Informatik I
Programmieren und Softwaretechnik

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

Erstellt: 2. Dezember 2002

URL für die Vorlesung:

http://hal.iwr.uni-heidelberg.de/lehre/inf1-ws02/index.html

Vorwort

In dieser Vorlesung geht es um

- eine Einführung in die Denkweise der Informatik,
- das Erlernen der Programmiersprache C++ und
- eine Vorstellung grundlegender Algorithmen.

Beim Erlernen des Programmierens ist es wichtig die grundlegenden Konzepte zu verstehen, die im wesentlichen unabhängig von der verwendeten Programmiersprache sind. Als Lehrsprache haben wir bewusst C++ gewählt, damit der Leser sofort mit einer in der Praxis eingesetzten Sprache umgehen lernt. Konstrukte werden behutsam eingeführt um den Leser mit der Komplexität von C++ nicht zu erschlagen. Bewusst wählen wir den Einstieg über die funktionale Programmierung um den Anfänger mit wenigen Konstrukten früh zum Programmiererfolg zu führen. Dieser Teil des Kurses folgt im wesentlichen dem Anfang des bewährten Buches [ASS98].

Danach gehen wir zur prozeduralen Programmierung über, begründen, warum Variablen und Iteration eingeführt werden, und üben das Modellieren der Daten. Auch die Probleme der prozeduralen Programmierung werden diskutiert.

Der Schwerpunkt des Kurses liegt dann auf einer Einführung in die objektorientierte Programmierung in C++ mit Klassen und Vererbung bis hin zur generischen Programmierung. Die Notwendigkeit neuer Konstrukte wird dabei immer am Beispiel demonstriert. Den Abschluß bilden Containerklassen und ihre Implementierung sowie ein ausführliches, größeres Programmierbeispiel.

Den Herren Volker Reichenberger und Thimo Neubauer möchte ich für viele Diskussionen über den Stoff danken, für alle Fehler bin ich natürlich selbst verantwortlich. Dieses Skript ist ein lebendiges Dokument und wird ständig verbessert. Konstruktive Kritik ist deswegen jederzeit willkommen (e-mail Adresse: `Peter.Bastian@iwr.uni-heidelberg.de`).

Heidelberg, im September 2002 P. Bastian

1 Einführung

Bevor wir uns mit real exisitierenden (und manchmal komplizierten) Computern beschäftigen beginnen wir erst mal mit einem Spiel und der Erläuterung einiger grundlegender Begriffe.

1.1 Formale Systeme: MIU

Als erstes wollen wir sogenannte *formale Systeme* betrachten. Zunächst stellen wir uns darunter ein System vor, welches starren Regeln folgt. Die Betrachtung solcher Systeme ist eine gute Übung für den Umgang mit Computern.

Das folgende Beispiel stammt aus dem Buch [Hof79], einem Informatikkultbuch aus den frühen 1980er Jahren (lesenswert!), und heisst *MIU-System*.

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

Beispiele für solche Wörter sind:

MI

MIIII

MIIUIIU

MU

...

Aber, nicht alle möglichen Wörter bestehend aus M, I oder U sind Wörter des MIU-Sytems, sondern nur solche, die nach gewissen Regeln aus dem Wort „MI“ erzeugt werden können.

Hier sind die Regeln:

Regel 1: Bei einer Kette, deren letzter Buchstabe I ist darf ein U hinten angefügt werden.

Beispiel: MI → MIU. Man sagt MIU wird aus MI abgeleitet. Die Ableitung wird durch den Pfeil → dargestellt.

Regel 2: Aus einer Kette der Form Mx darf man die Kette Mxx ableiten. Dabei steht x für eine beliebige Zeichenkette.

Beispiele: MI → MII, MIUUI → MIUUIIUUI.

Beachte, dass x immer die gesammte Kette ausser dem M umfasst.

Regel 3: Wenn in einer Kette die Teilkette III vorkommt, darf man diese durch ein U ersetzen.

Beispiele: MIII $\rightarrow$ MU, U$\underbrace{\text{III}}$IM $\rightarrow$ UUIM, UI$\underbrace{\text{III}}$M $\rightarrow$ UIUM.

Regel 4: Wenn in einer Kette die Teilkette UU vorkommt darf man sie streichen.

Beispiele: UUU $\rightarrow$ U, MUUUIII $\rightarrow$ MUIII.

Die Regeln dürfen natürlich nur genau in der angegebenen Weise verwendet werden. Offensichtlich gibt es Situationen in denen mehrere Regeln anwendbar sind. Dann darf man sich eine aussuchen.

Alle Wörter, die aus MI mit beliebiger Anwendung der Regeln erzeugt werden können heissen

„die Wörter des MIU-Systems“.

Beispiel für eine korrekte Ableitung:

MI	
$\rightarrow$ MIU	Regel 1
$\rightarrow$ MIUIU	Regel 2
$\rightarrow$ MIUIUIUIU	Regel 2

Man sagt dass MIUIUIUIU ein Wort des MIU-Systems ist genau dann wenn MIUIUIUIU aus MI ableitbar ist.

Nun kommen wir zur eigentlichen Aufgabe, dem *MU-Rätsel*. Dies ist ganz einfach die Frage:

Ist MU ein Wort des MIU-Systems?

Schreiben wir die Frage etwas mathematischer. Definiere

$$\mathbb{MIU} = \{x \mid x \text{ ist Wort des MIU-Systems}\}.$$

Dann lautet das MU-Rätsel schlicht:

$$\text{MU} \in \mathbb{MIU}?$$

Zunächst wird man etwas herumprobieren und zusehen ob man die Zeichenkette MU irgenwie bekommen kann. Wenn es einem zu langweilig geworden ist fragt man sich vielleicht, ob es nicht systematischer geht?

Kann man das MU-Rätsel systematisch lösen? Idee: Erzeuge der Reihe nach alle Wörter des MIU-Systems. Dazu ist es geschickt die Menge der erzeugbaren Wörter folgendermaßen darzustellen:

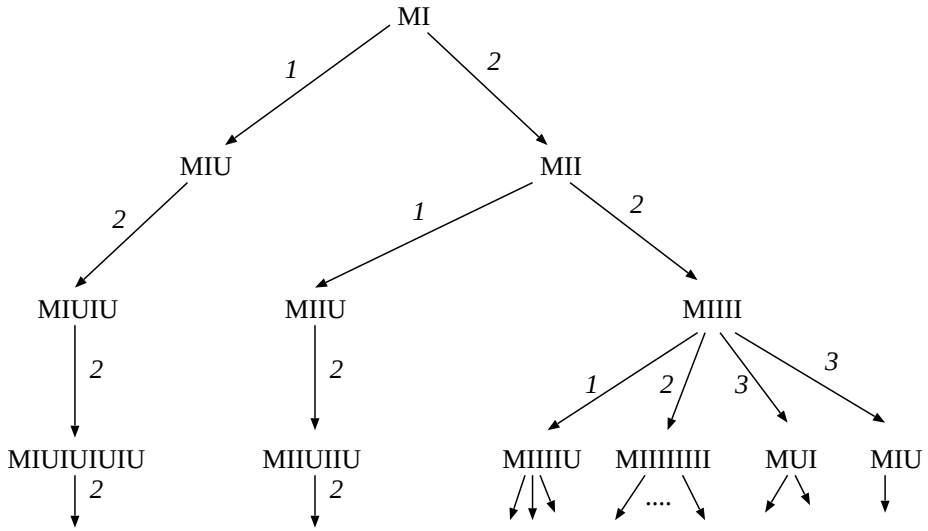


Abbildung 1: Anordnung aller Wörter des MIU-Systems in einem Baum.

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 aller Wörter, die durch Anwendung von zwei Regeln erzeugt werden können, usw.

Solche eine Struktur nennt man einen *Baum*. Bäume sind die Lieblingsstruktur der Informatiker und kommen noch häufiger vor.

Damit haben wir ein Schema (später: Algorithmus) welches uns erlaubt alle Wörter des MIU-Systems der Reihe nach zu erzeugen. Wir schließen:

> *Falls das Wort MU erzeugbar ist, wird unser Verfahren in endlicher Zeit die Antwort liefern.*

Was ist nun wenn MU aber *kein* Wort des MIU-Systems ist? Dann werden wir es mit obigem Verfahren nie erfahren!

Unser Verfahren ist also „nur zur Hälfte brauchbar“.

Man sagt: Die Menge $\mathbb{MIU}$ ist rekursiv aufzählbar, hingegen ist das Komplement $\overline{\mathbb{MIU}}$ nicht rekursiv aufzählbar.

1.2 Turingmaschine

Als weiteres Beispiel für ein „Regelsystem“ betrachten wir die Turingmaschine (TM) (Nein, ist kein eingetragenes Warenzeichen).

Diese wurde 1936 von Alan Turing zum theoretischen Studium der Berechenbarkeit eingeführt.

Weiterführende Literatur: [HU00].

Eine TM besteht aus einem festen Teil („Hardware“) und einem variablen Teil („Software“). TM meint 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 Schema aufgebaut und funktionieren nach den selben Regeln.

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.

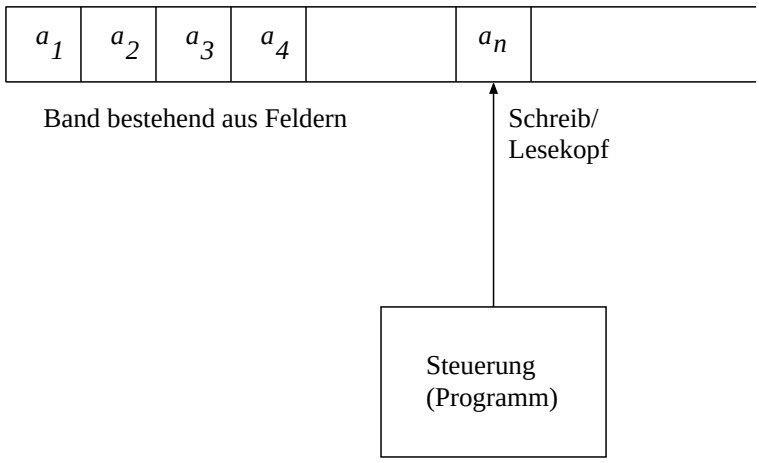


Abbildung 2: Die Turingmaschine.

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 endlich vielen Zuständen und einer Tabelle,

die beschreibt wie man von einem Zustand in einen anderen gelangen kann. Diese Tabelle nennt man auch Programm.

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

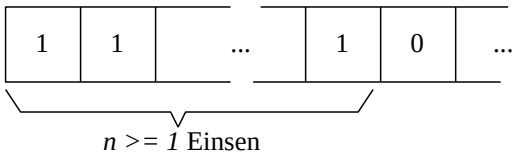


Abbildung 3: Eine Kette von Einsen.

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 2: Raten Sie was folgendes Programm macht. Eingabe wie in Beispiel 1. Tipp: Spielen Sie die Eingabe 10... durch.

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

Programme lassen sich übersichtlicher als Übergangsgraph darstellen. Jeder Knoten ist ein Zustand. Jeder Pfeil entspricht einer Zeile der Tabelle. Hier das Programm aus Beispiel 2 als Graph:

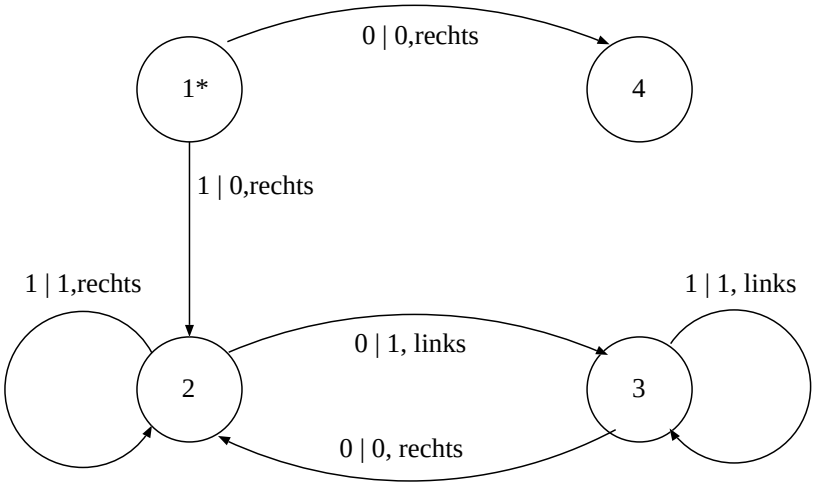


Abbildung 4: Programm als Übergangsgraph.

Beispiel 3: 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:

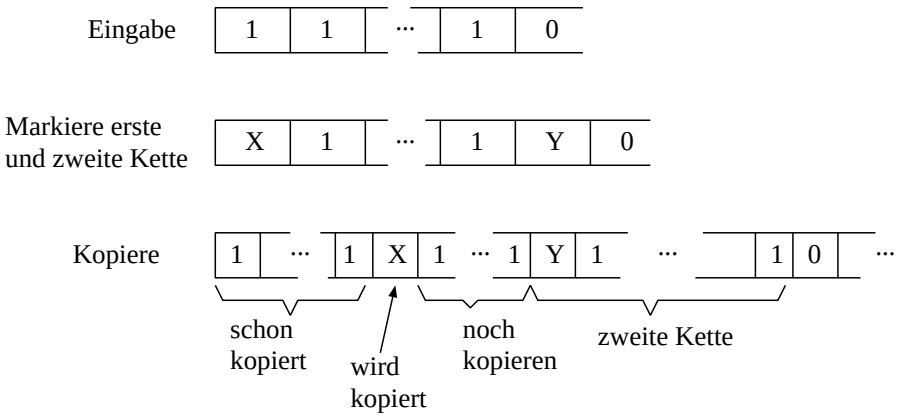


Abbildung 5: Idee zum Verdoppeln mit einer TM.

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

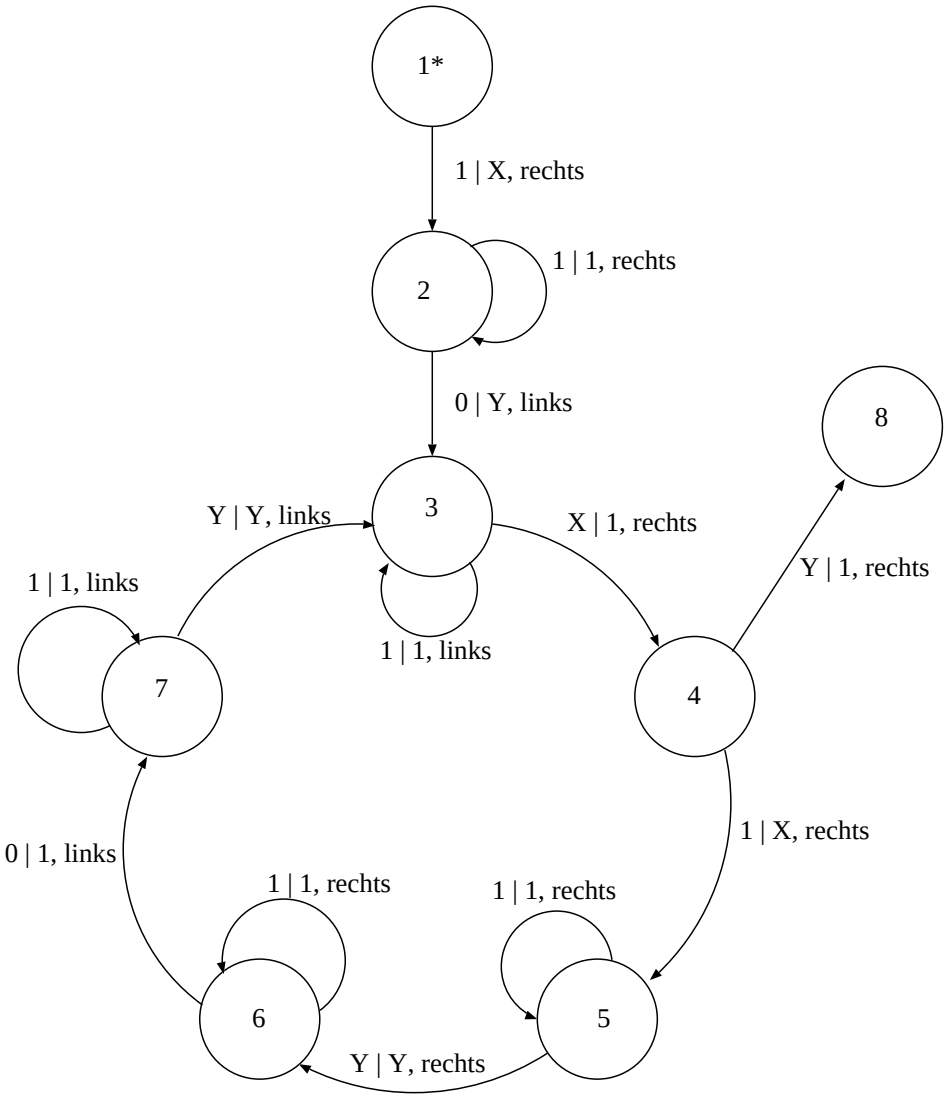


Abbildung 6: Verdoppeln einer Einserkette.

