeigervariablen referenziert werden.
Diese hat dann den Typ int** oder „Zeiger auf eine int*-Variable“.

Beispiel:

```
int     n = 3;
int     m = 5;
int*    x = &n;
int**   y = &x;
int***  z = &y;
```

Name	Wert	Typ
n	3	int
m	5	int
x	n	int*
y	x	int**
z	y	int***

Damit können wir schreiben

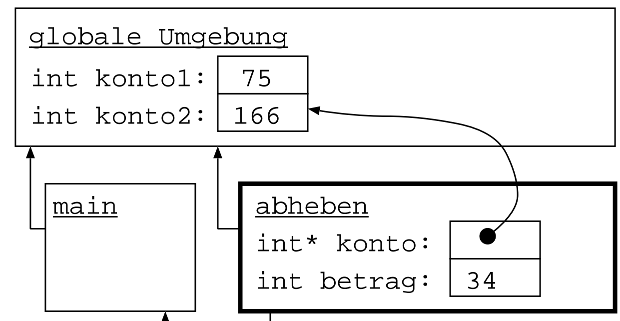
```
n = 4;    // das ist
*x = 4;   // alles
**y = 4;  // das
***z = 4; // gleiche !

x = &m;   // auch
*y = &m;  // das
**z = &m; // ist gleich !

y = &n;   // geht nicht, da n nicht vom Typ int*
y = &&n;  // geht auch nicht, da & nur
          // einmal angewandt werden kann
```

11.3 Call by reference

Programm: Die Realisation der Kontoverwaltung könnte wie folgt aussehen:

```
int konto1=100;
int konto2=200;

int abheben (int* konto, int betrag)
{
    *konto = *konto - betrag; // 1
    return *konto;            // 2
}

int main () {
    abheben(&konto1,25);      // 3
    abheben(&konto2,34);      // 4
}
```

Nach Marke 1, im zweiten Aufruf von abheben:



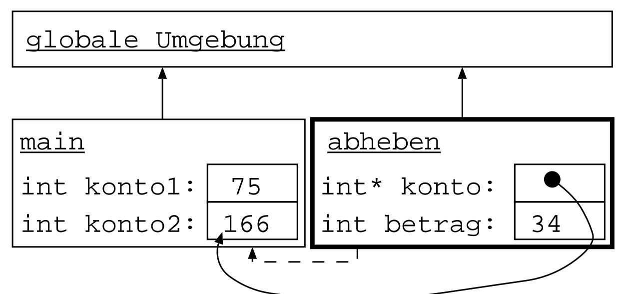
Definition: In der Funktion `abheben` nennt man `betrag` einen *call by value* Parameter und `konto` einen *call by reference* Parameter.

Bemerkung: Es gibt Computersprachen, die konsequent *call by value* verwenden (z.B. Lisp/-Scheme), und solche, die konsequent *call by reference* verwenden (z.B. Fortran). Algol 60 war die erste Programmiersprache, die beides möglich machte.

Bemerkung: Die Variablen `konto1`, `konto2` im letzten Beispiel müssen nicht global sein! Folgendes ist auch möglich:

```
int abheben (int* konto, int betrag)
{
    *konto = *konto - betrag; // 1
    return *konto;            // 2
}

int main () {
    int konto1=100;
    int konto2=200;

    abheben(&konto1,25);      // 3
    abheben(&konto2,34);      // 4
}
```

Nach (1), zweiter Aufruf von abheben



Bemerkung: `abheben` darf `konto1` in `main` verändern, obwohl dieser Name dort nicht sichtbar ist! Zeiger können also die Sichtbarkeitsregeln durchbrechen und — im Prinzip — kann somit auch jede lokale Variable von einer anderen Prozedur aus verändert werden.

Bemerkung: Es gibt im wesentlichen zwei Situationen in denen man Zeiger als Argumente von Funktionen einsetzt:

- Der Seiteneffekt ist explizit erwünscht wie in `abheben` (→ Objektorientierung).

- Man möchte das Kopieren großer Objekte sparen (→ `const` Zeiger).

11.3.1 Referenzen in C++

Beobachtung: Obige Verwendung von Zeigern als Prozedurparameter ist ziemlich umständlich: Im Funktionsaufruf müssen wir ein & vor das Argument setzen, innerhalb der Prozedur müssen wir den * benutzen.

Abhilfe: Wenn man in der Funktionsdefinition die Syntax `int& x` verwendet, so kann man beim Aufruf den Adressoperator & und bei der Verwendung innerhalb der Funktion den Dereferenzierungsoperator * weglassen. Dies ist wieder sogenannter „syntaktischer Zucker“.

Programm: (Konto mit Referenzen)

```
int abheben (int& konto, int betrag)
{
    konto = konto - betrag;  // 1
    return konto;            // 2
}

void main ()
{
    int konto1=100;
    int konto2=200;

    abheben(konto1,25);      // 3
    abheben(konto2,34);      // 4
}
```

Bemerkung: Referenzen können nicht nur als Funktionsargumente benutzt werden:

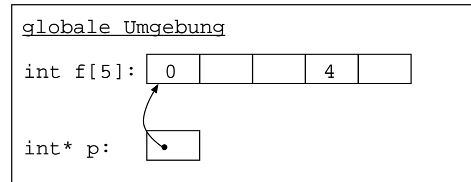
```
int  n=3;
int& r=n; // independent reference
r = 5;    // identisch zu n=5;
```

11.4 Zeiger und Felder

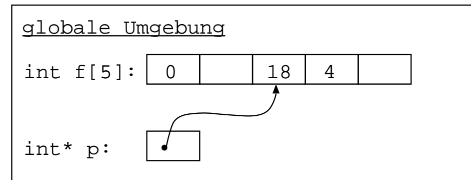
Beispiel: Zeiger und (eingebaute) Felder sind in C/C++ synonym:

```
int f[5];
int* p=f; // f hat Typ int*

...

p[0]=0;
p[3]=4;  // 1

p = &(f[2]);
*p = 18;     // p[0] = 18;
```

Nach Marke 1:



Am Ende



Bemerkung:

- Die Äquivalenz von eingebauten Feldern mit Zeigern ist eine höchst problematische Eigenschaft von C. Insbesondere führt sie dazu, dass man innerhalb einer mit einem Feld aufgerufenen Funktion die Länge dieses Felds nicht direkt zur Verfügung hat. Diese muss anderweitig bekannt sein, oder es muss auf eine *Bereichsüberprüfung*erzichtet werden, was unter anderem ein Sicherheitsproblem darstellt (siehe auch die Diskussion bei `char`-Feldern).
- In C++ werden daher bessere Feldstrukturen (`vector`, `string`, `valarray`) in der Standard-Bibliothek STL (*Standard Template Library*) zur Verfügung gestellt.

11.5 Zeiger auf zusammengesetzte Datentypen

Beispiel: Es sind auch Zeiger auf Strukturen möglich. Ist `p` ein solcher Zeiger, so kann man mittels `p-><Komponente>` eine Komponente selektieren:

```
struct rational {
    int n;
    int d;
} ;

int main () {
    rational  q;
    rational* p = &q;
    (*p).n = 5;   // Zuweisung an Komponente n von q
    p->n   = 5;   // identische Abkuerzung
}
```

11.6 Problematik von Zeigern

Beispiel: Betrachte folgendes Programm:

```
char* alphabet () {
    char buffer[27];
    for (int i=0; i<26; i++) buffer[i] = i+65;
    buffer[26]=0;
    return buffer;
}
int main () {
    std::cout << alphabet();
}
```

Beobachtung: Der Speicher für das lokale Feld ist schon freigegeben, aber den Zeiger darauf gibt es noch.

Bemerkung:

- Der gcc-Compiler warnt für das vorige Beispiel, dass ein Zeiger auf eine lokale Variable zurückgegeben wird. Er merkt allerdings schon nicht mehr, dass auch der Rückgabewert `buffer+2` problematisch ist.
- Zeiger sind ein sehr *maschinennahes* Konzept (vgl. Neumann-Architektur). In vielen Programmiersprachen (z.B. Lisp, Java, etc) sind sie daher für den Programmierer nicht sichtbar.
- Um die Verwendung von Zeigern sicher zu machen, muss man folgendes Prinzip beachten: *Speicher darf nur dann freigegeben werden, wenn keine Referenzen darauf mehr existieren.* Dies ist vor allem für die im nächsten Abschnitt diskutierte dynamische Speicherverwaltung wichtig.

11.7 Dynamische Speicherverwaltung

Bisher: Zwei Sorten von Variablen:

- Globale Variablen, die für die gesamte Laufzeit des Programmes existieren.
- Lokale Variablen, die nur für die Lebensdauer des Blockes/der Prozedur exisitieren.

Jetzt: Dynamische Variablen. Diese werden vom Programmierer explizit erzeugt und vernichtet. Dazu dienen die Operatoren `new` und `delete`. Dynamische Variablen haben keinen Namen und können (in C/C++) nur indirekt über Zeiger bearbeitet werden.

Beispiel:

```
int m;
rational* p = new rational;
p->n = 4; p->d = 5;
m = p->n;
delete p;
```

Bemerkung:

- Die Anweisung `rational *p = new rational` erzeugt eine Variable vom Typ `rational` und weist deren Adresse dem Zeiger `p` zu. Man sagt auch, dass die Variable dynamisch allokiert wurde.
- Dynamische Variablen werden nicht auf dem Stack der globalen und lokalen Umgebungen gespeichert, sondern auf dem so genannten Heap. Dadurch ist es möglich, dass dynamisch allokierte Variablen in einer Funktion allokiert werden und die Funktion überdauern.
- Dynamische Variablen sind notwendig, um Strukturen im Rechner zu erzeugen, deren Größe sich während der Rechnung ergibt (und von der aufrufenden Funktion nicht gekannt wird).
- Auch Felder können dynamisch erzeugt werden:

```
int  n = 18;
int* q = new int[n];  // Feld mit 18 int Eintraegen
q[5] = 3;
delete[] q;           // dynamisches Feld löschen
```

11.7.1 Probleme bei dynamischen Variablen

Beispiel: Wie schon im vorigen Abschnitt bemerkt, kann auf Zeiger zugegriffen werden, obwohl der Speicher schon freigegeben wurde:

```
int f ()
{
    rational* p = new rational;
    p->n = 50;
    delete p;      // Vernichte Variable
    return p->n;   // Ooops, Zeiger gibt es immer noch
}
```

Beispiel: Wenn man alle Zeiger auf dynamisch allokierten Speicher löscht, kann dieser nicht mehr freigegeben werden ($\rightsquigarrow$ u.U. Speicherüberlauf):

```
int f ()
{
    rational* p = new rational;
    p->n = 50;
    return p->n;   // Ooops, einziger Zeiger verloren
}
```

Problem: Es gibt zwei voneinander unabhängige Dinge, den Zeiger und die dynamische Variable. Beide müssen jedoch in konsistenter Weise verwendet werden. C++ stellt das nicht automatisch sicher!

Abhilfe:

- Nur verwenden, wenn unbedingt nötig (so bevorzugt man etwa Referenzen gegenüber Zeigern, wenn beide anwendbar sind)
- Manipulation der Variablen und Zeiger in Funktionen (später: Klassen) verpacken, die eine konsistente Behandlung sicherstellen.
- Benutzung spezieller Zeigerklassen (*smart pointers*).
- Die für den Programmierer angenehmste Möglichkeit ist die Verwendung von Garbage collection (=Sammeln von nicht mehr referenziertem Speicher).

11.8 Die einfach verkettete Liste

Zeiger und dynamische Speicherverwaltung benötigt man zur Erzeugung *dynamischer Datenstrukturen*.
Dies illustrieren wir am Beispiel der einfach verketteten Liste. Das komplette Programm befindet sich in der Datei `intlist.cc`.

Eine Liste natürlicher Zahlen

$$(12\ 43\ 456\ 7892\ 1\ 43\ 43\ 746)$$

zeichnet sich dadurch aus, dass

- die Reihenfolge der Elemente wesentlich ist, und
- Zahlen mehrfach vorkommen können.

Zur Verwaltung von Listen wollen wir folgende Operationen vorsehen

- Erzeugen einer leeren Liste.
- Einfügen von Elementen an beliebiger Stelle.
- Entfernen von Elementen.
- Durchsuchen der Liste.

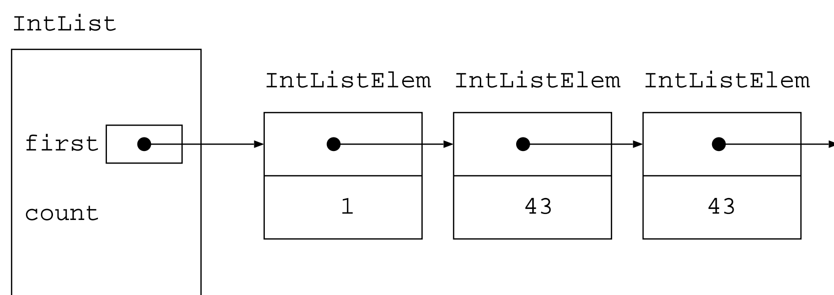
Bemerkung: Der Hauptvorteil gegenüber dem Feld ist, dass das Einfügen und Löschen von Elementen schneller geschehen kann (es ist eine O(1)-Operation, wenn die Stelle nicht gesucht werden muss).
Eine übliche Methode zur Speicherung von Listen (natürlicher Zahlen) besteht darin ein *Listenelement* zu definieren, das ein Element der Liste sowie einen Zeiger auf das nächste Listenelement enthält:

```
struct IntListElem {
    IntListElem* next;  // Zeiger auf nächstes Element
    int value;          // Daten zu diesem Element
} ;
```

Um die Liste als Ganzes ansprechen zu können definieren wir den folgenden zusammengesetzten Datentyp, der einen Zeiger auf das erste Element sowie die Anzahl der Elemente enthält:

```
struct IntList {
    int count;          // Anzahl Elemente in der Liste
    IntListElem* first; // Zeiger auf erstes Element der Liste
} ;
```

Das sieht also so aus:



Das Ende der Liste wird durch einen Zeiger mit dem Wert 0 gekennzeichnet.
Das klappt deswegen, weil 0 kein erlaubter Ort eines Listenelementes (irgendeiner Variable) ist.

Bemerkung: Die Bedeutung von 0 ist in C/C++ mehrfach überladen. In manchen Zusammenhängen bezeichnet es die Zahl 0, an anderen Stellen einen speziellen Zeiger. Auch der bool-Wert `false` ist synonym zu 0.

11.8.1 Initialisierung

Folgende Funktion initialisiert eine `IntList`-Struktur mit einer leeren Liste:

```
void empty_list (IntList* l)
{
    l->first = 0;   // Liste ist leer
    l->count = 0;
}
```

Bemerkung: Die Liste wird *call-by-reference* übergeben, um die Komponenten ändern zu können.

11.8.2 Durchsuchen

Hat man eine solche Listenstruktur, so gelingt das Durchsuchen der Liste mittels

```
IntListElem* find_first_x (IntList l, int x)
{
    for (IntListElem* p=l.first; p!=0; p=p->next)
        if (p->value==x) return p;
    return 0;
}
```

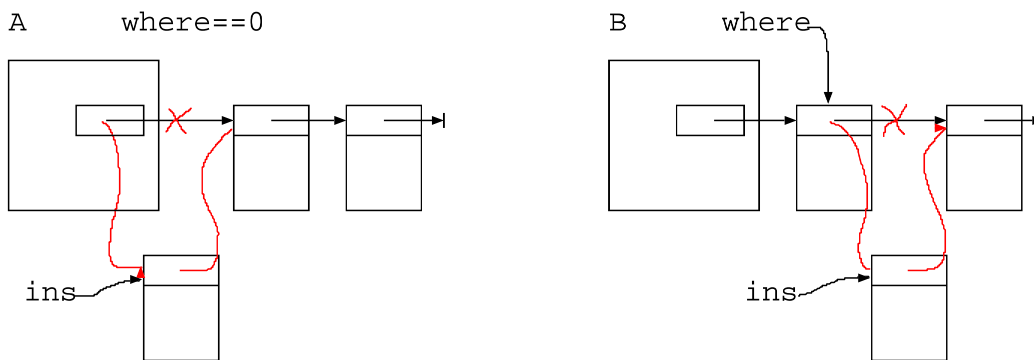
11.8.3 Einfügen

Beim Einfügen von Elementen unterscheiden wir zwei Fälle:

A Am Anfang der Liste einfügen.

B *Nach* einem gegebenem Element einfügen.

Für die beiden Fälle sind folgende Manipulationen der Zeiger erforderlich:



Programm: Folgende Funktion behandelt beide Fälle:

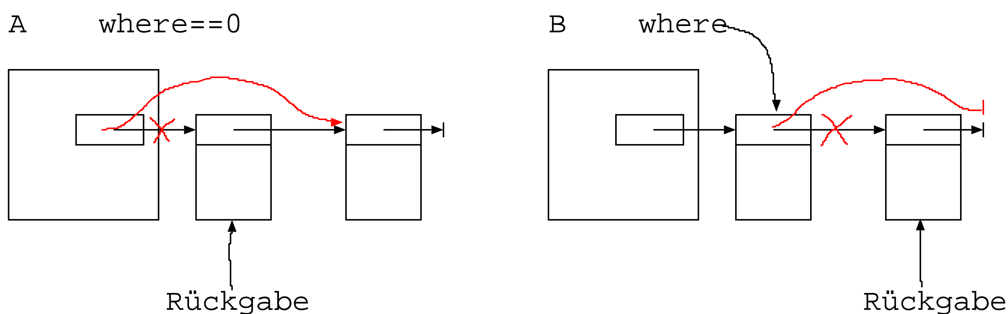
```
void insert_in_list (IntList* list, IntListElem* where,
                     IntListElem* ins)
{
    if (where==0)
    {   // fuege am Anfang ein
        ins->next = list->first;
        list->first = ins;
        list->count = list->count + 1;
    } else {
        // fuege nach where ein
        ins->next = where->next;
        where->next = ins;
        list->count = list->count + 1;
    }
}
```

11.8.4 Entfernen

Auch beim Entfernen von Elementen unterscheiden wir wieder ob

1. das erste Element gelöscht werden soll, oder
2. das Element *nach* einem gegebenem.

entsprechend graphisch:



Programm: Beide Fälle behandelt folgende Funktion:

```
IntListElem* remove_from_list (IntList* list, IntListElem* where)
{
    IntListElem* p;  // das entfernte Element

    // where==0 dann entferne erstes Element
    if (where==0) {
        p = list->first;
        if (p!=0)
        {
            list->first = p->next;
            list->count = list->count - 1;
        }
        return p;
    }

    // entferne Element nach where
    p = where->next;
    if (p!=0) {
        where->next = p->next;
        list->count = list->count - 1;
    }
    return p;
}
```

Bemerkung:

- Es wird angenommen, dass `where` ein Element der Liste ist. Wenn dies nicht erfüllt sein sollte, so ist Ärger garantiert!
- Alle Funktionen auf der Liste beinhalten *nicht* die Speicherverwaltung für die Objekte. Dies ist Aufgabe des benutzenden Programmteiles.

11.8.5 Kritik am Programmdesign

- Oft wird der Fall `where==0` verwendet werden, d.h. die Liste wird wie ein Stack verwendet. Dann sollte man aber auch die Schnittstelle anpassen, und die Befehle `push` und `pop` zur Verfügung stellen.
- Man wird auch Listen anderer Typen brauchen. Dies ist erst mit den später behandelten Werkzeugen wirklich befriedigend zu erreichen (Templates). Etwas mehr Flexibilität erhielte man aber über:

```
typedef int list_element_type;
```

und Verwendung dieses Datentyps später.

- Ist es sinnvoll, die Länge der Liste mitzuschleppen? Wenn nein, so sollte man darüber nachdenken, ob man nicht Listen generell als Zeiger auf Listenelemente definiert:

```
struct list_element {
    list_element* next;      // Zeiger auf nächstes
    list_element_type value; // Datum dieses Elements
} ;
typedef list_element* list;
```

Bemerkung: Eine alternative Listenimplementation in dieser Richtung kann man im Programm `nn_liste.cc` auf der Vorlesungshomepage finden.

11.8.6 Listenvarianten

- Bei der doppelt verketteten Liste ist auch der Vorgänger erreichbar und die Liste kann auch in umgekehrter Richtung durchlaufen werden.
- Listen, die auch ein (schnelles) Einfügen am Ende erlauben, sind zur Implementation von Warteschlangen (Queues) nützlich.
- Manchmal kann man *zirkuläre Listen* gebrauchen. Diese sind für simple Speicherverwaltungsmechanismen (reference counting) problematisch.
- In dynamisch typisierten Sprachen können Elemente beliebigen Typ haben (zum Beispiel wieder Listen), und die Liste wird zum Spezialfall einer Baumstruktur. Am elegantesten ist dieses Konzept wohl in der Sprache Lisp (=*List Processing*) verwirklicht.

11.9 Endliche Menge

Im Gegensatz zu einer Liste kommt es bei einer endlichen Menge

$$\{34\ 567\ 43\ 1\}$$

1. nicht auf die Reihenfolge der Mitglieder an und
2. können Elemente auch nicht doppelt vorkommen!

11.9.1 Schnittstelle

Als Operationen auf einer Menge benötigen wir

- Erzeugen einer leeren Menge.
- Einfügen eines Elementes.

- Entfernen eines Elementes.
- Mitgliedschaft in der Menge testen.

Wir wollen zur Realisierung der Menge die eben vorgestellte einfach verkettete Liste verwenden.

11.9.2 Datentyp und Initialisierung

Datentyp: (Menge von Integer-Zahlen)

```
struct IntSet {
    IntList list;
} ;
```

Man versteckt damit auch, dass `IntSet` mittels `IntList` realisiert ist.

Programm: (Leere Menge)

```
IntSet* empty_set () {
    IntSet* s = new IntSet;
    empty_list(&s->list);
    return s;
}
```

11.9.3 Test auf Mitgliedschaft

Programm:

```
bool is_in_set (IntSet* s, int x) {
    for (IntListElem* p=s->list.first; p!=0; p=p->next)
        if (p->value==x) return true;
    return false;
}
```

Bemerkung:

- Dies nennt man sequentielle Suche. Der Aufwand ist $O(n)$ wenn die Liste n Elemente hat.
- Später werden wir bessere Datenstrukturen kennenlernen, mit denen man das in $O(\log n)$ Aufwand schaffen (Baum).

11.9.4 Einfügen in eine Menge

Idee: Man testet, ob das Element bereits in der Menge ist, ansonsten wird es am Anfang der Liste eingefügt.

```
void insert_in_set (IntSet* s, int x)
{
    if (!is_in_set(s,x))
    {
        IntListElem* p = new IntListElem;
        p->value = x;
        insert_in_list(&s->list,0,p);
    }
}
```

Bemerkung: Man beachte, dass diese Funktion auch die `IntListElem`-Objekte dynamisch erzeugt.

11.9.5 Ausgabe

Programm: (Ausgabe der Menge)

```
void print_set (IntSet* s)
{
    cout << "{";
    for (IntListElem* p=s->list.first; p!=0; p=p->next)
        cout << " " << p->value;
    cout << " }" << endl;
}
```

Bemerkung: Kritik/Denkanstoß: eigentlich kann man diese Funktion auch für Listen gebrauchen. Vielleicht hätte man Mengen also besser doch als besondere Listen ansehen sollen?

11.9.6 Entfernen

Idee: Man sucht zuerst den Vorgänger des zu löschenden Elementes in der Liste und wendet dann die entsprechende Funktion für Listen an.

```
void remove_from_set (IntSet* s, int x)
{
    // Hat es ueberhaupt Elemente?
    if (s->list.first==0) return;

    // Teste erstes Element
```

```
    if (s->list.first->value==x) {
        IntListElem* p=remove_from_list(&s->list,0);
        delete p;
        return;
    }

    // Suche in Liste, teste immer Nachfolger
    // des aktuellen Elementes
    for (IntListElem* p=s->list.first; p->next!=0; p=p->next)
        if (p->next->value==x) {
            IntListElem* q=remove_from_list(&s->list,p);
            delete q;
            return;
        }
}
```

11.9.7 Vollständiges Programm

Programm: (useintset.cc)

```
#include<iostream.h>
#include"intlist.cc"
#include"intset.cc"

int main () {
    IntSet* s = empty_set();
    print_set(s);
    for (int i=1; i<12; i=i+1) insert_in_set(s,i);
    print_set(s);
    for (int i=2; i<30; i=i+2) remove_from_set(s,i);
    print_set(s);
}
```

12 Klassen

12.1 Motivation

Bisher:

- Funktionen bzw. Prozeduren (Funktion, bei welcher der Seiteneffekt wesentlich ist) als *aktive* Entitäten
- Daten als *passive* Entitäten.

Beispiel:

```
int konto1=100;
int konto2=200;
int abheben (int& konto, int betrag) {
    konto = konto - betrag;
    return konto;
}
```

Kritik:

- Auf welchen Daten operiert `abheben`? Es könnte mit jeder `int`-Variablen arbeiten.
- Wir könnten `konto1` auch ohne die Funktion `abheben` manipulieren.
- Nirgends ist der Zusammenhang zwischen den globalen Variablen `konto1`, `konto2` und der Funktion `abheben` erkennbar.

Idee: Verbinde Daten und Funktionen zu einer Einheit!

12.2 Klassendefinition

Diese Verbindung von Daten und Funktionen wird durch Klassen (*classes*) realisiert:

Beispiel: Klasse für das Konto:

```
class Konto {
public:
    int kontostand ();
    int abheben (int betrag);
private:
    int k;
} ;
```

Syntax: **(Klassendefinition)** Die allgemeine Syntax der Klassendefinition lautet

<Klasse> ::= class <Name> { <Rumpf> } ;

Im Rumpf werden sowohl Variablen als auch Funktionen aufgeführt. Bei den Funktionen genügt der Kopf. Die Funktionen einer Klasse heißen Methoden (*methods*). Alle Komponenten (Daten und Methoden) heißen Mitglieder.

Bemerkung:

- Die Klassendefinition
  - beschreibt, aus welchen Daten eine Klasse besteht,
  - und welche Operationen auf diesen Daten ausgeführt werden können.
- Klassen sind (in C++) keine normalen Datenobjekte. Sie sind nur zur Kompilierungszeit bekannt und belegen daher keinen Speicherplatz.

12.3 Objektdefinition

Die Klasse kann man sich als Bauplan vorstellen. Nach diesem Bauplan werden Objekte (*objects*) erstellt, die dann im Rechner existieren. Objekte heißen auch Instanzen (*instances*) einer Klasse.
Objektdefinitionen sehen aus wie Variablendefinitionen, wobei die Klasse wie ein neuer Datentyp erscheint. Methoden werden wie Komponenten eines zusammengesetzten Datentyps selektiert und mit Argumenten wie eine Funktion versehen.

Beispiel:

```
Konto k1;
Konto *pk=&k1;

k1.abheben(25);
cout << pk->kontostand() << endl;
```

Bemerkung: Objekte haben einen internen Zustand, der durch die Datenmitglieder repräsentiert wird. *Objekte haben ein Gedächtnis!*

12.4 Kapselung

Der Rumpf einer Klassendefinition zerfällt in zwei Teile:

1. einen *öffentlichen* Teil, und
2. einen privaten Teil.

Der öffentliche Teil einer Klasse ist die Schnittstelle (*interface*) der Klasse zum restlichen Programm. Diese sollte für den Benutzer der Klasse ausreichende Funktionalität bereitstellen. Der private Teil der Klasse enthält Mitglieder, die zur Implementierung der Schnittstelle benutzt werden.

Bezeichnung: Diese Trennung nennt man Kapselung (*encapsulation*).