Wir erkennen die drei wesentlichen Komponenten von Berechnungsprozessen:

- Grundoperationen
- Selektion
- Wiederholung

1.3 Problem, Algorithmus, Programm

Problem: Beschreibt die zu lösende Aufgabe. Die Problemstellung enthält nicht wie das Problem gelöst werden kann.

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

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

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. Z. B. ist in obigem Beispiel jedes $n \in \mathbb{N}$ erlaubt. Ein Algorithmus beschreibt also ein Verfahren zur Lösung (möglicherweise) unendlich vieler Probleme.
- 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.

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.

Programm: Ein Programm ist eine Formalisierung des Algorithmenbegriffes. Ein Programm kann auf einer Maschine (z. B. einer 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 folgendes Schema:

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

Frage: Was kann man mit einer TM alles berechnen?

Charakterisierung über Simulation: Es ist möglich eine TM mit einem PC zu simulieren, d. h. man kann einen Algorithmus angeben der zu jeder TM ein PC-Programm erzeugt, welches das Verhalten der TM Schritt für Schritt nachvollzieht. Wir schließen daraus: Ein PC (mit unendlich viel Speicher) kann alles berechnen was eine TM kann.

Das ist vielleicht nicht so beeindruckend ...

Aber es geht auch umgekehrt! Behauptung: Zu einem PC mit gegebenem Programm kann man eine TM angeben, die die Berechnung des PCs nachvollzieht! Allerdings habe ich keine Zeit das zu zeigen ... Wir schließen: PC und TM können dieselbe Klasse von Problemen lösen.

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.

Die Church'sche These lautet daher:

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

Dabei heisst intuitiv berechenbar, dass man einen Algorithmus angeben kann.

Mehr davon in der Vorlesung „Theoretische Informatik".

Was bringt uns das?

Mit der TM hat man das Prinzip von Berechnungsvorgängen vollständig verstanden. Man braucht keine zusätzlichen Operationen oder Eigenschaften um alle vorstellbaren Berechnungen durchführen zu können. Dass reale Computer und ihre Programmiersprachen dennoch viel komplizierter sind hat „nur" mit praktischen Aspekten wie Geschwindigkeit oder Arbeitseffizienz zu tun.

1.4 Reale Computer

Von Neumann Rechner

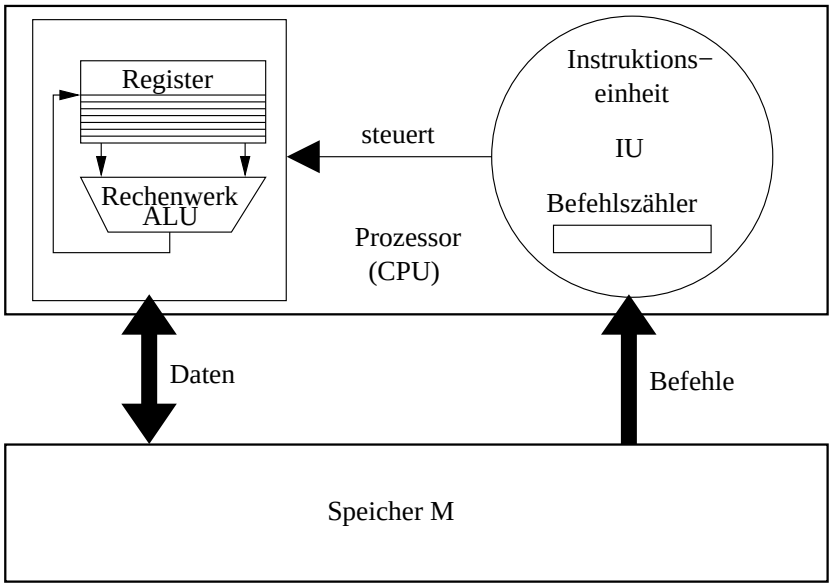


Abbildung 7: Von Neumann Rechner.

Praktische Computer basieren meist auf dem von John von Neumann 1945 eingeführten Konzept.

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 Neumann'schen 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 „Theoretische Informatik“.

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.

Zweck der Maschinenspreche ist deren möglichst effiziente Ausführung.

Die weitaus meisten Programme werden heute in sogenannten „höheren Programmiersprachen“ erstellt.

Sinn einer höheren Programmiersprache ist, dass der Programmierer Programme möglichst

- schnell (in Sinne benötigter Programmiererzeit)
- effizient (im Sinne benötigter Ausführungszeit) 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 der 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).

Warum gibt es verschiedene Programmiersprachen?

Das Erstellen von Comuterprogrammen ist eine sehr komplizierte und zeitaufwendige Sache.

Die Leistungsfähigkeit von Computern wächst sehr schnell.

Moore’sches Gesetz: Die Leistung von Mikroprozessoren verdoppelt sich etwa alle zwei Jahre.

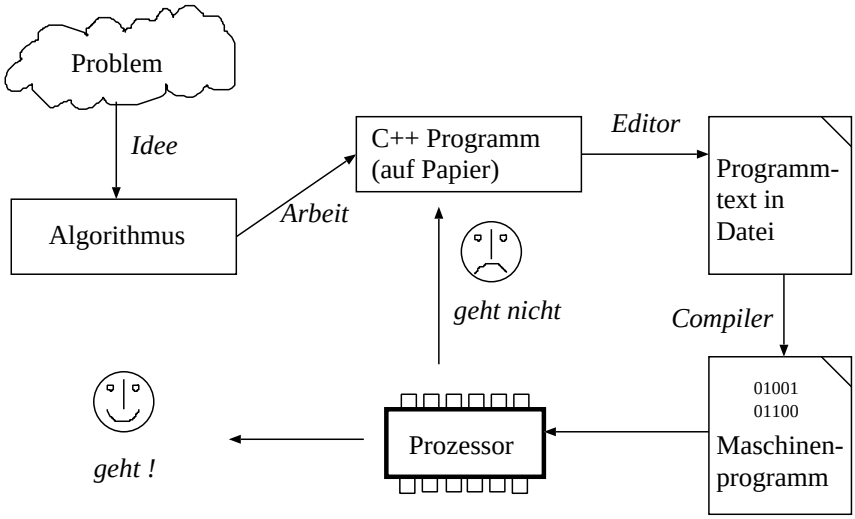


Abbildung 8: Workflow bei der Programmerstellung.

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)

(Der Linux Kernel ist der zentrale Teil des Betriebssystemes.)

Offensichtlich wächst auch die Länge des Programmtextes (hier am Beispiel des Linux-Kernel) im selben Maße mit!

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

Diesen Effekt versucht man durch bessere Programmiertechnik, Sprachen und Softwareentwurfsprozesse auszugleichen.

In dieser Vorlesung wollen wir die Grundlagen der objektorientierten Programmierung (in C++) erlernen.

Diese Konzepte können auch in anderen Programmiersprachen realisiert werden.

TEIL I

FUNKTIONALE PROGRAMMIERUNG

2 Auswertung von Ausdrücken

2.1 Ausdrücke

Wir beginnen ganz bescheiden mit der Auswertung von arithmetischen Ausdrücken der Form wie wir sie aus der Mathematik kennen:

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

Mit anderen Worten: Wir wollen unseren Computer als Taschenrechner verwenden.

Hier ist ein sehr simples C++ Programm, dass dies leistet

Programm 2.1 (Unser erstes Programm) `erstes.cc`

```
#include"iostream.h"

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

Tipp: In der interaktiven Variante des Textes können Sie die Programme durch Klicken auf den Link rechts oberhalb des Programmtextes herunterladen.

Ein C++ Programm ist, genau wie ein Wort des MIU-Systems, eine regelkonforme Zeichenkette. Nur sind die Regeln etwas komplizierter und das Alphabet umfasst mehr als M, I und U.

Dieses Programm enthält schon eine Menge Konstruktionselemente. Was diese alle genau bedeuten lernen wir im Verlauf der Vorlesung. Im Moment interessiert uns nur die Auswertung des Ausdrucks in der vorletzten Zeile.

Das Übersetzen des Programmes gelingt (auf meinem Rechner) mittels

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

und die Ausführung produziert

```
> erstes
-213
```

Unser simples Programm

- berechnet den Wert des Ausdrucks zwischen << ... << und
- druckt ihn auf der Konsole aus.

Wir fragen nun zunächst: 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:

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

Die linke Schreibweise nennt man *Infix-Schreibweise*, die rechte *Prefix-Schreibweise*.

Die vier Grundoperationen $+, -, *, /$ betrachten wir als *atomar*.

Im Rechner gibt es entsprechende Baugruppen, die diese atomaren Operationen realisieren (die Addition behandeln wir gegen Ende der Vorlesung, der Rest folgt in Informatik II).

Wir sehen nun auch wie ein aus atomaren Operationen zusammengesetzter Ausdruck von „innen nach aussen“ ausgewertet werden kann

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

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

2.2 Funktionen

C++ erlaubt es uns, zu den schon eingebauten Funktionen wie $+, -, *, /$ noch weitere hinzuzufügen. Diese neuen Funktionen können wir mit Hilfe der bereits bekannten Funktionen realisieren.

Hier ist die *Definition* einer Funktion, die das Quadrat der eingegebenen Zahl berechnet:

quadratfunktion.cc

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

Die erste Zeile, der *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.

C++ ist eine *streng typgebundene* Sprache, d. h. jedem *Namen* (z. B. `x` oder `quadrat` ist ein *Typ* zugeordnet.

Diese Typzuordnung kann später im Programm nicht mehr geändert werden (*statische Typbindung, static typing*).

Den Rest der Funktionsdefinition in geschweiften Klammern bezeichnet man als *Funktionsrumpf*. Er sagt was die Funktion tut.

Der Typ `int` soll $\mathbb{Z}$, der Menge der ganzen Zahlen, entsprechen. Natürlich gibt es eine Reihe weiterer eingebauter Typen wie

`float`, `double`, `char`, `bool`.

Hier ist ein vollständiges Programm, welches die Quadratfunktion nutzt:

Programm 2.2 (Beispiel mit Quadrat) quadrat.cc

```
#include"iostream.h"

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

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

Wo immer eine Zahl in einem Ausdruck stehen darf können wir also auch `quadrat(...)` schreiben. Das Argument der Quadratfunktion kann wiederum ein zusammengesetzter Ausdruck sein.

Wenn wir das Programm `quadrat.cc` genau betrachten, erkennen wir dass `main` eine Funktion ohne Argumente und mit Rückgabetyp `int` ist.

Die Funktion `main` ist die Funktion mit der die Programmausführung beginnt. Wir wissen schon von der TM, dass wir für den Programmstart eine spezielle Vereinbarung treffen müssen.

2.3 Selektion

Bis jetzt erlaubt unsere Programmiersprache nur die Funktionsdefinition und die Auswertung zusammengesetzter Ausdrücke. Dies ist nicht genug im Sinne der Berechenbarkeit.

Das fehlende Element ist die Möglichkeit den Programmverlauf in Abhängigkeit der Daten zu steuern.

Z. B. haben wir noch nicht die Möglichkeit die Absolutwertfunktion zu programmieren:

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

Um Selektion ausdrücken zu können führen wir eine spezielle dreistellige Funktion `cond` ein.

Der Absolutwert sieht dann so aus:

Programm 2.3 (Absolutwert) `absolut.cc`

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

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

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

Die `cond` Funktion erhält drei Argumente: Einen *Bool'schen Ausdruck* und zwei normale Ausdrücke. Ein Bool'scher Ausdruck hat einen der beiden Werte „wahr“ oder „falsch“ als Ergebnis. Ist der Wert „wahr“ so ist das Resultat der `cond` Funktion der Wert des zweiten Arguments ansonsten der des Dritten.

3 Syntaxbeschreibung mit Backus-Naur Form

3.1 EBNF

C++ Programme sind nach bestimmten Regeln erzeugte Zeichenketten, ähnlich wie das beim MIU-System der Fall war.

Die Regeln nach denen wohlgeformte C++ Programme erzeugt werden nennt man *Syntax* (eigentlich: Lehre vom Satzbau).

Zur Definition der Syntax von Programmiersprachen hat man eine spezielle Schreibweise, die erweiterte Backus-Naur Form (EBNF), entwickelt.

Da die EBNF Regeln zur Bildung wohlgeformter Zeichenketten beschreibt muss man genau aufpassen welche Zeichen zu den Regeln und welche Zeichen zu der abzuleitenden Zeichenkette gehören.

In der EBNF unterscheiden wir folgende Zeichen bzw. Zeichenketten:

- Unterstrichene Zeichen oder Zeichenketten sind Teil der zu bildenden, wohlgeformten Zeichenkette. Unterstrichene Zeichen 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.

Nun beginnen wir mit der Beschreibung von Regeln. Hier ein Beispiel mit zwei Regeln:

$$\begin{aligned} &<\text{A}> \;::=\; \underline{\text{a}}\; <\text{A}>\; \underline{\text{b}} \\ &<\text{A}> \;::=\; < \epsilon > \end{aligned}$$

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.

Hat man das Symbol <A> gegeben so darf man es nach der ersten Regel durch das terminale Zeichen $\underline{\text{a}}$ gefolgt von dem Symbol <A> gefolgt von $\underline{\text{b}}$ ersetzen.

Die Anwendung einer Regel funktioniert somit ganz ähnlich wie die Ableitung im MIU-System. Die zweite Regel besagt schlicht, dass <A> gestrichen werden darf.

Ausgehend vom Symbol <A> kann man somit folgende Zeichenketten erzeugen:

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

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

Das Zeichen „ | “ („oder“) erlaubt die Zusammenfassung mehrer Regeln in einer Zeile. Somit bedeutet

$$<\text{A}> \;::=\; \underline{\text{a}}\; <\text{A}>\; \underline{\text{b}} \;\mid\; <\epsilon>$$

dass <A> durch $\underline{\text{a}}$ <A> $\underline{\text{b}}$ oder $<\epsilon>$ ersetzt werden kann.

Als Beispiel hier die Regel für eine Ziffer:

$$<\text{Ziffer}> \;::=\; \underline{0} \mid \underline{1} \mid \underline{2} \mid \underline{3} \mid \underline{4} \mid \underline{5} \mid \underline{6} \mid \underline{7} \mid \underline{8} \mid \underline{9}$$

Die Klammern dienen der Notation von Option und Wiederholung. So ist

$$<\text{A}> ::= [\; <\text{B}>\;]$$

identisch zu

$$<\text{A}> ::= <\text{B}> \;\mid\; <\epsilon>$$

und

$$<\text{A}> ::= \{\; <\text{B}>\; \}$$

bezeichnet die n-malige Wiederholung mit $n \geq 0$:

$$<\text{A}> ::= <\text{A}>\; <\text{B}> \;\mid\; <\epsilon>$$

Schließlich ist

$$<\text{A}> ::= \{\; <\text{B}>\; \}^{+}$$

die n-malige Wiederholung mit $n \geq 1$:

$$<\text{A}> ::= <\text{A}>\; <\text{B}> \;\mid\; <\text{B}>$$

3.2 Syntaxbeschreibung für FC++

Damit können wir nun die komplette Syntax der bisher besprochenen Programme angeben. Wir wollen diesen (winzigen) Ausschnitt von C++ als FC++ (funktionales C++) bezeichnen.

Allerdings wäre es sehr aufwendig den ganzen Sprachumfang von C++ so zu beschreiben. Wir werden nur die Syntax der wichtigsten Konstrukte angeben und dies oft auch nur in eingeschränkter Form. Für eine komplette Syntax verweisen wir auf die Literatur, z. B. [Str97].

Beginnen wir mit einer Zahl. Mit Hilfe der obigen Definition einer Ziffer können wir die Syntax einer Zahl angeben:

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

Eine einfache Syntax für Ausdrücke könnte man folgendermassen schreiben:

Syntax 3.1 (Ausdrücke)

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

Die Regeln für <Buchstabe> und <Buchstabe oder Zahl> haben wir weggelassen.

Diese Regeln sind zu simpel um alle gültigen C++ Ausdrücke zu erfassen. Man darf nämlich z. B. auch Klammern weglassen.

Hier die Syntax einer Funktionsdefinition in EBNF:

Syntax 3.2 (Funktionsdefinition)

<Funktion>	::=	<Typ> <Name: > (<formale Parameter>)
		{ <Funktionsrumpf> }
<Typ>	::=	<Identifikator>
<Name: >	::=	<Identifikator>
<formale Parameter>	::=	$< \epsilon >$ | <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 etwas auf dem Bildschirm auszudrucken.

Die Syntax der Funktionsdefinition ist in C++ eigentlich wesentlich komplizierter. Diese einfache Variante soll aber hier genügen.

Hier ist noch die Syntax für die Selektion:

Syntax 3.3 (Cond Funktion)

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

Beachte dass der Test auf Gleichheit als `==` geschrieben wird!

Beispiel für einen Bool'schen Ausdruck:

$(x > 3) \wedge (x \leq 7)$ entspricht `(x>3)&&(x<=7)`.

Damit können wir nun auch die Syntax unserer eingeschränkten C++ Programme vollständig angeben.

Syntax 3.4 (Einfaches C++ Programm)

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

Unsere C++ Programme bestehen somit aus einer Folge von `#include` Zeilen und $n > 0$ Funktionsdefinitionen. Dabei muss genau eine Funktion den Namen `main` haben. Include gibt es in zwei Varianten: Mit spitzen Klammern oder Anführungsstrichen.

Bemerkung 3.5 (Leerzeichen) C++ Programme erlauben das Einfügen von Leerzeichen, Zeilenvorschüben und Tabulatoren um Programme für den Menschen lesbarer zu gestalten. Diese Zeichen sind in der C/C++ Literatur als Whitespace bekannt. Hierbei gibt es folgendes zu beachten:

- Identifikatoren, Zahlen, Schlüsselwörter und Operatorzeichen dürfen keinen Whitespace enthalten:
  - `zaehler` statt `zae hler`,
  - `893371` statt `89 3371`,
  - `return` statt `re tur n`,
  - `&&` statt `& &`.

- Folgen zwei Identifikatoren, 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 Syntaxbeschreibung mit EBNF ist nicht mächtig genug um selbst 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.

Hinweis: Mit Hilfe von EBNF lassen sich sogenannte kontextfreie Sprachen definieren. Entscheidend ist, dass in EBNF-Regeln links immer nur genau ein nichtterminales Symbol steht. Im Unterschied definiert das MIU-System keine kontextfreie Sprache. 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. Obige Regel „Kein Funktionsname darf doppelt vorkommen“ lässt sich mit einer kontextfreien Sprache nicht formulieren und wird deshalb extra gestellt.

3.3 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 */
```

In der ersten Form ignoriert der Übersetzer den Rest der Zeile. Mit der zweiten Form kann man Kommentare über mehrere Zeilen machen.

Wir wollen nun genauer untersuchen wie Programme bestehend aus den bisher vorgestellten Konstrukten abgearbeitet werden.

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*.

Ähnlich wie bei der TM brauchen wir eine Vorstellung davon was bei der Ausführung eines Programmes passiert.

Der bisher behandelte Ausschnitt aus C++ erlaubt nur die Auswertung von Ausdrücken. Dabei stellen wir uns auch die Operatoren +, -, * und / wieder als zweistellige Funktionen vor.

Das sogenannte Substitutionsmodell beschreibt wie Funktionen ausgewertet werden.

Definition 4.1 (Substitutionsmodell) Wir haben folgende Fallunterscheidung entsprechend dem Aufbau von Ausdrücken:

1. Der auszuwertende Ausdruck hat die Form <Zahl>

   dann ist der Wert des Ausdruckes einfach die Zahl.

2. Es sei ein Ausdruck der Form <Name: > $\underline{(}$ $< a_1 >, < a_2 >, \ldots, < a_n >$ $\underline{)}$ auszuwerten, wobei <Name: > der Name einer Elementarfunktion ist. Dann

   (a) Werte die Argumente aus. Diese sind wieder Ausdrücke. Unsere Definition ist also rekursiv!

   (b) Werte die Elementarfunktion für die Argumente aus und liefere den Wert

3. Es sei ein Ausdruck der Form <Name: > $\underline{(}$ $< a_1 >, < a_2 >, \ldots, < a_n >$ $\underline{)}$ auszuwerten, wobei <Name: > nicht `cond` ist und $< a_1 >$ bis $< a_n >$ die Argumentausdrücke sind. Die Auswertung erfolgt in drei Schritten:

   (a) Werte die Argumente aus.

   (b) Ersetze im Rumpf der Funktion <Name: > jeden formalen Parameter durch den entsprechenden Wert des zugehörigen, soeben ausgewerteten Ausdrucks.

   (c) Werte den Rumpf der Funktion aus. Der erhaltene Wert ist der Wert der Funktion.

4. Der auszuwertende Ausdruck sei <u>cond</u> $\underline{(}$ $< a_1 >, < a_2 >, < a_3 >$ $\underline{)}$

   In diesem Fall

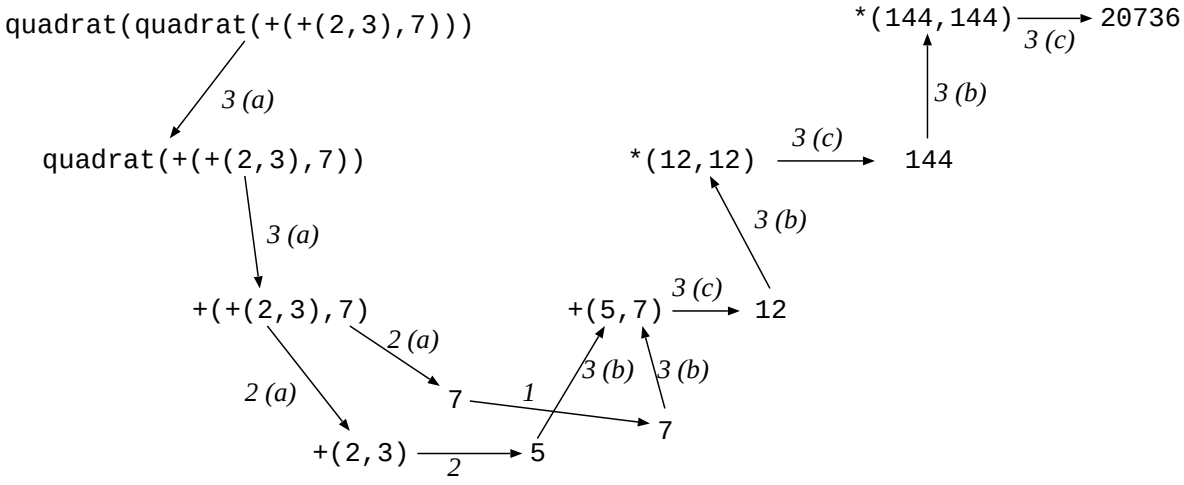


Abbildung 9: Auswertung eines Ausdruckes mit dem Substitutionsmodell.

(a) Werte das erste Argument aus.

(b) Ist der Wert des ersten Ausdruckes 1, so werte das zweite Argument aus und liefere diesen Wert als Ergebnis. Ansonsten werte das dritte Argument aus und liefere den Wert zurück. Wichtig ist, dass nur eines der beiden Argumente $< a_2 >$ oder $< a_3 >$ ausgewertet wird.

Hier der Vorgang an einem einfachen Beispiel:

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

Hier ein komplizierteres Beispiel:

```
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
```

Diese Auswertung wird in Abbildung 9 nochmal ausführlich mit Angabe der Regeln ausgeführt:

Natürlich kann die Quadratfunktion auch selbst in Ausdrücken vorkommen:

```
quadrat(3)+quadrat(9) = +(quadrat(3),quadrat(9))
                      = +( *(3,3)   ,quadrat(9))
                      = +(    9     ,  *(9,9)  )
                      = +(    9     ,    81    )
                      = 90
```

Der Computer verwendet *nicht* das Substitutionsmodell zur Auswertung von Ausdrücken. Es dient hier nur dazu, eine einfache Vorstellung von der Auswertung zu vermitteln. Selbstverständlich ermittelt das Substitutionsmodell das korrekte Ergebnis.

Später lernen wir noch eine andere Auswertungsmethode kennen, die in der Praxis verwendet wird. Wenden Sie bis dahin das Substitutionsmodell an.

Übrigens: Die Namen der formalen Parameter sind egal, sie entsprechen sogenannten *gebundenen Variablen* in der Mathematik.

5 Funktionen und Prozesse

5.1 Lineare Rekursion und Iteration

Wir betrachten das Problem der Berechnung der Fakultätsfunktion. Diese ist für $n \geq 1$ definiert als

$$n! = 1 * 2 * 3 * \ldots * n,$$

also das Produkt der Zahlen $1 \ldots n$.

Alternativ kann die Fakultätsfunktion auch so definiert werden:

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

Hierbei handelt es sich um eine *rekursive* Definition, da die Funktion mit Hilfe der Fakultätsfunktion selbst definiert ist.

Die Definition ist sinnvoll, da jede Anwendung der Definition den Argumentwert um eins erniedrigt, schließlich erreicht man $n = 1$ wofür die Fakultät explizit definiert ist.

In C++ kann man solche rekursiven Definitionen auch umsetzen:

Programm 5.1 (Rekursive Berechnung der Fakultät) `fakultaet.cc`

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

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

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

Betrachten wir die Auswertung von `fakultaet(5)`:

```
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
```

Dies bezeichnen wir als *linear rekursiven Prozess.*

Es wird ein immer längerer Ausdruck mit *verzögerten* Operationen aufgebaut. Die Länge dieses Ausdruckes nimmt *linear* mit n zu.

Es gibt noch eine andere Art die Fakultät zu berechnen. Dazu betrachten wir folgendes Tableau von Werten von n und $n!$:

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

Wir beginnen bei 1 und bauen der Reihe nach die Werte von 1!, 2!, 3!, usw. auf und wenn n erreicht ist sind wir fertig.

Wir stellen uns eine Funktion `fakIter` vor mit formalen Parametern `zaehler` und `produkt`. `Zaehler` nimmt nacheinander die Werte $1, 2, 3, \ldots$ an. `Produkt` soll das Produkt aller bisherigen Werte von `zaehler` enthalten. Dann berechnet `fakIter` das nächste Produkt mittels

```
produkt ← produkt*zaehler
zaehler ← zaehler+1
```

Zusätzlich ist in `fakIter` noch das Erreichen des Endzählerstandes zu testen. Hier ist das vollständige Programm:

Programm 5.2 (Iterative Fakultätsberechnung) `fakultaetiter.cc`

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

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;
}
```

Betrachten wir jetzt wie das Programm `fakultaet(5)` berechnet:

```
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
```

Hier spricht man von einem *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.

Der Zustand beschreibt die bisher abgelaufene Berechnung vollständig. Von einem Zustand kann man ohne Kenntnis der Vorgeschichte aus weiterrechnen.

Bei einem linear iterativen Prozess ist die Anzahl der durchlaufenen Zustände proportional zu n. Die Informationsmenge, die man zur Darstellung des Zustandes benötigt, ist konstant.

Bei einem linear rekursiven Prozess hingegen steigt die Informationsmenge die man zur Darstellung des Zustandes benötigt linear an.

Beide Arten von Prozessen wurden durch rekursive Funktionen beschrieben! Rekursive Funktionen müssen also nicht unbedingt rekursive Prozesse beschreiben!

Bei geeigneter Implementierung rekursiver Funktionen ist der Speicherplatzbedarf der Berechnung proportional zu n bei linear rekursiven Prozessen und konstant bei linear iterativen Prozessen.

Obige Realisierung hat jedoch in C++ in beiden Fällen einen Speicherbedarf der linear in n wächst. Wir werden noch kennenlernen wie man iterative Prozesse anders formulieren kann.

5.2 Baumrekursion

Die *Fibonacci Zahlen* sind ein weiteres Beispiel für rekursiv definierte Zahlen:

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

Hier ist das entsprechende Programm:

Programm 5.3 (Fibonacci rekursiv) `fibonacci.cc`

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

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

int main ()
{
```

```
    cout << fib(40) << endl;
}
```

Sehen wir uns an wie `fib(5)` mit dem Substitutionsmodell ausgewertet wird:

```
  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
```

Offensichtlich wächst der Ausdruck wesentlich schneller als bei der Fakultätsfunktion.

Wie schnell wächst der Ausdruck bei der Auswertung der Fibonaccifunktion?

Dazu stellen wir die Auswertung der Fibonaccifunktion mit einem Baum dar (Sie erinnern sich: die Lieblingsstruktur der Informatiker):

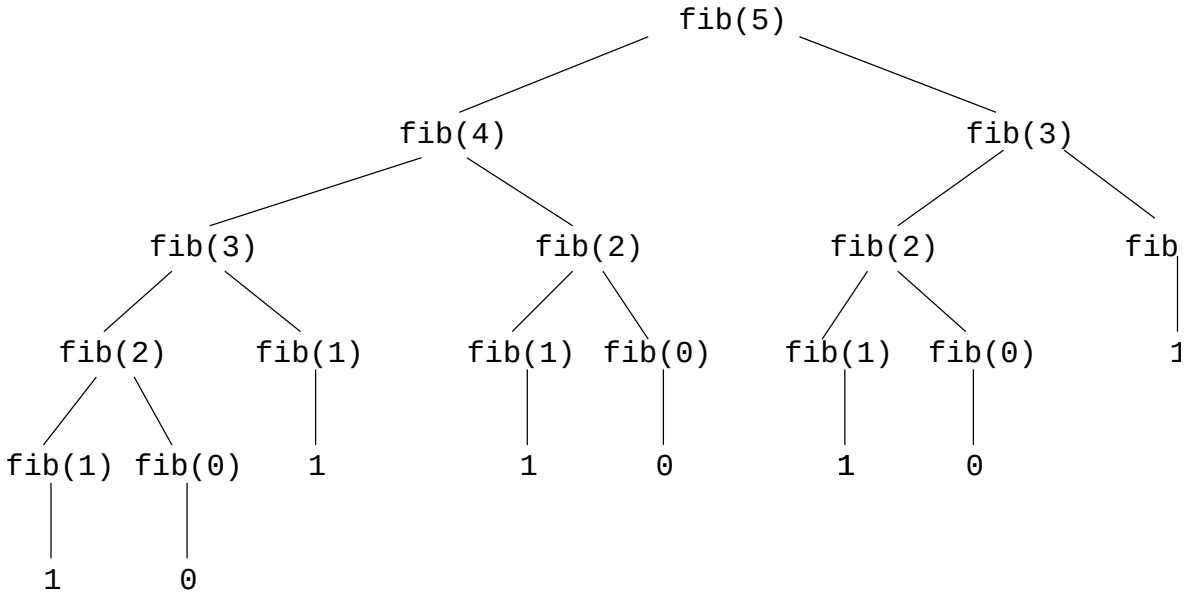


Abbildung 10: Zur Auswertung der Fibonaccifunktion.

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

Definition 5.4 *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.*

Der Knoten heisst „Wurzel“ des neu gebildeten Baumes. Die Wurzeln der Unterbäume auf die ein Knoten verweist heissen „Kinder“ des Knotens. Ein Knoten ohne Kinder heisst „Blatt“. Für jedes Kind ist der darauf verweisende Knoten der „Vaterknoten“.

Die Definition ist rekursiv: Mehrere Bäume werden mittels eines Knotens zu einem größeren Baum kombiniert.

Die Anzahl der Kinder kann von Knoten zu Knoten unterschiedlich sein.

Bäume bei denen alle Knoten höchstens zwei Kinder haben heissen *binäre Bäume*. Die Auswertung der Fibonaccifunktion wurde mit einem binären Baum dargestellt.

Da sich der Rekursionsprozess bei der Fibonaccifunktion als Baum darstellen lässt heisst dieser Prozess *baumrekursiv*.

Wie schnell wächst nun der Aufwand für die rekursive Auswertung der Fibonaccifunktion?

Wir können in guter Näherung annehmen, dass der Aufwand proportional zur Anzahl der Funktionsauswertungen ist, die wir mit $A(n)$ bezeichnen wollen.

Diese wollen wir nun nach *unten* abschätzen. Dazu folgendes Bild:

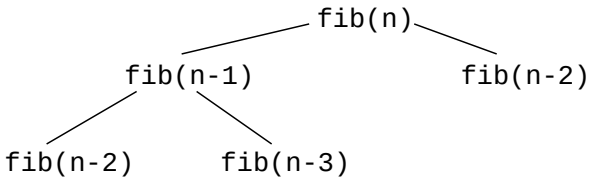


Abbildung 11: Zum Aufwand bei der Auswertung der Fibonaccifunktion.

Offensichtlich wird `fib(n-2)` mindestens zweimal ausgwertet. Deshalb gilt

$$A(n) \geq 2A(n-2)$$

Für $n = 0, 1$ benötigt man jeweils genau eine Funktionsauswertung.

Nun wollen wir die Rekursionsformel $A(n) \geq 2A(n-2)$ auflösen. Sei / die Division ganzer Zahlen ohne Rest dann behaupten wir:

$$A(n) \geq 2^{n/2}, \quad \forall n \geq 0$$

Die Behauptung bestätigen wir durch vollständige Induktion. Für $n = 0, 1$ rechnen wir nach

$$A(0) \geq 2^{0/2} = 1, \quad A(1) \geq 2^{1/2} = 2^0 = 1$$

Sei die Annahme für $n-2$ bewiesen, so gilt

$$A(n) \geq 2A(n-2) \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2-1+1} = 2^{n/2}.$$

Der Aufwand des baumrekursiven Prozesses zur Auswertung der Fibonaccifunktion steigt *exponentiell* mit n.

Genauer zeigt man, dass die Anzahl der Blätter im Auswertungsbaum für `fib(n)` gleich ist

$$\mathrm{Fib}(n+1) = \mathrm{Int}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n \Big/ \sqrt{5}\right).$$

Betrachten wir nochmal die Fibonaccifolge

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

Natürlich können wir die n-te Fibonaccizahl auch mit einem iterativen Prozess berechnen indem wir alle Zahlen aufzählen und bei der n-ten anhalten. Hier ist das Programm:

Programm 5.5 (Fibonacci iterativ) `fibiter.cc`