Bemerkung:

- Sowohl öffentlicher als auch privater Teil können sowohl Methoden als auch Daten enthalten.
- Öffentliche Mitglieder einer Klasse können von jeder Funktion eines Programmes benutzt werden (etwa die Methode `abheben` in `Konto`).
- Private Mitglieder können nur von den Methoden der Klasse selbst benutzt werden.

Beispiel:

```
Konto k1;
k1.abheben(-25);       // OK
k1.k = 1000000;        // Fehler !, k private
```

Bemerkung: Kapselung erlaubt uns, das Prinzip der versteckten Information (*information hiding*) zu realisieren. David L. Parnas [CACM, 15(12): 1059-1062, 1972] hat dieses Grundprinzip im Zusammenhang mit der modularen Programmierung so ausgedrückt:

1. One must provide the intended user with all the information needed to use the module correctly, and with nothing more.
2. One must provide the implementor with all the information needed to complete the module, and with nothing more.

Bemerkung: Insbesondere sollte eine Klasse alle Implementierungsdetails „verstecken“, die sich möglicherweise in Zukunft ändern werden. Da Änderungen der Implementierung meist Änderung der Datenmitglieder bedeutet, sind diese normalerweise nicht öffentlich!

Zitat: *Brooks* [The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, 1975, page 102]:

> ...but much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies.

Regel: *Halte Datenstrukturen geheim!*

Bemerkung:

- Die „Geheimhaltung“ durch die Trennung `public`/`private` ist kein perfektes Verstecken der Implementation, weil der Benutzer der Klasse ja die Klassendefinition einsehen kann/muss.

- Sie erzwingt jedoch bei gutwilligen Benutzern ein regelkonformes Verwenden der Bibliothek.
- Andererseits schützt sie nicht gegenüber böswilligen Benutzern! (z.B. sollte man nicht erwarten, dass ein Benutzer der Bibliothek ein private-Feld `password` nicht auslesen kann!)

12.5 Konstruktoren und Destruktoren

Objekte werden – wie jede Variable – erzeugt und zerstört, sei es automatisch oder unter Programmiererkontrolle.
Diese Momente erfordern oft spezielle Beachtung, so dass *jede* Klasse die folgenden Operationen zur Verfügung stellt:

- Einen Konstruktor, der aufgerufen wird, nachdem der Speicher für ein Objekt bereitgestellt wurde. Der Konstruktor hat die Aufgabe, die Datenmitglieder des Objektes geeignet zu initialisieren.
- Einen Destruktor, der aufgerufen wird, bevor der vom Objekt belegte Speicher freigegeben wird. Der Destruktor kann entsprechende Aufräumarbeiten durchführen (Beispiele folgen).

Bemerkung:

- Ein Konstruktor ist eine Methode mit demselben Namen wie die Klasse selbst und kann mit beliebigen Argumenten definiert werden. Er hat *keinen* Rückgabewert.
- Ein Destruktor ist eine Methode, deren Name mit einer Tilde $\sim$ beginnt, gefolgt vom Namen der Klasse. Ein Destruktor hat weder Argumente noch einen Rückgabewert.
- Gibt der Programmierer keinen Konstruktor und/oder Destruktor an, so erzeugt der Übersetzer Default-Versionen. Der Default-Konstruktor hat keine Argumente.

Beispiel: Ein Beispiel für eine Klassendefinition mit Konstruktor und Destruktor:

```
class Konto {
public:
  Konto (int start);     // Konstruktor
  ~Konto ();             // Destruktor
  int kontostand ();
  int abheben (int betrag);
private:
  int k;
} ;
```

Der Konstruktor erhält ein Argument, welches das Startkapital des Kontos sein soll (Implementierung folgt gleich). Erzeugt wird so ein Konto mittels

```
Konto k1(1000);  // Argumente des Konstruktors nach Objektname
```

12.6 Implementierung der Klassenmethoden

Bisher haben wir noch nicht gezeigt, wie die Klassenmethoden implementiert werden. Dies ist Absicht, denn wir wollten deutlich machen, dass man nur die Definition einer Klasse *und die Semantik ihrer Methoden* wissen muss, um sie zu verwenden.
Nun wechseln wir auf die Seite des Implementierers einer Klasse. Hier nun ein vollständiges Programm mit Klassendefinition und Implementierung der Klasse `Konto`:

Programm: (Konto.cc)

```
#include<iostream>
using namespace std;

class Konto {
public:
    Konto (int start);      // Konstruktor
    ~Konto ();              // Destruktor
    int kontostand ();
    int abheben (int betrag);
private:
    int bilanz;
} ;

Konto::Konto (int startkapital)
{
    bilanz = startkapital;
    cout << "Konto mit " << bilanz << " eingerichtet" << endl;
}

Konto::~Konto ()
{
    cout << "Konto mit " << bilanz << " aufgelöst" << endl;
}

int Konto::kontostand () {
    return bilanz;
}

int Konto::abheben (int betrag)
{
    bilanz = bilanz - betrag;
    return bilanz;
}

int main ()
{
    Konto k1(100), k2(200);
```

```
    k1.abheben(50);
    k2.abheben(300);
}
```

Bemerkung:

- Die Definitionen der Klassenmethoden sind normale Funktionsdefinitionen, nur der Funktionsname lautet

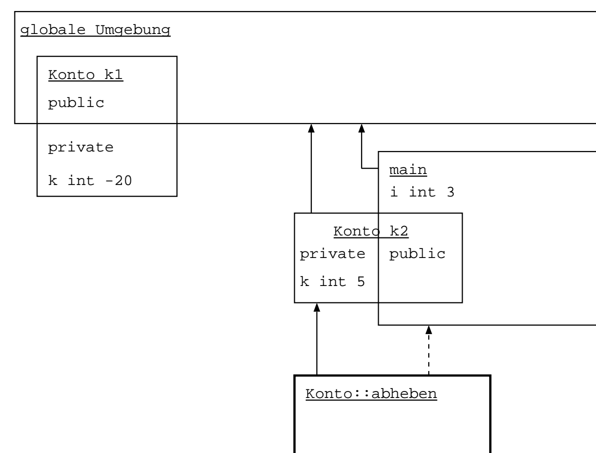
  <Klassenname>::<Methodenname>

- Klassen bilden einen eigenen *Namensraum*. So ist `abheben` keine global sichtbare Funktion. Der Name `abheben` ist nur innerhalb der Definition von `Konto` sichtbar.
- Außerhalb der Klasse ist der Name erreichbar, wenn ihm der Klassenname gefolgt von zwei Doppelpunkten (*scope resolution operator*) vorangestellt wird.

12.7 Klassen im Umgebungsmodell

```
class Konto; // wie oben

Konto k1(0);

void main ()
{
  int i=3;
  Konto k2(0);

  k1.abheben(20);
  k2.abheben(-5);
}
```

In `k2.abheben(-5)`



Bemerkung:

- Jedes Objekt definiert eine eigene Umgebung.
- Die öffentlichen Daten einer Objektumgebung überlappen mit der Umgebung, in der das Objekt definiert ist, und sind dort auch sichtbar.
- Der Methodenaufruf erzeugt eine neue Umgebung unterhalb der Umgebung des zugehörigen Objektes

Folgerung:

- Öffentliche Daten von `k1` sind global sichtbar.

- Öffentliche Daten von `k2` sind in `main` sichtbar.
- Private Daten von `k1` und `k2` sind von Methoden der Klasse `Konto` zugreifbar (jede Methode eines Objektes hat Zugriff auf die Mitglieder aller Objekte dieser Klasse, sofern bekannt).

Bemerkung: Die Lebensdauer von Objekten (bzw. Objektvariablen) ist genauso geregelt wie die von anderen Variablen.

12.8 Beispiel: Monte-Carlo objektorientiert

Wir betrachten nochmal das Beispiel der Bestimmung von π mit Hilfe von Zufallszahlen.

Bestandteile:

- Zufallsgenerator: Liefert bei Aufruf eine Zufallszahl.
- Experiment: Führt das Experiment einmal durch und liefert im Erfolgsfall 1, sonst 0.
- Monte-Carlo: Führt Experiment N mal durch und berechnet relative Häufigkeit.

12.8.1 Zufallsgenerator

Programm: Der Zufallsgenerator lässt sich hervorragend als Klasse formulieren. Er kapselt die aktuelle Zufallszahl als internen Zustand.

```
class Zufall {
public:
    Zufall (unsigned int anfang);
    unsigned int ziehe_zahl ();
private:
    unsigned int x;
} ;

Zufall::Zufall (unsigned int anfang) {
    x = anfang;
}

// Implementierung ohne lange Arithmetik
// siehe Numerical Recipes, Kap. 7.
unsigned int Zufall::ziehe_zahl ()
{
    // a = 7^5, m = 2^31-1
    unsigned int ia = 16807,  im = 2147483647;
    unsigned int iq = 127773, ir = 2836;
    unsigned int k = x/iq;
    x = ia*(x-k*iq)-ir*k;
    if (x<0) x = x+im;
    return x;
}
```

Vorteile:

- Durch die Angabe des Konstruktors ist sichergestellt, dass der Zufallsgenerator initialisiert werden muss. Beachte: Wenn ein Konstruktor angegeben ist, so gibt es keinen Default-Konstruktor!

- Die Realisierung des Zufallsgenerators ist nach außen nicht sichtbar (`x` ist private). Beispielsweise könnte man nun problemlos die Implementation so abändern, dass man intern mit längeren Zahlen arbeitet.

12.8.2 Klasse für das Experiment

Programm:

```
class Experiment {
public:
    Experiment (Zufall& z); // Konstruktor
    int durchfuehren ();    // einmal ausfuehren
private:
    Zufall& zg; // Merke Zufallsgenerator
} ;

Experiment::Experiment (Zufall& z) : zg(z) {}

unsigned int ggT (unsigned int a, unsigned int b)
{
    if (b==0) return a;
    else      return ggT(b,a%b);
}

int Experiment::durchfuehren ()
{
    unsigned int x1 = zg.ziehe_zahl();
    unsigned int x2 = zg.ziehe_zahl();
    if (ggT(x1,x2)==1)
        return 1;
    else
        return 0;
}
```

Bemerkung: Die Klasse `Experiment` enthält (eine Referenz auf) ein Objekt einer Klasse als Unterobjekt. Für diesen Fall gibt es eine spezielle Form des Konstruktors, die weiter unten erläutert wird.

12.8.3 Monte-Carlo-Funktion und Hauptprogramm

Programm:

```
#include <iostream>
using namespace std;
```

```
#include "Zufall.cc"
#include "Experiment.cc"

double montecarlo (Experiment& e, int N)
{
    int erfolgreich=0;

    for (int i=0; i<N; i=i+1)
        erfolgreich = erfolgreich+e.durchfuehren();

    return ((double)erfolgreich)/((double)N);
}

int main ()
{
    Zufall z(93267); // ein Zufallsgenerator
    Experiment e(z); // ein Experiment

    cout << sqrt(6.0/montecarlo(e,1000000)) << endl;
}
```

Diskussion:

- Es gibt keine globale Variable mehr! `Zufall` kapselt den Zustand intern.
- Wir könnten auch mehrere unabhängige Zufallsgeneratoren haben.
- Die Funktion `montecarlo` kann nun mit dem Experiment parametrisiert werden. Dadurch kann man das Experiment leicht austauschen: beispielsweise erhält man π auch, indem man Punkte in $(-1,1)^2$ würfelt und misst, wie oft sie im Einheitskreis landen.

12.9 Initialisierung von Unterobjekten

Ein Objekt kann Objekte anderer Klassen als Unterobjekte enthalten. Um in diesem Fall die ordnungsgemäße Initialisierung des Gesamtobjekts sicherzustellen, gibt es eine erweiterte Form des Konstruktors selbst.

Syntax:**(Erweiterter Konstruktor)** Ein Konstruktor für eine Klasse mit Unterobjekten hat folgende allgemeine Form:

<Konstruktor> ::= <Klassenname>::<Klassenname> (<ArgListe>) :
<UnterObjekt> (<ArgListe>)
{, <UnterObjekt> (<ArgListe>)}
{ <Rumpf> }

Die Aufrufe nach dem : sind Konstruktoraufrufe für die Unterobjekte. Deren Argumente sind Ausdrücke, die die formalen Parameter des Konstruktors des Gesamtobjektes enthalten können.

Eigenschaften:

- Bei der Ausführung jedes Konstruktors (egal ob einfacher, erweiterter oder default) werden *erst* die Konstruktoren der Unterobjekte ausgeführt und dann der Rumpf des Konstruktors.
- Wird der Konstruktoraufruf eines Unterobjektes im erweiterten Konstruktor weggelassen, so wird dessen argumentloser Konstruktor aufgerufen. Gibt es keinen solchen, wird ein Fehler gemeldet.
- Beim Destruktor wird erst der Rumpf abgearbeitet, dann werden die Destruktoren der Unterobjekte aufgerufen. Falls man keinen programmiert hat, wird die Default-Version verwendet.
- Dies nennt man hierarchische Konstruktion/Destruktion.

Erinnerung: Eingebaute Datentypen und Zeiger haben keine Konstruktoren und werden nicht initialisiert.

Anwendung: `Experiment` enthält eine Referenz als Unterobjekt. Mit einer Instanz der Klasse `Experiment` wird auch diese Referenz erzeugt. Referenzen müssen aber *immer* initialisiert werden, daher muss die erweiterte Form des Konstruktors benutzt werden.
Es ist in diesem Fall nicht möglich, die Referenz im Rumpf des Konstruktors zu initialisieren.

12.10 Selbstreferenz

Innerhalb jeder Methode einer Klasse `T` ist ein Zeiger `this` vom Typ `T*` definiert, der auf das Objekt zeigt, dessen Methode aufgerufen wurde.

Beispiel: Folgende Routine ist eine gleichwertige Definition von `abheben`:

```
int Konto::abheben (int betrag) {
    this->k = this->k - betrag;
    return this->k; // neuer kontostand
}
```

Bemerkung: Anders ausgedrückt, ist die alte Form von `abheben` syntaktischer Zucker für die Form mit `this`. Der Nutzen von `this` wird sich später zeigen (Verkettung von Operationen).

12.11 Überladen von Funktionen und Methoden

C++ erlaubt es, mehrere Funktionen *gleichen Namens* aber mit unterschiedlicher Signatur (Zahl und Typ der Argumente) zu definieren.

Beispiel:

```
int summe ()                  {return 0;}
int summe (int i)             {return i;}
int summe (int i, int j)      {return i+j;}
double summe (double a, double b) {return a+b;}

int main () {
    int i[2];
    double x[2];
    short c;

    i[1] = summe();          // erste Version
    i[1] = summe(3);         // zweite Version
    i[0] = summe(i[0],i[1]); // dritte Version
    x[0] = summe(x[0],x[1]); // vierte Version
    i[0] = summe(i[0],c);    // dritte Version
    i[0] = summe(x[0],i[1]); // Fehler, mehrdeutig
}
```

Dabei bestimmt der Übersetzer anhand der Zahl und Typen der Argumente, welche Funktion aufgerufen wird. Der Rückgabewert ist dabei unerheblich.

Bezeichnung: Diesen Mechanismus nennt man *Überladen* von Funktionen.

12.11.1 Automatische Konversion

Schwierigkeiten entstehen durch automatische Konversion eingebauter numerischer Typen. Der Übersetzer geht nämlich in folgenden Stufen vor:

1. Versuche passende Funktion ohne Konversion oder mit trivialen Konversionen (z. B. Feldname nach Zeiger) zu finden. Man spricht von exakter Übereinstimmung. Dies sind die ersten vier Versionen oben.

2. Versuche innerhalb einer Familie von Typen ohne Informationsverlust zu konvertieren und so eine passende Funktion zu finden. Z. B. ist erlaubt `bool` nach `int`, `short` nach `int`,`int` nach `long`, `float` nach `double`, etc. Im obigen Beispiel wird `c` in Version 5 nach `int` konvertiert.

3. Versuche Standardkonversionen (Informationsverlust!) anzuwenden: `int` nach `double`, `double` nach `int` usw.

4. Gibt es verschiedene Möglichkeiten auf *einer* der vorigen Stufen, so wird ein Fehler gemeldet.

Tip: Verwende Überladen möglichst nur so, dass die Argumente mit einer der definierten Signaturen exakt übereinstimmen!

12.11.2 Überladen von Methoden

Auch Methoden einer Klasse können überladen werden. Dies benutzt man gerne für den Konstruktor, um mehrere Möglichkeiten der Initialisierung eines Objektes zu ermöglichen:

```
class Konto {
  public:
    Konto ();           // Konstruktor 1
    Konto (int start); // Konstruktor 2
    int konto_stand ();
    int abheben (int betrag);
  private:
    int k;      // Zustand
} ;

Konto::Konto () { k = 0; }
Konto::Konto (int start) { k = start; }
```

Jetzt können wir ein Konto auf zwei Arten erzeugen:

```
  Konto k1;       // Hat Wert 0
  Konto k2(100); // Hundert Euro
```

Bemerkung:

- Eine Klasse muss einen Konstruktor ohne Argumente haben, wenn man Felder dieses Typs erzeugen will.
- Ein Default-Konstruktor wird nur erzeugt, wenn kein Konstruktor explizit programmiert wird.

Das Überladen von Funktionen ist eine Form von Polymorphismus womit man meint:

Eine Schnittstelle, viele Methoden.

Aber: Es ist sehr verwirrend, wenn überladene Funktionen sehr verschiedene Bedeutung haben. Dies sollte man vermeiden.

12.12 Objektorientierte und funktionale Programmierung

Folgendes Scheme-Programm ließ sich nur schlecht in C++ übertragen, weil die Erzeugung lokaler Funktionen nicht möglich war:

Programm: (lokal erzeugte Funktion in Scheme)

```
(define (inkrementierer n)
  (lambda (x)
    (+ x n)))

(map (inkrementierer 5)
     '(1 2 3))  => (6 7 8)
```

Mit den jetzt verfügbaren Klassen kann diese Funktionalität dagegen nachgebildet werden:

Programm: (Inkrementierer.cc)

```
#include <iostream>
using namespace std;

class Inkrementierer {
public:
    Inkrementierer (int n) {inkrement = n;}
    int eval (int n) {return n+inkrement;}
private:
    int inkrement;
};

void schleife (Inkrementierer &ink) {
    for (int i=1; i<10; i++)
        cout << ink.eval(i) << endl;
}

int main () {
    Inkrementierer ink(10);
    schleife (ink);
}
```

Bemerkung:

- Man beachte die Definition der Methoden innerhalb der Klasse. Dies ist zwar kürzer, legt aber die Implementation der Schnittstelle offen.
- Die innerhalb einer Klasse definierten Methoden werden „inline“ (d.h. ohne Funktionsaufruf) übersetzt. Bei Änderungen solcher Methoden muss daher aufrufender Code neu übersetzt werden!

- Man sollte dieses Feature daher nur mit Vorsicht verwenden (z.B. bei nur lokal verwendeten Klassen oder wenn das Inlining gewünscht wird).

- Eine erweiterte Schnittstelle zur Simulation funktionaler Programme erhält man in der STL (*Standard Template Library*) mit `#include <functional>`.

Bemerkung: Umgekehrt erhält man eine einfache Form von Objektorientiertheit, sobald man in funktionalen Sprachen Zuweisungen zulässt.

Programm: OOP-Simulation in Scheme durch „Message-passing“

```
(define (konto bilanz)
    (lambda (message)
      (case message
        ((kontostand) bilanz)
        ((abheben) (lambda (betrag)
                     (set! bilanz (- bilanz betrag)))))))
(define konto1 (konto 100))
(define konto2 (konto 200))
(konto1 'kontostand)   ; => 100
((konto1 'abheben) 20)
(konto1 'kontostand)   ; => 80
(konto2 'kontostand)   ; => 200
```

Bemerkung:

- (Sehr) vereinfacht gilt also „OOP=FP+Zuweisung“, wobei OOP = objektorientierte Programmierung, FP = funktionale Programmierung.

- Jedoch hat die Simulation von OOP durch Funktionen Nachteile. Dies sind insbesondere:

  - Effizienzproblem: Laufzeit-Abfragen sind zum Finden von Methoden (=Dispatch) nötig
  - Probleme bei der Introspektion (z.B. im Debugger): zu wenig Information ist vorhanden, was die Funktionen eigentlich darstellen.

  Aus diesem Grund sind separate OO-Sprachkonstrukte auch in funktionalen Sprachen üblich.

12.13 Operatoren

In C++ hat man auch bei selbstgeschriebenen Klassen die Möglichkeit einem Ausdruck wie a+b eine Bedeutung zu geben:

Idee: Interpretiere den Ausdruck a+b als a.operator+(b), d. h. die Methode operator+ des Objektes a (des linken Operanden) wird mit dem Argument b (rechter Operand) aufgerufen:

```
class X {
public:
  X operator+ (X b);
};
X X::operator+ (X b) { .... }
X a,b,c;
c = a+b;
```

Bemerkung:

- operator+ ist also ein ganz normaler Methodenname, nur die Methode wird aus der Infixschreibweise heraus aufgerufen.
- Diese Technik ist insbesondere bei Klassen sinnvoll, die mathematische Konzepte realisieren, wie etwa rationale Zahlen, Vektoren, Polynome, Matrizen, gemischtzahlige Artihmetik, Arithmetik beliebiger Genauigkeit, usw.
- Auch eckige Klammern [], Dereferenzierung ->, Vergleichsoperatoren <,>,== und sogar die Zuweisung = können (um-)definiert werden.
- Man sollte diese Technik zurückhaltend verwenden. Zum Beispiel sollte man + nur überladen, wenn die Operation wirklich eine Addition im mathematischen Sinn ist.

12.14 Anwendung: rationale Zahlen objektorientiert

Definition der Klasse (Rational.cc):

```
class Rational {
private:
    int n,d;
public:
    // (lesender) Zugriff auf Zaehler und Nenner
    int numerator ();
    int denominator ();

    // Konstruktoren
    Rational (int num, int denom); // rational
    Rational (int num);            // ganz
    Rational ();                   // Null
```

```
        // Ausgabe
        void print ();

        // Operatoren
        Rational operator+ (Rational q);
        Rational operator- (Rational q);
        Rational operator* (Rational q);
        Rational operator/ (Rational q);
  };
```

Programm: Implementierung der Methoden (`RationalImp.cc`):

```
int Rational::numerator () {
    return n;
}

int Rational::denominator () {
    return d;
}

void Rational::print () {
    cout << n << "/" << d << endl;
}

// ggT zum kuerzen
int ggT (int a, int b) {
    return (b==0) ? a : ggT(b,a%b);
}

// Konstruktoren
Rational::Rational (int num, int denom)
{
    int t = ggT(num,denom);
    if (t!=0)
    {
        n=num/t;
        d=denom/t;
    }
    else
    {
        n = num;
        d = denom;
    }
}

Rational::Rational (int num) {
    n=num;
```

```
    d=1;
}

Rational::Rational () {
    n=0;
    d=1;
}

// Operatoren
Rational Rational::operator+ (Rational q) {
    return Rational(n*q.denominator()+q.numerator()*d,
                    d*q.denominator());
}

Rational Rational::operator- (Rational q) {
    return Rational(n*q.d-q.n*d, d*q.d);
}

Rational Rational::operator* (Rational q) {
    return Rational(n*q.numerator(), d*q.denominator());
}

Rational Rational::operator/ (Rational q) {
    return Rational(n*q.denominator(),
                    d*q.numerator());
}
```

Programm: Lauffähiges Beispiel (`UseRational.cc`):

```
#include <iostream>
using namespace std;
#include "Rational.cc"
#include "RationalImp.cc"

int main () {
    Rational p(3,4), q(5,3), r;

    p.print(); q.print();
    r = (p+q*p)*p*p;
    r.print();

    return 0;
}
```

Bemerkung:

- Wie schon früher erwähnt, ist die Implementierung einer leistungsfähigen gemischtzahligen Arithmetik eine hochkomplexe Aufgabe, für welche die Klasse `Rational` nur ein

erster Ansatz sein kann.

- Sehr notwendig wäre auf jeden Fall die Verwendung von Ganzzahlen beliebiger Länge anstatt von `int` als Bausteine für `Rational`.

12.15 Beispiel: Turingmaschine

Ein großer Vorteil der objektorientierten Programmierung ist, dass man seine Programme sehr „problemnah" formulieren kann. Als Beispiel zeigen wir, wie man eine Turingmaschine realisieren könnte. Diese besteht aus den drei Komponenten

- Band
- Programm
- eigentliche Turingmaschine

Es bietet sich daher an, diese Einheiten als Klassen zu definieren.

12.15.1 Band

Programm: (`Band.h`)

```
class Band {
public:
    Band (char *name, char init);
    char lese ();
    void schreibe_links (char symbol);
    void schreibe_rechts (char symbol);
    void drucke ();
private:
    enum {N=100000};
    char band[N];
    int pos;
    int maxpos;
} ;
```

12.15.2 TM-Programm

Programm: (`Programm.h`)

```
// Eine Klasse, die das Programm einer Turingmaschine realisiert
// Zustaende sind vom Typ int
// Bandalphabet ist der Typ char

class Programm {
public:
```

```
    enum R {links, rechts};

    Programm (char *name);

    char Ausgabe (int zustand, char symbol);
    R Richtung (int zustand, char symbol);
    int Folgezustand (int zustand, char symbol);

    int Anfangszustand ();
    int Endzustand ();

    void drucke ();

private:
    bool FindeZeile(int zustand, char symbol);

    enum {N=1000};

    int zeilen;
    int Qaktuell[N];
    char eingabe[N];
    char ausgabe[N];
    R richtung[N];
    int Qfolge[N];
    int letztesQ;
    int letzteEingabe;
    int letzteZeile;
} ;
```

Bemerkung: Man beachte die Definition des lokalen Datentyps `R` durch `enum`. Andererseits wird eine Form von `enum`, bei der den Konstanten gleich Zahlen zugewiesen werden, verwendet, um die Konstante `N` innerhalb der Klasse `Programm` zur Verfügung zu stellen.

12.15.3 Turingmaschine

Programm: (`TM.h`)

```
// Klasse, die eine Turingmaschine realisiert

class TM {
public:
    TM (Programm& p, Band& b);
    void Schritt ();
    bool Endzustand ();
private:
    Programm& prog;
    Band& band;
```

```
    int q;
} ;
```

Programm: (TM.cc)

```
TM::TM (Programm& p, Band& b) : prog(p), band(b)
{
    q=p.Anfangszustand();
}

void TM::Schritt ()
{
    // lese Bandsymbol
    char s = band.lese();

    // schreibe Band
    if (prog.Richtung(q,s)==Programm::links)
        band.schreibe_links(prog.Ausgabe(q,s));
    else
        band.schreibe_rechts(prog.Ausgabe(q,s));

    // bestimme Folgezustand
    q = prog.Folgezustand(q,s);
}

bool TM::Endzustand ()
{
    if (q==prog.Endzustand()) return true; else return false;
}
```

12.15.4 Turingmaschinen-Hauptprogramm

Programm: (Turingmaschine.cc)

```
#include<iostream>
#include<fstream>
using namespace std;

#include"Band.h"
#include"Band.cc"
#include"Programm.h"
#include"Programm.cc"
#include"TM.h"
#include"TM.cc"
```

```cpp
int main (int argc, char *argv[])
{
    if (argc<3)                    // Auswerten der Kommandozeile
    {
        cout << "tm <Programmdatei> <Banddatei>" << endl;
        return 0;
    }

    Programm p(argv[1]);           // lese TM Programm ein
    p.drucke();                    // drucke Programm

    Band b(argv[2],'0');           // lese Band ein
    b.drucke();                    // drucke Band

    TM tm(p,b);                    // baue eine Maschine

    while (!tm.Endzustand())       // Solange nicht Endzustand
    {
        tm.Schritt() ;             // mache einen Schritt
        b.drucke();                // und drucke Band
    }

    return 0;                      // fertig.
}
```

Experiment: Programm zum Verdoppeln einer Einserkette (`TM.prog`):

```
1 1  X R 2
2 1  1 R 2
2 0  Y L 3
3 1  1 L 3
3 X  1 R 4
4 Y  1 R 8
4 1  X R 5
5 1  1 R 5
5 Y  Y R 6
6 1  1 R 6
6 0  1 L 7
7 1  1 L 7
7 Y  Y L 3
8
```

Kritik:

- Das Band könnte seine Größe dynamisch verändern.