```
#include"iostream.h"
#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 ()
{
    cout << fib(40) << endl;
}
```

Die entsprechende Auswertung im Substitutionsmodell ist

```
  fib(5)
= fibIter(1,0,5)
= fibIter(1,1,4)
= fibIter(2,1,3)
= fibIter(3,2,2)
= fibIter(5,3,1)
= fibIter(8,5,0)
= 5
```

Der Zustand der Berechnung kann wieder durch einen festen Satz von (drei) Zustandsvariablen beschrieben werden.

Die Zahl der durchlaufenen Zustände (Berechnungsschritte) ist proportional zu n. Es handelt sich also um einen linear iterativen Prozess.

Damit ist diese Variante deutlich schneller als die baumrekursive!

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,
- im allgemeinen die Größe der Eingabe in Bits.

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

- Rechenzeit,
- Anzahl auszuführender Operationen,
- Speicherbedarf,
- Plattenplatz.

Dann schreibt man $R(n) = \Omega(f(n))$ falls es von n unabhängige Konstanten c_1, n_1 gibt so dass

$$R(n) \geq c_1 f(n) \quad \forall n \geq n_1.$$

Für den baumrekursiven Prozess oben war also $A(n) = \Omega(\sqrt{2^n})$.

Man schreibt $R(n) = O(f(n))$ falls es von n unabhängige Konstanten c_2, n_2 gibt so dass

$$R(n) \leq c_2 f(n) \quad \forall n \geq n_2.$$

Schließlich schreibt man $R(n) = \Theta(f(n))$ falls

$$R(n) = \Omega(f(n)) \text{ und } R(n) = O(f(n)).$$

Beispiele:

$R(n) = \Theta(1)$	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

Wechselgeld

Oben haben wir zwei verschiedene Varianten kennengelernt die n-te Fibonaccizahl zu berechnen: Erst rekursiv, dann iterativ.

Es könnte nun der Eindruck entstehen: Rekursive Prozesse sind zu vermeiden, da diese Probleme mit iterativen Prozessen viel schneller gelöst werden können.

Dass es nicht so einfach ist rekursive in iterative Prozesse zu übersetzen zeigt folgendes Beispiel.

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

Wir überlegen uns folgende rekursive Lösung des Problems: 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.

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 also

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

Beispiel: Es seien 5 Cent in 1 und 2 Centstücke zu wechseln

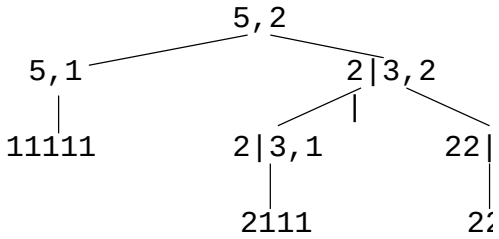


Abbildung 12: Wechseln von 5 in 1 und 2 Cent.

Es ergibt sich ein baumrekursiver Prozess.

Hier ist das zugehörige C++ Programm

Programm 5.6 (Wechselgeld zählen) `wechselgeld.cc`

```
#include"iostream.h"
#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==5,50,0)))));
}

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,5);
}

int main ()
{
    cout << wechselgeld(300) << endl;
}
```

Hier einige Werte

```
wechselgeld(50)  = 342
wechselgeld(100) = 2498
wechselgeld(200) = 24405
wechselgeld(300) = 102522
```

Es ist hier nicht so einfach ein Programm zu finden welches das Problem mit einem iterativen Prozess löst.

Baumrekursive Prozesse erlauben oft verblüffend einfach Lösungen von relativ komplexen Problemen auch wenn sie äußerst ineffizient sein können.

Der größte gemeinsame Teiler

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

Den ggT braucht man etwa um rationale Zahlen zu kürzen:

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

Eine Möglichkeit zur Berechnung des ggT ist die Zerlegung beider Zahlen in Primfaktoren, der ggT ist dann das Produkt aller gemeinsamer Faktoren.

Ein sehr viel effizienterer, rekursiver Algorithmus stammt schon von Euklid, deswegen heißt er auch *Euklischer Algorithmus*. Die Rekursion basiert auf den folgenden Überlegungen.

Jedes $a \in \mathbb{N}$ lässt sich darstellen als $a = q \cdot b + r$, wobei

- q der Quotient bei Teilen ohne Rest und
- r der Rest $a \bmod b$ ist.

Wir unterscheiden nun zwei Fälle

I) Gilt $r = 0$, so ist a durch b teilbar und somit $\text{ggT}(a, b) = b$.

II) Sei nun $0 < r < b$. Wir behaupten folgende Aussage: $0 < s \in \mathbb{N}$ *ist ein Teiler von* a *und* b *genau dann wenn* s *ein Teiler von* $r = a \bmod b$ *und* b *ist.*

Beweis:

i) Aus s teilt a und b folgt s teilt r und b. Dass s das b teilt ist bereits eine Voraussetzung. Es bleibt noch das r. Es gilt

$$\frac{a}{s} = \frac{q \cdot b + r}{s} = q\frac{b}{s} + \frac{r}{s}$$

Da a/s und b/s natürliche Zahlen sind muss auch r/s natürlich sein.

ii) Aus s teilt r und b folgt s teilt a und b. Wieder ist nur s teilt a zu zeigen. Mit der selben Rechnung wie in i) und der Voraussetzung $r/s \in \mathbb{N}$ und $b/s \in \mathbb{N}$ folgt sofort $a/s \in \mathbb{N}$.

Somit haben wir folgende Rekursion bewiesen:

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

Die Rekursion endet, da das zweite Argument stetig abnimmt, aber nicht negativ werden kann.

Programm 5.7 (Größter gemeinsamer Teiler) `ggT.cc`

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

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
= ggT(7,0)
= 7
```

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.

Man kann zeigen, dass die Anzahl der Rechenschritte wie $\Theta(\log n)$ wächst wobei $n = \min(a, b)$. Details siehe [ASS98].

5.3 Deklaration von Funktionen

Betrachte folgende etwas umständlich geratene Fakultätsberechnung:

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

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

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

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

Der C++-Übersetzer liefert eine Fehlermeldung beim Übersetzen dieses Programmes.

Das Problem ist, dass bei einem Funktionsaufruf der Name einer Funktion bekannt sein muss. Im obigen Programm rufen sich jedoch die Funktionen `g` und `h` gegenseitig auf und eine muss ja zuerst definiert werden.

Das Problem löst man mit einer *Funktionsdeklaration*. Deklarationen führen einen Namen in das Programm ein und verbinden ihn mit einem Typ.

Eine Funktionsdeklaration sieht aus wie eine Funktionsdefinition ohne Rumpf und wird durch `;` beendet.

Eine Deklaration alleine ist unvollständig, d. h. es muss an anderer Stelle im Programm noch eine Definition folgen. Eine Definition enthält bereits eine Deklaration.

Hier das korrekte Programm:

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

int h (int n);  // forward declaration

int g (int n)
{
```

```
    return cond( n<=1 , 1 , n*h(n-1) );
}

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

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

5.4 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 $a_i \in \{0, \ldots, \beta - 1\}$

Dann bedeutet

$$(a_{n-1}a_{n-2}\ldots a_1a_0)_\beta \equiv \sum_{i=0}^{n-1} a_i\beta^i$$

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

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

Am häufigsten wird $\beta = 2$, das Binärsystem, verwendet.

Will man vorzeichenbehaftete Zahlen darstellen gibt es verschiedene Verfahren dies zu tun.

Erste Möglichkeit: Zusätzliches Bit für das Vorzeichen.

Zweite Möglichkeit: *2er Komplement* (Setzt $\beta = 2$ voraus). 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	→	011
-1	→	111
2	→	[1]010

Gebräuchliche Zahlenbereiche für $\beta = 2$ 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

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?

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

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

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

Neben der Endlichkeit des Zahlenbereiches tritt nun das zusätzliche Problem der *Rundung* auf.

- 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.

Wir müssen also damit leben, dass reelle Zahlen im Rechner nur mit einer gewissen *Genauigkeit* dargestellt werden können.

Auch das Ergebnis elementarer Rechenoperationen $+, -, *, /$ von darstellbaren Zahlen kann nicht mehr darstellbar sein und muss *gerundet* werden. Hierauf geht man in der Numerik weiter ein.

Festkommazahlen haben eine feste Auflösung. Praktischerweise sollte aber die Auflösung an die absolute Größe der Zahl anpassbar sein. Dies gelingt mit der Fließkommaarithmetik, wie sie bei der wissenschaftlichen Darstellung üblich ist.

Ein Zahl wird repräsentiert als

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

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.

Typische Wortlängen sind:

Bei `float`: 23 Bit Mantisse, 8 Bit Exponent, 1 Bit Vorzeichen entsprechen 6 Nachkommastellen dezimal in der Mantisse.

Bei `double`: 52 Bit Mantisse, 11 Bit Exponent, 1 Bit Vorzeichen entsprechen 16 Nachkommastellen dezimal in der Mantisse.

Rundungsfehler

$\beta = 10, n = 3$: 3.141 wird gerundet auf 3.14×10^0. Der Fehler beträgt maximal 0.005, man sagt $0.5ulp$, *ulp* heisst *units last place*.

Auslöschung

Vorsicht ist vor allem bei der Subtraktion angebracht. Dort 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 $b = 3.34, a = 1.22, c = 2.28$.

Exakt:

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

Mit Rundung der Zwischenergebnisse ergibt sich:

$$\ldots 11.2 - 11.1 = 0.1$$

Der Fehler ist somit $0.1 - 0.0292 = 0.0708$. Dies ist mehr als zehnmal größer als der maximale Fehler 0.005 in den Argumenten a, b, c!

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) 5)/3`	Fließkommadivision
`((int) 5.0)/((int) 3.0)`	Ganzzahldivision

Weitere Details zu arithmetischen Operationen und Fließkommazahlen findet man in [HP96, Anhang A], [Gol91], oder in der Vorlesung „Einführung in die Numerik“.

Wurzelberechnung mit dem Newtonverfahren

Wir wollen nun etwas mit den Fließkommazahlen experimentieren und betrachten das Problem der Lösung der nichtlinearen Gleichung

$$f(x) = a.$$

Eine Anwendung wäre $f(x) = x^2$, also das Berechnen der Quadratwurzel.

Hier ist ein Unterschied zwischen Mathematik und Informatik: In der Mathematik genügt es völlig x als diejenige Zahl zu *definieren* für die $x^2 = a$ gilt.

Aber wie *berechnet* man den Zahlenwert wirklich? Wir benötigen einen *Algorithmus*, also eine endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer Elementaroperationen.

Zur numerischen Lösung der allgemeinen nichtlinearen Gleichung gehen wir aus von der *Taylorreihe*

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

dabei ist x_n ein Schätzwert für die Lösung x der Gleichung.

Dieser Schätzwert soll nun mittels

$$x_{n+1} = x_n + h$$

um die Korrektur h verbessert werden.

Wir wollen $x_{n+1} = x$, also schreiben wir

$$a = f(x_{n+1}) = f(x_n + h) \approx f(x_n) + hf'(x_n)$$

woraus wir eine Beziehung zur Bestimmung von h erhalten

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

Zusammen ergibt das die *Iterationsvorschrift*

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

Speziell für die Quadratwurzel erhalten wir mit $f(x) = x^2$ und $f'(x) = 2x$ die Vorschrift

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

Nun ist noch zu klären wann wir die Iteration abbrechen. Dazu testen wir

$$|f(x) - a| < \varepsilon$$

mit einer vorgegebenen (kleinen) Zahl ε.

Programm 5.8 (Quadratwurzelberechnung) `newton.cc`

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

double abs (double x)
{
    return cond( x<0 , -x , x );
}

bool gut_genug (double xn, double a)
{
    cout << xn*xn<< endl;
    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;
}
```

`cout.precision(20)` bewirkt, dass 20 Stellen nach dem Komma bei Fließkommazahlen ausgegeben werden.

Im Konvergenztest wird die aktuelle Iterierte jeweils ausgedruckt.

Dieses Programm entspricht nicht genau unseren Regeln. Es enthält schon eine „Sequenz von Anweisungen“ in der Funktion `gut_genug` sowie in `main`.

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
```

Offensichtlich konvergiert das Newtonverfahren sehr schnell, hier ist die Folge der Werte x_n^2:

```
1
2.25
2.0069444444444446418
2.0000060073048828713
2.0000000000045106141
2.0000000000000004441
```

Man kann zeigen, dass sich unter gewissen Voraussetzungen an die Funktion f die Zahl der gültigen Ziffern im Ergebnis mit jedem Schritt verdoppelt, man sieht dies sehr gut in den ersten vier Schritten.

Was passiert im fünften Schritt?

Weitere Details in Numerik I.

Zusammenfassung

Bisher besteht unser Berechnungsmodell aus folgenden Bestandteilen:

1. Auswertung von zusammengesetzten Ausdrücken aus Zahlen, Funktionsaufrufen und Selektionen.
2. Konstruktion neuer Funktionen.

Namen treten dabei in genau zwei Zusammenhängen auf:

1. Als Symbole für Funktionen.
2. Als formale Parameter in Funktionen.

Im Substitutionsmodell werden bei der Auswertung einer Funktion die formalen Parameter im Rumpf durch die aktuellen Werte ersetzt und dann der Rumpf ausgewertet.

Unter Vernachlässigung von Resourcenbeschränkungen (endlich große Zahlen, endlich viele rekursive Funktionsauswertungen) kann man sich überlegen, dass dieses Berechnungsmodell äquivalent zur Turingmaschine ist.

Den bisher gepflegten Programmierstil nennt man *funktionale Programmierung.*

Trotzdem erweist es sich in der Praxis als nützlich weitere Konstruktionselemente einzuführen um einfachere, übersichtlichere und wartbarere Programme schreiben zu können.

Der Preis dafür ist, dass wir uns von unserem einfachen Substitutionsmodell verabschieden müssen!

TEIL II

PROZEDURALE PROGRAMMIERUNG

6 Kontrollfluß

6.1 Lokale Variablen und die Zuweisung

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).

C++ erlaubt es neue Name-Wert Zuordnungen einzuführen, wie etwa in

```
float umfang (float r)
{
    const double pi=3.14159265; // pi wird Wert 3.14159265 zugeordnet
    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;
}
```

Einem Namen kann nur *einmal* ein Wert zugeordnet werden (deswegen `const`).

Die Auswertung solcher Funktionsrümpfe erfordert eine Erweiterung des Substitutionsmodells:

- Ersetze formale Parameter durch aktuelle Parameter.
- Erzeuge Name-Wert Zuordnungen, die keine unbekannten Namen mehr enthalten. Ersetze neue Namen im Rest des Rumpfes durch den Wert.
- Iteriere bis keine unbekannten Namen mehr vorhanden.

Eine völlig neue Situation entsteht jedoch, *wenn wir erlauben, dass die Zuordnung von Werten zu Namen geändert werden kann*:

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

Zeile 1 definiert die *Variable* `y`, die Werte vom Typ `int` annehmen kann.

Zeile 2 nennt man eine Zuweisung. Die links des =-Zeichens stehende Variable erhält den Wert des rechts stehenden Ausdrucks als neuen Wert. Nur der Wert einer Variablen aber nicht ihr Typ kann geändert werden!

Die Einführung der Zuweisung bringt eine Reihe weitreichender Änderungen in unserer bisherigen Vorstellung über den Ablauf einer Berechnung mit sich.

Betrachte

```
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
}
```

In Zeile 1 erhält `y` den Wert 3. Damit erhält `x1` in Zeile 2 den Wert `3*x`. Dann erhält `y` in Zeile 3 den Wert 5 und somit `x2` in Zeile 4 den Wert `5*x`.

Dies ist eine völlig neue Situation! Obwohl `x1` und `x2` durch den selben Ausdruck `y*x` definiert werden haben sie im allgemeinen verschiedene Werte.

Dies widerspricht der Vorstellung in unserem bisherigen Substitutionsmodell, bei dem ein Name im ganzen Funktionsrumpf durch seinen Wert ersetzt werden kann.

Außerdem hängt das Ergebnis der Berechnung von der *Reihenfolge* der Ausführung der Zuweisungen ab. Vertauschen wir etwa die Zeilen 3 und 4 so erhält `x2` einen anderen Wert. Auch dieses Verhalten ist in den bisherigen Programmen nicht möglich gewesen: Die Reihenfolge der Auswertung von Ausdrücken im Substitutionsmodell war egal!

Wir haben in den letzten beiden Beispielprogrammen natürlich implizit angenommen, dass die Namensdefinitionen und Zuweisungen *der Reihe nach* abgearbeitet werden und diese Tatsache ist für das Ergebnis der Berechnung wichtig!

Wie beobachtet, hängt der Wert eines Ausdrucks von der aktuellen Belegung der Variablen mit Werten zum Zeitpunkt des Auswertens ab.

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

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

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

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.

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* begreifen.

Die Zuweisung

```
y = 5;
```

ändert die Bindungstabelle derart, dass nach Ausführung der Zuweisung

$$w(\mathtt{y}) = 5$$

gilt (hier ist das Gleichheitszeichen gemeint!).

Fassen wir die Syntax von Variablendefinition und Zuweisung noch einmal zusammen:

Syntax 6.1 (Variablendefinition)

<VarDef> ::= [const] <Typ> <Variablenname> [= <Ausdruck>]

Elemente in eckigen Klammern sind optional. Mit dem Schlüsselwort `const` wird dem Namen ein nicht änderbarer Wert zugeordnet. In diesem Fall *muss* der zweite optionale Teil, die *Initialisierung*, vorhanden sein. Ohne `const` wird eine Variable definiert. Dann *kann* die Initialisierung weggelassen werden. In diesem Fall ist der Wert der Variablen bis zur ersten Zuweisung unbestimmt. Es ist eine gute Idee Variablen grundsätzlich zu initialisieren!

Wir erlauben zunächst Variablendefinitionen nur innerhalb von Funktionsdefinitionen. Diese Variablen bezeichnet man als *lokale Variablen*.

Syntax 6.2 (Zuweisung)

<Zuweisung> ::= <Variablenname> = <Ausdruck>

Der Ausdruck rechts wird relativ zur aktuellen Umgebung ausgewertet. Dann wird die Umgebung geändert so dass die Variable an den neuen Wert gebunden ist.

Was ist gleich?

Die Einführung der Zuweisung hat auch Auswirkungen auf den Gleichheitsbegriff.

Im Substitutionsmodell fassen wir Namen x, y als Werte auf, d. h. mathematisch:

$$x, y \in \mathbb{Z}$$

und $x = y$ bedeutet eben, dass x und y das gleiche Element in $\mathbb{Z}$ bezeichnen.

Sind x, y Variablen, werden deren Werte durch $w : \{x, y\} \to \mathbb{Z}$ bestimmt und „Gleichheit der Variablen“ würden wir dann als verstehen wollen als :

$$w(x) = w(y).$$

Lokale Umgebung

Wie müssen wir uns die Umgebung im Kontext mehrerer Funktionen vorstellen?

```
int g (int x)
{
    int y = 3;                  // Zeile 1
    int z = 8;                  // Zeile 2
    return h(z*(x+y));          // Zeile 3
}

int h (int x)
{
    int y = 7;                  // Zeile 1
    return cond(x<1000,g(x+y), // Zeile 2
                88);            // Zeile 3
}
```

Bei jeder Auswertung einer Funktion wird eine eigene, *lokale* Umgebung erzeugt. 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 `z` auch wenn `h` von `g` aufgerufen wurde.

Wird eine Funktion n mal rekursiv aufgerufen so existieren n verschiedene Umgebungen für diese Funktion.

Eine Funktion hat kein Gedächtnis! Wir meinen damit: Wird eine Funktion mehrmals mit gleichen Argumenten aufgerufen so sind auch die Ergebnisse gleich.

6.2 Anweisungsfolgen (Sequenz)

Bei der funktionalen Programmierung haben wir uns nur mit der Auswertung von Ausdrücken beschäftigt.

Mit Einführung der Zuweisung verallgemeinern wir die Auswertung von Ausdrücken zur *Ausführung von Anweisungen*. Dies nennt man auch *imperative Programmierung*.

Wir kennen schon eine Reihe wichtiger Anweisungen:

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

Jede Anweisung endet mit einem Semikolon.

Überall wo eine Anweisung stehen darf, kann auch eine ganze *Folge von Anweisungen*, auch *Sequenz* genannt, stehen:

Syntax 6.3 (Sequenz)

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

Anweisungsfolgen stehen in geschweiften Klammern.

Anweisungen werden der Reihe nach abgearbeitet.

Auch ein Funktionsrumpf kann eine Anweisungsfolge enthalten.

Beispiel

```
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)
}
```

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

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.

Beachte, dass die Reihenfolge in

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

nicht vertauscht werden darf.

In der funktionalen Programmierung mussten wir weder auf die Reihenfolge achten noch irgendwelche „Hilfsvariablen“ einführen.

6.3 Bedingte Anweisung (Selektion)

Als Ersatz für die Cond-Funktion führen wir nun die bedingte Anweisung ein. Diese erlaubt uns die Auswahl zwischen zwei Alternativen in Abhängigkeit eines logischen Ausdrucks.

Syntax 6.4 (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).

Als Beispiel formulieren wir die Absolutfunktion um.

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

ist vollkommen äquivalent zu

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

Im Prinzip macht die Headerdatei cond.h genau diese Ersetzung!

6.4 Schleifen

Wir lernen nun Möglichkeiten kennen um iterative Prozesse einfacher formulieren zu können.

Erinnern wir uns an die iterative Berechnung der Fakultätsfunktion. Die Anweisungsfolge

```
ergebnis = zaehler*ergebnis;
zaehler  = zaehler+1;
```

muss wiederholt werden solange `zaehler` nicht größer als der Endwert ist.

Diese Wiederholung erreicht man mit folgendem Konstrukt:

Programm 6.5 (Fakultät mit Schleife) `fakwhile.cc`

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

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

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 `zaehler` $= n + 1$ gelten und daher ist `ergebnis` $= n!$.

Man beachte das Zusammenspiel von

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

Wird nur einer dieser Punkte geändert so funktioniert das Programm (wahrscheinlich) nicht mehr.

Syntax 6.6 (While-Schleife)

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

Die Anweisung wird solange ausgeführt wie die Bedingung erfüllt ist.

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

Syntax 6.7 (For-Schleife)

<ForSchleife> ::= for (<Init> ; <BoolAusdr> ; <Increment>)
<Anweisung>
<Init> ::= <VarDef> | <Zuweisung>
<Increment> ::= <Zuweisung>

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

Hier ist diese Variante der Fakultätsberechnung:

Programm 6.8 (Fakultät mit For-Schleife) `fakfor.cc`

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

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

    return ergebnis;
}
```

Variable `zaehler` gibt es nur innerhalb der `for`-Schleife.

Die Initialisierungsanweisung enthält Variablendefinition und Initialisierung.

Wie in Programm 6.5 wird die Inkrementanweisung am Ende des Schleifendurchlaufes ausgeführt.

Beispiele

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

Die folgende Funktion zur Berechnung der Fibonaccizahlen verallgemeinert die Funktion `f5` von oben:

Programm 6.9 (Fibonacci mit For-Schleife) `fibfor.cc`

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

    if (n==0)
        return 0;
    else
        for (int i=2; i<=n; i=i+1)
        {
            t = a+b;
            a = b;
            b = t;
        }

    return b;
}
```

Hier lautet die Schleifeninvariante

$$(2 \leq \texttt{i} \leq \texttt{n}+1) \wedge (\texttt{b} = (\text{Fib}(\texttt{i}-1))$$

Für $n > 1$ wird die Schleife $n-1$ mal durchlaufen, der Aufwand ist linear in n.

Programm 6.10 (Newton mit While-Schleife) `newtonwhile.cc`

```
double abs (double x)
{
    if (x<0) return -x; else return x;
}

double wurzel (double a)
{
```

```
    double x=1.0;

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

7 Benutzerdefinierte Datentypen

Unsere bisherigen Programme haben nur mit Zahlen (unterschiedlichen Typs) gearbeitet.

„Richtige“ Programme bearbeiten allgemeinere Daten. Beispiele:

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

Im Sinne der Berechenbarkeit ist das keine Einschränkung, denn in beliebig großen Zahlen lassen sich beliebige Daten kodieren (Gödelisierung). Nur ist das furchtbar umständlich ...

Deswegen erlauben praktisch alle Programmiersprachen dem Programmierer die Definition neuer Datentypen.

7.1 Aufzählungstyp

Erlaubt die Definition eines Datentyps, der aus endlich vielen Werten besteht. Jedem Wert ist ein Name zugeordnet.

Hier ein Beispiel aus einem Programm zur Verwaltung von Büchern:

Programm 7.1 (Beispiel Aufzählungstyp) `enum.cc`

```
#include"iostream.h"

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

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

Schließlich die Syntax des Aufzählungstyps

Syntax 7.2 (Aufzählungstyp)

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

Ein Aufzählungstyp ist eine endliche Menge.

7.2 Felder

Bis jetzt haben wir nur eingebaute Datentypen usw. verwendet.

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

Ein Feld besteht aus einer *festen Anzahl* von Elementen eines Grundtyps.

Die Elemente sind angeordnet, d. h. mit einer Nummerierung versehen. Die Nummerierung ist fortlaufend und beginnt bei 0.

Als Beispiel dient uns das mathematische Konstrukt eines Vektors:

$$x \in \mathbb{R}^3, \quad x = (x_0, x_1, x_2)^T.$$

In C++ würden wir dies ausdrücken als

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

wobei man auf die einzelnen Komponenten mittels

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

zugreifen kann. Ansonsten verhält sich `x[1]` wie jede andere Variable vom Typ `double`.

Der Index in eckigen Klammern kann selbst wieder Wert eines Ausdruckes mit dem Typ `int` sein. In diesem Fall wird der Indexausdruck relativ zur aktuellen Umgebung ausgewertet und zwar sowohl auf der rechten wie auf der linken Seite des Zuweisungszeichens.

Syntax 7.3 (Felddefinition)

<FeldDef> ::= <Typ> <Name: > [<Anzahl>]

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

Eine Felddefinition darf wie eine Variablendefinition verwendet werden.

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.

Es genügt nur die Vielfachen der Primzahlen zu nehmen (Primfaktorzerlegung).

Alle Vielfachen der Zahlen bis $\sqrt{N}$ erzeugen die Primzahlen bis N, denn

Eine Zahl $x < N$ die nicht prim, ist hat mindestens einen Teiler $1 < a < \sqrt{N}$: Da x nicht prim ist gilt $x = ab$ mit $1 < a, b < x$, $a, b \in \mathbb{N}$. Angenommen es wäre $a \geq \sqrt{N} \wedge b \geq \sqrt{N}$ so wäre $x = ab \geq N$ im Widerspruch zur Voraussetzung.

Programm 7.4 (Sieb des Eratosthenes) `eratosthenes.cc`

```
#include<iostream.h>

double abs (double x)
{
    if (x<0) return -x; else return x;
}

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

    while (abs(x*x-a)>1E-3)
        x = 0.5*(x+a/x);
    return x;
}

int main ()
{
    bool prim[50000];                       // bool ist ein Datentyp
                                            // mit den Werten true und false
    for (int i=0; i<50000; i=i+1)
        prim[i]=true;

    for (int i=2; i<wurzel(50000); i=i+1)
        if (prim[i])
            for (int j=2*i; j<50000; j=j+i)
```

```
                prim[j]=false;

    for (int i=0; i<50000; i=i+1)
        if (prim[i]) cout << i << endl;
}
```

7.3 Zeichen und Zeichenketten

Zur Verarbeitung von einzelnen Zeichen gibt es den Datentyp `char`:

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

Er kann kann genau ein Zeichen aufnehmen.

Die Initialisierung benutzt die einfachen Anführungsstriche.

Der Datentyp `char` ist kompatibel mit `int`. Man kann mit ihm rechnen:

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

`char` umfasst nur die Werte $-127 \ldots 128$.

Den druckbaren Zeichen entsprechen allerdings nur die Werte $32 \ldots 127$:

Programm 7.5 (ASCII) `ASCII.cc`

```
#include"iostream.h"

int main ()
{
    char c;

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

Die Abbildung der Zahlen $32 \ldots 127$ auf druckbare Zeichen nennt man den *American Standard Code for Information Interchange* oder kurz ASCII.

Die Zeichen kleiner 32 dienen Steuerzwecken wie Zeilenende, Papiervorschub, Piepston, etc. (Auch diese sind Teil des ASCII).

Zeichenketten realisiert man am einfachsten mittels einem Feld:

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

Zur Initialisierung dienen die doppelten Anführungsstriche.

Das Feld muss groß genug sein um die Zeichenkette aufnehmen zu können, plus ein Zeichen, welches das Ende der Kette anzeigt.

Dies illustriert folgendes kleine Programm:

Programm 7.6 (Zeichenketten C-style) `Cstring.cc`

```
#include"iostream.h"

int main ()
{
    char name[64] = "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.

Hier ein Beispiel:

Programm 7.7 (Zeichenketten C++-style) `CCstring.cc`

```
#include"iostream"
#include"string"

int main ()
{
    std::string vorname = "Peter";
    std::string nachname = "Bastian";
    std::string name;

    // Addieren von Zeichenketten
    name = vorname + " " + nachname;

    // ausgeben
    cout << name << endl;

    // es geht auch umstaendlich
    for (unsigned int i=0; i<name.size(); i=i+1) // size() : Anzahl Zeichen
        cout << name[i];                   // name[i] liefert i-tes Zeichen
    cout << endl;                          // neue Zeile
}
```

7.4 Typedef

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

Hier ein Beispiel:

```
typedef int MyInteger;
```

Damit hat der Datentyp `int` auch den Namen `MyInteger` erhalten.

`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: Soll das Programm portabel auf verschiedenen Rechnern laufen so kann man sein Programm mit `MyInteger` schreiben und an zentraler Stelle geeignet definieren.

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;
```

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

Die `typedef`-Anweisung nutzen wir in unserem nächsten Beispiel um die Problemgröße zu parametrisieren.

7.5 Das Acht-Damen-Problem

Als weitere Anwendung von Rekursion und Feldern als Argumente von Funktionen betrachten wir das Acht-Damen-Problem:

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

Eine Dame kann horizontal, vertikal und diagonal schlagen:

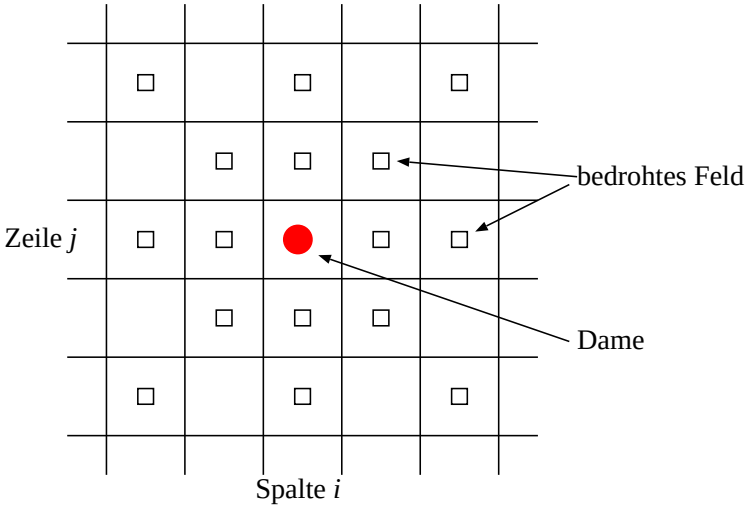


Abbildung 13: Bedrohte Felder einer Dame.

Ist die Dame an der Stelle (i, j), so bedroht sie alle Positionen (i', j') mit

- $i = i'$ oder $j = j'$
- $(i - i') = (j - j')$ oder $(i - i') = -(j - j')$

Offensichtlich kann bei jeder Lösung in jeder Zeile des Schachbrettes nur genau eine Dame stehen.

Wir wollen die Lösung sukzessive aufbauen indem wir erst in der ersten Zeile eine Dame platziern, dann in der zweiten Zeile usw.

Die Platzierung der ersten n Damen können wir durch ein `int`-Feld der Länge n beschreiben, wobei jede Komponente die Spaltenposition der Dame enthält.

Programm 7.8 (Acht-Damen-Problem) achtdamen.cc