- Statt eines einseitig unendlichen Bandes könnten wir auch ein zweiseitig unendliches Band realisieren.
- Das Finden einer Tabellenzeile könnte durch bessere Datenstrukturen beschleunigt werden.

Aber: Diese Änderungen betreffen jeweils nur die *Implementierung* einer einzelnen Klasse (`Band` oder `Programm`) und beeinflussen die Implementierung anderer Klassen nicht!

12.16 Abstrakter Datentyp

Eng verknüpft mit dem Begriff der Schnittstelle ist das Konzept des abstrakten Datentyps (ADT). Ein ADT besteht aus

- einer Menge von Objekten, und
- einem Satz von Operationen auf dieser Menge, sowie
- einer genauen Beschreibung der Semantik der Operationen.

Bemerkung:

- Das Konzept des ADT ist unabhängig von einer Programmiersprache, die Beschreibung kann in natürlicher (oder mathematischer) Sprache abgefasst werden.
- Der ADT beschreibt, *was* die Operationen tun, aber nicht, *wie* sie das tun. Die Realisierung ist also nicht Teil des ADT!
- Die Klasse ist der Mechanismus zur Konstruktion von abstrakten Datentypen in C++. Allerdings fehlt dort die Beschreibung der Semantik der Operationen! Diese kann man als Kommentar unter die Methoden schreiben.
- In manchen Sprachen (z.B. Eiffel, PLT Scheme) ist es möglich, die Semantik teilweise zu berücksichtigen (Design by Contract: zur Funktionsdefinition kann man Vorbedingungen und Nachbedingungen angeben).

12.16.1 Beispiel 1: Positive m-Bit-Zahlen im Computer

Der ADT „Positive m-Bit-Zahl" besteht aus

- Der Teilmenge $P_m = \{0, 1, \ldots, 2^m - 1\}$ der natürlichen Zahlen.
- Der Operation $+_m$ so dass für $a, b \in P_m$: $a +_m b = (a + b) \bmod 2^m$.
- Der Operation $-_m$ so dass für $a, b \in P_m$: $a -_m b = ((a - b) + 2^m) \bmod 2^m$.
- Der Operation $*_m$ so dass für $a, b \in P_m$: $a *_m b = (a * b) \bmod 2^m$.
- Der Operation $/_m$ so dass für $a, b \in P_m$: $a/_m b = q$, q die größte Zahl in P_m so dass $q *_m b \leq a$.

Bemerkung:

- Die Definition dieses ADT stützt sich auf die Mathematik (natürliche Zahlen und Operationen darauf).
- In C++ (auf einer 32-Bit Maschine) entsprechen `unsigned char`, `unsigned short`, `unsigned int` den Werten $m = 8, 16, 32$.

12.16.2 Beispiel 2: ADT Stack

- Ein Stack S über X besteht aus einer geordneten Folge von n Elementen aus X: $S = \{s_1, s_2, \ldots, s_n\}$, $s_i \in X$. Die Menge aller Stacks $\mathcal{S}$ besteht aus *allen möglichen Folgen der Länge* $n \geq 0$.
- Operation $new : \emptyset \to \mathcal{S}$, die einen leeren Stack erzeugt.
- Operation $empty : \mathcal{S} \to \{w, f\}$, die prüft ob der Stack leer ist.
- Operation $push : \mathcal{S} \times X \to \mathcal{S}$ zum Einfügen von Elementen.
- Operation $pop : \mathcal{S} \to \mathcal{S}$ zum Entfernen von Elementen.
- Operation $top : \mathcal{S} \to X$ zum Lesen des obersten Elementes.
- Die Operationen erfüllen folgende Regeln:
  1. $empty(new()) = w$
  2. $empty(push(S, x)) = f$
  3. $top(push(S, x)) = x$
  4. $pop(push(S, x)) = S$

Bemerkung:

- Die einzige Möglichkeit einen Stack zu erzeugen ist die Operation new.
- Die Regeln erlauben uns formal zu zeigen, welches Element nach einer beliebigen Folge von $push$ und pop Operationen zuoberst im Stack ist:

$$top(pop(push(push(push(new(), x_1), x_2), x_3))) =$$

$$top(push(push(new(), x_1), x_2)) = x_2$$

- Auch nicht gültige Folgen lassen sich erkennen:

$$pop(pop(push(new(), x_1))) = pop(new())$$

  und dafür gibt es keine Regel!

Bemerkung: Abstrakte Datentypen, wie Stack, die Elemente einer Menge X aufnehmen, heißen auch Container. Wir werden noch eine Reihe von Containern kennenlernen: Feld, Liste (in Varianten), Queue, usw.

12.16.3 Beispiel 3: Das Feld

Wie beim Stack wird das Feld über einer Grundmenge X erklärt. Auch das Feld ist ein Container.
Das charakteristische an einem Feld ist der indizierte Zugriff. Wir können das Feld daher als eine Abbildung einer Indexmenge $I \subset \mathbb{N}$ in die Grundmenge X auffassen.
Die *Indexmenge* $I \subseteq \mathbb{N}$ sei beliebig, aber im folgenden fest gewählt. Zur Abfrage der Indexmenge gebe es folgende Operationen:

- Operation min liefert kleinsten Index in I.
- Operation max liefert größten Index in I.
- Operation $isMember : \mathbb{N} \to \{w, f\}$. $isMember(i)$ liefert wahr falls $i \in I$, ansonsten falsch.

Den ADT Feld definieren wir folgendermaßen:

- Ein Feld f ist eine Abbildung der Indexmenge I in die Menge der möglichen Werte X, d. h. $f : I \to X$. Die Menge aller Felder $\mathcal{F}$ ist die Menge aller solcher Abbildungen.
- Operation $new : X \to \mathcal{F}$. $new(x)$ erzeugt neues Feld mit Indexmenge I (und initialisiert mit x, siehe unten).
- Operation $read : \mathcal{F} \times I \to X$ zum Auswerten der Abbildung.
- Operation $write : \mathcal{F} \times I \times X \to \mathcal{F}$ zum Manipulieren der Abbildung.
- Die Operationen erfüllen folgende Regeln:
    1. $read(new(x), i) = x$ für alle $i \in I$.
    2. $read(write(f, i, x), i) = x$.
    3. $read(write(f, i, x), j) = read(f, j)$ für $i \neq j$.

Bemerkung:

- In unserer Definition darf $I \subset \mathbb{N}$ beliebig gewählt werden. Es sind also auch nichtzusammenhängende Indexmengen erlaubt.
- Als Variante könnte man die Manipulation der Indexmenge erlauben (die Indexmenge sollte dann als weiterer ADT definiert werden).

13 Klassen und dynamische Speicherverwaltung

Erinnerung: Nachteile von eingebauten Feldern in C/C++:

- Ein eingebautes Feld kennt seine Größe nicht, diese muss immer extra mitgeführt werden, was ein Konsistenzproblem mit sich bringt.
- Bei dynamischen Feldern ist der Programmierer für die Freigabe des Speicherplatzes verantwortlich.
- Eingebaute Felder sind äquivalent zu Zeigern und können daher nur *by reference* übergeben werden.
- Eingebaute Felder prüfen nicht, ob der Index im erlaubten Bereich liegt.
- Manchmal bräuchte man Verallgemeinerungen, z.B. andere Indexmengen.

13.1 Klassendefinition

Unsere Feldklasse soll Elemente des Grundtyps `float` aufnehmen. Hier ist die Klassendefinition:

Programm: (SimpleFloatArray.cc)

```
class SimpleFloatArray {
public:
    SimpleFloatArray (int s, float f);
    // Erzeuge ein neues Feld mit s Elementen, I=[0,s-1]

    SimpleFloatArray (const SimpleFloatArray&);
    // Copy-Konstruktor

    SimpleFloatArray& operator= (const SimpleFloatArray&);
    // Zuweisung von Feldern

    ~SimpleFloatArray();
    // Destruktor: Gebe Speicher frei

    virtual float& operator[](int i);
    // Indizierter Zugriff auf Feldelemente
    // keine Ueberpruefung ob Index erlaubt

    int numIndices ();
    // Anzahl der Indizes in der Indexmenge

    int minIndex ();
    // kleinster Index
```

```
    int maxIndex ();
    // größter Index

    bool isMember (int i);
    // Ist der Index in der Indexmenge?

private:
    int n;        // Anzahl Elemente
    float *p;     // Zeiger auf built-in array
} ;
```

Bemerkung: Man beachte, dass diese Implementierung das eingebaute Feld nutzt.

13.2 Konstruktor

Programm: (SimpleFloatArrayImp.cc)

```
SimpleFloatArray::SimpleFloatArray (int s, float v)
{
    n = s;
//    try {
        p = new float[n];
//    }
//     catch (std::bad_alloc) {
//         cout << "nicht genug Speicher!" << endl;
//         return;
//         }
    for (int i=0; i<n; i=i+1) p[i]=v;
}

SimpleFloatArray::~SimpleFloatArray () { delete[] p; }

int SimpleFloatArray::numIndices () { return n; }

int SimpleFloatArray::minIndex () { return 0; }

int SimpleFloatArray::maxIndex () { return n-1; }

bool SimpleFloatArray::isMember (int i)
{
    return (i>=0 && i<n);
}
```

13.2.1 Ausnahmen

Bemerkung:

- Oben kann in der Operation `new` das Ereignis eintreten, dass nicht genug Speicher vorhanden ist. Dann setzt diese Operation eine sogenannte Ausnahme (*exception*), die in der `catch`-Anweisung abgefangen wird.
- *Gute* Fehlerbehandlung (Reaktionen auf Ausnahmen) ist in einem großen, professionellen Programm sehr wichtig!
- *Die Schwierigkeit ist, dass man oft an der Stelle des Erkennens des Ereignisses nicht weiss, wie man darauf reagieren soll.*
- Genau dies ist hier der Fall (und die lokale Reaktion ist eher schädlich) $\rightsquigarrow$ besser weglassen.

13.3 Indizierter Zugriff

Erinnerung: Die Operationen $read$ und $write$ des ADT Feld werden bei eingebauten Felder durch den Operator `[]` und die Zuweisung realisiert:

```
x = 3*a[i]+17.5;
a[i] = 3*x+17.5;
```

Unsere neue Klasse soll sich in dieser Beziehung wie ein eingebautes Feld verhalten. Dies gelingt durch die Definition eines Operators `operator[]`:

Programm:

```
float& SimpleFloatArray::operator[] (int i)
{
    return p[i];
}
```

Bemerkung:

- `a[i]` bedeutet, dass der `operator[]` von `a` mit dem Argument `i` aufgerufen wird.
- Der Rückgabewert von `operator[]` muss eine Referenz sein, damit `a[i]` auf der linken Seite der Zuweisung stehen kann. Wir wollen ja das i-te Element des Feldes verändern und keine Kopie davon.

13.4 Copy–Konstruktor

Schließlich ist zu klären, was beim Kopieren von Feldern passieren soll. Hier sind zwei Situationen zu unterscheiden:

1. Es wird ein *neues* Objekt erzeugt. Dies ist der Fall bei
   - Funktionsaufruf mit call by value: der aktuelle Parameter wird auf den formalen Parameter kopiert.
   - Objekt wird als Funktionswert zurückgegeben: Ein Objekt wird in eine temporäre Variable im Stack des Aufrufers kopiert.
   - Initialisierung von Objekten mit existierenden Objekten bei der Definition, also

     `SimpleFloatArray a=b;`
2. Kopieren eines Objektes auf ein bereits existierendes Objekt, das ist die Zuweisung.

Im ersten Fall wird von C++ der sogenannte Copy-Konstruktor aufgerufen. Ein Copy-Konstruktor ist ein Konstruktor der Gestalt

<Klassenname> (const <Klassenname> &);

Als Argument wird also eine Referenz auf ein Objekt desselben Typs übergeben. Dabei bedeutet `const`, dass das Argumentobjekt nicht manipuliert werden darf.

Programm: (SimpleFloatArrayCopyCons.cc)

```
SimpleFloatArray::SimpleFloatArray (const SimpleFloatArray& a) {
    n = a.n;
    p = new float[n];
    for (int i=0; i<n; i=i+1)
        p[i]=a.p[i];
}
```

Bemerkung:

- Unser Copy-Konstruktor allokiert ein neues Feld und kopiert alle Elemente des Argumentfeldes.
- Damit gibt es *immer nur jeweils einen Zeiger auf ein dynamisch erzeugtes, eingebautes Feld*. Der Destruktor kann dieses eingebaute Feld gefahrlos löschen!

Beispiel:

```
int f () {
  SimpleFloatArray a(100,0.0); // Feld mit 100 Elementen
  SimpleFloatArray b=a;        // Aufruf Copy-Konstruktor
  ... // mach etwas schlaues
} // Destruktor rufen delete[] ihres eingeb. Feldes auf
```

Bemerkung:

- Hier hat man mit der dynamischen Speicherverwaltung der eingebauten Felder nichts mehr zu tun, und es können auch keine Fehler passieren.
- Dieses Verhalten des Copy-Konstruktors nennt man *deep copy*.
- Alternativ könnte der Copy-Konstruktor nur den Zeiger in das neue Objekt kopieren (*shallow copy*). Hier dürfte der Destruktor das Feld aber nicht einfach freigeben, weil noch Referenzen bestehen könnten! (Abhilfen: *reference counting*, *garbage collection*)

13.5 Zuweisungsoperator

Bei einer Zuweisung a=b soll das Objekt rechts des =-Zeichens auf das *bereits initialisierte* Objekt links des =-Zeichens kopiert werden. In diesem Fall ruft C++ den `operator=` des links stehenden Objektes mit dem rechts stehenden Objekt als Argument auf.

Programm: (SimpleFloatArrayAssign.cc)

```
SimpleFloatArray& SimpleFloatArray::operator=
                    (const SimpleFloatArray& a)
{
    if (&a!=this) {// nur bei verschiedenen Objekten was tun
        if (n!=a.n) {
            // allokiere fuer this ein Feld der Groesse a.n
            delete[] p; // altes Feld loeschen
            n = a.n;
            p = new float[n];
        }
        for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }
    return *this; // Gebe Referenz zurueck damit a=b=c klappt
}
```

Bemerkung:

- Haben beide Felder unterschiedliche Größe, so wird für das Feld links vom Zuweisungszeichen ein neues eingebautes Feld der korrekten Größe erzeugt.
- Der Zuweisungsoperator ist in C/C++ so definiert, dass er gleichzeitig den zugewiesenen Wert hat. Somit werden Ausdrücke wie `a = b = 0` oder `return tabelle[i]=n` möglich.

13.6 Hauptprogramm

Programm: (UseSimpleFloatArray.cc)

```
#include<iostream>
using namespace std;
#include "SimpleFloatArray.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

void show (SimpleFloatArray f) {
    cout << "#( ";
    for (int i=f.minIndex(); i<=f.maxIndex(); i++)
        cout << f[i] << " ";
    cout << ")" << endl;
}

int main () {
    SimpleFloatArray a(10,0.0); // erzeuge Felder
    SimpleFloatArray b(5,5.0);

    for (int i=a.minIndex(); i<=a.maxIndex(); i++) a[i] = i;

    show(a); // call by value, ruft Copy-Konstruktor
    b = a;   // ruft operator= von b
    show(b);

    // hier wird der Destruktor beider Objekte gerufen
}
```

Bemerkung:

- Jeder Aufruf der Funktion `show` kopiert das Argument mittels des Copy-Konstruktors. (Für Demonstrationszwecke: eigentlich sollte man in `show` eine Referenz verwenden!)
- Entscheidend ist, dass der Benutzer gar nicht mehr mit dynamischer Speicherverwaltung konfrontiert wird.

13.7 Default-Methoden

Für folgende Methoden einer Klasse `T` erzeugt der Übersetzer automatisch Default-Methoden, sofern man keine eigenen definiert:

- Argumentloser Konstruktor `T ();`
  Dieser wird erzeugt, wenn man keinen anderen Konstruktor außer dem Copy-Konstruktor angibt.

- Copy-Konstruktor `T (const T&);`
  Kopiert alle Mitglieder in das neue Objekt (*memberwise copy*).
- Destruktor `~T ();`
- Zuweisungsoperator `T& operator= (const T&);`
  Kopiert alle Mitglieder des Quellobjektes auf das Zielobjekt.
- Adress-of-Operator (`&`) mit Standardbedeutung.

Bemerkung:

- Der Konstruktor (ob default oder selbstdefiniert) ruft rekursiv die Konstruktoren von selbstdefinierten Unterobjekten auf.
- Ebenso der Destruktor.
- Enthält ein Objekt Zeiger auf andere Objekte und ist für deren Speicherverwaltung verantwortlich, so wird man wahrscheinlich alle oben genannten Methoden speziell schreiben müssen (außer dem `&`-Operator). Die Klasse `SimpleFloatArray` illustriert dies.

14 Vererbung von Schnittstelle und Implementierung

14.1 Motivation: Polynome

Definition: Ein Polynom $p : \mathbb{R} \to \mathbb{R}$ ist eine Funktion der Form

$$p(x) = \sum_{i=0}^{n} p_i x^i,$$

Wir betrachten hier nur den Fall reellwertiger Koeffizienten $p_i \in \mathbb{R}$, und verlangen $p_n \neq 0$. n heißt dann Grad des Polynoms.

Operationen:

- Konstruktion.
- Manipulation der Koeffizienten.
- Auswerten des Polynoms an einer Stelle x.
- Addition zweier Polynome

$$p(x) = \sum_{i=0}^{n} p_i x^i, \quad q(x) = \sum_{j=0}^{m} q_j x^j$$

$$r(x) = p(x) + q(x) = \sum_{i=0}^{max(n,m)} \underbrace{(p_i^* + q_i^*)}_{r_i} x^i$$

$$p_i^* = \begin{cases} p_i & i \leq n \\ 0 & \text{sonst} \end{cases}, \quad q_i^* = \begin{cases} q_i & i \leq m \\ 0 & \text{sonst} \end{cases}.$$

- Multiplikation zweier Polynome

$$\begin{aligned} r(x) = p(x) * q(x) &= \left(\sum_{i=0}^{n} p_i x^i\right)\left(\sum_{j=0}^{m} q_j x^j\right) \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} p_i q_j x^{i+j} \\ &= \sum_{k=0}^{m+n} \underbrace{\left(\sum_{\{(i,j)|i+j=k\}} p_i q_j\right)}_{r_k} x^k \end{aligned}$$

14.2 Implementation

Für den Koeffizientenvektor $p_0, \ldots, p_n$ wäre offensichtlich ein Feld der adäquate Datentyp. Wir wollen unser Feld `SimpleFloatArray` benutzen. Eine Möglichkeit wäre folgender Zugang:

Programm:

```
class Polynomial {
private:
    SimpleFloatArray coefficients;
public:
   ...
}
```

Alternativ kann man Polynome als Exemplare von `SimpleFloatArray` mit zusätzlichen Eigenschaften ansehen, was im folgenden ausgeführt wird.

14.3 Öffentliche Vererbung

Programm: Definition der Klasse `Polynomial` mittels Vererbung:

```
class Polynomial :
    public SimpleFloatArray {
public:
    Polynomial (int n);
    // konstruiere Polynom vom Grad n

    // Default-Destruktor ist ok
    // Default-Copy-Konstruktor ist ok
    // Default-Zuweisung ist ok

    int degree ();
    // Grad des Polynoms

    float eval (float x);
    // Auswertung

    Polynomial operator+ (Polynomial q);
    // Addition von Polynomen

    Polynomial operator* (Polynomial q);
    // Multiplikation von Polynomen

    bool operator== (Polynomial q);
    // Gleichheit

    void print ();
```

```
    // drucke Polynom
} ;
```

Syntax: Die Syntax der *öffentlichen Vererbung* lautet:

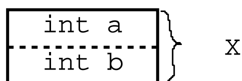
<ÖAbleitung> ::= class <Klasse2> : public <Klasse1> {
<Rumpf>
} ;

Bemerkung:

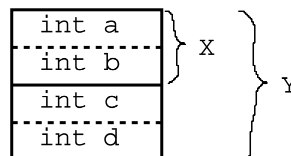
- Klasse 2 erhält ein Objekt von Klasse 2 als Unterobjekt.
- Alle *öffentlichen Mitglieder* der Klasse 1 (Basisklasse) mit Ausnahme von Konstruktoren, Destruktor und Zuweisungsoperatoren sind auch öffentliche Mitglieder der Klasse 2 (abgeleitete Klasse). Sie operieren auf dem Unterobjekt.
- Im Rumpf kann Klasse 2 weitere Mitglieder vereinbaren.
- Daher spricht man auch von einer Erweiterung einer Klasse durch *öffentliche Ableitung*.
- Alle privaten Mitglieder der Klasse 1 sind *keine* Mitglieder der Klasse 2. Damit haben auch Methoden der Klasse 2 *keinen* Zugriff auf private Mitglieder der Klasse 1.
- Eine Klasse kann mehrere Basisklassen haben (Mehrfachvererbung), diesen Fall behandeln wir hier aber nicht.

14.4 Beispiel zu public/private und öffentlicher Vererbung

```
class X {
public:
  int a;
  void A();
private:
  int b;
  void B();
} ;
```




```
class Y : public X {
public:
  int c;
  void C();
private:
  int d;
  void D();
} ;
```



```
X x;                          Y y;
                              y.a = 1; // OK
x.a = 5;    // OK             y.c = 2; // OK
x.b = 10;   // Fehler         y.b = 4; // Fehler
                              y.d = 8; // Fehler
void X::A ()
{                             void Y::C() {
  B();      // OK               d = 8; // OK
  b = 3;    // OK               b = 4; // Fehler
}                               A();   // OK
                                B();   // Fehler
                              }
```

14.5 Ist-ein-Beziehung

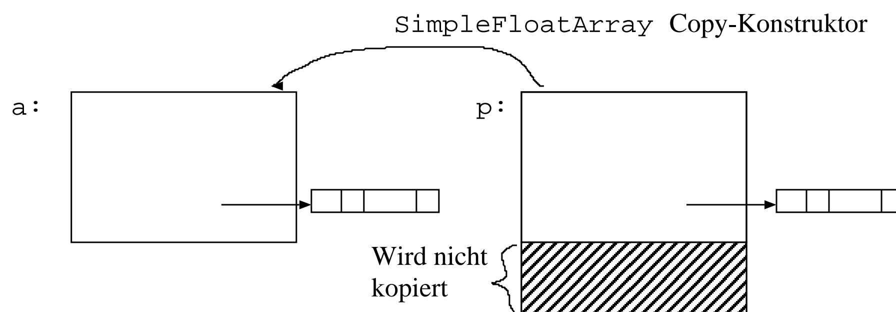
Ein Objekt einer abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt. Daher darf ein Objekt der abgeleiteten Klasse für ein Objekt der Basisklasse eingesetzt werden. Allerdings sind dann nur Methoden der Basisklasse für das Objekt aufrufbar.

Beispiel:

```
void g (SimpleFloatArray a) { a[3] = 1.0; }
Polynomial p(10);
SimpleFloatArray b(100,0.0);
g(p);  // (1) OK
p = b; // (2) Fehler
```

Bemerkung:

- Im Fall (1) wird bei Aufruf von `g(p)` der Copy-Konstruktor des formalen Parameters `a`, also `SimpleFloatArray`, benutzt, um das `SimpleFloatArray`-Unterobjekt von `p` auf den formalen Parameter `a` vom Typ `SimpleFloatArray` zu kopieren.
- Falls `Polynomial` weitere Datenmitglieder hätte, so würde die Situation so aussehen:



In diesem Fall spricht man von *slicing*.

- Im Fall (2) soll einem Objekt der abgeleiteten Klasse ein Objekt der Basisklasse zugewiesen werden. Dies ist nicht erlaubt, da nicht klar ist, welchen Wert etwaige zusätzliche Datenmitglieder der abgeleiteten Klasse bekommen sollen.

14.6 Konstruktoren, Destruktor und Zuweisungsoperatoren

Programm: (`PolynomialKons.cc`)

```
Polynomial::Polynomial (int n) : SimpleFloatArray(n+1,0.0)
{}
```

Bemerkung:

- Die syntaktische Form entspricht der Initialisierung von Unterobjekten wie oben beschrieben.
- Die Implementierung des Copy-Konstruktors kann man sich sparen, da der Default-Copy-Konstruktor das Gewünschte leistet, dasselbe gilt für Zuweisungsoperator und Destruktor.

14.7 Auswertung

Programm: Auswertung mit Horner-Schema (`PolynomialEval.cc`)

```
// Auswertung
float Polynomial::eval (float x)
{
    float sum=0.0;

    // Hornerschema
    for (int i=maxIndex(); i>=0; i=i-1)
        sum = sum*x + operator[](i);
    return sum;
}
```

Bemerkung: Statt `operator[]` könnte man `(*this)[i]` schreiben.

14.8 Weitere Methoden

Programm: (Grad, Addition, Multiplikation, Drucken)

```
// Grad auswerten
int Polynomial::degree ()
{
    return maxIndex();
```

```
}

// Addition von Polynomen
Polynomial Polynomial::operator+ (Polynomial q)
{
    int nr=degree(); // mein grad

    if (q.degree()>nr) nr=q.degree();

    Polynomial r(nr); // Ergebnispolynom

    for (int i=0; i<=nr; i=i+1)
    {
        if (i<=degree())
            r[i] = r[i]+(*this)[i];
        if (i<=q.degree())
            r[i] = r[i]+q[i];
    }

    return r;
}

// Multiplikation von Polynomen
Polynomial Polynomial::operator* (Polynomial q)
{
    Polynomial r(degree()+q.degree()); // Ergebnispolynom

    for (int i=0; i<=degree(); i=i+1)
        for (int j=0; j<=q.degree(); j=j+1)
            r[i+j] = r[i+j] + (*this)[i]*q[j];

    return r;
}

// Drucken
void Polynomial::print ()
{
    if (degree()<0)
        cout << 0;
    else
        cout << (*this)[0];

    for (int i=1; i<=maxIndex(); i=i+1)
        cout << "+" << (*this)[i] << "*x^" << i;

    cout << endl;
```

```
}
```

14.9 Gleichheit

Gleichheit ist kein einfaches Konzept, wie man ja schon an Zahlen sieht: ist `0==0.0`? Oder `(int) 1000000000 == (short) 1000000000`? Gleichheit für selbstdefinierte Datentypen ist daher Sache des Programmierers:

Programm: (PolynomialEqual.cc)

```
bool Polynomial::operator== (Polynomial q)
{
    if (q.degree()>degree())
    {
        for (int i=0; i<=degree(); i=i+1)
            if ((*this)[i]!=q[i]) return false;
        for (int i=degree()+1; i<=q.degree(); i=i+1)
            if (q[i]!=0.0) return false;
    }
    else
    {
        for (int i=0; i<=q.degree(); i=i+1)
            if ((*this)[i]!=q[i]) return false;
        for (int i=q.degree()+1; i<=degree(); i=i+1)
            if ((*this)[i]!=0.0) return false;
    }

    return true;
}
```

Bemerkung: Im Gegensatz dazu ist Gleichheit von Zeigern immer definiert. Zwei Zeiger sind gleich, wenn sie auf *dasselbe* Objekt zeigen:

```
Polynomial p(10), q(20);

Polynomial* z1 = &p;
Polynomial* z2 = &p;
Polynomial* z3 = &q;

if (z1==z2) ... // ist wahr
if (z1==z3) ... // ist falsch
```

14.10 Benutzung von `Polynomial`

Folgendes Beispiel definiert das Polynom

$$p = 1 + x$$

und druckt p, p^2 und p^3.

```
#include<iostream>
using namespace std;

// alles zum SimpleFloatArray
#include "SimpleFloatArray.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// Das Polynom
#include "Polynomial.cc"
#include "PolynomialImp.cc"
#include "PolynomialKons.cc"
#include "PolynomialEqual.cc"
#include "PolynomialEval.cc"

int main ()
{
    Polynomial p(2),q(10);

    p[0] = 1.0;
    p[1] = 1.0;
    p.print();

    q = p*p;
    q.print();

    q = p*p*p;
    q.print();
}
```

mit der Ausgabe:

```
1+1*x^1
1+2*x^1+1*x^2
1+3*x^1+3*x^2+1*x^3
```

14.11 Diskussion

- Diese Implementation hat die wesentliche Schwachstelle, dass der Grad bei führenden Nullen mathematisch nicht korrekt ist. Angenehmer wäre, wenn Konstanten den Grad

0 hätten, lineare Polynome den Grad 1 und der Koeffizientenvektor allgemein die Länge Grad+1 hätte. Das Nullpolynom hätte dann den Grad -1 (mathematisch noch besser wäre wohl $-\infty$, dies ist aber als `int` nicht darstellbar).

- Die Abhilfe besteht darin, dafür zu sorgen, dass der Konstruktor nur Polynome mit korrektem Grad erzeugt. Dieser Zugang ist im Programm `nn_polynomial.cc` auf der Vorlesungshomepage ausgeführt.
- In `nn_polynomial-2.cc` ist eine alternative Implementierung mit Koeffizientenvektor ausgeführt. Man sieht, dass sie in diesem Fall mindestens genausogut ist wie die Vererbung.

14.12 Private Vererbung

Wenn man nur die Implementierung von `SimpleFloatArray` nutzen will, ohne die Methoden öffentlich zu machen, so kann man dies durch private Vererbung erreichen:

Programm:

```
class Polynomial : private SimpleFloatArray {
  public:
    ...
}
```

Bemerkung: Die Konstruktion von Objekten der Klasse `Polynomial` müsste dann allerdings ohne [] geschehen, z.B. durch einen Konstruktor
`Polynomial::Polynomial (SimpleFloatArray &coeffs) { ... }`.

14.12.1 Eigenschaften der privaten Vererbung

Bemerkung: Private Vererbung bedeutet:

- Ein Objekt der abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.
- Alle *öffentlichen* Mitglieder der Basisklasse werden *private* Mitglieder der abgeleiteten Klasse.
- Alle privaten Mitglieder der Basisklasse sind keine Mitglieder der abgeleiteten Klasse.
- Ein Objekt der abgeleiteten Klasse kann *nicht* für ein Objekt der Basisklasse eingesetzt werden!

14.13 Zusammenfassung

Wir haben somit drei verschiedene Möglichkeiten kennengelernt, um die Klasse `SimpleFloatArray` für `Polynomial` zu nutzen:

1. Als öffentliches oder privates Datenmitglied
2. Mittels öffentlicher Vererbung
3. Mittels privater Vererbung

Bemerkung: Je nach Situation ist die eine oder andere Variante angemessener. Hier hängt viel vom guten Geschmack des Programmierers ab. In diesem speziellen Fall würde ich persönlich Möglichkeit 1 bevorzugen (ebenso für das im Bastian-Skript diskutierte Beispiel der Polynominterpolation).

15 Methodenauswahl und virtuelle Funktionen

15.1 Motivation: Feld mit Bereichsprüfung

Problem: Die für die Klasse `SimpleFloatArray` implementierte Methode `operator[]` prüft nicht, ob der Index im erlaubten Bereich liegt. Zumindest in der Entwicklungsphase eines Programmes wäre es aber nützlich, ein Feld mit Indexüberprüfung zu haben.

Abhilfe: Ableitung einer Klasse `CheckedSimpleFloatArray`, bei der sich `operator[]` anders verhält.

Programm: Klassendefinition (`CheckedSimpleFloatArray.cc`):

```
class CheckedSimpleFloatArray :
    public SimpleFloatArray {
public:
    CheckedSimpleFloatArray (int s, float f);
    CheckedSimpleFloatArray (const CheckedSimpleFloatArray&);
    CheckedSimpleFloatArray& operator= (const
        CheckedSimpleFloatArray&);
    // Default-Destruktor ist OK

    float& operator[](int i);
    // Indizierter Zugriff mit Indexprüfung
} ;
```

Methodendefinition (`CheckedSimpleFloatArrayImp.cc`):

```
CheckedSimpleFloatArray::CheckedSimpleFloatArray (int s, float f)
    : SimpleFloatArray (s,f)
{}

CheckedSimpleFloatArray::CheckedSimpleFloatArray
  (const CheckedSimpleFloatArray& a) : SimpleFloatArray(a)
{}

CheckedSimpleFloatArray& CheckedSimpleFloatArray::operator=
  (const CheckedSimpleFloatArray& rhs) {
    SimpleFloatArray::operator=(rhs);
    return *this;
}

float& CheckedSimpleFloatArray::operator[] (int i)
{
    assert (i>=minIndex() && i<=maxIndex());
    return SimpleFloatArray::operator[](i);
}
```

Verwendung (`UseCheckedSimpleFloatArray.cc`):

```
#include<iostream>
#include<cassert>
using namespace std;

#include "SimpleFloatArray.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

#include "CheckedSimpleFloatArray.cc"
#include "CheckedSimpleFloatArrayImp.cc"

void g (SimpleFloatArray& a) {
    cout << a[1] << "  " << a[11] << endl;
}

int main () {
    CheckedSimpleFloatArray a(10,0);
    g(a);
    cout << a[11] << endl;
}
```

Bemerkung:

- Leider gibt es (in C++) keine Vererbung von Konstruktoren, Destruktor und Zuweisungsoperator, und die Default-Varianten haben nicht das korrekte Verhalten. Daher müssen diese explizit angegeben werden.
- In der Funktion `main` funktioniert die Bereichsprüfung dann wie erwartet.
- In der Funktion `g` wird hingegen keine Bereichsprüfung durchgeführt, *auch wenn sie mit einem Objekt vom Typ `CheckedSimpleFloatArray` aufgerufen wird*!
- Der Grund ist, dass zur Übersetzungszeit von `g` nicht bekannt ist, dass sie mit einer Referenz auf ein Objekt einer öffentlich abgeleiteten Klasse aufgerufen wird.
- Meistens ist dies aber *nicht das gewünschte Verhalten* (vgl. dazu auch die späteren Beispiele).

15.2 Virtuelle Funktionen

Idee: Gib dem Compiler genügend Information, so dass er schon bei der Übersetzung von `SimpleFloatArray`-Methoden ein flexibles Verhalten von `[]` möglich macht. In C++ geschieht dies, indem man Methoden *in der Basisklasse* als virtuell (*virtual*) kennzeichnet.

Programm:

```
class SimpleFloatArray {
public:
    ...
    virtual float& operator[](int i);
    ...
private:
    ...
} ;
```

Beobachtung: Mit dieser Änderung funktioniert die Bereichsprüfung auch in der Funktion g in `UseCheckedSimpleFloatArray.cc`: wird sie mit einer Referenz auf ein `CheckedSimpleFloatArray`-Objekt aufgerufen, so wird der Bereichstest durchgeführt, bei Aufruf mit einer Referenz auf ein `SimpleFloatArray`-Objekt aber nicht.

Bemerkung:

- Die Einführung einer virtuellen Funktion erfordert also Änderungen in bereits existierendem Code, nämlich der Definition der Basisklasse!
- Die Implementierung der Methoden bleibt jedoch unverändert.

Implementation: Diese Auswahl der Methode in Abhängigkeit vom tatsächlichen Typ des Objekts kann man dadurch erreichen, dass jedes Objekt entweder Typinformation oder einen Zeiger auf eine Tabelle mit den für seine Klasse virtuell definierten Funktionen mitführt.

Bemerkung:

- Wird eine als virtuell markierte Methode in einer abgeleiteten Klasse neu implementiert, so wird die Methode der *abgeleiteten Klasse* verwendet, wenn das Objekt für ein Basisklassenobjekt eingesetzt wird.
- Die Definition der Methode in der abgeleiteten Klasse muss genau mit der Definition in der Basisklasse übereinstimmen, ansonsten wird *überladen*!
- Das Schlüsselwort `virtual` muss in der abgeleiteten Klasse nicht wiederholt werden, es ist aber guter Stil dies zu tun.
- Die Eigenschaften virtueller Funktionen lassen sich nur nutzen, wenn auf das Objekt über Referenzen oder Zeiger zugegriffen wird! Bei einem Aufruf (*call-by-value*) von

  ```
  void f (SimpleFloatArray a)
  {
      cout << a[1] << "  " << a[11] << endl;
  }
  ```

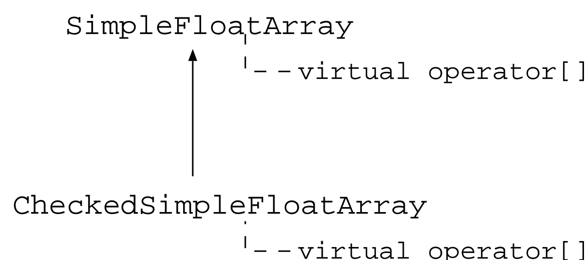
  erzeugt der Copy-Konstruktor ein Objekt `a` vom Typ `SimpleFloatArray` (Slicing!) und innerhalb von `f()` wird entsprechend dessen `operator[]` verwendet.

- Virtuelle Funktionen stellen wieder eine Form des Polymorphismus dar („eine Schnittstelle — viele Methoden").
- Der Zugriff auf eine Methode über die Tabelle virtueller Funktionen ist deutlich ineffizienter, was für Objektorientierung auf niedriger Ebene eine Rolle spielen kann.
- In vielen Sprachen (z.B. Smalltalk, Objective C, Common Lisp/CLOS) verhalten sich alle Methoden „virtuell".
- In der Programmiersprache Java ist das virtuelle Verhalten der Normalfall, das Default-Verhalten von C++-Methoden kann man aber durch Hinzufügen des Schlüsselworts `static` erreichen.

16 Abstrakte Klassen

16.1 Motivation

Hatten:



Beobachtung: Beide Klassen besitzen dieselben Methoden und unterscheiden sich nur in der Implementierung von `operator[]`. Wir könnten ebenso `SimpleFloatArray` von `CheckedSimpleFloatArray` ableiten. Das Klassendiagramm drückt diese Symmetrie aber nicht aus.

Grund: `SimpleFloatArray` stellt sowohl die Definition der Schnittstelle eines ADT *Feld* dar, als auch eine Implementierung dieser Schnittstelle. Es ist aber sinnvoll, diese beiden Aspekte zu trennen.

16.2 Schnittstellenbasisklassen

Idee: Definiere eine möglichst allgemeine Klasse `FloatArray`, von der sowohl `SimpleFloatArray` als auch `CheckedSimpleFloatArray` abgeleitet werden.

Bemerkung: Oft will und kann man für (virtuelle) Methoden in einer solchen Basisklasse keine Implementierung angeben. In C++ kennzeichnet man sie dann mit dem Zusatz `= 0` am Ende. Solche Funktionen bezeichnet man als rein virtuelle (*engl.: pure virtual*) Funktionen.

Beispiel:

```
class FloatArray {
public:
    virtual ~FloatArray() {};
    virtual float& operator[](int i) = 0;
    virtual int numIndices () = 0;
    virtual int minIndex () = 0;
    virtual int maxIndex () = 0;
    virtual bool isMember (int i) = 0;
} ;
```

Bezeichnung: Klassen, die mindestens eine rein virtuelle Funktion enthalten, nennt man abstrakt. Das Gegenteil ist eine konkrete Klasse.

Bemerkung:

- Man kann keine Objekte von abstrakten Klassen instanzieren. Aus diesem Grund haben abstrakte Klassen auch *keine Konstruktoren*.
- Sehr wohl kann man aber Zeiger und Referenzen dieses Typs haben, die dann aber auf Objekte abgeleiteter Klassen zeigen.
- Eine abstrakte Klasse, die der Definition einer Schnittstelle dient, bezeichnen wir nach Barton/Nackman als Schnittstellenbasisklasse (*interface base class*).
- Schnittstellenbasisklassen enthalten üblicherweise keine Datenmitglieder und die Methoden sind rein virtuell.
- Die Implementierung dieser Schnittstelle erfolgt in abgeleiteten Klassen.

Bemerkung: (Virtueller Destruktor) Eine Schnittstellenbasisklasse sollte einen *virtuellen* Destruktor

```
virtual ~FloatArray();
```

mit einer Dummy-Implementierung

```
FloatArray::~FloatArray () {}
```

besitzen, damit man dynamisch erzeugte Objekte abgeleiter Klassen durch die Schnittstelle der Basisklasse löschen kann. Beispiel:

```
void g (FloatArray* p)
{
    delete p;
}
```

Bemerkung: Der Destruktor darf nicht rein virtuell sein, da der Destruktor abgeleiteter Klassen einen Destruktor der Basisklasse aufrufen will.

16.3 Beispiel: geometrische Formen

Aufgabe: Wir wollen mit zweidimensionalen geometrischen Formen arbeiten. Dies sind von einer Kurve umschlossene Flächen wie Kreis, Rechteck, Dreieck, ...

Programm: Eine mögliche C++-Implementierung wäre folgende:

```
#include<iostream>
using namespace std;

const double pi = 3.1415926536;

class Shape
{
public:
    virtual ~Shape () {};
    virtual double area () = 0;
    virtual double diameter () = 0;
    virtual double circumference () = 0;
};

// works on every shape
double circumference_to_area (Shape &shape) {
    return shape.circumference()/shape.area();
}

class Circle : public Shape {
public:
    Circle (double r) {radius = r;}
    virtual double area () {return pi*radius*radius;}
    virtual double diameter () {return 2*radius;}
    virtual double circumference () {return 2*pi*radius;}
private:
    double radius;
};

class Rectangle : public Shape {
public:
    Rectangle (double aa, double bb) {a = aa; b = bb;}
    virtual double area () {return a*b;}
    virtual double diameter () {return sqrt(a*a+b*b);}
    virtual double circumference () {return 2*(a+b);}
private:
    double a, b;
};

int main ()
{
```

```
    Rectangle unit_square(1.0, 1.0);
    Circle unit_circle(1.0);
    Circle unit_area_circle(1.0/sqrt(pi));

    cout << "Das Verhältnis von Umfang zu Fläche beträgt\n";
    cout << "Einheitsquadrat:    "
         << circumference_to_area(unit_square) << endl;
    cout << "Kreis mit Fläche 1: "
         << circumference_to_area(unit_area_circle) << endl;
    cout << "Einheitskreis:      "
         << circumference_to_area(unit_circle) << endl;

    return 0;
}
```

Ergebnis: Wir erhalten als Ausgabe des Programms:

```
Das Verhältnis von Umfang zu Fläche beträgt
Einheitsquadrat:    4
Kreis mit Fläche 1: 3.54491
Einheitskreis:      2
```

16.4 Beispiel: Nochmals funktionales Programmieren

Hatten: Definition einer Inkrementierer-Klasse in `Inkrementierer.cc`. Nachteile waren:

- Andere Funktionen erfordern andere Klassen.
- Syntax `ink.eval(...)` nicht optimal.

Dies wollen wir nun mit Hilfe einer Schnittstellenbasisklasse und der Verwendung von `operator()` verbessern.

Programm:

```
#include <iostream>
using namespace std;

class Function {
public:
    virtual ~Function () {};
    virtual int operator() (int) = 0;
};

class Inkrementierer : public Function {
public:
    Inkrementierer (int n) {inkrement = n;}
    int operator() (int n) {return n+inkrement;}
private:
    int inkrement;
};

void schleife (Function &func) {
    for (int i=1; i<10; i++)
        cout << func(i) << " ";
    cout << endl;
}

class Quadrat : public Function {
public:
    int operator() (int n) {return n*n;}
} ;

int main () {
    Inkrementierer ink(10);
    Quadrat quadrat;
    schleife (ink);
    schleife (quadrat);
}
```

Bemerkung: Unangenehm ist jetzt eigentlich nur noch, dass der Typ der Funktion auf `int name(int)` festgelegt ist. Dies wird bald durch Schablonen (*Templates*) behoben werden.

16.5 Beispiel: Exotische Felder

Programm: Wir definieren folgende Schnittstellenbasisklasse:

```
class FloatArray {
public:
    virtual ~FloatArray() {};
    virtual float& operator[](int i) = 0;
    virtual int numIndices () = 0;
    virtual int minIndex () = 0;
    virtual int maxIndex () = 0;
    virtual bool isMember (int i) = 0;
} ;
```

Von dieser kann man leicht `SimpleFloatArray` ableiten. Außerdem passt die Schnittstelle auf weitere Feldtypen, was wir im folgenden zeigen wollen.

16.5.1 Dynamisches Feld

Wir wollen jetzt ein Feld mit variabel großer, aber zusammenhängender Indexmenge $I = \{o, o+1, \ldots, o+n-1\}$ mit $o, n \in \mathbb{Z}$ und $n \geq 0$ definieren. Wir gehen dazu folgendermaßen vor:

- Der Konstruktor fängt mit einem Feld der Länge 0 an ($o = n = 0$).
- `operator[]` prüft, ob $i \in I$ gilt; wenn nein, so wird der Indexbereich erweitert, ein entsprechendes Feld allokiert, die Werte aus dem alten Feld werden in das neue kopiert und das alte danach freigegeben.

Programm:

```
class DynamicFloatArray : public FloatArray {
public:
    DynamicFloatArray () {n=0; o=0; p=new float[0];}

    virtual ~DynamicFloatArray() {delete[] p;}
    virtual float& operator[](int i);
    virtual int numIndices ()    {return n;}
    int minIndex ()              {return o;}
    int maxIndex ()              {return o+n-1;}
    bool isMember (int i)        {return (i>=o) && (i<o+n);}
private:
    int n;       // Anzahl Elemente
    int o;       // Ursprung der Indexmenge
    float *p;    // Zeiger auf built-in array
```

```
};

float& DynamicFloatArray::operator[] (int i)
{
    if (i<o || i>=o+n)
    {   // resize
        int new_o, new_n;
        if (i<o) {
            new_o = i;
            new_n = n+o-i;
        }
        else {
            new_o = o;
            new_n = i-o+1;
        }
        float *q = new float[new_n];
        for (int i=0; i<new_n; i=i+1) q[i]=0.0;
        for (int i=0; i<n; i=i+1)
            q[i+o-new_o] = p[i];
        delete[] p;
        p = q;
        n = new_n;
        o = new_o;
    }
    return p[i-o];
}
```

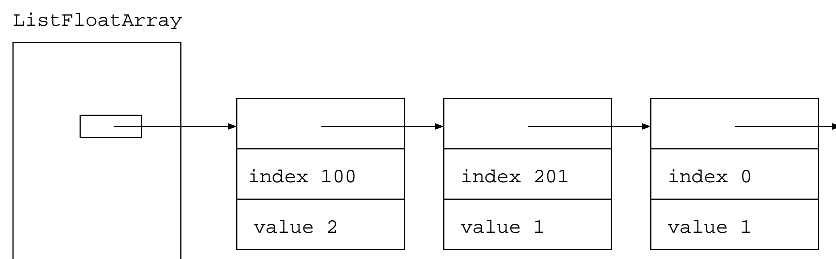
16.5.2 Listenbasiertes Feld

Problem: Wenn man `DynamicFloatArray` zur Darstellung von Polynome verwendet, so werden Polynome mit vielen Nullkoeffizienten, z.B.

$$p(x) = x^{100} + 1 \text{ oder } q(x) = p^2(x) = x^{200} + 2x^{100} + 1$$

sehr ineffizient verwaltet.

Abhilfe: Speichere die Elemente des Feldes als einfach verkettete *Liste* von Index–Wert–Paaren:



Programm:

```
class ListFloatArray :
    public FloatArray {      // Ableitung
public:
    ListFloatArray ();       // leeres Feld

    ~ListFloatArray();       // ersetzt ~FloatArray !!

    virtual float& operator[](int i);
    virtual int numIndices ();
    virtual int minIndex ();
    virtual int maxIndex ();
    virtual bool isMember (int i);
private:
    struct FloatListElem { // lokal benutzte Datenstruktur
        struct FloatListElem *next; // naechstes Element
        int index;                  // der Index
        float value;                // der Wert
    };

    int n;                 // Anzahl Elemente
    FloatListElem *p;      // einfach verkettete Liste

    FloatListElem* insert (int i, float v); // einfuegen
    FloatListElem* find (int i);  // finde Index
} ;

// private Hilfsfunktionen
ListFloatArray::FloatListElem*
ListFloatArray::insert (int i, float v)
{
    FloatListElem* q = new FloatListElem;

    q->index = i;
    q->value = v;
    q->next = p;
    p = q;
    n = n+1;
    return q;
}

ListFloatArray::FloatListElem* ListFloatArray::find (int i)
{
    for (FloatListElem* q=p; q!=0; q = q->next)
        if (q->index==i)
            return q;
```

```
    return 0;
}

// Konstruktoren
ListFloatArray::ListFloatArray ()
{
    n = 0; // alles leer
    p = 0;
}

// Destruktor
ListFloatArray::~ListFloatArray ()
{
    FloatListElem* q;

    while (p!=0)
    {
        q = p;        // q ist erstes
        p = q->next; // entferne q aus Liste
        delete q;
    }
}

float& ListFloatArray::operator[] (int i)
{
    FloatListElem* r=find(i);
    if (r==0)
        r=insert(i,0.0); // erzeuge index, r nicht mehr 0
    return r->value;
}

int ListFloatArray::numIndices ()
{
    return n;
}

int ListFloatArray::minIndex ()
{
    if (p==0) return 0;
    int min=p->index;
    for (FloatListElem* q=p->next; q!=0; q = q->next)
        if (q->index<min) min=q->index;
    return min;
}

int ListFloatArray::maxIndex ()
```

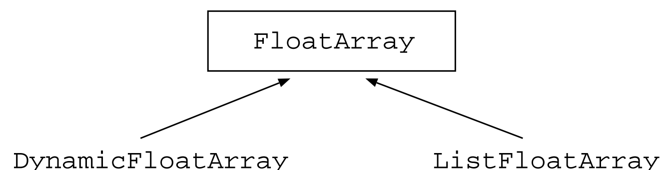
```
{
    if (p==0) return 0;
    int max=p->index;
    for (FloatListElem* q=p->next; q!=0; q = q->next)
        if (q->index>max) max=q->index;
    return max;
}

bool ListFloatArray::isMember (int i)
{
    return (find(i)!=0);
}
```

Bemerkung:

- Das Programm verwendet eine unsortierte Liste. Man hätte auch eine sortierte Liste verwenden können (Vorteile?).
- Für die Index–Wert–Paare wird innerhalb der Klassendefinition der zusammengesetzte Datentyp `FloatListElem` definiert.
- Die privaten Methoden dienen der Manipulation der Liste und werden in der Implementierung der öffentlichen Methoden verwendet.

16.5.3 Anwendung

Wir haben somit folgendes Klassendiagramm:



Da sowohl `DynamicFloatArray` als auch `ListFloatArray` die durch `FloatArray` definierte Schnittstelle erfüllen, kann man nun Methoden für `FloatArray` schreiben, die auf beiden abgeleiteten Klassen funktionieren.
Als Beispiel betrachten wir folgendes Programm, welches `FloatArray` wieder zur Polynom-Multiplikation verwendet (der Einfachheit halber ohne es in eine Klasse `Polynomial` zu packen).

Programm:

```
#include <iostream>
using namespace std;

#include "FloatArray.cc"
#include "DFA.cc"
```

```
#include "LFA.cc"

void polyshow (FloatArray& f) {
    for (int i=f.minIndex(); i<=f.maxIndex(); i=i+1)
        if (f.isMember(i)  && f[i]!=0.0)
            cout << "+" << f[i] << "*x^" << i;
    cout << endl;
}

void polymul (FloatArray& a, FloatArray& b, FloatArray& c) {
    // Loesche a
    for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1)
        if (a.isMember(i))
            a[i] = 0.0;

    // a = b*c
    for (int i=b.minIndex(); i<=b.maxIndex(); i=i+1)
        if (b.isMember(i))
            for (int j=c.minIndex(); j<=c.maxIndex(); j=j+1)
                if (c.isMember(j))
                    a[i+j] = a[i+j]+b[i]*c[j];
}

int main ()
{
    // funktioniert mit einer der folgenden Zeilen:
    // DynamicFloatArray f,g;
    ListFloatArray f,g;

    f[0] = 1.0; f[100] = 1.0;

    polymul(g,f,f); polymul(f,g,g);
    polymul(g,f,f); polymul(f,g,g); // f=(1+x^100)^16

    polyshow(f);
}
```

Ausgabe:

```
+1*x^0+16*x^1000+120*x^2000+560*x^3000+1820*x^4000+4368*x^5000+8008*x^6000
+11440*x^7000+12870*x^8000+11440*x^9000+8008*x^10000+4368*x^11000
+1820*x^12000+560*x^13000+120*x^14000+16*x^15000+1*x^16000
```

Bemerkung:

- Man kann nun sehr „spät“, nämlich erst in der `main`-Funktion entscheiden, mit welcher Art Felder man tatsächlich arbeiten will.

- Je nachdem, wie vollbesetzt der Koeffizientenvektor ist, ist entweder `DynamicFloatArray` oder `ListFloatArray` günstiger.
- Schlecht ist noch die Weise, in der allgemeine Schleifen über das Feld implementiert werden. Die Anwendung auf `ListFloatArray` ist sehr ineffektiv! Eine Abhilfe werden wir bald kennenlernen (Iteratoren).

16.6 Zusammenfassung

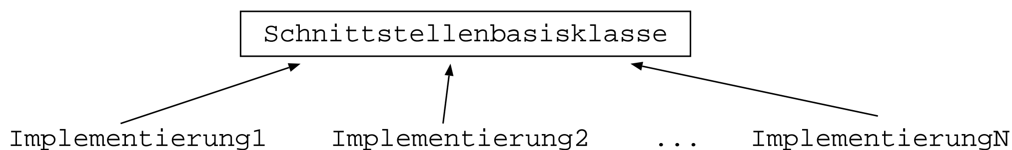
In diesem Abschnitt haben wir gezeigt wie man mit Hilfe von Schnittstellenbasisklassen eine Trennung von

- Schnittstellendefinition und
- Implementierung

erreicht.
Dies gelingt durch

- rein virtuelle Funktionen in Verbindung mit
- Vererbung.

Typischerweise erhält man Klassendiagramme der Form:



Man *erzeugt* Objekte konkreter (abgeleiteter) Klassen und *benutzt* diese Objekte durch die Schnittstellenbasisklasse:

Create objects, use interfaces!

17 Generische Programmierung

17.1 Funktionsschablonen

Definition: Eine Funktionsschablone entsteht, indem man die Präambel

```
template<class T>
```

einer Funktionsdefinition voranstellt. In der Schablonendefinition kann man `T` dann wie einen vorhandenen Datentyp verwenden.

Programm: Vertauschen des Inhalts zweier gleichartiger Referenzen:

```
template<class T> void swap (T& a, T& b) {
  T t = a;
  a = b;
  b = t;
}

int main () {
  int a=10, b=20;
  swap(a,b);
}
```

Bemerkung:

- Bei der *Übersetzung* von `swap(a,b)` generiert der Übersetzer die Version `swap(int& a, int& b)` und übersetzt sie (es sei denn, es gibt schon genau so eine Funktion).
- Wie beim Überladen von Funktionen wird die Funktion nur anhand der Argumente ausgewählt. Der Rückgabewert spielt keine Rolle.
- Im Unterschied zum Überladen generiert der Übersetzer für jede vorkommende Kombination von Argumenten eine Version der Funktion (keine automatische Typkonversion).