```
#include<iostream.h>

typedef int Spalten[8];     // Name fuer die Spalten

bool gute_position (int n, Spalten J, int j)  // n,J  : Brett mit n Damen
{                                             // (n,j): neue Position
    for (int k=0; k<n; k=k+1)
    {                                         // (k,J[k]) schlägt (n,j)?
        if (J[k]==j) return false;            // gleiche Spalte
        if (n-k==J[k]-j) return false;        // diagonal
        if (n-k==j-J[k]) return false;        // diagonal
    }
    return true;
}

int platziere_dame (int n, Spalten J, int a)  // n,J : Brett mit n Damen
{                                             // a   : Größe des Brettes
    if (n==a)                                 // Brett ist voll
    {
        cout << endl << endl;                 // Drucke Lösung
        for (int i=0; i<n; i=i+1)             // als Seiteneffekt
        {
            for (int j=0; j<n; j=j+1)
                if (j!=J[i]) cout << '.';
                else cout << 'D';
            cout << endl;
        }
        return 1;                             // Eine Mögl. gefunden
    }
    else                                      // es gibt noch Platz
    {
        int c=0;                              // Zähler für Anzahl Mögl.
        for (int j=0; j<a; j=j+1)             // Prüfe alle (n,j)
            if (gute_position(n,J,j))         // Wenn Position OK ist,
            {
                J[n]=j;                       // setze Dame
                c = c+platziere_dame(n+1,J,a);// und rufe rekursive auf
            }
        return c;                             // liefere Anzahl Mögl.
    }
}

int main ()
{
    Spalten J;                                // Ein leeres Feld
    cout << "Es wurden " << platziere_dame(0,J,8)
         << " Loesungen gefunden" << endl;
}
```

Dieses Verfahren ist als *Backtracking* bekannt.

Beachte, wie das das Feld benutzt wird um eine variable Zahl von Argumenten zu übergeben.

Eine Lösung des Acht-Damen-Problems. Für $n = 8$ gibt es 92 Lösungen:

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

Hier eine Lösung für $n = 28$:

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

7.6 Zusammengesetzte Datentypen

Bisher haben wir einfache Datentypen und Felder kennengelernt. In diesem Abschnitt behandeln wir eine weitere Möglichkeit neue Datentypen zu konstruieren: den *zusammengesetzten Datentyp*.

Bei zusammengesetzten Datentypen kann man eine beliebige Anzahl möglicherweise verschiedener, auch zusammengesetzter, Datentypen zu einem neuen Datentyp kombinieren.

Diese nennt man auch *Strukturen*.

Hier ist ein simples Beispiel bei dem zwei `int`-Zahlen zu einem neuen Datentyp `Rational` kombiniert werden:

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

Der neue Datentyp kann wie gewohnt in Variablendefinitionen verwendet werden:

```
Rational p;
```

Die Namen der einzelnen Komponenten in der Strukturdefinition werden zu *Selektoren* um auf die einzelnen Komponenten zuzugreifen:

```
p.n = 3;
p.d = 4; // gibt die Zahl 3/4
```

Hier ist die allgemeine Syntax:

Syntax 7.9 (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. Wie immer beschreibt das nur einen kleinen Ausschnitt der Möglichkeiten, die man in C++ hat.

Als Beispiel hier ein paar Funktionen um mit rationalen Zahlen zu rechnen.

Programm 7.10 (Rationale Zahlen, die erste) Rational1.cc

```
#include <iostream.h>

struct Rational {
    int n;
    int d;
} ;

Rational kon_rat (int n, int d)              // konstruiere rationale
{                                            // Zahl aus zwei int
    Rational t;

    t.n = n; t.d = d;
    return t;
}

Rational add_rat (Rational p, Rational q)    // die Addition
{
    return kon_rat(p.n*q.d+q.n*p.d,p.d*q.d);
}

Rational sub_rat (Rational p, Rational q)    // die Subtraktion
{
    return kon_rat(p.n*q.d-q.n*p.d,p.d*q.d);
}

Rational mul_rat (Rational p, Rational q)    // die Multiplikation
{
    return kon_rat(p.n*q.n,p.d*q.d);
}

Rational div_rat (Rational p, Rational q)    // die Division
{
    return kon_rat(p.n*q.d,p.d*q.n);
}

void dru_rat (Rational p)                    // Ausgabe als Bruch
{
    cout << p.n << '/' << p.d << endl;
}

int main ()
{
    Rational p=kon_rat(3,4);                 // Beispiel
    Rational q=kon_rat(5,3);
    Rational r;

    dru_rat(p); dru_rat(q);
    r = sub_rat(add_rat(mul_rat(p,q),p),mul_rat(p,p)); // p*q+p-p*p
    dru_rat(r);
}
```

Das Programm druckt folgende Ausgabe:

```
3/4
5/3
1104/768
```

Offensichtlich ist das Ergebnis nicht gekürzt, Damit kann ein Zahlenüberlauf früher als nötig eintreten.

Da wir in allen Funktionen die Funktion `kon_rat` zur Konstruktion einer rationalen Zahl einsetzen, können wir dort leicht eine Normalisierung einbauen, etwa

```
Rational kon_rat (int n, int d)
{
    Rational t;

    if (n*d<0) // Vorzeichen unterschiedlich?
    {
        n=-abs(n); d=abs(d);
    }
    else
    {
        n=abs(n); d=abs(d);
    }
    int g=ggT(abs(n),d); // kuerzen
    t.n = n/g; t.d = d/g;
    return t;
}
```

Angenommen wir wollen nicht nur mit rationalen Zahlen rechnen sondern auch mit komplexen, so können wir das natürlich ganz genauso machen.

Wir könnten natürlich auch auf die Idee kommen die Zahlen zu mischen. Solange wir nur mit rationalen Zahlen rechnen soll das Ergebnis rational bleiben. Wird eine rationale mit einer komplexen kombiniert muss vorher die rationale in eine komplexe Zahl konvertiert werden.

In C++ gibt es eine Reihe von Möglichkeiten *gemischtzahlige Arithmetik* zu realisieren.

Wir könnten für jede mögliche Kombination von Argumenten eine andere Funktion schreiben. Dies ist sehr umständlich.

Wir stellen jetzt eine Variante vor, die mit *etikettierten Daten* arbeitet, d. h. jede Zahl trägt eine Markierung, auch Etikette genannt, die ihren Typ angibt.

Die Etikette selbst ist am besten ein Aufzählungstyp, damit das Programm lesbar bleibt.

Dann benötigen wir noch eine Möglichkeit auszudrücken, dass eine Variable *entweder* eine rationale oder eine komplexe Zahl ist.

Dies erreicht man mit der varianten Struktur:

Syntax 7.11 (Variante Struktur)

<UnionDef> ::= union <Name: > { { <Komponente> ; }$^+$ } ;

Ein `union`-Datentyp ist eine Überlagerung der einzelnen Komponenten. Man kann ihn als genau eine der angegebenen Komponenten verwenden. C++ stellt sicher, dass eine Variable dieses Types jede der möglichen Komponenten aufnehmen kann, aber eben nur genau eine zu einer Zeit.

Hier eine simple Realisierung gemischtzahliger Arithmetik.

Programm 7.12 (Gemischtzahlige Arithmetik, die erste) `Mixed1.cc`

```
#include <iostream.h>

enum Kind {rational, complex};

struct Rational {                          // rationale Zahl
    int n;
    int d;
} ;

struct Complex {                           // komplexe Zahl
    float re;
    float im;
} ;

union Combination {                        // vereinige beide
    Rational p;
    Complex  c;
} ;

struct Mixed {                             // gemischte Zahl
    Kind a;                                // welche bist Du?
    Combination com;                       // benutze je nach Art
} ;

Mixed kon_rat (int n, int d)               // erzeuge rationale Zahl
{                                          // aus zwei int
    Mixed t;

    t.a = rational;
    t.com.p.n = n; t.com.p.d = d;
    return t;
}
```

```
Mixed kon_kom (float re, float im)          // erzeuge komplexe Zahl
{                                           // aus zwei float
    Mixed t;

    t.a = complex;
    t.com.c.re = re; t.com.c.im = im;
    return t;
}

Mixed konvert (Mixed z)                     // konvertiere rationale Zahl
{                                           // in komplexe Zahl
    if (z.a==complex) return z;             // ist schon eine
    Mixed t;
    t.a = complex;
    t.com.c.re = z.com.p.n;
    t.com.c.re = t.com.c.re/z.com.p.d;      // Fliesskommadivision
    t.com.c.im = 0.0;                       // kein Imaginaerteil
    return t;
}

Mixed add (Mixed a, Mixed b)
{
    if (a.a==rational && b.a==rational)     // beide sind rational
        return kon_rat(a.com.p.n*b.com.p.d+b.com.p.n*a.com.p.d,
                       a.com.p.d*b.com.p.d);
    if (a.a==rational)                      // a rational, b komplex
        a = konvert(a);
    if (b.a==rational)                      // b rational, a komplex
        b = konvert(b);
    return kon_kom(a.com.c.re+b.com.c.re,a.com.c.im+b.com.c.im);
}

void dru (Mixed a)
{
    if (a.a==rational)
        cout << a.com.p.n << '/' << a.com.p.d << endl;
    if (a.a==complex)
        cout << a.com.c.re << "+i*" << a.com.c.im << endl;
}

int main ()
{
    Mixed p=kon_rat(3,4);
    Mixed z=kon_kom(3.14,0.66);
    Mixed r;

    dru(p); dru(z);
    r = add(p,z);
    dru(r);
}
```

Diese Lösung hat ein paar Nachteile:

Alle Variablen des Typs `Mixed` brauchen gleichviel Speicherplatz.

Jede arithmetische Operation enthält Wissen über alle Varianten. Kommt eine neue hinzu so ist jede Funktion zu modifizieren.

Im objektorientierten Teil werden wir das Problem der gemischtzahligen Arithmetik nochmal (besser?) behandeln.

8 Globale Variablen und das Umgebungsmodell

8.1 Globale Variablen

Bisher gilt für alle unsere Beispiele

Funktionen haben kein Gedächtnis!

d. h. ruft man eine Funktion zweimal mit den selben Argumenten auf, so liefert sie auch das selbe Ergebnis.

Dies liegt daran, dass

- Funktionen nur von den formalen Parametern abhängen.
- Die lokale Umgebung zwischen Funktionsaufrufen nicht erhalten bleibt.

Das werden wir jetzt ändern!

Warum wollen wir das?

Betrachten wir folgendes Problem: Wir wollen die Abstraktion eines Kontos schaffen.

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

Programm 8.1 (Konto) `konto.cc`

```
#include<iostream.h>

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;
}
```

mit der Ausgabe

```
75
50
25
```

Die Variable `konto` ist ausserhalb jeder Funktion definiert.

Die Variable `konto` wird zu Beginn des Programmes erzeugt und *nie* mehr zerstört.

Die Variable `konto` „kennen“ alle Funktionen. Man sie eine *globale Variable*.

Oben haben wir eingeführt, dass Ausdrücke relativ zu einer Umgebung ausgeführt werden. Bis jetzt gab es allerdings nur lokale Umgebungen einer Funktion.

In welcher Umgebung ist dann `konto`?

Deshalb betrachten wir nun den Begriff der Umgebung noch genauer.

Vorher noch eine Warnung !

Jede Funktion kann globale Variablen verändern (nennt man Seiteneffekt) ohne dass man dies am Rückgabewert erkennen könnte.

Jede Funktion kann von Werten globaler Variablen abhängen.

Damit kann — im Prinzip — jede Stelle eines Programmes Einfluss auf jede andere Stelle im Programm haben. Dies ist eine Freiheit mit der man bewusst umgehen muss:

Keine globale Variable ohne Grund!

Unten folgt ein Beispiel welches illustriert dass globale Variablen durchaus sinnvoll im Sinne der Abstraktion sein können.

8.2 Das Umgebungsmodell

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

Definition 8.2 (Umgebung)

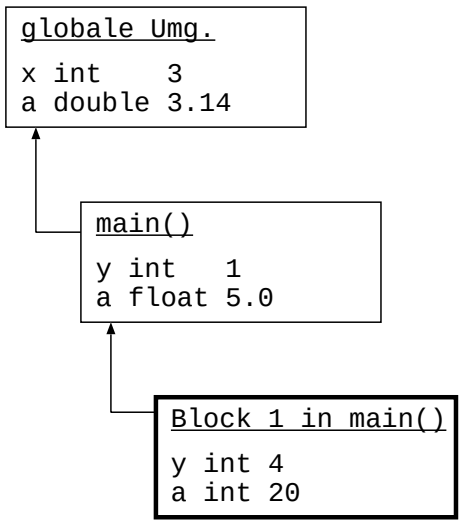
1. *Eine Umgebung enthält eine Bindungstabelle, d. h. eine Zuordnung von Namen zu Werten.*
2. *Es kann beliebig viele Umgebungen geben. Umgebungen werden während des Programmlaufes implizit (automatisch) oder explizit (bewusst) erzeugt bzw. zerstört.*
3. *Die Menge der Umgebungen bildet eine Baumstruktur. Die Wurzel dieses Baumes heisst globale Umgebung.*
4. *Zu jedem Zeitpunkt des Programmablaufes gibt es eine aktuelle Umgebung. Die Auswertung von Ausdrücken erfolgt relativ zur aktuellen Umgebung.*
5. *Die Auswertung relativ zur aktuellen Umgebung versucht den Wert eines Namens in dieser Umgebung zu ermitteln, schlägt dies fehl wird in der nächst höheren (umschliessenden) Umgebung gesucht usw. bis die globale Umgebung erreicht wird.*

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

Das beschreiben wir mit einigen Beispielen. Betrachte

```
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:



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 sagt man, dass das erste Vorkommen die weiteren verdeckt.

Eine Zuweisung wirkt immer auf den *sichtbaren* Namen. Mit vorangestelltem `::` erreicht man die Namen der globalen Umgebung.

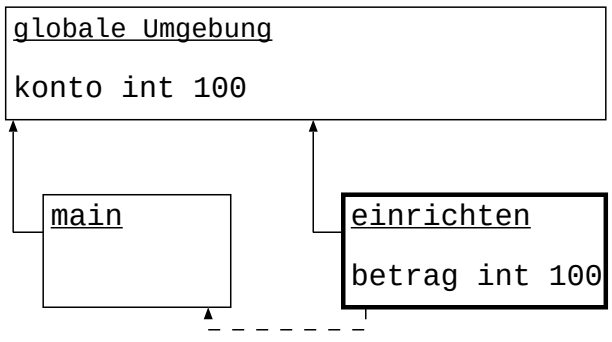
Eine Anweisungsfolge in geschweiften Klammern bildet einen *Block*. Ein Block ist eine eigene Umgebung, die von { erzeugt wird und mit } wieder beendet wird.

Eine Schleife `for (int i=0; ...` wird in einer eigenen Umgebung ausgeführt. Die Variable `i` gibt es im Rest der Funktion nicht.