Programm: Beispiel: Maximum

```
template<class T> T max (T a, T b) {
    if (a<b) return b; else return a;
}
```

Bemerkung: Hier muss für den Typ `T` ein `operator<` definiert sein.

17.1.1 Beispiel: wieder funktionales Programmieren

Problem: Der Aufruf virtueller Funktionen erfordert Entscheidungen zur Laufzeit, was in einigen (wenigen) Fällen zu langsam sein kann.

Abhilfe: Verwendung von Funktionsschablonen.

Programm: (Funktionales Programmieren mit Schablonen)

```
#include <iostream>
using namespace std;

class Inkrementierer {
public:
    Inkrementierer (int n) {inkrement = n;}
    int operator() (int n) {return n+inkrement;}
private:
    int inkrement;
};

class Quadrat {
public:
    int operator() (int n) {return n*n;}
} ;

template<class T>
void schleife (T &func) {
    for (int i=1; i<10; i++)
        cout << func(i) << " ";
    cout << endl;
}

int main () {
    Inkrementierer ink(10);
    Quadrat quadrat;
    schleife (ink);
    schleife (quadrat);
}
```

Bemerkung:

- Hier werden ebenfalls automatisch die passenden Varianten der Funktion `schleife` erzeugt.
- Leider haben wir aber keine Schnittstellendefinition mehr.

Bezeichnung: Man nennt diese Technik auch statischen Polymorphismus, da die Methodenauswahl zur Übersetzungszeit erfolgt. Im Gegensatz dazu bezeichnet man die Verwendung virtueller Funktionen als dynamischer Polymorphismus.

Empfehlung: Wenden Sie diese oder ähnliche Techniken (wie etwa die sogenannten *expression templates*) nur an, wenn es unbedingt notwendig ist. Untersuchen Sie auch vorher das Laufzeitverhalten (Profiling), denn laut Donald E. Knuth (ursprünglich wohl von C.A.R. Hoare) gilt:

> *Premature optimization is the root of all evil!*

17.2 Klassenschablonen

Problem: Unsere selbstdefinierten Felder und Listen sind noch zu inflexibel. So hätten wir beispielsweise auch gerne Felder von `int`-Zahlen.

Bemerkung: Dieses Problem rührt von der statischen Typbindung von C/C++ her und tritt bei Sprachen mit dynamischer Typbindung (Scheme, Python, ...) nicht auf. Allerdings ist es für solche Sprachen viel schwieriger hocheffizienten Code zu generieren.

Abhilfe: Die C++-Lösung für dieses Problem sind parametrisierte Klassen, die auch Klassenschablonen (*class templates*) genannt werden.

Definition: Eine Klassenschablone entsteht indem man der Klassendefinition die Präambel `template<class T>` voranstellt. In der Klassendefinition kann dann der Parameter `T` wie ein Datentyp verwendet werden.

Beispiel:

```
// Schablonendefinition
template<class T>
class SimpleArray {
  public:
      SimpleArray (int s, T f);
...
}
// Verwendung
SimpleArray<int> a(10,0);
SimpleArray<float> b(10,0.0);
```

Bemerkung:

- `SimpleArray` alleine ist kein Datentyp!
- `SimpleArray<int>` ist ein neuer Datentyp, d. h. Sie können Objekte dieses Typs erzeugen, oder ihn als Parameter/Rückgabewert einer Funktion verwenden.
- Der Mechanismus arbeitet wieder zur *Übersetzungszeit* des Programmes. Bei *Übersetzung* der Zeile

  ```
  SimpleArray<int> a(10,0);
  ```

generiert der Übersetzer den Programmtext für `SimpleArray<int>`, der aus dem Text der Klassenschablone `SimpleArray` entsteht indem alle Vorkommen von `T` durch `int` ersetzt werden. Anschließend wird diese Klassendefinition übersetzt.

- Da der Übersetzer selbst C++-Programmcode generiert spricht man auch von generischer Programmierung.
- Den Vorgang der Erzeugung einer konkreten Variante einer Klasse zur Übersetzungszeit nennt man auch Template-Instanzierung.
- Der Name Schablone (*template*) kommt daher, dass man sich die parametrisierte Klasse als Schablone vorstellt, die zur Anfertigung konkreter Varianten benutzt wird.

Programm: (SimpleArray.cc)

```
template <class T>
class SimpleArray {
public:
    SimpleArray (int s, T f);
    SimpleArray (const SimpleArray<T>&);
    SimpleArray<T>& operator= (const SimpleArray<T>&);
    ~SimpleArray();

    T& operator[](int i);
    int numIndices ();
    int minIndex ();
    int maxIndex ();
    bool isMember (int i);

private:
    int n;    // Anzahl Elemente
    T *p;     // Zeiger auf built-in array
} ;
```

Bemerkung: Syntaktische Besonderheiten:

- Wird die Klasse selbst als Argument oder Rückgabewert im Rumpf der Definition benötigt schreibt man `SimpleArray<T>`.
- Im Namen des Konstruktors bzw. Destruktors taucht *kein* `T` auf. Der Klassenparameter parametrisiert den Klassennamen, nicht aber die Methodennamen.
- Die Definition des Destruktors (als Beispiel) lautet dann:

```
SimpleArray<T>::~SimpleArray () { delete[] p; }
```

Programm: Methodenimplementierung (`SimpleArrayImp.cc`):

```
// Destruktor
template <class T>
inline SimpleArray<T>::~SimpleArray () { delete[] p; }

// Konstruktor
template <class T>
SimpleArray<T>::SimpleArray (int s, T v) {
    n = s;
    p = new T[n];
    for (int i=0; i<n; i=i+1) p[i]=v;
}

// Copy-Konstruktor
template <class T>
SimpleArray<T>::SimpleArray (const SimpleArray<T>& a) {
    n = a.n;
    p = new T[n];
    for (int i=0; i<n; i=i+1)
        p[i]=a.p[i];
}

// Zuweisungsoperator
template <class T>
SimpleArray<T>& SimpleArray<T>::operator= (const SimpleArray<T>& a)
{
    if (&a!=this) {
        if (n!=a.n) {
            delete[] p;
            n = a.n;
            p = new T[n];
        }
        for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }
    return *this;
}

template <class T>
inline T& SimpleArray<T>::operator[] (int i) { return p[i];}

template <class T>
inline int SimpleArray<T>::numIndices () { return n; }

template <class T>
inline int SimpleArray<T>::minIndex () { return 0; }

template <class T>
```

```
inline int SimpleArray<T>::maxIndex () { return n-1; }

template <class T>
inline bool SimpleArray<T>::isMember (int i) {
    return (i>=0 && i<n);
}

template <class T>
ostream& operator<< (ostream& s, SimpleArray<T>& a) {
    s << "#( ";
    for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1)
        s << a[i] << " ";
    s << ")" << endl;
    return s;
}
```

Programm: Verwendung (`UseSimpleArray.cc`):

```
#include<iostream>
using namespace std;

#include "SimpleArray.cc"
#include "SimpleArrayImp.cc"

int main ()
{
    SimpleArray<float> a(10,0.0); // erzeuge Felder
    SimpleArray<int> b(25,5);

    for (int i=a.minIndex(); i<=a.maxIndex(); i++)
        a[i] = i;
    for (int i=b.minIndex(); i<=b.maxIndex(); i++)
        b[i] = i;

    cout << a << endl << b << endl;

    // hier wird der Destruktor gerufen
}
```

17.2.1 Beispiel: Feld fester Größe

Bemerkung:

- Eine Schablone kann auch mehr als einen Parameter haben.
- Als Schablonenparameter sind nicht nur Klassennamen, sondern z.B. auch Konstanten von eingebauten Typen erlaubt.

Anwendung: Ein Feld fester Größe könnte folgendermaßen definiert und verwendet werden:

```
template <class T, int m>
class SimpleArrayCS {
public:
    SimpleArrayCS (T f);
    ...
private:
    T p[m]; // built-in array fester Groesse
} ;

...

SimpleArrayCS<int,5> a(0);
SimpleArrayCS<float,3> a(0.0);
...
```

Bemerkung:

- Die Größe ist hier auch zur Übersetzungszeit festgelegt und muss nicht mehr gespeichert werden.
- Da nun keine Zeiger auf dynamisch allokierte Objekte verwendet werden sind für Copy-Konstruktor, Zuweisung und Destruktor die Defaultmethoden ausreichend.
- Der Compiler kann bei bekannter Feldgröße unter Umständen effizienteren Code generieren, was vor allem für kleine Felder interessant ist (z.B. Vektoren im $\mathbb{R}^2$ oder $\mathbb{R}^3$). *Es ist ein wichtiges Kennzeichen von C++, dass Objektorientierung bei richtigem Gebrauch auch für sehr kleine Datenstrukturen ohne Effizienzverlust angewendet werden kann.*

17.2.2 Beispiel: Smart Pointer

Problem: Dynamische erzeugte Objekte können ausschließlich über Zeiger verwaltet werden. Wie wir bereits diskutiert, ist die konsistente Verwaltung des Zeigers (bzw. der Zeiger) und des Objekts nicht einfach.

Abhilfe: Entwurf mit einem neuen Datentyp, der anstatt eines Zeigers verwendet wird. Mittels Definition von `operator*` und `operator->` kann man erreichen, dass sich der neue Datentyp wie ein eingebauter Zeiger benutzen lässt. In Copy-Konstruktor und Zuweisungsoperator wird dann *reference counting* eingebaut.

Bezeichnung: Ein Datentyp mit dieser Eigenschaft wird intelligenter Zeiger (*smart pointer*) genannt.

Programm: (Zeigerklasse zum *reference counting*)

```
template<class T>
class Ptr {
    struct RefCntObj {
        int count;
        T* obj;
        RefCntObj (T* q) { count = 1; obj = q; }
    };
    RefCntObj* p;

    void report () {
        cout << "refcnt = " << p->count << endl;
    }
    void increment () {
        ++(p->count);
        report();
    }
    void decrement () {
        --(p->count);
        report();
        if (p->count==0) {
            delete p->obj;
            delete p;
        }
    }

  public:
    Ptr () { p=0; }

    Ptr (T* q) {
        p = new RefCntObj(q);
        report();
    }

    Ptr (const Ptr<T>& y) {
        p = y.p;
        if (p!=0) increment();
    }

    ~Ptr () {
        if (p!=0) decrement();
    }

    Ptr<T>& operator= (const Ptr<T>& y) {
        if (p!=y.p) {
            if (p!=0) decrement();
            p = y.p;
```

```
            if (p!=0) increment();
        }
        return *this;
    }

    T& operator* () { return *(p->obj); }

    T* operator-> () { return p->obj; }
};
```

Programm: (Anwendungsbeispiel)

```
#include<iostream>
using namespace std;

#include"Ptr.h"

int g (Ptr<int> p) {
    return *p;
}

int main ()
{
    Ptr<int> q = new int(17);
    cout << *q << endl;
    int x = g(q);
    cout << x << endl;
    Ptr<<int> z = new int(22);
    q = z;
    cout << *q << endl;
    // nun wird alles automatisch geloescht !
}
```

Bemerkung:

- Man beachte die sehr einfache Verwendung durch Ersetzen der eingebauten Zeiger (die natürlich nicht weiterverwendet werden sollten!).
- Nachteil: mehr Speicher wird benötigt (das `RefCntObj`)
- Es gibt verschiedene Möglichkeiten, *reference counting* zu implementieren, die sich bezüglich Speicher- und Rechenaufwand unterscheiden.
- Die hier vorgestellte Zeigerklasse funktioniert (wegen `delete[]`) nicht für Felder. Im Bastian-Skript ist auch noch die Version für Felder enthalten.
- *Reference counting* funktioniert *nicht* für Datenstrukturen mit Zykeln $\rightsquigarrow$ andere Techniken zur automatischen Speicherverwaltung notwendig.

18 Effizienz generischer Programmierung

18.1 Beispiel: Bubblesort

Aufgabe: Ein Feld von Zahlen $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ ist zu sortieren. Die Sortierfunktion liefert als Ergebnis eine Permutation $a' = (a'_0, a'_1, a'_2, \ldots, a'_{n-1})$ der Feldelemente zurück, so dass

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$$

Idee: Der Algorithmus Bubblesort ist folgendermaßen definiert:

- Gegeben sei ein Feld $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ der Länge n.
- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden. Beispiel:

	17	3	8	16
$i=0$	3	17	8	16
$i=1$	3	8	17	16
$i=2$	3	8	16	17

  Am Ende eines solchen Durchlaufes steht die größte der Zahlen sicher ganz rechts und ist damit an der richtigen Position.

- Damit bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Satz: t_{cs} sei eine obere Schranke für einen Vergleich und einen swap und n bezeichne die Länge des Felds. Falls t_{cs} nicht von n abhängt, so hat Bubblesort eine asymptotische Laufzeit von $O(n^2)$.

Beweis:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} t_{cs} = t_{cs} \sum_{i=0}^{n-1} i = t_{cs} \frac{(n-1)n}{2} = O(n^2)$$

Programm:

```
/* ist in namespace std schon enthalten:
  template <class T> void swap (T& a, T&b) {
    T t = a;
    a = b;
    b = t;
}
*/

template <class C> void bubblesort (C& a) {
    for (int i=a.maxIndex(); i>=a.minIndex(); i=i-1)
```

```
        for (int j=a.minIndex(); j<i; j=j+1)
            if (a[j+1]<a[j])
                swap(a[j+1], a[j]);
}
```

Bemerkung:

- Die Funktion `bubblesort` benötigt, dass auf *Elementen* des Feldes der Vergleichsoperator `operator<` definiert ist.
- Die Funktion benutzt die *öffentliche Schnittstelle* der Feldklassen, die wir programmiert haben, d. h. für `C` können wir jede unserer Feldklassen einsetzen!

18.2 Effizienz

Mit folgender Routine kann man Laufzeiten verschiedener Programmteile messen:

Programm: (`timestamp.cc`)

```
#include <ctime>
// Setzt Marke und gibt Zeitdifferenz zur letzten Marke zurueck
clock_t last_time;
double time_stamp () {
    clock_t current_time = clock();
    double duration =
        ((double)(current_time-last_time)) / CLOCKS_PER_SEC;
    last_time = current_time;
    return duration;
}
```

Dies wenden wir auf Bubblesort an:

Programm: Bubblesort für verschiedene Feldtypen (`UseBubblesort.cc`)

```
#include<iostream>
using namespace std;

// SimpleFloatArray mit virtuellem operator[]
#include "SimpleFloatArrayV.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// templatisierte Variante mit variabler Groesse
#include "SimpleArray.cc"
#include "SimpleArrayImp.cc"

// templatisierte Variante mit Compile-Zeit Groesse
```

```
#include "SimpleArrayCS.cc"
#include "SimpleArrayCSImp.cc"

// dynamisches listenbasiertes Feld
#include "FloatArray.cc"
#include "ListFloatArrayDerived.cc"
#include "ListFloatArrayImp.cc"

// Zeitmessung
#include "timestamp.cc"

// generischer bubblesort
#include "bubblesort.cc"

// Zufallsgenerator
#include "Zufall.cc"

const int n = 2000;

static Zufall z(93576);

template <class T>
void initialisiere (T & a) {
    for (int i=0; i<n; i=i+1)
        a[i] = z.ziehe_zahl();
}

int main ()
{
    SimpleArrayCS<float,n> a(0.0);
    SimpleArray<float> b(n,0.0);
    SimpleFloatArray c(n,0.0);
    ListFloatArray d;

    initialisiere (a); initialisiere (b);
    initialisiere (c); initialisiere (d);

    time_stamp();
    cout << "SimpleArrayCS ...";
    bubblesort(a);
    cout << time_stamp() << " sec" << endl;
    cout << "SimpleArray ...";
    bubblesort(b);
    cout << time_stamp() << " sec" << endl;
    cout << "SimpleFloatArray ...";
    bubblesort(c);
```

```
    cout << time_stamp() << " sec" << endl;
    cout << "ListFloatArray ...";
    bubblesort(d);
    cout << time_stamp() << " sec" << endl;
}
```

Ergebnis:

n	1000	2000	4000	8000	16000	32000
built-in array	0.01	0.04	0.14	0.52	2.08	8.39
`SimpleArrayCS`	0.01	0.03	0.15	0.58	2.30	9.12
`SimpleArray`	0.01	0.05	0.15	0.60	2.43	9.68
`SimpleArray` ohne inline	0.04	0.15	0.55	2.20	8.80	35.31
`SimpleFloatArrayV`	0.04	0.15	0.58	2.28	9.13	36.60
`ListFloatArray`	4.62	52.38	—	—	—	—

Bemerkung:

- Die ersten fünf Zeilen zeigen deutlich den $O(n^2)$-Aufwand: Verdopplung von n bedeutet vierfache Laufzeit.
- Die Zeilen fünf und vier zeigen die Laufzeit für die Variante mit einem virtuellem `operator[]` bzw. eine Version der Klassenschablone, bei der das Schlüsselwort `inline` vor der Methodendefinition des `operator[]` weggelassen wurde. Diese beiden Varianten sind etwa viermal langsamer als die vorherigen.
- Eine Variante mit eingebautem Feld (nicht vorgestellt, ohne Klassen) ist am schnellsten, gefolgt von den zwei Varianten mit Klassenschablonen, die unwesentlich langsamer sind.
- `ListFloatArray` ist die listenbasierte Darstellung des Feldes mit Index-Wert-Paaren. Diese hat Komplexität $O(n^3)$, da nun die Zugriffe auf die Feldelemente Komplexität $O(n)$ haben.

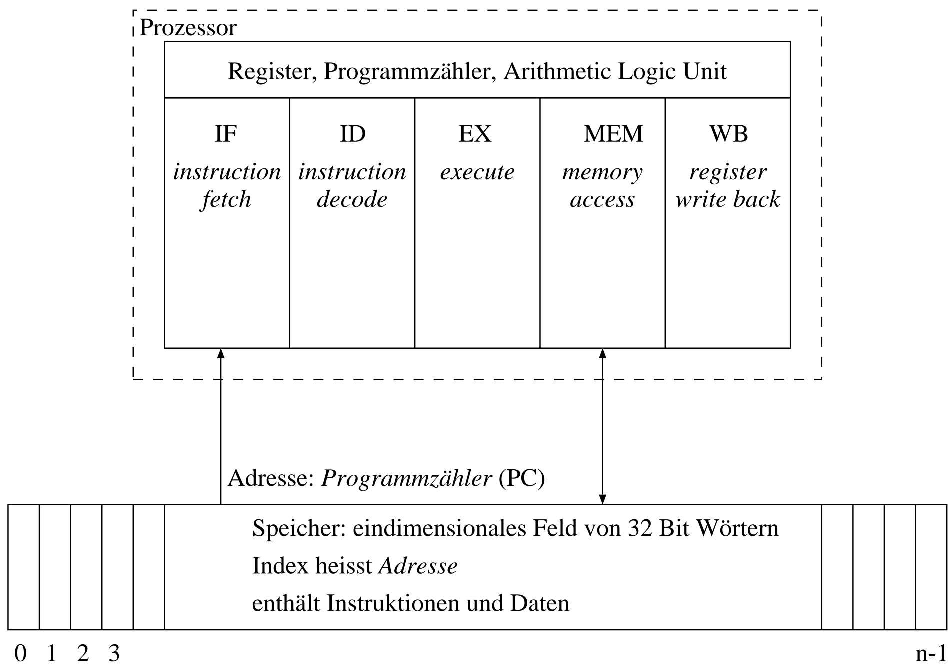
Frage: Warum sind die Varianten auf Schablonenbasis (mit inlining) schneller als die Variante mit virtueller Methode?

18.3 RISC

Bezeichnung: RISC steht für *Reduced Instruction Set Computer* und steht für eine Kategorie von Prozessorarchitekturen mit verhältnismäßig einfachem Befehlssatz. Gegenpol: CISC=*Complex Instruction Set Computer*.

Geschichte: RISC stellt heutzutage den Großteil aller Prozessoren dar (vor allem bei eingebetteten Systemen (Handy, PDA, Spielekonsole, etc), wo das Verhältnis Leistung/Verbrauch wichtig ist). Für PCs ist allerdings noch mit den Intel-Chips die CISC-Technologie dominant (mittlerweile wurden aber auch dort viele RISC-Techniken übernommen).

18.3.1 Aufbau eines RISC-Chips



18.3.2 Befehlszyklus

Bezeichnung: Ein typischer RISC-Befehl lässt sich in Teilschritte unterteilen, die von verschiedener Hardware (in der CPU) ausgeführt werden:

1. IF: Holen des nächsten Befehls aus dem Speicher. Ort: *Programmzähler*.
2. ID: Dekodieren des Befehls, Auslesen der beteiligten Register.
3. EX: Eigentliche Berechnung (z. B. Addieren zweier Zahlen).
4. MEM: Speicherzugriff (entweder Lesen oder Schreiben).
5. WB: Rückschreiben der Ergebnisse in Register.

Dies nennt man Befehlszyklus (*instruction cycle*).

18.3.3 Pipelining

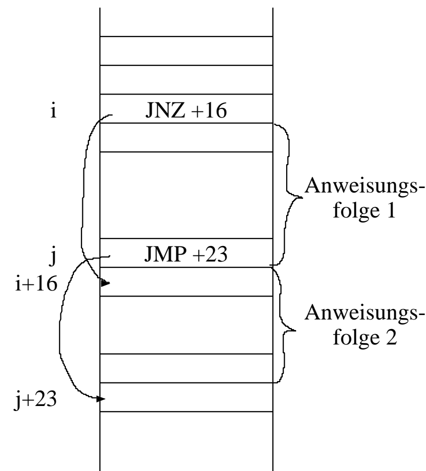
Diese Stadien werden nun für aufeinanderfolgende Befehle überlappend ausgeführt (Pipelining).

IF	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6	Instr7
ID	—	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6
EX	—	—	Instr1	Instr2	Instr3	Instr4	Instr5
MEM	—	—	—	Instr1	Instr2	Instr3	Instr4
WB	—	—	—	—	Instr1	Instr2	Instr3

18.3.4 Probleme mit Pipelining

Sehen wir uns an, wie eine if-Anweisung realisiert wird:

```
if (a==0)
{
   <Anweisungsfolge 1>
}
else
{
   <Anweisungsfolge 2>
}
```



Problem: Das Sprungziel des Befehls `JNZ +16` steht erst am Ende der dritten Stufe der Pipeline (EX) zur Verfügung, da ein Register auf 0 getestet und 16 auf den PC addiert werden muss.

Frage: Welche Befehle sollen bis zu diesem Punkt weiter angefangen werden?

Antwort:

- Gar keine, dann bleiben einfach drei Stufen der Pipeline leer (pipeline stall).
- Man *rät* das Sprungziel (branch prediction unit) und führt die nachfolgenden Befehle spekulativ aus (ohne Auswirkung nach aussen). Notfalls muss man die Ergebnisse dieser Befehle wieder verwerfen.

Bemerkung: Selbst das Ziel eines unbedingten Sprungbefehls stünde wegen der Addition des Offset auf den PC erst nach der Stufe EX zur Verfügung (es sei denn, man hat extra Hardware dafür).

18.3.5 Funktionsaufrufe

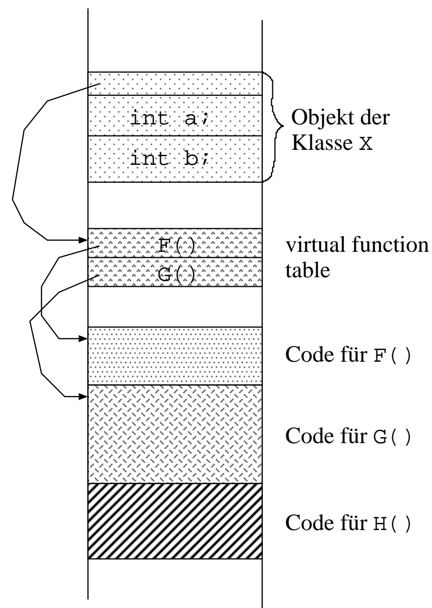
Ein Funktionsaufruf (Methodenaufruf) besteht normalerweise aus folgenden Operationen:

- Sicherung der Rücksprungadresse auf dem Stack
- ein unbedingter Sprungbefehl
- der Rücksprung an die gespeicherte Adresse
- + eventuelle Sicherung von Registern auf dem Stack

Diese Liste gilt genauso für CISC-Architekturen. Ein Funktionsaufruf ist also normalerweise mit erheblichem Aufwand verbunden.

18.3.6 Realisierung virtueller Funktionen

```
class X {
public:
  int a;
  int b;
  virtual void F();
  virtual void G();
  void H();
} ;
```



Bemerkung:

- Für jede Klasse gibt es eine Tabelle mit Zeigern auf den Programmcode für die virtuellen Funktionen dieser Klasse. Diese Tabelle heißt *virtual function table* (VFT).
- Jedes Objekt einer Klasse, die virtuelle Funktionen enthält, besitzt einen Zeiger auf die VFT der zugehörigen Klasse. Dies entspricht im wesentlichen der Typinformation, die bei Sprachen mit dynamischer Typbindung den Daten hinzugefügt ist.
- Beim Aufruf einer virtuellen Methode generiert der Übersetzer Code, welcher der VFT des Objekts die Adresse der aufzurufenden Methode entnimmt und dann den Funktionsaufruf durchführt. Welcher Eintrag der VFT zu entnehmen ist, ist zur *Übersetzungszeit* bekannt.
- Der Aufruf nichtvirtueller Funktionen geschieht ohne VFT. Klassen (und ihre zugehörigen Objekte) ohne virtuelle Funktionen brauchen keinen Zeiger auf eine VFT.
- Für den Aufruf virtueller Funktionen ist immer ein Funktionsaufruf notwendig, da erst zur Laufzeit bekannt ist, welche Methode auszuführen ist.

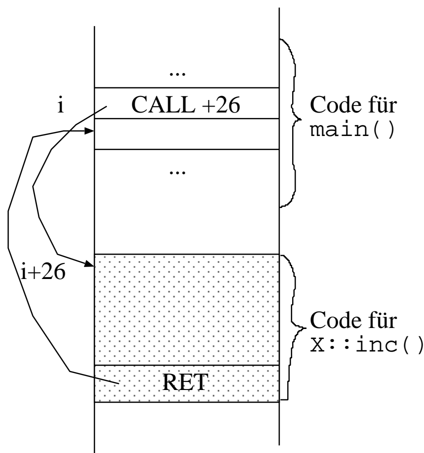
18.3.7 Inlining

Problem: Der Funktionsaufruf sehr kurzer Funktionen ist relativ langsam.

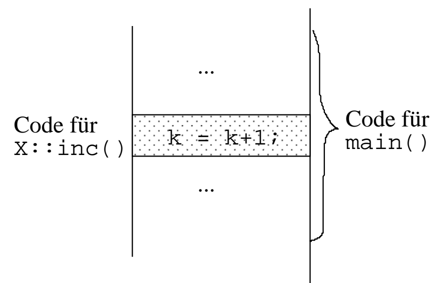
Beispiel:

```
class X {                              void main ()
public:                                {
  void inc();                            X x;
private:                                 x.inc();
  int k;                               }
} ;

inline void X::inc ()
{
    k = k+1;
}
```

Ohne das Schlüsselwort `inline` in der Methodendefinition generiert der C++-Übersetzer einen Funktionsaufruf für `inc()`:



Mit dem Schlüsselwort `inline` in der Methodendefinition setzt der Übersetzer den Code der Methode am Ort des Aufrufes direkt ein falls dies möglich ist:



Bemerkung:

- Inlining ändert nichts an der Semantik des Programmes.
- Das Schlüsselwort `inline` ist nur ein *Vorschlag* an den Compiler. Z. B. wird es für rekursive Funktionen ignoriert.
- Virtuelle Funktionen können nicht inline ausgeführt werden, da die auszuführende Methode zur Übersetzungszeit nicht bekannt ist.
- *Aber:* Änderungen der Implementation einer Inline-Funktion in einer Bibliothek machen normalerweise die erneute Übersetzung von anderen Programmteilen notwendig!