Was passiert bei einem Funktionsaufruf? Betrachte

```
int konto;

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

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

Nach Marke 2:

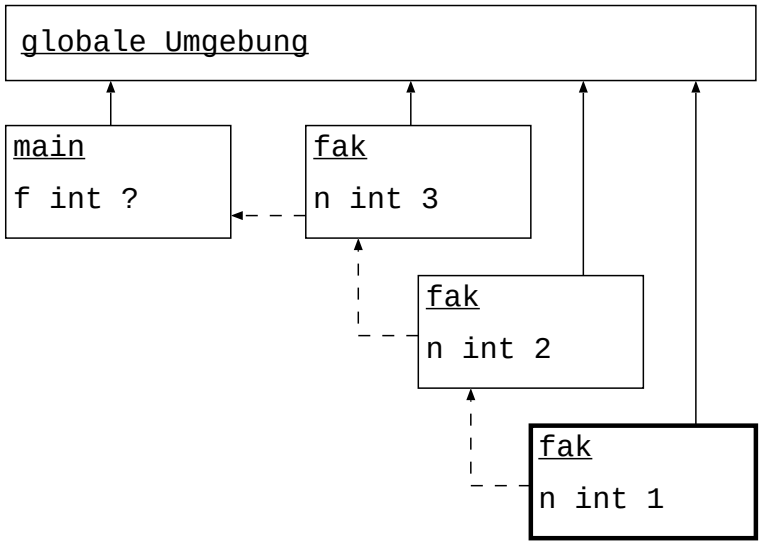


Jeder Funktionsaufruf startet eine neue Umgebung unterhalb der globalen Umgebung. Dies ist dann die aktuelle Umgebung.

Am Ende einer Funktion wird ihre Umgebung vernichtet und die aktuelle Umgebung wird die, in der der Aufruf stattfand (gestrichelte Linie).

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

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

Am Anfang von `fak`:



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:

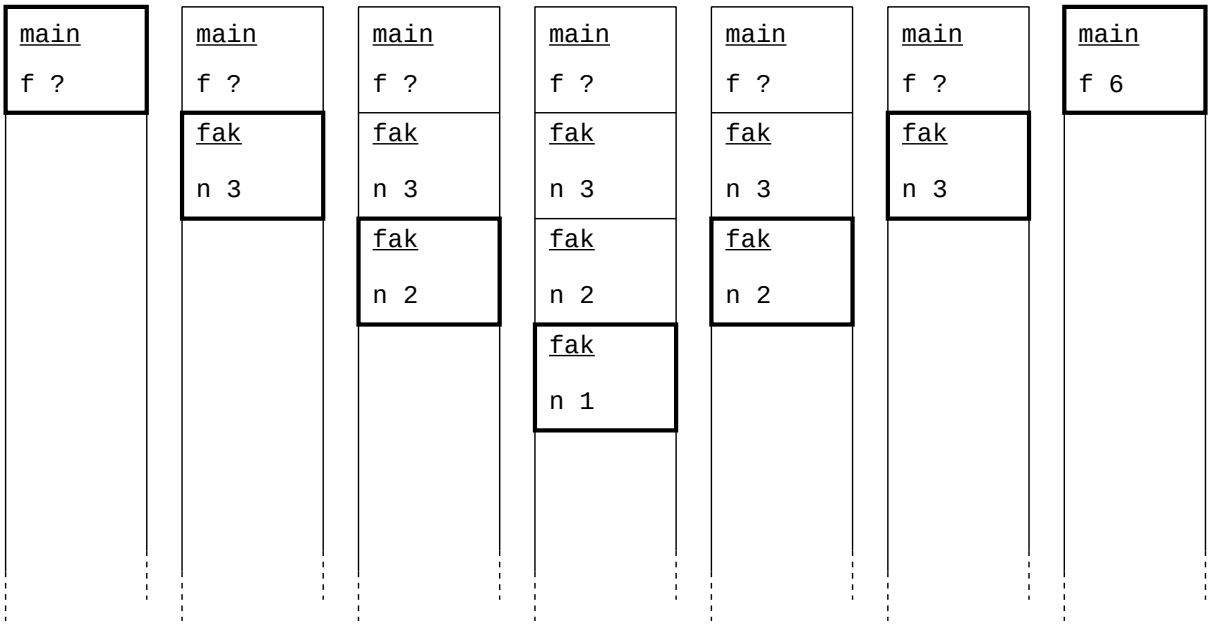
- Namen im aktuellen Block und allen umschließenden Blöcken innerhalb der Funktion sofern nicht verdeckt.
- Namen in der globalen Umgebung.

Im obigen Beispiel gibt es zusätzlich noch eine „versteckte“ Variable für den Rückgabewert einer Funktion. Die `return`-Anweisung ist eine Zuweisung an diese Variable in der Umgebung in der der Aufruf stattfand.

Wie werden die Umgebungen verwaltet?

Immer die zuletzt erzeugte Umgebung wird als erstes wieder gelöscht.

Betrachten wir die Berechnung von `fak(3)`



Eine solche Struktur nennt man einen *Stapel* oder *Stack* oder *LIFO* (last in first out):

Definition 8.3 (Stapel) *Ein Stapel ist eine Struktur, die folgende Operationen zur Verfügung stellt:*

- *Erzeugen eines leeren Stapels.*
- *Einfügen eines neuen Elementes.*
- *Löschen des zuletzt eingefügten Elementes.*
- *Ist der Stapel leer, kann auch kein Element gelöscht werden.*

Diese Technik zur Bearbeitung rekursiver Funktionen kann auch als allgemeines Verfahren zur Konversion rekursiver in nicht rekursive Programme genutzt werden.

Als Beispiel hier das Wechselgeld-Programm:

Programm 8.4 (Wechselgeld nicht rekursiv) `wg-stack.cc`

```
#include"iostream.h"

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

struct Arg {                     // Struktur fuer Stapelelemente
    int betrag;                  // das sind die Argumente der
```

```
    int muenzarten; };          // rekursiven Variante

const int N = 1000;             // Stapelgroesse

int wechselgeld2 (int betrag) {
    Arg stapel[N];              // hier ist der Stapel
    int i=0;                    // der "stack pointer"
    int anzahl=0;               // Anzahl Moeglichkeiten (das Ergebnis)
    int b,m;                    // Hilfsvariablen in Schleife

    stapel[i].betrag = betrag; // initialisiere Stapel
    stapel[i].muenzarten = 5;  // Startwert der rek. Variante
    i = i+1;                    // ein Element mehr

    while (i>0) {               // Solange Stapel nicht leer
        i = i-1;                // stapel[i] ist dann oberstes Element
        b = stapel[i].betrag;  // lese Argumente
        m = stapel[i].muenzarten;

        if ( b==0 )
            anzahl = anzahl+1; // Eine Moeglichkeit gefunden
        else if ( b>0 && m>0 ) {
            if (i>=N) {cout<<"Stapel zu klein"<<endl; return anzahl;}
            stapel[i].betrag = b;            // Betrag b
            stapel[i].muenzarten = m-1;      // mit m-1 Muenzarten wechseln
            i = i+1;

            if (i>=N) {cout<<"Stapel zu klein"<<endl; return anzahl;}
            stapel[i].betrag = b-nennwert(m);// Betrag b-nennwert(m)
            stapel[i].muenzarten = m;        // mit m Muenzarten
            i = i+1;
        }
    }

    return anzahl;              // Stapel ist jetzt leer
}

int main () {
    cout << wechselgeld2(300) << endl;
}
```

Bemerkung: Natürlich hat sich durch diese Transformation die Komplexität nicht geändert.

8.3 Beispiel: Monte-Carlo Methode zur Bestimmung von π

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

Satz 8.5 *Die Wahrscheinlichkeit q, dass zwei natürliche Zahlen u, v keinen gemeinsamen Teiler haben ist $\frac{6}{\pi^2}$. Zu dieser Aussage siehe [Knu98, Vol. 2, Theorem D].*

Um π zu bestimmen gehen wir wie folgt vor:

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

Was sind denn Zufallszahlen und wie erzeugt man welche im Rechner?

Eine Zufallsfolge ist eine Folge von Zahlen $< x_i > \in \mathbb{N}, i = 1, 2, \ldots$ mit $0 \leq x_i < m$, die keine erkennbare Ordnung hat.

Man erwartet dass die Zahlen in der Folge eine gewisse Verteilung haben, z. B. dass jede Zahl gleich oft vorkommt wenn man die Folge genügend lang macht:
$$\lim_{n\to\infty} \frac{|\{i | 1 \leq i \leq n \wedge x_i = k\}|}{n} = \frac{1}{m}, \quad \forall k = 0, \ldots, m-1.$$

Die populärste Methode zur Erzeugung von Folgen zufälliger Zahlen ist folgende Iterationsvorschrift:
$$x_{n+1} = (ax_n + c) \bmod m.$$
Dabei sind a, c frei wählbare Konstanten.

Alles wesentliche zu Zufallszahlen steht in [Knu98, Vol. 2, Kapitel 3].

Programm 8.6 (π mit Monte Carlo Methode) montecarlo1.cc

```
#include<iostream.h>
#include<math.h>

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);
}

void main ()
{
    cout.precision(20);
    cout << sqrt(6.0/montecarlo(1000000)) << endl;
}
```

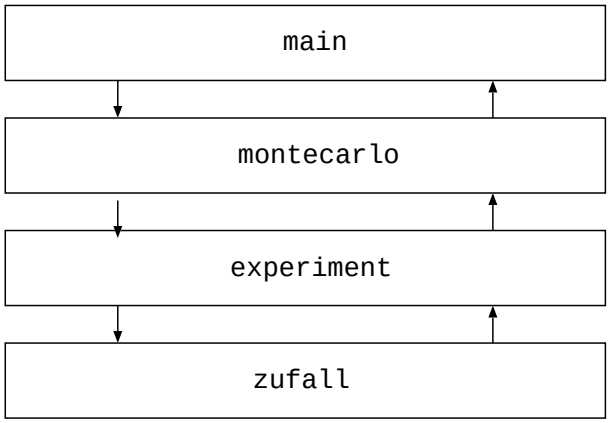
Programm 8.7 (MC ohne globale Variable) montecarlo2.cc

```
#include<iostream.h>
#include<math.h>

unsigned int zufall (unsigned int x)
{
    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);
}

struct ex_resultat {
    unsigned int x;
    int erfolg;
} ;

ex_resultat experiment (unsigned int x)
{
    unsigned int x1,x2;
    ex_resultat t;

    x1 = zufall(x); x2 = zufall(x1); t.x = x2;
    if (ggT(x1,x2)==1)
        t.erfolg=1;
    else
        t.erfolg=0;
    return t;
}

struct mc_resultat {
    unsigned int x;
    double p;
} ;

mc_resultat montecarlo (int N, unsigned int x)
{
    int erfolgreich=0;
    ex_resultat er;
    mc_resultat mr;
```

```
    er.x = x;
    for (int i=0; i<N; i=i+1)
    {
        er = experiment(er.x);
        erfolgreich = erfolgreich+er.erfolg;
    }
    mr.x = er.x;
    mr.p = ((double)erfolgreich)/((double)N);
    return mr;
}

void main ()
{
    mc_resultat mr;

    cout.precision(20);
    mr=montecarlo(1000,93267);
    cout << sqrt(6.0/mr.p) << endl;
    mr=montecarlo(1000000,mr.x);
    cout << sqrt(6.0/mr.p) << endl;
}
```

Wollen wir ohne globale Variable auskommen so müssen wir die aktuelle Iterierte des Zufallsgenerators von `main` aus durch alle Funktionen „durchtunneln“.

Um die Iterierte wieder zurückzubekommen müssen wir zusammengesetzte Datentypen einführen.



Bei Änderungen am Zufallsgenerator (z. B. bei einer Vorschrift mit der Struktur $x_n = f(x_{n-1}, x_{n-2})$) wären alle anderen Funktionen auch betroffen.

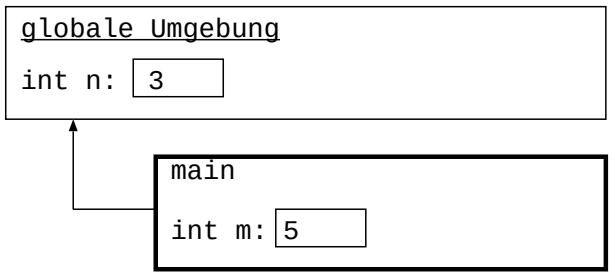
Hier sind globale Variablen ein Mittel zur Verbesserung der Programmstruktur!

9 Zeiger und dynamische Datenstrukturen

9.1 Zeiger

Wir wollen uns eine Umgebung als Sammlung von Schubladen (Orten) vorstellen, die Werte aufnehmen können. Jede Schublade hat einen Namen (der Variablenname), einen Typ (sagt was hinein kann) und einen Wert (Inhalt, was drin ist):

```
int n=3;

void main ()
{
    int m=5;

}
```



Es wäre nun praktisch wenn man so etwas wie „ Nehme den Wert (Inhalt) *dieser* Schublade (Variable) und zähle eins dazu“ ausdrücken könnte.

Anwendung: Im Konto-Beispiel möchten wir nicht nur ein Konto sondern viele Konten verwenden. Natürlich will man nur einen Satz von Funktionen schreiben, die Konten manipulieren. Hierzu benötigt man einen Mechanismus, der einem auszudrücken erlaubt, *welches* Konto verändert werden soll.

Dieser Mechanismus sind Zeigervariablen !

Den Wert einer Zeigervariablen stellen wir uns als einen Pfeil oder eben *Zeiger* auf eine andere Variable vor.

```
int *x;
```

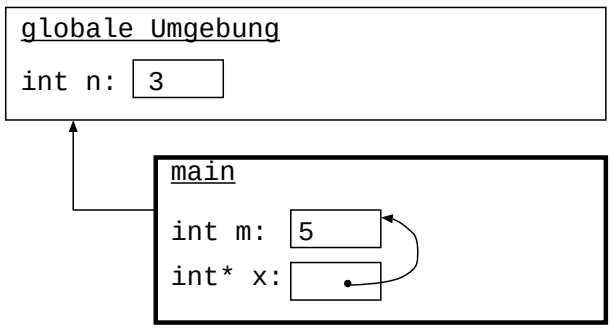
vereinbart, dass x auf Variablen (Schubladen) vom Typ `int` zeigen kann. Man sagt `x` habe den Typ `int*`,

Die Zuweisung `x = &n;` lässt `x` „auf den Ort von `n` zeigen“.

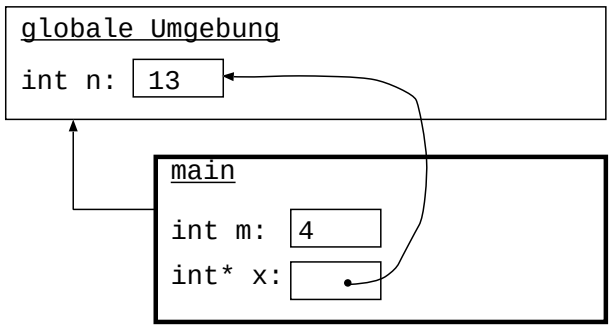
Die Zuweisung `*x = 4;` verändert den Wert der Schublade „auf die `x` zeigt“.

```
int n=3;

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

Nach (2)



Nach (5)



9.2 Zeiger und die Bindungstabelle

(Kleine Wiederholung) 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:

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

Statt $\mathtt{n}, \mathtt{m}$ könnten wir genausogut $\square$ und $\lozenge$ nehmen.

Die Zuweisung $\mathtt{n} = 3$; („=“ 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 $\mathtt{n} = \mathtt{m}+1$; dann soll nach der Zuweisung

$$w(\mathtt{n}) = w(\mathtt{m}) + 1$$

gelten. Auf jeden Namen wird also w angewandt.

Betrachte folgende Situation: Wir haben mehrere verschiedene Konten (z. B. `n` und `m`) und wir möchten einen allgemeinen Code schreiben, der einen Betrag von einem Konto abhebt. In einer Variable wollen wir angeben, von *welchem* Konto abgehoben werden soll.

Wir könnten das Problem lösen, wenn *Namen selbst wieder als Werte von (anderen) Namen zugelassen wären*, wie etwa in folgender Bindungstabelle:

Name	Wert	Typ
`n`	3	`int`
`m`	5	`int`
`x`	`n`	?

Wie können wir den Wert von `x` setzen? Eine Zuweisung `x = n;` tut es nicht, da dies ja die Bindungstabelle so ändert, dass

$$w(\texttt{x}) = w(\texttt{n}).$$

Wir müssten rechts die Anwendung von w *verhindern* können. Dazu führen wir einen speziellen einstelligen Operator & („Adressoperator“) ein:

$$\texttt{x} = \&\texttt{n};$$

ändert die Bindungstabelle so dass

$$w(\texttt{x}) = \texttt{n}.$$

Wie weisen wir nun dem Konto, dessen Name der Wert von `x` ist, den neuen Wert 4 zu? D. h. wir wollen dass nach der Zuweisung

$$w(\underbrace{w(\texttt{x})}_{\texttt{n}}) = 4$$

gilt.

Auf der linken Seite ist also die Abbildung w einmal mehr anzuwenden. Dies leistet der neue einstellige Operator $*$ („Dereferenzierungsoperator“), also

$$*\texttt{x} = 4;$$

Zusammengefasst:

- 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.

C++ ist eine streng typgebundene Sprache, d. h. jedem Namen muss ein Typ zugeordnet werden. Der Typ ist für die gesamte Lebensdauer des Namens nicht veränderbar. Welchen Typ hat dann unsere Variable `x`?

Wir vereinbaren `int*` als Typ von `x`. Dies steht für „Name einer `int`-Variable“.

Werte von `x` sind Namen von `int`-Variablen. `x` heisst *Zeigervariable.*

Beispiel:

```
int n=100;    // Konto n
int m=200;    // Konto m

int* x = &m;  // x hat m als Wert
*x = *x-25;   // 25 von m abheben
x = &n;       // x hat nun n als Wert
*x = *x-30;   // 30 von n abheben

n = *x-40;    // nochmal 40 von n abheben
*&m = *&n;    // umstaendlich fuer m = n !
```

Wir können das ganze natürlich iterieren, d. h. auch `x` kann wieder Wert einer Variablen sein.

Diese hat dann den Typ `int**` oder „Name einer `int*`-Variable“

usw.:

```
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
```

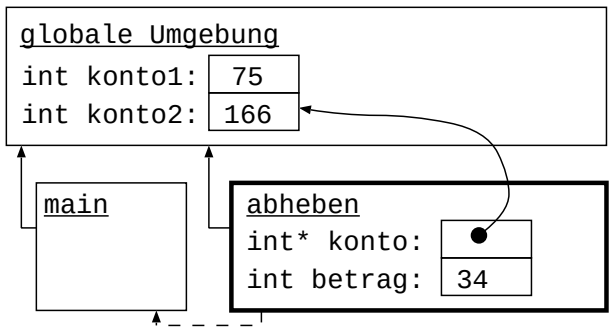
9.3 Call by reference

Realisieren wir nun das Bankenprogramm, das mehrere Konten verwalten kann.