Bemerkung: Es sei auch nochmal eindringlich an Knuth's Wort *"Premature optimization is the root of all evil"* erinnert. Bevor Sie daran gehen, Ihr Programm durch Elimination virtueller Funktionen und Inlining unflexibler zu machen, sollten Sie folgendes tun:

1. Überdenken Sie den Algorithmus!

2. Messen Sie, wo der „Flaschenhals“ wirklich liegt (Profiling notwendig).
3. Überlegen Sie, ob die erreichbare Effizienzsteigerung den Aufwand wert ist.

Beispielsweise ist die *einzig sinnvolle Verbesserung* für das Sortierbeispiel am Anfang dieses Abschnitts das Verwenden eines besseren Algorithmus!

18.4 Zusammenfassung

- Klassenschablonen definieren parametrisierte Datentypen und sind daher besonders geeignet, um allgemein verwendbare Konzepte (ADT) zu implementieren.
- Funktionsschablonen definieren parametrisierte Funktionen, die auf verschiedenen Datentypen (mit gleicher Schnittstelle) operieren.
- In beiden Fällen werden konkrete Varianten der Klassen/Funktionen zur Übersetzungszeit erzeugt und übersetzt (generische Programmierung).
- Diese Techniken sind für Sprachen mit *dynamischer Typbindung* meist unnötig. Solche Sprachen brauchen aber in vielen Fällen Typabfragen zur Laufzeit, was dazu führt, dass der erzeugte Code nicht mehr hocheffizient ist.

18.4.1 Nachteile der generischen Programmierung

- Es wird viel Code erzeugt. Die Übersetzungszeiten template-intensiver Programme können unerträglich lang sein.
- Es ist keine *getrennte Übersetzung* möglich. Der Übersetzer muss die Definition aller vorkommenden Schablonen kennen. Dasselbe gilt für Inline-Funktionen.
- Das Finden von Fehlern in Klassen/Funktionenschablonen ist erschwert, da der Code für eine konkrete Variante nirgends existiert. Empfehlung: testen Sie zuerst mit einem konkreten Datentyp und machen Sie dann eine Schablone daraus.

19 Containerklassen

19.1 Motivation

Bezeichnung: Klassen, die eine Menge anderer Objekte verwalten (man sagt aggregieren) nennt man Containerklassen.

Beispiele: Wir hatten: Liste, Stack, Feld. Weitere sind: *binärer Baum* (*binary tree*), Warteschlange (*queue*), Abbildung (*map*), ...

Bemerkung: Diese Strukturen treten sehr häufig als Komponenten in größeren Programmen auf. Ziel von Containerklassen ist es, diese Bausteine in wiederverwendbarer Form zur Verfügung zu stellen (*code reuse*).

Vorteile:

- Weniger Zeitaufwand in Entwicklung und Fehlersuche.
- Klarere Programmstruktur, da man auf einer höheren Abstraktionsebene arbeitet.

Werkzeug: Das Werkzeug zur Realisierung effizienter und flexibler Container in C++ sind Klassenschablonen.

Bemerkung: In diesem Abschnitt sehen wir uns eine Reihe von Containern an. Die Klassen sind vollständig ausprogrammiert und zeigen, wie man Container implementieren *könnte*. In der Praxis verwendet man allerdings die *Standard Template Library* (STL), welche Container in professioneller Qualität bereitstellt.

Ziel: Sie sind am Ende dieses Kapitels motiviert die STL zu verwenden und können die Konzepte verstehen.

19.2 Listenschablone

Bei diesem Entwurf ist die Idee das Listenelement und damit auch die Liste als Klassenschablone zu realisieren. In jedem Listenelement wird ein Objekt der Klasse `T`, dem Schablonenparameter, gespeichert. Allerdings ist die Liste nun homogen, d. h. alle Listenelement speichern Objekte des gleichen Typs.

Programm: Definition und Implementation (`Liste.cc`)

```
template<class T>
class List {
public:
    // Infrastruktur
    List() { _first=0; }
    ~List();

    // Listenelement als nested class
    class Link {
```

```
        Link* _next;
    public:
        T item;
        Link (T& t) {item=t;}
        Link* next () {return _next;}
        friend class List<T>;
    };

    Link* first() {return _first;}
    void insert (Link* where, T t);
    void remove (Link* where);
private:
    Link* _first;
};

template<class T> List<T>::~List()
{
    Link* p = _first;
    while (p!=0)
    {
        Link* q = p;
        p = p->next();
        delete q;
    }
}

template<class T>
void List<T>::insert (List<T>::Link* where, T t)
{
    Link* ins = new Link(t);
    if (where==0)
    {
        ins->_next = _first;
        _first = ins;
    }
    else
    {
        ins->_next = where->_next;
        where->_next = ins;
    }
}

template<class T>
void List<T>::remove (List<T>::Link* where)
{
    Link* p;
```

```
    if (where==0)
    {
        p = _first;
        if (p!=0) _first = p->_next;
    }
    else
    {
        p = where->_next;
        if (p!=0) where->_next = p->_next;
    }
    delete p;
}
```

Programm: Verwendung (`UseListe.cc`)

```
#include"iostream.h"
#include"new.h"
#include"Liste.cc"

int main () {
    List<int> list;

    list.insert(0,17); list.insert(0,34); list.insert(0,26);

    for (List<int>::Link* l=list.first(); l!=0; l=l->next())
        cout << l->item << endl;
    for (List<int>::Link* l=list.first(); l!=0; l=l->next())
        l->item = 23;
}
```

Bemerkung:

- Diese Liste ist homogen, d.h. alle Objekte im Container haben den gleichen Typ. Eine heterogene Liste könnte man als Liste von Zeigern auf eine gemeinsame Basisklasse realisieren.
- Speicherverwaltung wird von der Liste gemacht. Listen können kopiert und als Parameter übergeben werden.
- Zugriff auf die Listenelemente erfolgt über eine offengelegte *nested class*.

19.3 Iteratoren

Problem: Eine Grundoperation aller Container ist das Durchlaufen aller Objekte in dem Container. Um Container austauschbar verwenden zu können, sollten sie daher in gleicher Weise durchlaufen werden können. Die Schleife für eine Liste sah aber ganz anders aus als bei einem Feld.

Abhilfe: Diese Abstraktion realisiert man mit Iteratoren. Iteratoren sind zeigerähnliche Objekte, die auf ein Objekt im Container zeigen (obwohl der Iterator nicht als Zeiger realisiert sein muss).

Prinzip:

```
template <class T> class Container {
public:
    class Iterator {  // nested class definition
        ...
    public:
        Iterator();
        bool operator!= (Iterator x);
        bool operator== (Iterator x);
        Iterator operator++ ();      // prefix
        Iterator operator++ (int);   // postfix
        T& operator* () const;
        T* operator-> () const;
        friend class Container<T>;
    } ;
    Iterator begin () const;
    Iterator end () const;
    ... // Spezialitäten des Containers
} ;

// Verwendung
Container<int> c;
for (Container<int>::Iterator i=c.begin(); i!=c.end(); i++)
    cout << *i << endl;
```

Bemerkung:

- Der `Iterator` ist als Klasse innerhalb der Containerklasse definiert. Dies nennt man eine geschachtelte Klasse (*nested class*).
- Damit drückt man aus, dass `Container` und `Iterator` zusammengehören. Jeder Container wird seine eigene Iteratorklasse haben.
- Innerhalb von `Container` kann man `Iterator` wie jede andere Klasse verwenden.
- `friend class Container<T>` bedeutet, dass die Klasse `Container<T>` auch Zugriff auf die *privaten* Datenmitglieder der Iteratorklasse hat.
- Die Methode `begin()` des Containers liefert einen Iterator, der auf das erste Element des Containers zeigt.

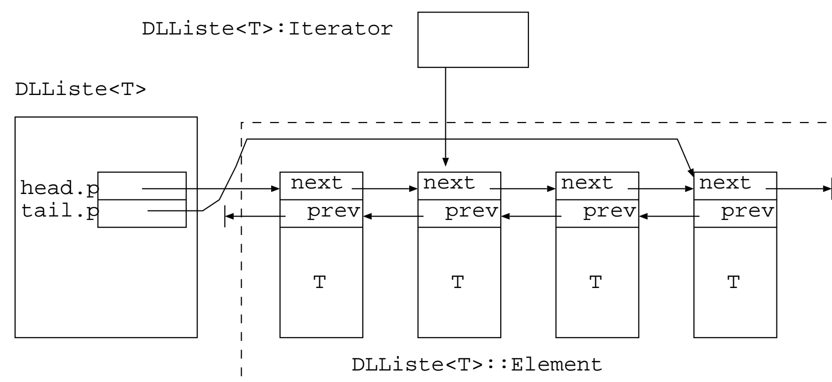
- `++i` bzw. `i++` stellt den Iterator auf das *nächste* Element im Container. Zeigte der Container auf das letzte Element, dann ist der Iterator gleich dem von `end()` gelieferten Iterator.
- `++i` bzw. `i++` manipulieren den Iterator für den den sie aufgerufen werden. Als Rückgabewert liefert `++i` den neuen Wert, `i++` jedoch den alten Wert.
- Bei der Definition unterscheiden sie sich dadurch, dass der Postfix-Operator noch ein `int`-Argument erhält, das aber keine Bedeutung hat.
- `end()` liefert einen Iterator, der auf „das Element nach dem letzten Element“ des Containers zeigt (siehe oben).
- `*i` liefert eine Referenz auf das Objekt im Container, auf das der Iterator `i` zeigt. Damit kann man sowohl `x = *i` als auch `*i = x` schreiben.
- Ist das Objekt im Container von einem zusammengesetzten Datentyp (also `struct` oder `class`), so kann mittels `i-><Komponente>` eine Komponente selektiert werden. Der Iterator verhält sich also wie ein Zeiger.

19.4 Doppelt verkettete Liste

Anforderungen:

- Vorwärts- *und* Rückwärtsdurchlauf
- Das Einfügen vor oder nach einem Element soll eine O(1)-Operation sein. Die Position wird durch einen Iterator angegeben.
- Das Entfernen eines Elementes soll eine O(1)-Operation sein. Das zu entfernende Element wird wieder durch einen Iterator angegeben
- Für die Berechnung der Größe der Liste akzeptieren wir einen O(N) Aufwand. (Die Version aus dem Bastian-Skript erreicht hingegen O(1) durch Mitführen der Länge.)

19.4.1 Struktur



Bemerkung:

- Intern werden die Listenelemente durch den Datentyp `Element` repräsentiert. Dieser private, geschachtelte, zusammengesetzte Datentyp ist außerhalb der Klasse nicht sichtbar.
- Die Einfügeoperationen erhalten Objekte vom Typ `T`, erzeugen dynamisch ein Listenelement und *kopieren* das Objekt in das Listenelement.
- Damit kann man Listen für beliebige Datentypen erzeugen. Die Manipulation gelingt mit Hilfe der Iteratorschnittstelle. Der Iterator kapselt insbesondere den Zugriff auf die außerhalb der Liste nicht bekannten `Element`-Objekte.

19.4.2 Implementation

Programm: `DLL.cc`

```
template<class T>
class DLList {
public:
    struct Element;
    class Iterator {
    private:
        Element* p;
    public:
        Iterator() { p=0; }
        Iterator(Element* q) { p=q; }
        bool operator!= (Iterator x) { return p!=x.p; }
        bool operator== (Iterator x) { return p==x.p; }
        Iterator operator++ () {
            p=p->next;
            return *this;
        }
        Iterator operator++ (int) {
            Iterator tmp = *this;
            p=p->next;
            return tmp;
        }
        Iterator operator-- () {
            p=p->prev;
            return *this;
        }
        Iterator operator-- (int) {
            Iterator tmp = *this;
            p=p->next;
            return tmp;
```

```
        }
        T& operator* () { return p->item; }
        T* operator-> () { return &(p->item); }
        friend class DLList<T>;
    };
    Iterator begin () const {return head;}
    Iterator end () const {return Iterator();}
    Iterator rbegin () const {return tail;}
    Iterator rend () const {return Iterator();}

    DLList();
    DLList (const DLList<T>& list);
    DLList<T>& operator= (const DLList<T>&);
    ~DLList();

    Iterator insert (Iterator i, T t);  // insert before i
    void erase (Iterator i);
    void append (const DLList<T>& l);
    void clear ();
    bool empty () const;
    int size () const;
    Iterator find (T t) const;

private:
    struct Element {
        Element* next;
        Element* prev;
        T item;
        Element (T &t) {
            item = t;
            next = prev = 0;
        }
    };
    Iterator head;      // erstes Element der Liste
    Iterator tail;      // letztes Element der Liste
};


// Insertion
template<class T>
typename DLList<T>::Iterator
DLList<T>::insert (Iterator i, T t)
{
    Element* e = new Element(t);
    if (empty())
    {
```

```
            assert (i.p==0);
            head.p = tail.p = e;
        }
        else
        {
            e->next = i.p;
            if (i.p!=0)
            {   // insert before i
                e->prev = i.p->prev;
                i.p->prev = e;
                if (head==i)
                    head.p=e;
            }
            else
            {   // insert at end
                e->prev = tail.p;
                tail.p->next = e;
                tail.p = e;
            }
        }
        return Iterator(e);
    }

template<class T>
void DLList<T>::erase (Iterator i)
{
    if (i.p==0) return;

    if (i.p->next!=0)
        i.p->next->prev = i.p->prev;
    if (i.p->prev!=0)
        i.p->prev->next = i.p->next;

    if (head==i) head.p=i.p->next;
    if (tail==i) tail.p=i.p->prev;

    delete i.p;
}

template<class T>
void DLList<T>::append (const DLList<T>& l) {
    for (Iterator i=l.begin(); i!=l.end(); i++)
        insert(end(),*i);
}

template<class T>
```

```
bool DLList<T>::empty () const {
    return begin()==end();
}

template<class T>
void DLList<T>::clear () {
    while (!empty())
        erase(begin());
}

// Constructors
template<class T> DLList<T>::DLList () {}

template<class T>
DLList<T>::DLList (const DLList<T>& list) {
    append(list);
}

// Assignment
template<class T>
DLList<T>&
DLList<T>::operator= (const DLList<T>& l) {
    if (this!=&l) {
        clear();
        append(l);
    }
    return *this;
}

// Destructor
template<class T> DLList<T>::~DLList() { clear(); }

// Size method
template<class T> int DLList<T>::size () const {
    int count = 0;
    for (Iterator i=begin(); i!=end(); i++)
        count++;
    return count;
}

template<class T>
typename DLList<T>::Iterator DLList<T>::find (T t) const {
    DLList<T>::Iterator i = begin();
    while (i!=end())
    {
        if (*i==t) break;
```

```
        i++;
    }
    return i;
}

template <class T>
ostream& operator<< (ostream& s, DLList<T>& a) {
    s << "(";
    for (typename DLList<T>::Iterator i=a.begin();
         i!=a.end(); i++)
    {
        if (i!=a.begin()) cout << " ";
        s << *i;
    }
    s << ")" << endl;
    return s;
}
```

19.4.3 Verwendung

Programm: UseDLL.cc

```
#include<cassert>
#include<iostream>
using namespace std;

#include"DLL.cc"
#include"Zufall.cc"

int main ()
{
    Zufall z(87124);
    DLList<int> l1,l2,l3;

    // Erzeuge 3 Listen mit je 5 Zufallszahlen
    for (int i=0; i<5; i=i+1)
        l1.insert(l1.end(), i);
    for (int i=0; i<5; i=i+1)
        l2.insert(l2.end(), z.ziehe_zahl());
    for (int i=0; i<5; i=i+1)
        l3.insert(l3.end(), z.ziehe_zahl());

    // Loesche alle geraden in der ersten Liste
    DLList<int>::Iterator i,j;
    i=l1.begin();
    while (i!=l1.end())
```

```
    {
        j=i; // merke aktuelles Element
        ++i; // gehe zum naechsten
        if (*j%2==0) l1.erase(j);
    }

    // Liste von Listen ...
    DLList<DLList<int> > ll;
    ll.insert(ll.end(),l1);
    ll.insert(ll.end(),l2);
    ll.insert(ll.end(),l3);
    cout << ll << endl << "Länge: " << ll.size() << endl;
}
```

19.4.4 Diskussion

- Den Rückwärtsdurchlauf durch eine Liste erreicht man durch:

  ```
  for (DLList<int>::Iterator i=c.rbegin(); i!=c.rend(); i--)
      cout << *i << endl;
  ```

- Die Objekte (vom Typ T) werden beim Einfügen in die Liste kopiert. Abhilfe: Liste von Zeigern auf die Objekte, z. B. `DLList<int *>`.
- Die Schlüsselworte `const` in der Definition von `begin`, `end`, ... bedeuten, dass diese Methoden ihr Objekt nicht ändern.
- Ärgerlicherweise muss man an einigen Stellen ein `typename` vor den Iterator-Typ einfügen, weil der Parser (Syntaxanalysierer) das Wort nicht eindeutig einem Typ zuordnen kann.

19.4.5 Beziehung zur STL-Liste

Die entsprechende STL-Schablone heißt `list` und unterscheidet sich von unserer Liste unter anderem in folgenden Punkten:

- Man erhält die Funktionalität durch `#include <list>`.
- Die Iterator-Klasse heißt `iterator` statt `Iterator`.
- Es gibt zusätzlich einen `const_iterator`. Auch unterscheiden sich Vorwärts- und Rückwärtsiteratoren (`reverse_iterator`).
- Sie hat einige Methoden mehr, z.B. `push_front`, `push_back`, `front`, `back`, `pop_front`, `pop_back`, `sort`, `reverse`, ...
- Die Ausgabe über „`cout <<`“ ist nicht definiert.

19.5 Feld

Wir fügen nun die Iterator-Schnittstelle unserer `SimpleArray<T>`-Schablone hinzu.

Programm:

```
template <class T> class Array {
public:
    class Iterator {
    private:
        T* p;
        Iterator(T* q) {p=q;}
    public:
        Iterator() {p=0;}
        bool operator!= (Iterator x) {
            return (p!=x.p);
        }
        bool operator== (Iterator x) {
            return (p==x.p);
        }
        Iterator operator++ () {
            p++;
            return *this;
        }
        Iterator operator++ (int) {
            Iterator tmp = *this;
            ++*this;
            return tmp;
        }
        T& operator* () const {return *p;}
        T* operator-> () const {return p;}
        friend class Array<T>;
    };
    Iterator begin () const {
        return Iterator(p);
    }
    Iterator end () const {
        return Iterator(&(p[n]));
    }

    Array(int m) {
        n = m;
        p = new T[n];
    }
    Array (const Array<T>&);
    Array<T>& operator= (const Array<T>&);
    ~Array() {
        delete[] p;
```

```
    }
    int size () const {
        return n;
    }
    T& operator[](int i) {
        return p[i];
    }
    typedef T MemberType;   // Merke Grundtyp
private:
    int n;   // Anzahl Elemente
    T *p;    // Zeiger auf built-in array
} ;

// Copy-Konstruktor
template <class T>
Array<T>::Array (const Array<T>& a) {
    n = a.n;
    p = new T[n];
    for (int i=0; i<n; i=i+1)
        p[i]=a.p[i];
}

// Zuweisung
template <class T>
Array<T>& Array<T>::operator= (const Array<T>& a) {
    if (&a!=this) {
        if (n!=a.n) {
            delete[] p;
            n = a.n;
            p = new T[n];
        }
        for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }
    return *this;
}

// Ausgabe
template <class T>
ostream& operator<< (ostream& s, Array<T>& a) {
    s << "array " << a.size() <<
        " elements = [" << endl;
    for (int i=0; i<a.size(); i++)
        s << "    " << i << "  " << a[i] << endl;
    s << "]" << endl;
    return s;
}
```

Bemerkung:

- Der Iterator ist als Zeiger auf ein Feldelement realisiert.
- Die Schleife

  ```
  for (Array<int>::Iterator i=a.begin(); i!=a.end(); i++) ...
  ```

  entspricht nach Inlining der Methoden einfach

  ```
  for (int* p=a.p; p!=&a[100]; p=p+1) ...
  ```

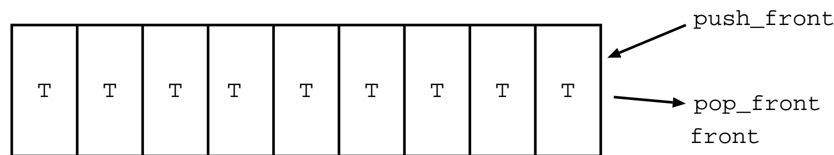
  und ist somit nicht langsamer als handprogrammiert!
- Man beachte auch die Definition von `MemberType`. Dies ist praktisch innerhalb eines Template `template <class C>`, wo der Datentyp eines Containers `C` dann als `C::MemberType` erhalten werden kann.

Programm: Gleichzeitige Verwendung `DLList/Array` (`UseBoth.cc`):

```
#include<cassert>
#include<iostream>
using namespace std;

#include"Array.cc"
#include"DLL.cc"
#include"Zufall.cc"

int main () {
    Zufall z(87124);
    Array<int> a(5);
    DLList<int> l;

    // Erzeuge Array und Liste mit 5 Zufallszahlen
    for (int i=0; i<5; i=i+1) a[i] = z.ziehe_zahl();
    for (int i=0; i<5; i=i+1)
        l.insert(l.end(), z.ziehe_zahl());

    // Benutzung
    for (Array<int>::Iterator i=a.begin();
         i!=a.end(); i++)
        cout << *i << endl;
    cout << endl;
    for (DLList<int>::Iterator i=l.begin();
         i!=l.end(); i++)
        cout << *i << endl;
}
```

Bemerkung: Die STL-Version von `Array` erhält man mit `#include<vector>`. Die Klassenschablone heißt `vector` anstatt `Array`.

19.6 Stack



Schnittstelle:

- Konstruktion eines Stack.
- Einfügen eines Elementes vom Typ `T` oben (`push`).
- Entfernen des obersten Elementes (`pop`).
- Inspektion des obersten Elementes (`top`).
- Test ob Stack voll oder leer (`empty`).

Programm: Implementation über DLList (`Stack.cc`)

```
template<class T>
class Stack : private DLList<T> {
public :
    // Default-Konstruktoren + Zuweisung OK

    bool empty () {return DLList<T>::empty();}
    void push (T t) {
        insert(begin(),t);
    }
    T top () {return *begin();}
    void pop () {erase(begin());}
} ;
```

Bemerkung:

- Wir haben den Stack als Spezialisierung der doppelt verketteten Liste realisiert. Etwas effizienter wäre die Verwendung einer einfach verketteten Liste gewesen.
- Auffallend ist, dass die Befehle `top`/`pop` getrennt existieren (und `pop` keinen Wert zurückliefert). Verwendet werden diese Befehle nämlich meist gekoppelt, so dass auch eine Kombination `pop` ← `top`+`pop` nicht schlecht wäre (siehe Bastian-Skript).

Programm: Anwendung: (`UseStack.cc`)

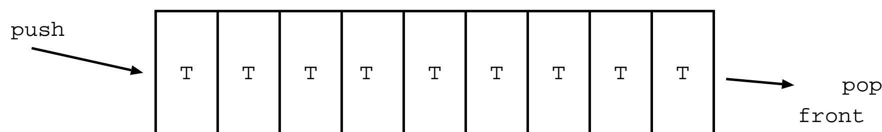
```
#include<cassert>
#include<iostream>
using namespace std;
```

```
#include"DLL.cc"
#include"Stack.cc"

int main ()
{
    Stack<int> s1;
    for (int i=1; i<=5; i++)
        s1.push(i);

    Stack<int> s2(s1);
    s2 = s1;
    while (!s2.empty())
    {
        cout << s2.top() << endl;
        s2.pop();
    }
}
```

Bemerkung: Die STL-Version erhält man durch `#include<stack>`. Die Klassenschablone heißt dort `stack` und hat im wesentlichen dieselbe Schnittstelle.

19.7 Queue

Eine Queue ist eine Struktur, die Einfügen an einem Ende und Entfernen nur am anderen Ende erlaubt:



Anwendung: Warteschlangen.

Schnittstelle:

· Konstruktion einer leeren Queue
· Einfügen eines Elementes vom Typ `T` am Ende
· Entfernen des Elementes am Anfang
· Inspektion des Elementes am Anfang
· Test ob Queue leer

Programm: (`Queue.cc`)

```
template<class T>
class Queue : private DLList<T> {
public :
    // Default-Konstruktoren + Zuweisung OK
    bool empty () {
```

```
        return DLList<T>::empty();
    }
    T front () {
        return *begin();
    }
    T back () {
        return *rbegin();
    }
    void push (T t) {
        insert(end(),t);
    }
    T pop () {
        erase(begin());
    }
} ;
```

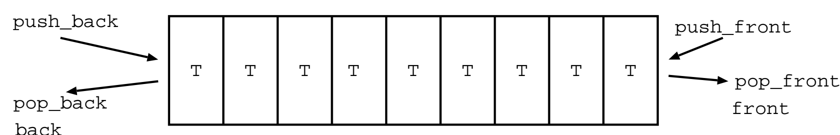
Bemerkung: Die STL-Version erhält man durch `#include<queue>`. Die Klassenschablone heißt dort `queue` und hat im wesentlichen dieselbe Schnittstelle wie `Queue`.

Programm: Anwendung der STL-Version: (`UseQueueSTL.cc`)

```
#include<queue>
#include<iostream>
using namespace std;

int main () {
    queue<int> q;
    for (int i=1; i<=5; i++)
        q.push(i);

    while (!q.empty()) {
        cout << q.front() << endl;
        q.pop();
    }
}
```

19.8 DeQueue

Eine DeQueue (*double ended queue*) ist eine Struktur, die Einfügen und Entfernen an beiden Enden erlaubt:



Schnittstelle:

· Konstruktion einer leeren `DeQueue`
· Einfügen eines Elementes vom Typ `T` am Anfang oder Ende
· Entfernen des Elementes am Anfang oder am Ende
· Inspektion des Elementes am Anfang oder Ende
· Test ob `DeQueue` leer

Programm: (`DeQueue.cc`)

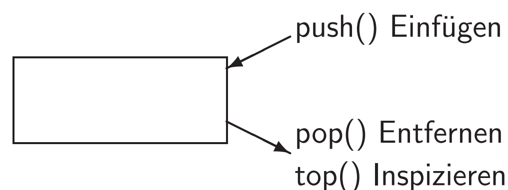
```
template<class T>
class DeQueue : private DLList<T> {
public :
    // Default-Konstruktoren + Zuweisung ok
    bool empty ();
    void push_front (T t);
    void push_back (T t);
    T pop_front ();
    T pop_back ();
    T front ();
    T back ();
} ;
```

Bemerkung: Die STL-Version erhält man auch hier mit `#include<queue>`. Die Klassenschablone heißt `deque`.

19.9 Prioritätswarteschlangen

Bezeichnung: Eine *Prioritätswarteschlange* ist eine Struktur, in die man Objekte des Grundtyps `T` einfüllen kann und von der jeweils das *kleinste* (MinPriorityQueue) bzw. das *größte* (MaxPriorityQueue) der eingegebenen Elemente als nächstes entfernt werden kann. Bei gleich großen Elementen verhält sie sich wie eine Queue.

Bemerkung: Auf dem Grundtyp `T` muß dazu die Relation $<$ mittels dem `operator<` zur Verfügung stehen.



Schnittstelle:

- Konstruktion einer leeren MinPriorityQueue.
- Einfügen eines Elementes vom Typ `T` (`push`).

- Entfernen des kleinsten Elementes im Container (`pop`).
- Inspektion des kleinsten Elementes im Container (`top`).
- Test ob MinPriorityQueue leer (`empty`).

Programm: Hier die Klassendefinition:

```
template<class T>
class MinPriorityQueue : private DLList<T> {
private:
    typename DLList<T>::Iterator find_minimum();
public :
    // Default-Konstruktoren + Zuweisung OK
    bool empty ();
    void push (T t);  // Einfuegen
    void pop ();      // Entferne kleinstes
    T top ();         // Inspiziere kleinstes
} ;
```

Und die Implementation:

```
template<class T>
bool MinPriorityQueue<T>::empty () {
    return DLList<T>::empty();
}

template<class T>
void MinPriorityQueue<T>::push (T t) {
    insert(begin(),t);
}

template<class T>
typename DLList<T>::Iterator MinPriorityQueue<T>::find_minimum () {
    typename DLList<T>::Iterator min=begin();
    for (typename DLList<T>::Iterator i=begin(); i!=end(); i++)
        if (*i<*min) min=i;
    return min;
}

template<class T>
inline void MinPriorityQueue<T>::pop () {
    erase(find_minimum());
}

template<class T>
inline T MinPriorityQueue<T>::top () {
    return *find_minimum();
}
```

Bemerkung:

- Unsere Implementierung arbeitet mit einer einfach verketteten Liste. Das Einfügen hat Komplexität $O(1)$, das Entfernen/Inspizieren jedoch $O(n)$.
- Bessere Implementationen verwenden einen Heap, was zu einem Aufwand der Ordnung $O(\log n)$ führt.
- Analog ist die Implementation der `MaxPriorityQueue`.

Bemerkung: Die STL-Version erhält man auch durch `#include<queue>`. Die Klassenschablone heißt `priority_queue` und implementiert eine `MaxPriorityQueue`. Man kann allerdings den Vergleichsoperator auch als Template-Parameter übergeben (etwas lästig).

19.10 Set

Ein Set (Menge) ist ein Container mit folgenden Operationen:

- Konstruktion einer leeren Menge.
- Einfügen eines Elementes vom Typ `T`.
- Entfernen eines Elementes.
- Test auf Enthaltensein.
- Test ob Menge leer.

Programm: Klassendefinition:

```
template<class T>
class Set : private DLList<T> {
public :
    // Default-Konstruktoren + Zuweisung OK

    typedef typename DLList<T>::Iterator Iterator;
    Iterator begin();
    Iterator end();

    bool empty ();
    bool member (T t);
    void insert (T t);
    void remove (T t);
    // union, intersection, ... ?
} ;
```

Implementation:

```
template<class T>
typename Set<T>::Iterator Set<T>::begin() {
    return DLList<T>::begin();
}

template<class T>
typename Set<T>::Iterator Set<T>::end() {
    return DLList<T>::end();
}

template<class T>
bool Set<T>::empty () {
    return DLList<T>::empty();
}

template<class T>
inline bool Set<T>::member (T t) {
    return find(t)!=end();
}

template<class T>
inline void Set<T>::insert (T t)
{
    if (!member(t))
        DLList<T>::insert(begin(),t);
}

template<class T>
inline void Set<T>::remove (T t)
{
    typename DLList<T>::Iterator i = find(t);
    if (i!=end())
        erase(i);
}
```

Bemerkung:

- Die Implementierung hier basiert auf der doppelt verketteten Liste von oben (private Ableitung!).
- Einfügen, Suchen und Entfernen hat die Komplexität $O(n)$.
- Wir lernen später Implementierungen kennen, die den Aufwand $O(\log n)$ für alle Operationen haben.

19.11 Map

Bezeichnung: Eine Map ist ein assoziatives Feld, das Objekten eines Typs `Key` Objekte eines Typs `T` zuordnet.

Beispiel: Telephonbuch:

Meier	→	504423
Schmidt	→	162300
Müller	→	712364
Huber	→	8265498

Diese Zuordnung könnte man realisieren mittels:

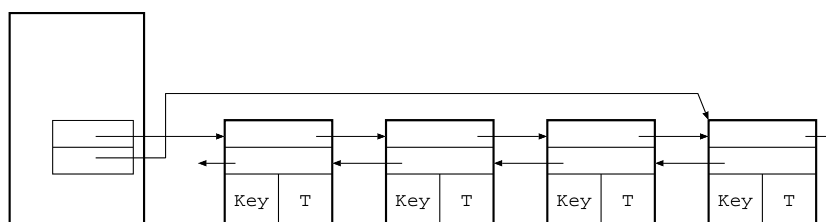
```
Map<string,int> telefonbuch;
telefonbuch["Meier"] = 504423;
...
```

Programm: Definition der Klassenschablone (`Map.cc`)

```
// Existiert schon als std::pair
// template<class Key, class T>
// struct pair {
//     Key first;
//     T second;
// } ;

template<class Key, class T>
class Map : private DLList<pair<Key,T> > {
public :

    T& operator[](const Key& k);

    typedef typename DLList<pair<Key,T> >::Iterator Iterator;
    Iterator begin () const;
    Iterator end () const;
    Iterator find (const Key& k);
} ;
```

Bemerkung:

- In dieser Implementation von Map werden je zwei Objekte der Typen `Key` (*Schlüssel*) und `T` (*Wert*) zu einem Paar vom Typ `pair<Key,T>` kombiniert und in eine doppelt verkettete Liste eingefügt:



- Ein Objekt vom Typ `Key` kann nur einmal in einem Paar vorkommen. (Daher ist das Telephonbuch kein optimales Beispiel.)
- Wir haben einen Iterator zum Durchlaufen des Containers.
- Auf dem Schlüssel muss der Gleichheitsoperator `operator==` definiert sein.
- `find(Key k)` liefert den Iterator für den Wert, ansonsten `end()`.
- Der Aufwand einer Suche ist wieder $O(n)$. Bald werden wir aber eine Realisierung von `Map` kennenlernen, die Einfügen und Suchen in $O(\log n)$ Schritten ermöglicht.

19.12 Anwendung: Huffman-Kode

Problem: Wir wollen eine Zeichenfolge, z.B.

'ABRACADABRASIMSALABIM'

durch eine Folge von Zeichen aus der Menge $\{0, 1\}$ darstellen (kodieren).

Dazu wollen wir jedem der 9 Zeichen aus der Eingabekette eine Folge von Bits zuzuordnen. Am einfachsten ist es, einen Kode fester Länge zu konstruieren. Mit n Bits können wir 2^n verschiedene Zeichen kodieren. Im obigem Fall genügen also 4 Bit, um jedes der 9 verschiedenen Zeichen in der Eingabekette zu kodieren, z. B.

A	0001	D	0100	M	0111
B	0010	I	0101	R	1000
C	0011	L	0110	S	1010

Die Zeichenkette wird dann kodiert als

$$\underbrace{0001}_{\text{A}}\underbrace{0010}_{\text{B}}\underbrace{1000}_{\text{R}}\ldots$$

Insgesamt benötigen wir $21 \cdot 4 = 84$ Bits (ohne die Übersetzungstabelle!).

Beobachtung: Kommen manche Zeichen häufiger vor als andere (wie etwa bei Texten in natürlichen Sprachen) so kann man Platz sparen, indem man Kodes variabler Länge verwendet.

Beispiel: Morsekode.

Beispiel: Für unsere Beispielzeichenkette 'ABRACADABRASIMSALABIM' wäre folgender Code gut:

A	1	D	010	M	100
B	10	I	11	R	101
C	001	L	011	S	110

Damit kodieren wir unsere Beispielkette als

$$\underbrace{1}_{\text{A}}\underbrace{10}_{\text{B}}\underbrace{101}_{\text{R}}\underbrace{1}_{\text{A}}\underbrace{001}_{\text{C}}\ldots$$

Schwierigkeit: Bei der Dekodierung könnte man diese Bitfolge auch interpretieren als

$$\underbrace{110}_{\text{S}}\underbrace{101}_{\text{R}}\underbrace{100}_{\text{M}}\ldots$$

Abhilfe: Es gibt zwei Möglichkeiten das Problem zu umgehen:

1. Man führt zusätzliche Trennzeichen zwischen den Zeichen ein (etwa die Pause beim Morsekode).

2. Man sorgt dafür, dass kein Kode für ein Zeichen der Anfang (*Präfix*) eines anderen Zeichens ist. Einen solchen Kode nennt man *Präfixkode*.

Frage: Wie sieht der optimale Präfixkode für eine gegebene Zeichenfolge aus, d. h. ein Kode der die gegebene Zeichenkette mit einer Bitfolge minimaler Länge kodiert.

Antwort: Huffmankodes! (Sie sind benannt nach ihrem Entdecker David Huffman, der auch die Optimalität dieser Codes gezeigt hat.)

Beispiel: Für unsere Beispiel-Zeichenkette ist ein solcher Huffmankode

A	11	D	10011	M	000
B	101	I	001	R	011
C	1000	L	10010	S	010

Die kodierte Nachricht lautet hier

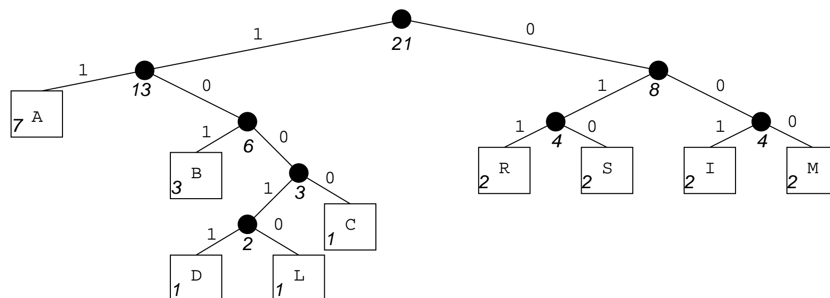
$$1110101111110001110011111010111101000100001011100101110100100\,0$$

und hat nur noch 61 Bits!

19.12.1 Trie

Einem Präfixkode kann man einen binären Baum zuordnen, der Trie genannt wird. In den Blättern stehen die zu kodierenden Zeichen. Ein Pfad von der Wurzel zu einem Blatt kodiert das entsprechende Zeichen.

Beispiel:



Bemerkung: Zeichen, die häufig vorkommen, stehen nahe bei der Wurzel. Zeichen, die seltener vorkommen, stehen tiefer im Baum.

19.12.2 Konstruktion von Huffmankodes

1. Zähle die Häufigkeit jedes Zeichens in der Eingabefolge. Erzeuge für jedes Zeichen einen Knoten mit seiner Häufigkeit. Packe alle Knoten in eine Menge E.

2. Solange die Menge E nicht leer ist: Entferne die zwei Knoten l und r mit geringster Häufigkeit aus E. Erzeuge einen neuen Knoten n mit l und r als Söhnen und der Summe der Häufigkeiten beider Söhne. Ist E leer ist n die Wurzel des Huffmanbaumes, sonst stecke n in E.

19.12.3 Implementation

Programm: (Huffman-Kodierung mit STL)

```
#include <functional>
#include <iostream>
#include <map>
#include <queue>
#include <string>

using namespace std;

// There are no general binary trees in the STL.
// But we do not use much of this structure
// anyhow...
struct node {
    struct node *left;
    struct node *right;
    char symbol;
    int weight;
    node (char c, int i) {  // leaf constructor
        symbol = c;
        weight = i;
        left = right = 0;
    }
    node (node* l, node *r) { // internal node constructor
        symbol = 0;
        weight = l->weight + r->weight;
        left = l;
        right = r;
    }
    bool isleaf() {return symbol!=0;}
    bool operator> (const node &a) const {
        return weight > a.weight;
    }
};

// construct the Huffman trie for this message
node *huffman_trie (string message) {
    // count multiplicities
    map<char,int> cmap;
    for (string::iterator i=message.begin(); i!=message.end(); i++)
        if (cmap.find(*i)!=cmap.end())
            cmap[*i]++;
        else
            cmap[*i]=1;
```

```
    // generate leaves with multiplicities
    priority_queue<node, vector<node>, greater<node> > q;
    for (map<char,int>::iterator i=cmap.begin(); i!=cmap.end(); i++)
        q.push (node (i->first, i->second));

    // build Huffman tree (trie)
    while (q.size()>1)
    {
        node *left = new node(q.top());
        q.pop();
        node *right = new node(q.top());
        q.pop();
        q.push(node (left, right));
    }
    return new node(q.top());
}

// recursive filling of the encoding table 'code'
void fill_encoding_table (string s, node *i,
                          map<char,string>& code) {
    if (i->isleaf())
        code[i->symbol]=s;
    else
    {
        fill_encoding_table (s+"0", i->left, code);
        fill_encoding_table (s+"1", i->right, code);
    }
}

// encoding
string encode (map<char,string> code, string& message) {
    string encoded = "";
    for (string::iterator i=message.begin(); i!=message.end(); i++)
        encoded += code[*i];
    return encoded;
}

// decoding
string decode (node* trie, string& encoded) {
    string decoded = "";
    node* node = trie;
    for (string::iterator i=encoded.begin(); i!=encoded.end(); i++)
    {
        if (!node->isleaf())
            node = (*i=='0') ? node->left : node->right;
        if (node->isleaf())
```

```
            {
                decoded.push_back(node->symbol);
                node = trie;
            }
        }
    return decoded;
}

int main () {
    string message = "ABRACADABRASIMSALABIM";

    // generate Huffman trie
    node* trie = huffman_trie(message);

    // generate and show encoding table
    map<char,string> table;
    fill_encoding_table ("", trie, table);
    for (map<char,string>::iterator i=table.begin(); i!=table.end();
        i++)
        cout << i->first << "␣" << i->second << endl;

    // encode and decode
    string encoded = encode(table, message);
    cout << "Encoded:␣" << encoded <<
        "␣[" << encoded.size() << "␣Bits]" << endl ;
    cout << "Decoded:␣" << decode(trie, encoded) << endl;
}
```

Ausgabe: Wir erhalten einen anderen Huffman-Code als oben angegeben (der aber natürlich genauso effizient kodiert):

```
A 11
B 100
C 0010
D 1010
I 010
L 0011
M 1011
R 011
S 000
Encoded: 1110001111001011101011100011... [61 Bits]
Decoded: ABRACADABRASIMSALABIM
```

20 Effiziente Algorithmen und Datenstrukturen

Beobachtung: Einige der bisher vorgestellten Algorithmen hatten einen sehr hohen Aufwand (z.B. $O(n^2)$ bei Bubblesort, $O(n)$ bei Einfügen/Löschen aus der Priority-Queue). Wir haben auch gesehen, dass z.B. die STL-Sortierung viel schneller ist.

Frage: Wie erreicht man diese Effizienz?

Ziel: In diesem Kapitel lernen wir Algorithmen und Datenstrukturen kennen, mit denen man hohe (in vielen Fällen sogar optimale) Effizienz erreichen kann.

20.1 Heap

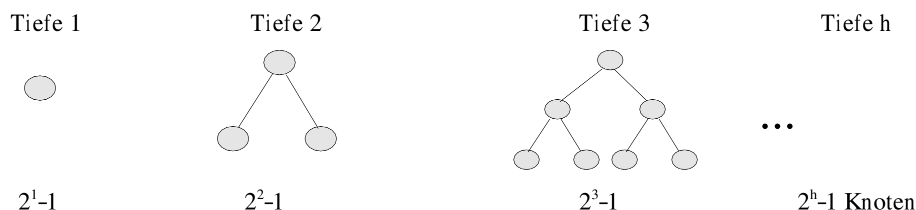
Die Datenstruktur Heap erlaubt es, Einfügen und Löschen in einer *Prioritätswarteschlange* mit $O(\log n)$ Operationen zu realisieren. Sie ist auch Grundlage eines schnellen Sortierverfahrens (Heapsort).

Definition: Ein Heap ist

- ein *fast vollständiger binärer Baum*
- Jedem Knoten ist ein Schlüssel zugeordnet. Auf der Menge der Schlüssel ist eine totale Ordnung (z.B. durch einen Operator $\leq$) definiert.
  Totale Ordnung: reflexiv ($a \leq a$), transitiv ($a \leq b, b \leq c \Rightarrow a \leq c$), total ($a \leq b \vee b \leq a$).
- Der Baum ist partiell geordnet, d.h. der Schlüssel jeden Knotens ist *nicht kleiner* als die Schlüssel in seinen Kindern (Heap-Eigenschaft).

Bezeichnung: Ein *vollständiger binärer Baum* ist ein

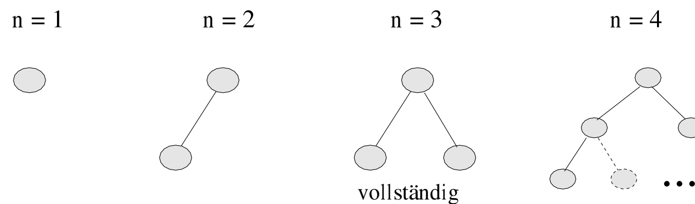
- binärer Baum der Tiefe h mit maximaler Knotenzahl,
- bei dem sich alle Blätter auf der gleichen Stufe befinden.



Bezeichnung: Ein *fast vollständiger binärer Baum* ist ein binärer Baum mit folgenden Eigenschaften:

- alle Blätter sind auf den beiden höchsten Stufen
- maximal ein innerer Knoten hat nur ein Kind
- Blätter werden von links nach rechts aufgefüllt.

Ein solcher Baum mit n Knoten hat eine eindeutige Struktur:

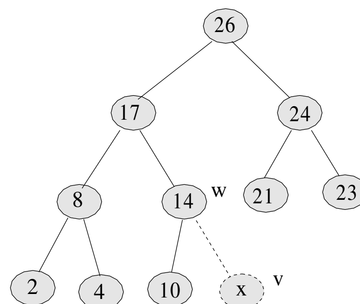


20.1.1 Einfügen

Problem: Gegeben ist ein Heap mit n Elementen und neues Element x. Konstruiere daraus einen um x erweiterter Heap mit $n + 1$ Elementen.

Beobachtung: Die Struktur des Baumes mit $n+1$ Elementen liegt fest. Wenn man daher x an der neuen Position v einfügt, so kann die Heapeigenschaft nur im Knoten $w = \mathrm{Vater}(v)$ verletzt sein.

Beispiel:



Algorithmus: Wiederherstellen der Heapeigenschaft in maximal $\log_2 n$ Vertauschungen:

Falls $Inhalt(w) < Inhalt(v)$ dann
    tausche Inhalt
    Falls w nicht die Wurzel ist:
        setze $w = Vater(w); v = Vater(v);$
    sonst $\longrightarrow$ fertig
sonst $\longrightarrow$ fertig

20.1.2 Reheap

Die im folgenden beschriebene Reheap-Operation wird beim Entfernen der Wurzel gebraucht.

Problem: Gegeben ist ein fast vollständiger Baum mit Schlüsseln, so dass die Heapeigenschaft in allen Knoten exklusive der Wurzel gilt. Ziel ist die Transformation in einen echten Heap.

Algorithmus:

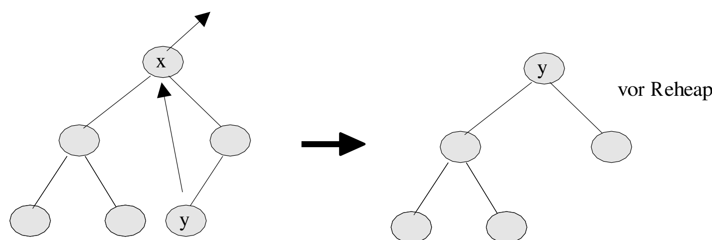
1. Tausche Schlüssel in der Wurzel mit dem größeren der beiden Kinder.
2. Wenn die Heap-Eigenschaft für dieses Kind nicht erfüllt ist, so wende den Algorithmus rekursiv an, bis ein Blatt erreicht wird.

20.1.3 Entfernen des Wurzelelements

Algorithmus:

- Ersetze den Wert in der Wurzel (Rückgabewert) durch das letzte Element des fast vollständigen binären Baumes.
- Verkleinere den Heap und rufe Reheap auf

Beispiel:



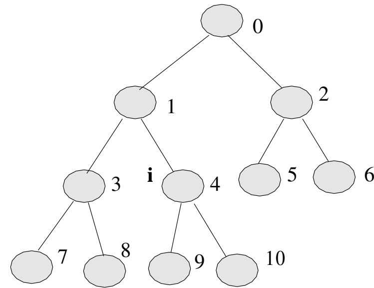
20.1.4 Komplexität

Ein fast vollständiger Baum mit n Knoten hat genau $\lceil \mathrm{ld}(n+1) \rceil$ Ebenen. Somit benötigt das Einfügen maximal $\lceil \mathrm{ld}(n+1) \rceil - 1$ Vergleiche und die Operation reheap maximal $2(\lceil \mathrm{ld}(n+1) \rceil - 1)$ Vergleiche.

20.1.5 Datenstruktur

Beobachtung: Die Knoten eines fast vollständigen binären Baumes können sehr effizient in einem Feld gespeichert werden:

Numeriert man die Knoten wie folgt:



Dann gilt für Knoten i: Linkes Kind: $2i+1$
Rechtes Kind: $2(i+1)$
Vater: $\lfloor \frac{i-1}{2} \rfloor$

20.1.6 Implementation

Programm: Definition und Implementation (`Heap.cc`)

```
template<class T>
class Heap {
public:
    bool empty ();
    void push (T x);
    void pop ();
    T top ();
private:
    vector<T> data;
    void reheap (int i);
} ;

template<class T>
void Heap<T>::push (T x) {
    int i = data.size();
    data.push_back(x);
    while (i>0 && data[i]>data[(i-1)/2])
    {
        swap(data[i],data[(i-1)/2]);
        i = (i-1)/2;
    }
}

template <class T>
void Heap<T>::reheap (int i)  {
    int n = data.size();
    while (2*i+1<n)
    {
        int l = 2*i+1;
        int r = l+1;
```

```
        int k = ((r<n) && (data[r]>data[l])) ? r : l;
        if (data[k]<=data[i]) break;
        swap(data[k], data[i]);
        i = k;
    }
}

template<class T>
void Heap<T>::pop () {
    swap(data.front(), data.back());
    data.pop_back();
    reheap(0);
}

template<class T>
T Heap<T>::top () {
    return data[0];
}

template<class T>
inline bool Heap<T>::empty () {
    return data.size()==0;
}
```

Programm: Anwendung (`Heap.cc`)

```
#include<vector>
#include<iostream>
using namespace std;

#include"Heap.cc"
#include"Zufall.cc"

int main ()
{
    Zufall z(87124);
    Heap<int> h;

    for (int i=0; i<5; i=i+1) {
        int k = z.ziehe_zahl();
        cout << k << endl;
        h.push(k);
    }
    cout << endl;
    while (!h.empty()) {
        cout << h.top() << endl;
        h.pop();
```

```
    }
}
```

Beobachtung: Mit Hilfe der Heap-Struktur lässt sich sehr einfach ein recht guter Sortieralgorithmus erzeugen. Dazu ordnet man Elemente einfach in einen Heap ein und extrahiert sie wieder. Dies wird später noch genauer beschrieben.

21 Sortieren

21.1 Das Sortierproblem

Gegeben: Eine „Liste“ von Datensätzen $(D_0, \ldots, D_{n-1})$. Zu jedem Datensatz D_i gehört ein Schlüssel $k_i = k(D_i)$. Auf der Menge der Schlüssel sei eine *totale* Ordnung durch einen Operator $\leq$ definiert.

Definition: Eine Permutation von $I = \{0, \ldots, n-1\}$ ist eine bijektive Abbildung $\pi : I \to I$.

Gesucht: Eine Permutation $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$, so dass gilt

$$k_{\pi(0)} \leq \ldots \leq k_{\pi(n-1)}$$

Bemerkung: In der Praxis hat man:

- Die Datensätze sind normalerweise in einer Liste oder einem Feld gespeichert. Wir betrachten im folgenden den Fall des Felds.
- Oft braucht man die Permutation π nicht weiter, und es reicht aus, als Ergebnis einer Sortierfunktion eine sortierte Liste/Feld zu erhalten.
- Die Relation $\leq$ wird durch einen beliebigen Vergleichsoperator definiert.
- Für große Datensätze sortiert man lieber ein Feld von Zeigern.
- Internes Sortieren: Alle Datensätze sind im Hauptspeicher.
- Externes Sortieren: Sortieren von Datensätzen, die auf Platten, Bändern gespeichert sind.

21.2 Sortierverfahren mit quadratischer Komplexität

21.2.1 Selectionsort (Sortieren durch Auswahl)

Idee:

- Gegeben sei ein Feld $a = (a_0, \ldots, a_{n-1})$ der Länge n.
- Suche das Minimum im Feld und tausche mit dem erstem Element.
- Danach steht die kleinste der Zahlen ganz links, und es bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Programm: Selectionsort (`Selectionsort.cc`)

```
template <class C>
void selectionsort (C& a)
{
    for (int i=0; i<a.size()-1; i=i+1)
    {   // i Elemente sind sortiert
        int min = i;
        for (int j=i+1; j<a.size(); j=j+1)
            if (a[j]<a[min]) min=j;
        swap(a[i],a[min]);
    }
}
```

Bemerkung:

- Komplexität: $\frac{n^2}{2}$ Vergleiche, n Vertauschungen $\to O(n^2)$.
- Die Zahl von Datenbewegungen ist optimal, das Verfahren ist also zu empfehlen, wenn möglichst wenige Datensätze bewegt werden sollen.

21.2.2 Bubblesort

Idee: (Siehe den Abschnitt über Effizienz generischer Programmierung.)

- Gegeben sei ein Feld $a = (a_0, \ldots, a_{n-1})$ der Länge n.
- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden.
- Nach einem solchen Durchlauf steht die größte der Zahlen ganz rechts, und es bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Programm: Bubblesort mit STL (`Bubblesort.cc`)

```
template <class C>
void bubblesort (C& a) {
    for (int i=a.size()-1; i>=0; i--)
        for (int j=0; j<i; j=j+1)
            if (a[j+1]<a[j])
                swap(a[j+1], a[j]);
}
```

Bemerkung:

- Komplexität: in führender Ordnung $\frac{n^2}{2}$ Vergleiche, $\frac{n^2}{2}$ Vertauschungen

21.2.3 Insertionsort (Sortieren durch Einfügen)

Beschreibung: Der bereits sortierte Bereich liegt links im Feld und das nächste Element wird jeweils soweit nach links bewegt, bis es an der richtigen Stelle sitzt.

Programm: Insertionsort mit STL (`Insertionsort.cc`)

```
template <class C>
void insertionsort (C& a) {
    for (int i=1; i<a.size(); i=i+1) {
        // i Elemente sind sortiert
        int j=i;
        while (j>0 && a[j-1]>a[j]) {
            swap(a[j], a[j-1]);
            j=j-1;
        }
    }
}
```

Bemerkung:

- Komplexität: $\frac{n^2}{2}$ Vergleiche, $\frac{n^2}{2}$ Vertauschungen $\to O(n^2)$.
- Ist das Feld bereits sortiert, endet der Algorithmus nach $O(n)$ Vergleichen. Sind in ein bereits sortiertes Feld mit n Elementen m weitere Elemente einzufügen, so gelingt dies mit Insertionsort in $O(nm)$ Operationen. Dies ist optimal für sehr kleines m ($m \ll \log n$).

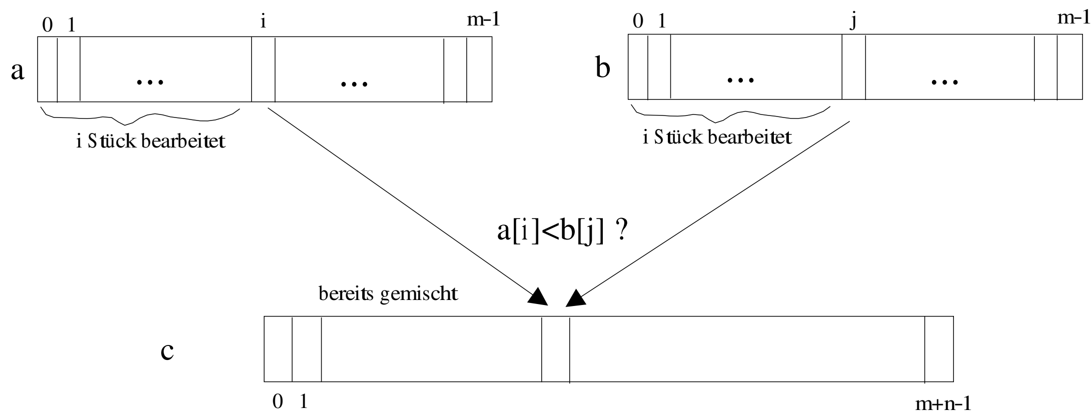
21.3 Sortierverfahren optimaler Ordnung

In diesem Abschnitt betrachten wir Sortierverfahren, die den Aufwand $O(n \log n)$ haben. Man kann zeigen, dass dieser Aufwand für allgemeine Listen von Datensätzen optimal ist.

Erinnerung: $\log x = \log_e x$, $\operatorname{ld} x = \log_2 x$. Wegen $\operatorname{ld} x = \frac{\log x}{\log 2} = 1.44\ldots \cdot \log x$ gilt $O(\log n) = O(\operatorname{ld} n)$.

21.3.1 Mergesort (Sortieren durch Mischen)

Beobachtung: Zwei bereits sortierte Felder der Länge m bzw. n können sehr leicht (mit Aufwand $O(m+n)$) zu einem sortierten Feld der Länge $m+n$ „vermischt“ werden:



Dies führt zu folgendem Algorithmus vom Typ „Divide and Conquer“:

Algorithmus:

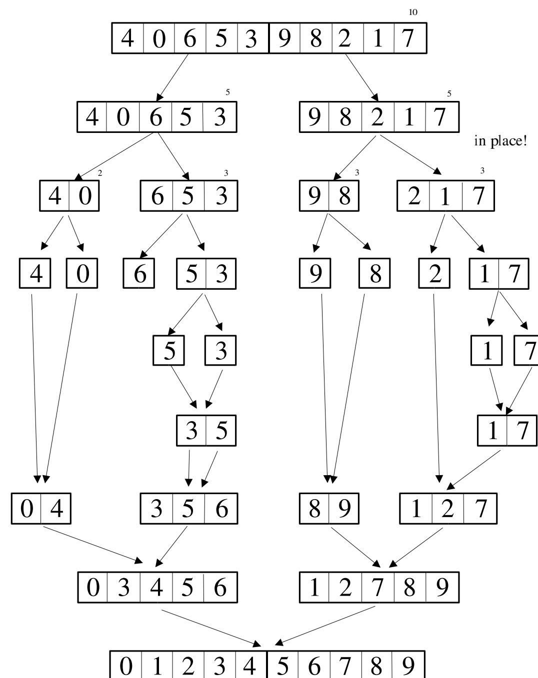
- Gegeben ein Feld der Länge n.
- Ist $n = 1$, so ist nichts zu tun, sonst
- zerlege in zwei Felder $a1$ mit Länge $n1 = n/2$ (ganzzahlige Division) und $a2$ mit Länge $n2 = n - n1$),
- sortiere $a1$ und $a2$ (Rekursion) und
- mische $a1$ und $a2$.

Programm: Mergesort mit STL (`Mergesort.cc`)

```
template <class C>
void rec_merge_sort (C& a, int o, int n)
{    // sortiere Eintraege [o,o+n-1]
     if (n==1) return;

     // teile und sortiere rekursiv
     int n1=n/2;
     int n2=n-n1;
     rec_merge_sort(a,o,n1);
     rec_merge_sort(a,o+n1,n2);

     // zusammenfuegen
     C b(n); // Hilfsfeld
     int i1=o, i2=o+n1;
     for (int k=0; k<n; k=k+1)
          if ((i2>=o+n) || (i1<o+n1 && a[i1]<=a[i2]))
               b[k] = a[i1++];
          else
               b[k] = a[i2++];
```

```
    // umkopieren
    for (int k=0; k<n; k=k+1) a[o+k] = b[k];
}

template <class C>
void mergesort (C& a)
{
    rec_merge_sort(a,0,a.size());
}
```

Beispiel:



Bemerkung:

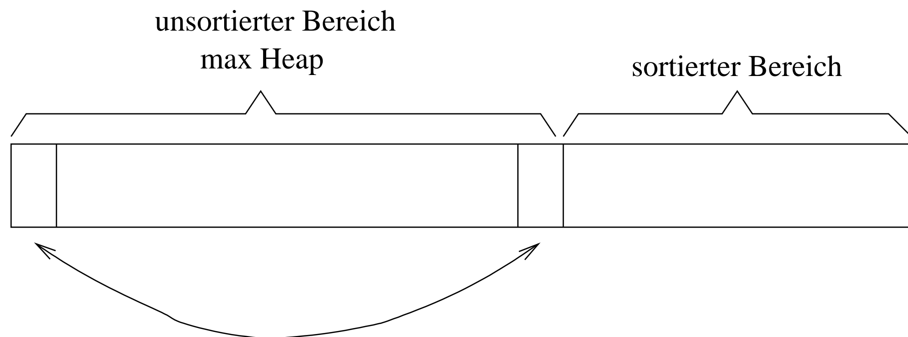
- Mergesort benötigt zusätzlichen Speicher von der Größe des zu sortierenden Felds.
- Die Zahl der Vergleiche ist aber (in führender Ordnung) $n \operatorname{ld} n$.
  Beweis für $n = 2^k$: Für die Zahl der Vergleiche $V(n)$ gilt (Induktion)

$$
\begin{aligned}
V(1) &= 0 = 1 \operatorname{ld} 1 \\
V(n) &= 2V(\frac{n}{2}) + n - 1 \leq 2(k-1)\frac{n}{2} + n \leq kn
\end{aligned}
$$

  Man kann zeigen, dass dies optimal ist.
- Mergesort ist stabil, d.h. Datensätze mit gleichen Schlüsseln verbleiben in derselben Reihenfolge wie zuvor

21.3.2 Heapsort

Idee: Transformiere das Feld in einen Heap, und dann wieder in eine sortierte Liste. Wegen der kompakten Speicherweise für den Heap kann dies ohne zusätzlichen Speicherbedarf geschehen, indem man das Feld wie folgt unterteilt:



Bemerkung:

- Die Transformation des Felds in einen Heap kann auf zwei Weisen geschehen:
  1. Der Heap wird von vorne durch `push` aufgebaut.
  2. Der Heap wird von hinten durch `reheap` aufgebaut.

  Da wir `reheap` sowieso für die pop-Operation brauchen, wählen wir die zweite Variante.
- Heapsort hat in führender Ordnung die Komplexität von $2n \operatorname{ld} n$ Vergleichen. Der zusätzliche Speicheraufwand ist unabhängig von n (in-situ-Verfahren)!
- Im Gegensatz zu Mergesort ist Heapsort nicht stabil.

Programm: Heapsort mit STL (`Heapsort.cc`)

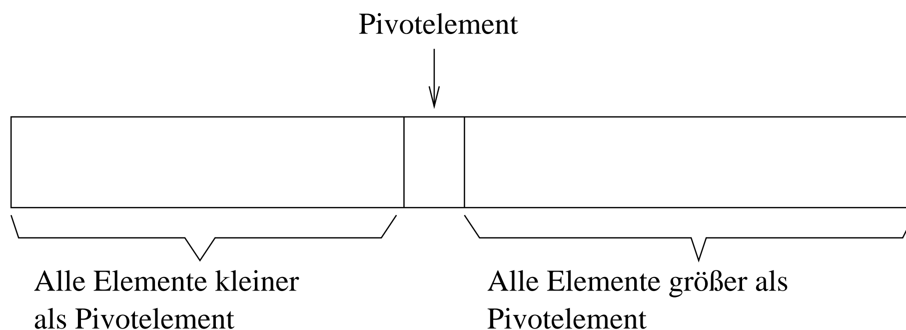
```
template <class C>
inline void reheap (C& a, int n, int i) {
    while (2*i+1<n)
    {
        int l = 2*i+1;
        int r = l+1;
        int k = ((r<n) && (a[r]>a[l])) ? r : l;
        if (a[k]<=a[i]) break;
        swap(a[k], a[i]);
        i = k;
    }
}

template <class C>
void heapsort (C& a)
```

```
{
    // build the heap by reheaping from the rear
    for (int i=a.size()-1; i>=0; i--)
        reheap(a, a.size(), i);
    // build the sorted list by popping the heap
    for (int i=a.size()-1; i>=0; i--) {
        swap(a[0],a[i]);
        reheap(a, i, 0);
    }
}
```

21.3.3 Quicksort

Beobachtung: Das Hauptproblem bei Mergesort war das speicheraufwendige Mischen. Man könnte es vermeiden, wenn man das Feld so in zwei (möglichst gleichgroße) Teile unterteilen könnte, das alle Elemente des linken Teilfelds kleiner oder gleich allen Elementen des rechten Teilfeldes sind.

Idee: Wähle „zufällig" ein beliebiges Element $q \in \{0, \dots, n-1\}$ aus, setze Pivot$= a[q]$ und zerlege das Feld so, dass das Eingabefeld folgende Gestalt hat:



Programm: Quicksort mit STL (`Quicksort.cc`)

```
template <class C>
int qs_partition (C& a, int l, int r, int q) {
    swap(a[q],a[r]);
    q=r;        // Pivot ist jetzt ganz rechts
    int i=l-1, j=r;

    while (i<j) {
        i=i+1; while (i<j && a[i]<=a[q]) i=i+1;
        j=j-1; while (i<j && a[j]>=a[q]) j=j-1;
        if (i<j)
            swap(a[i],a[j]);
        else
            swap(a[i],a[q]);
    }
```

```
    return i; // endgueltige Position des Pivot
}

template <class C>
void qs_rec (C& a, int l, int r) {
    if (l<r) {
        int i=qs_partition(a,l,r,r);
        qs_rec(a,l,i-1);
        qs_rec(a,i+1,r);
    }
}

template <class C>
void quicksort (C& a) {
    qs_rec(a,0,a.size()-1);
}
```

Bemerkung:

- Man kann im allgemeinen *nicht* garantieren, dass beide Hälften gleich groß sind.
- Im schlimmsten Fall wird das Pivotelement immer so gewählt, dass man ein einelementiges Teilfeld und den Rest als Zerlegung erhält. Dann hat Quicksort den Aufwand $O(n^2)$.
- Im besten Fall ist die Zahl der Vergleiche so gut wie Mergesort.
- Im „Mittel“ erhält man in führender Ordnung $1.386\, n \operatorname{ld} n$ Vergleiche.
- Auch Quicksort ist nicht stabil.

21.3.4 Anwendung

Mit folgendem Programm kann man die verschiedenen Sortierverfahren ausprobieren:

Programm:

```
#include<iostream>
#include<vector>
using namespace std;

#include"Bubblesort.cc"
#include"Selectionsort.cc"
#include"Insertionsort.cc"
#include"Mergesort.cc"
#include"Heapsort.cc"
#include"Quicksort.cc"
```

```
int main ()
{
    int n = 10000000;
    vector<int> a(n);
    for (int i=0; i<n; i++)
        a[i] = i*(n-i);
    quicksort(a);
    //for (int i=0; i<n; i++)
    //cout << a[i] << " ";
    //cout << endl;
}
```

21.4 Suchen

21.4.1 Binäre Suche in einem Feld

Idee: In einem sortierten Feld kann man Elemente durch sukzessives Halbieren schnell finden.

Bemerkung:

- Aufwand: in jedem Schritt wird die Länge des Suchbereichs halbiert. Der Aufwand beträgt daher $\lceil \mathrm{ld}(n) \rceil$ Vergleiche, dann ist man an einem Blatt. Anschließend braucht man noch einen Vergleich, um zu prüfen, ob das Blatt das gesuchte Element ist. $\Rightarrow \lceil \mathrm{ld}(n) \rceil + 1$ Vergleiche.
- Die binäre Suche ermöglicht also auch die Aussage, dass ein Element nicht enthalten ist!

Programm: Nicht-rekursive Formulierung (`Binsearch.cc`)

```
template <class C>
int binsearch (C& a, typename C::value_type x)
{   // returns either index (if found) or -1
    int l = 0;
    int r = a.size()-1;
    while (1) {
        int m = (l+r)/2;
        if ((m==l) || (m==r))
            return (a[m]==x) ? m : -1;
        if (x<a[m])
            r = m;
        else
            l = m;
    }
}
```

Bemerkung:

- Die binäre Suche beschleunigt nur das Finden.
- Einfügen und Löschen haben weiterhin Aufwand $O(n)$, da Feldelemente verschoben werden müssen.
- Binäre Suche geht auch nicht mit einer Liste (kein wahlfreier Zugriff).

21.4.2 Binäre Suchbäume

Beobachtung: Die binäre Suche im Feld kann als Suche in einem *binären Suchbaum* interpretiert werden (der aber in einem Feld abgespeichert wurde).

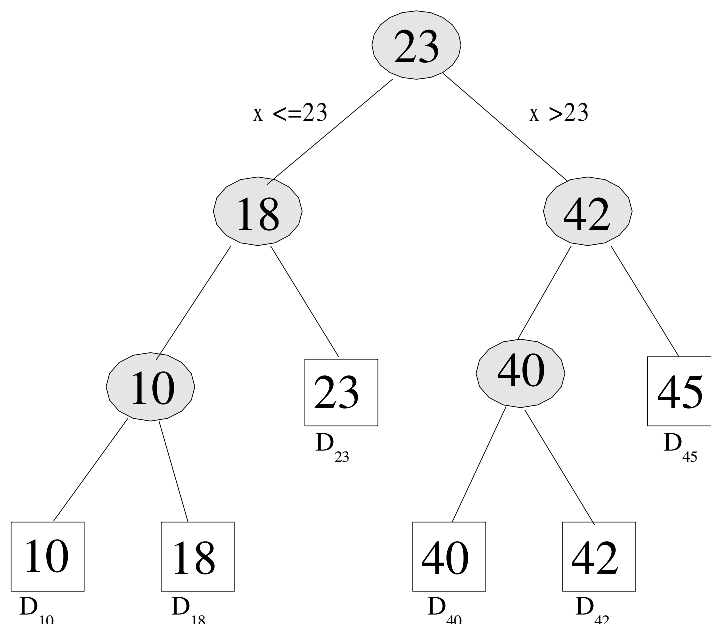
Idee: Die Verwendung einer echten Baum-Datenstruktur kann schnelles Einfügen und Entfernen von Knoten ermöglichen.

Definition: Ein binärer Suchbaum ist ein binärer Baum, in dessen Knoten Schlüssel abgespeichert sind und für den die Suchbaumeigenschaft gilt:

> Der Schlüssel in jedem Knoten ist größer als alle Schlüssel im linken Teilbaum und kleiner als alle Schlüssel im rechten Teilbaum.

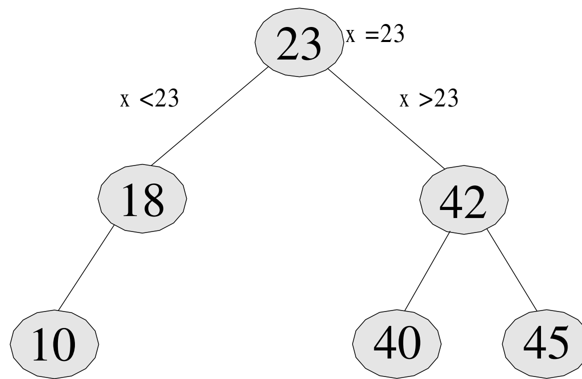
Bemerkung: Der Aufwand einer Suche entspricht der *Höhe* des Baumes.

Variante 1



- *Innere* Knoten enthalten nur Schlüssel
- (Verweise auf) Datensätze sind in den *Blättern* des Baumes gespeichert.

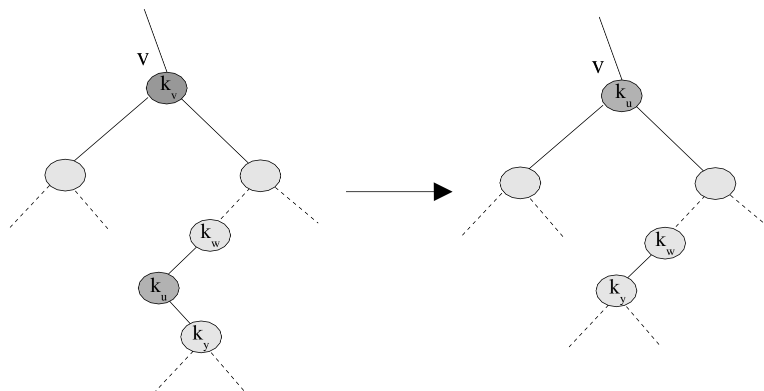
Variante 2 Man speichert Schlüssel und Datensatz genau einmal in einem Knoten:



Bemerkung: Innere Knoten unterscheiden sich von Blättern dann nur noch durch das Vorhandensein von Kindern.

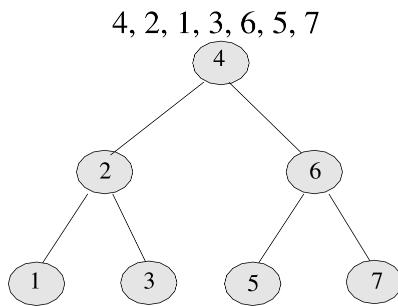
21.4.3 Einfügen und Löschen

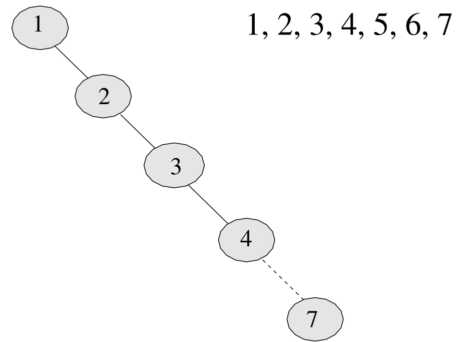
Einfügen Das Einfügen geschieht, indem man durch den Baum bis zu einem Blatt läuft, wo man den Datensatz/Schlüssel einfügen kann.
Löschen Das Löschen ist etwas komplizierter, weil verschiedene Situationen unterschiedlich behandelt werden müssen. Am schwierigsten ist das Löschen eines Knoten mit zwei Kindern: hier kann man den Knoten durch das kleinste Element im rechten Teilbaum ersetzen.



Problem: Die Gestalt des Suchbaumes und damit der Aufwand für seine Bearbeitung hängt entscheidend von der Reihenfolge des Einfügens (und eventuell Löschens) der Schlüssel ab!

Beispiel:





Der rechte Binärbaum entspricht im wesentlichen einer Listenstruktur! Der Aufwand zur Suche ist entsprechend $O(n)$.

21.4.4 Ausgeglichene Bäume

Beobachtung: Um optimale Effizienz zu gewährleisten, müssen sowohl Einfüge- als auch Löschoperation im Suchbaum sicherstellen, dass für die Höhe $H(n) = O(\log n)$ gilt. Dies kann auf verschiedene Weisen erreicht werden.

AVL-Bäume
Die *AVL-Bäume* wurden 1962 von Adelson-Velskii-Landis eingeführt. Es sind Binärbäume, die garantieren, dass sich die Höhen von rechtem und linken Teilbaum ihrer Knoten maximal um 1 unterscheiden (*Höhenbalancierung*).

(2,3)-Baum
Der (2,3)-Baum wurde 1970 von Hopcroft eingeführt. Er ist ein Spezialfall des später besprochenen (a,b)-Baums. Die Idee ist, mehr Schlüssel/Kinder pro Knoten zuzulassen.

B-Bäume
B-BäumeB-Baeume wurden 1970 von Bayer und McCreight eingeführt. Der B-Baum der Ordnung m ist gleich dem (a, b)-Baums mit $a = \lceil \frac{m}{2} \rceil$, $b = m$. Hier haben Knoten bis zu m Kinder. Für großes m führt das zu sehr flachen Bäumen. Der B-Baum wird oft zur Suche in externem Speicher verwendet. Dabei ist m die Anzahl der Schlüssel, die in einen Sektor der Platte passen (z.B. Sektorgröße 512 Byte, Schlüssel 4 Byte ⇒ m=128).

α-balancierter Baum
α-balancierte Bäume wurden um 1973 von Nievergelt und Reingold eingeführt. Idee: Die Größe $|T(v)|$ des Baums am Knoten v und des rechten (oder linken) Teilbaums $|T_l(v)|$ erfüllen

$$\alpha \leq \frac{|T_l(v)| + 1}{|T(v)| + 1} \leq 1 - \alpha$$

Dies garantiert wieder $H = O(\log n)$.

Rot-Schwarz-Bäume
Rot-Schwarz-Bäume wurden 1978 von Bayer, Guibas, Sedgewick eingeführt. Hier haben Knoten verschiedene „Farben“. Die Einfüge- und Löschoperationen erhalten dann gewisse Anforderungen an die Farbreihenfolge, welche wieder $H = O(\log n)$ garantieren. Man kann auch eine Äquivalenz zum (2,4)-Baum zeigen.

(a,b)-Bäume

(a,b)-Bäume(a,b)-Baeume wurden 1982 von Huddleston und Mehlhorn als Verallgemeinerung der B-Bäume und des (2,3)-Baums. Alle inneren Knoten haben hier mindestens a und höchstens b Kinder für $a \geq 2$ und $b \geq 2a - 1$. Außerdem befinden sich alle Blätter auf der gleichen Stufe. Dies garantiert dann wieder $H = O(\log n)$.

21.4.5 Implementation von (a,b)-Bäumen

Als Beispiel geben wir mit dem folgenden Programm eine mögliche Implementation von (a,b)-Bäumen an.

Programm: ab-tree.cc

```
#include<set>
#include<iostream>
#include<vector>
#include<cassert>
using namespace std;

const int m = 2;     // B-tree of order m
const int a = m;     // minimal number of keys
const int b = 2*m;   // maximal number of keys

template<class T>
struct Node {
    // data
    vector<T> keys;
    vector<Node *> children;
    Node* parent;
    // interface
    Node (Node* p) {parent = p;}
    bool is_leaf() {return children.size()==0;}
    Node* root () { return (parent==0) ? this : parent->root(); }
    Node* find_node (T item);
    int find_pos (T item);
    bool equals_item (int pos, T item);
} ;

// finds first position i such that keys[i]>=item
template<class T>
int Node<T>::find_pos (T item)
{
    int i = 0;
    while ((i<keys.size())&&(keys[i]<item)) i++;
    return i;
}

// checks if the key at position pos contains item
```

```
template<class T>
bool Node<T>::equals_item (int pos, T item) {
    return (pos<keys.size()) && !(item<keys[pos]);
}

// finds the node in which the item should be stored
template<class T>
Node<T>* Node<T>::find_node (T item) {
    if (is_leaf()) return this;
    int pos = find_pos(item);
    if (equals_item(pos, item))
        return this;
    else
        return children[pos]->find_node(item);
}

template<class VEC>
VEC subseq (VEC vec, int start, int end)
{
    int size = (vec.size()==0) ? 0 : end-start;
    VEC result(size);
    for (int i = 0; i<size; i++)
        result[i] = vec[i+start];
    return result;
}

// if necessary, split the node.  Returns 0 or a new root
template<class T>
Node<T>* balance (Node<T>* node)
{
    int n = node->keys.size();
    if (n<=b) return 0;
    T median = node->keys[a];
    // create a new node
    Node<T>* node2 = new Node<T>(node->parent);
    node2->keys = subseq(node->keys, a+1,
                         node->keys.size());
    node2->children = subseq(node->children, a+1,
                             node->children.size());
    for (int i=0; i<node2->children.size(); i++)
        node2->children[i]->parent = node2;
    // handle node
    node->keys = subseq(node->keys, 0, a);
    node->children = subseq(node->children, 0, a+1);

    Node<T>* parent = node->parent;
```

```
    if (parent==0)  // split the root!
    {
        Node<T>* root = new Node<T>(0);
        root->keys.push_back(median);
        root->children.push_back(node);
        root->children.push_back(node2);
        node->parent = root;
        node2->parent = root;
        return root;
    }
    // otherwise: insert in parent
    int pos=0;
    while (parent->children[pos]!=node) pos++;
    parent->keys.insert(parent->keys.begin()+pos, median);
    parent->children.insert(parent->children.begin()+pos+1, node2);
    // recursive call;
    return balance(parent);
}

template<class T>
void show (Node<T> *node)
{
    cout << node << ": ";
    if (node->children.size()>0)
    {
        cout << node->children[0];
        for (int i=0; i<node->keys.size(); i++)
            cout << " |" << node->keys[i] << "| "
                 << node->children[i+1];
    }
    else
        for (int i=0; i<node->keys.size(); i++)
            cout << node->keys[i] << " ";
    cout << endl;
    for (int i=0; i<node->children.size(); i++)
        show(node->children[i]);
}

// we could work with a root pointer, but for later use it is
// better to wrap it into a class

template<class T>
class abTree {
public:
    abTree () {root = new Node<T>(0);}
    void insert (T item);
```

```
    bool find (T item);
    void show () { ::show(root); }
  private:
    Node<T> *root;
};

template<class T>
void abTree<T>::insert (T item) {
    Node<T>* node = root->find_node(item);
    int i=node->find_pos(item);
    if (node->equals_item(i,item))
        node->keys[i] = item;
    else
    {
        node->keys.insert(node->keys.begin()+i, item);
        Node<T>* new_root = balance(node);
        if (new_root) root = new_root;
    }
}

template<class T>
bool abTree<T>::find (T item) {
    Node<T>* node = root->find_node(item);
    int i=node->find_pos(item);
    return node->equals_item(i, item);
}

int main ()
{
    abTree<int> tree;
    // insertion demo
    for (int i=0; i<5; i++) {
        tree.insert(i);
        tree.show();
    }
    // testing insertion and retrieval
    int n = 10;
    for (int i=0; i<n; i++)
        tree.insert(i*i);
    cout << endl;
    tree.show();
    for (int i=0; i<2*n; i++)
        cout << i << " " << tree.find(i) << endl;
    // performance test
    //abTree<int> set;
    set<int> set;  // should be faster
```

```
    int nn = 1000000;
    for (int i=0; i<nn; i++)
        set.insert(i*i);
    for (int i=0; i<nn; i++)
        set.find(i*i);
}
```

Bemerkung:

- Die Datensätze sind in allen Knoten gespeichert (Variante 2).
- Die Einfüge-Operation spaltet Knoten auf, wenn sie zu groß werden (mehr als b Schlüssel).
- Die Lösch-Operation ist nicht implementiert. Hier müssen Knoten vereinigt werden, wenn sie weniger als a Schlüssel enthalten.
- Die Effizienz ist zwar nicht schlecht, kommt aber nicht an die Effizienz der `set`-Implementation der STL heran. Ein wesentlicher Grund dafür ist die Verwendung variabel langer Vektoren zum Speichern von Schlüsseln und Kindern.

21.4.6 Literatur

Für eine weitergehende Darstellung von ausgeglichenen Bäumen sei auf das Buch „Grundlegende Algorithmen“ von Heun verwiesen. Noch mehr Informationen findet man im Buch „Introduction to Algorithms“ von Cormen, Leiserson, Rivest und Stein.