Wir möchten unserer Prozedur `abheben` ein Argument mitgeben, das bestimmt von welchem Konto der Betrag abgehoben werden soll. Genau das gelingt mit einer Zeigervariablen:

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

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

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

Nach (1), zweiter Aufruf von `abheben`

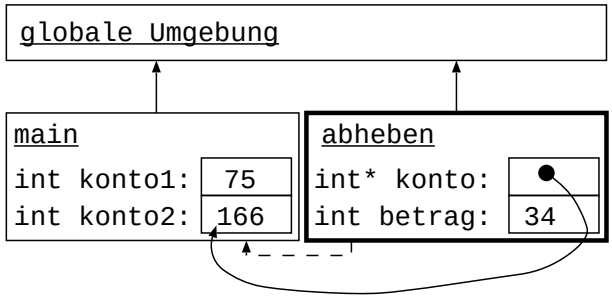


`betrag` nennt man einen *call by value* Parameter, `konto` einen *call by reference* Parameter.

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
}

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

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

Nach (1), zweiter Aufruf von `abheben`



`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.

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).

Referenzen in C++

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.

So genannte *Referenzen* vereinfachen dies:

```
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
}
```

```
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
}
```

Beide Programme verhalten sich identisch!

Referenzvariablen verhalten sich wie eine Zeigervariable, nur dass man den `*` *nicht* zu schreiben braucht.

Nur die Funktionsdefinition entscheidet, ob ein Parameter per Referenz übergeben wird, am Funktionsaufruf ist nichts Spezielles zu tun.

Referenzen können nicht nur als Funktionsargumente benutzt werden:

```
int  n=3;
int& r=n; // independent reference

r = 5;    // identisch zu n=5;
```

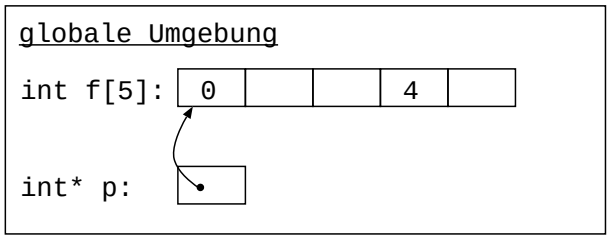
Aber: *Der Ort auf den eine Referenz zeigt kann nicht geändert werden. Unabhängige Referenzen müssen initialisiert werden.*

9.4 Zeiger und Felder

Zeiger und (eingebaute) Felder sind in C/C++ synonym, d. h. der Name eines Feldes kann einer Zeigervariablen zugewiesen werden und Zeigervariablen können indiziert werden:

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

...

p[0]=0;
p[3]=4;
```

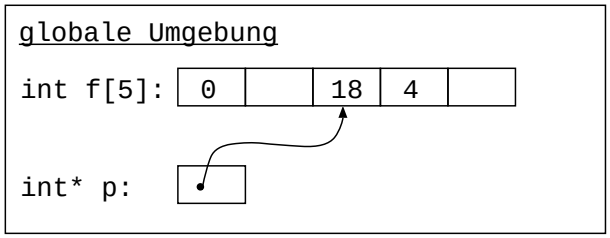
Am Ende



p kann auf ein beliebiges Feldelement zeigen.

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

Am Ende



In C/C++ wird bei eingebauten Feldern *keine* Bereichsprüfung durchgeführt.

9.5 Zeiger und zusammengesetzte Datentypen

Natürlich sind auch Zeiger auf Variablen zusammengesetzter Typen möglich:

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

void main ()
{
    rational  q;
    rational* p;

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

Ist `p` ein Zeiger auf eine Variable eines zusammengesetzten Datentyps so kann man mittels `p-><Komponente>` eine Komponente selektieren.

9.6 Dynamische Speicherverwaltung

Bis jetzt gibt es 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.

Es gibt noch eine dritte Art, die *dynamischen* Variablen.

Diese werden vom Programmierer explizit erzeugt und vernichtet. Dazu dienen die Operatoren `new` und `delete`.

Dynamische Variablen haben keinen Namen und können nur indirekt über Zeiger bearbeitet werden:

```
int m;
rational* p;

p = new rational;

p->n = 4; p->d = 5;
m = p->n;

delete p;
```

Die Anweisung `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 sind notwendig um Strukturen im Rechner zu erzeugen, deren Größe sich während der Rechnung ergibt (siehe unten).

Andererseits bergen sie eine Reihe von Fallen:

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

Hier wurde eine dynamische Variable vom Typ `rational` erzeugt, der einzige Zeiger auf diese Variable wird jedoch automatisch mit Beenden der Funktion gelöscht.

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

Hier wird mittels des Zeigers auf eine Variable zugegriffen, die bereits gelöscht worden ist.

Das 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.

Daher:

- Nur verwenden wenn unbedingt nötig.
- Manipulation der Variablen und Zeiger in Funktionen (später: Klassen) verpacken, die konsistente Behandlung sicherstellen.
- Benutzung spezieller Zeigerklassen (*smart pointers*).
- *Garbage collection*, löst zumindest das Problem nicht mehr zugänglicher Variablen.

Dynamische Variablen werden weder in der globalen noch einer lokalen Umgebung sondern auf dem so genannten *Heap* gespeichert.

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
```

9.7 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 (für Mathematiker: endliche Folge)

(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.

Übrigens: Wir nehmen hier kein Feld als Datenstruktur, da wir annehmen, dass die Anzahl der Elemente zur Übersetzungszeit des Programmes unbekannt ist.

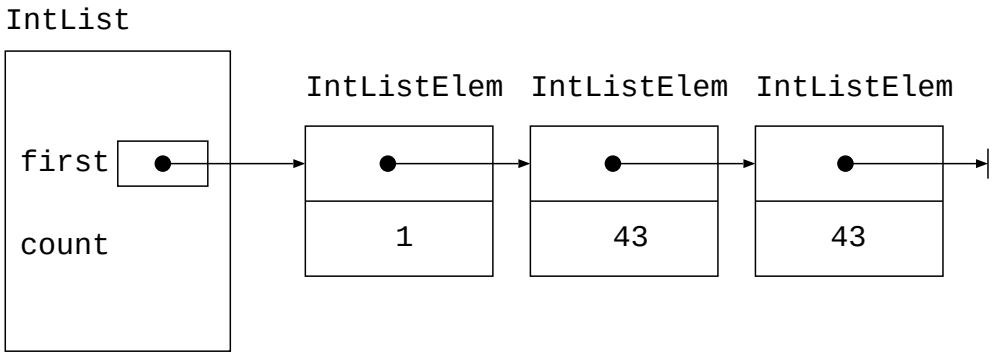
Eine übliche Methode zur Speicherung von Listen (natürlicher Zahlen) besteht darin ein *Listenelement* zu definieren, das ein Element der Liste sowie einen Zeiger auf das nächste Listenelement enthält:

```
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.

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

```
ListElem* 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;
}
```

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

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

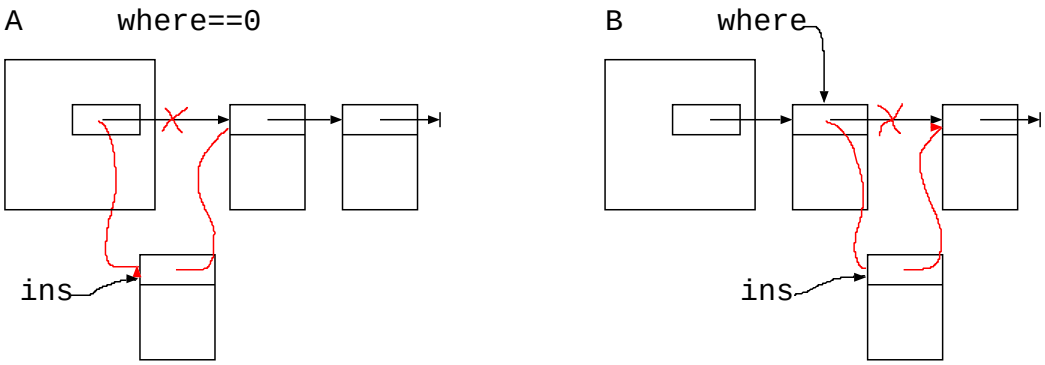
Beachte, dass die Liste call-by-reference übergeben wird um die Komponenten ändern zu können.

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:



Folgende Funktion behandelt beide Fälle. Hat `where` den Wert 0, so wird `ins` am Anfang eingefügt, sonst nach dem Element auf das `where` zeigt:

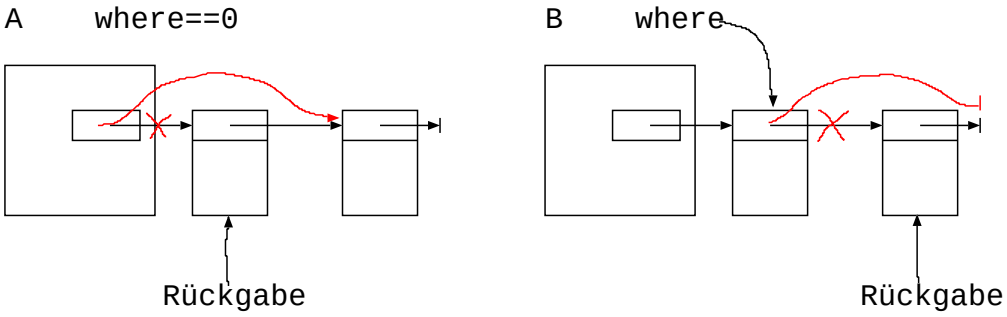
```
void insert_in_list (IntList* list, IntListElem* where, IntListElem* ins)
{
    if (where==0) // fuege am Anfang ein
    {
        ins->next = list->first;
        list->first = ins;
        list->count = list->count + 1;
    }
    else          // fuege nach where ein
    {
        ins->next = where->next;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

Auch beim Entfernen von Elementen unterscheiden wir wieder ob

A das erste Element gelöscht werden soll, oder

B das Element *nach* einem gegebenem.

entsprechend graphisch:



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;
}
```

Dabei wird angenommen, dass `where` ein Element der Liste ist.

Alle Funktionen auf der Liste beinhalten *nicht* die Speicherverwaltung für die Objekte. Dies ist Aufgabe des benutzenden Programmteiles.

Varianten der Liste: doppelt verkettet, einfügen am Ende, zirkuläre Listen, ...

9.8 Endliche Menge

Als Anwendungsbeispiel für die einfach verkettete Liste wollen wir die Abstraktion einer endlichen Menge realisieren.

Das komplette Programm befindet sich in der Datei `intset.cc`.

Im Gegensatz zu einer Liste kommt es bei einer endlichen Menge

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

- nicht auf die Reihenfolge der Mitglieder an und
- Elemente können auch nicht doppelt vorkommen!

Als Operationen auf einer Menge benötigen wir

- Erzeugen einer leeren Menge.
- Einfügen eines Elementes.
- Entfernen eines Elementes.
- Mitgliedschaft in der Menge testen.

Wir benutzen zur Realisierung der Menge die eben vorgestellte einfach verkettete Liste. Um deutlich zu machen, dass eine Menge keine Liste ist definieren wir einen neuen Datentyp:

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

Wir verstecken damit auch, dass wir `IntSet` mittels `IntList` realisieren.

Die Funktion für die leere Menge allokiert ein Objekt vom Typ `IntSet`, initialisiert es und liefert es zurück:

```
IntSet* empty_set ()
{
    IntSet* s = new IntSet;

    empty_list(&s->list);
    return s;
}
```

Der Test auf Mitgliedschaft durchläuft die Liste und liefert wahr, wenn das Element gefunden wird, ansonsten falsch:

```
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;
}
```

Dies nennt man *sequentielle Suche*. Der Aufwand ist $O(n)$ wenn die Liste n Elemente hat. Im statistischen Mittel wird man $n/2$ Listenelemente durchlaufen müssen bis man ein Element gefunden hat.

Später werden wir bessere Methoden kennenlernen, die das in $O(\log n)$ Aufwand schaffen.

Das Einfügen muss erst testen, ob das Element bereits in der Menge ist, ansonsten wird es am Anfang der Liste eingefügt.

Beachte, dass diese Funktion auch die `IntListElem`-Objekte dynamisch erzeugt:

```
void insert_in_set (IntSet* s, int x)
{
    // pruefe ob Element bereits in Liste
    // sonst fuege es ein
    if (!is_in_set(s,x))
    {
        IntListElem* p = new IntListElem;
        p->value = x;
        insert_in_list(&s->list,0,p);
    }
}
```

Eine Funktion zum Darstellen der Menge gibt es auch:

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

Zum Entfernen benötigt man einen Zeiger auf den Vorgänger des zu löschenden Elementes in der Liste:

```
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;
        }
}
```

Der `#include`-Befehl setzt die angegebene Datei an der Stelle ein, so als hätte man den Text dort selbst geschrieben:

```
#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);
}
```

```
{ }
{ 11 10 9 8 7 6 5 4 3 2 1 }
{ 11 9 7 5 3 1 }
```

10 Beispiel: Huffman Kodes

Als weitere Anwendung von Zeigern, zusammengesetzten Datentypen und dynamischer Speicherverwaltung betrachten wir die Kodierung von Zeichen durch Bitfolgen mit so genannten Huffman Kodes.

Betrachte folgendes Problem: Wir wollen die Zeichenfolge

'ABRACADABRASIMSALABIM'

durch eine Folge von Zeichen aus der Menge $\{0, 1\}$ darstellen (kodieren).

Jedem der 9 Zeichen aus der Eingabekette ist 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.

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.

Ein Beispiel ist der Morsekode.

Etwa könnten wir folgenden Kode in unserem obigen Beispiel verwenden:

A	1	D	010	M	100
B	10	I	11	R	101
C	001	L	011	S	110

Damit kodieren wir unsere Beispielkette als

$$\underbrace{1}_{\text{A}}\underbrace{10}_{\text{B}}\underbrace{101}_{\text{R}}\underbrace{1}_{\text{A}}\underbrace{001}_{\text{C}}\ldots$$

Allerdings bekommen wir Schwierigkeiten bei der Dekodierung, da wir diese Bitfolge auch interpretieren könnten als

$$\underbrace{110}_{\text{S}}\underbrace{101}_{\text{R}}\underbrace{100}_{\text{M}}\ldots$$

Hier gibt es zwei Möglichkeiten das Problem zu umgehen: Man führt zusätzliche Trennzeichen zwischen den Zeichen ein (etwa die Pause beim Morsekode), oder man sorgt dafür, dass kein Kode für ein Zeichen der Anfang (Präfix) eines anderen Zeichens ist. Einen solchen Kode nennt man *Präfixkode*.

Man kann sich fragen: Wie sieht der optimale Präfixkode für eine gegebene Zeichenfolge aus, d. h. ein Kode der die gegebene Zeichenkette mit einer Bitfolge minimaler Länge kodiert. Die Antwort sind die Huffmankodes!

Ein Huffmankode (sie sind nicht eindeutig) für unsere Beispielkette ist

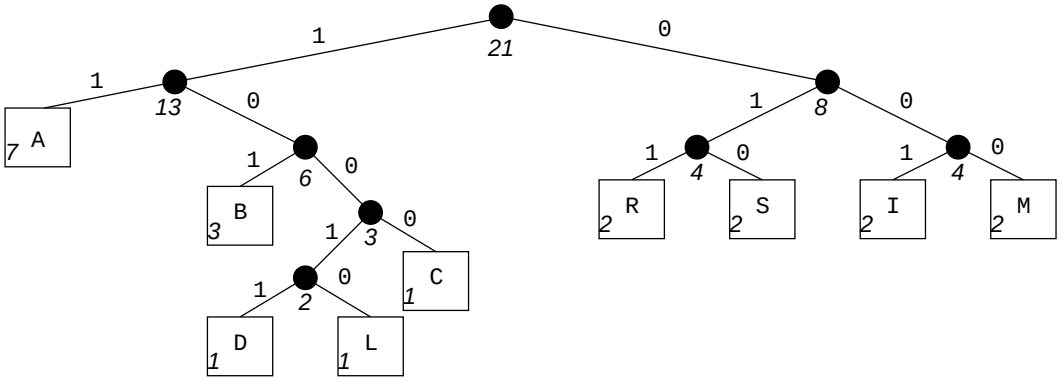
A	11	D	10011	M	000
B	101	I	001	R	011
C	1000	L	10010	S	010

Die kodierte Nachricht lautet

11101011111000111001111101011110100010000101110010111010010 00

und hat 61 Bits!

Ein Huffmankode wird durch einen binären Baum dargestellt, der hier auch *Trie* genannt wird. In den Blättern stehen die zu kodierenden Zeichen. Ein Pfad von der Wurzel zu einem Blatt kodiert das dem Blatt entsprechende Zeichen:



Zeichen, die häufig vorkommen stehen nahe bei der Wurzel, solche, die seltener vorkommen, stehen tiefer im Baum. Der Huffmanbaum wird wie folgt konstruiert:

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.

Nun zum Programm zur Erzeugung des Huffmanbaumes. Dies ist sicher weder das eleganteste noch das effizienteste Programm zur Erzeugung von Huffmankodes aber immerhin illustriert es eine Reihe der bisher besprochenen Konstruktionselemente.

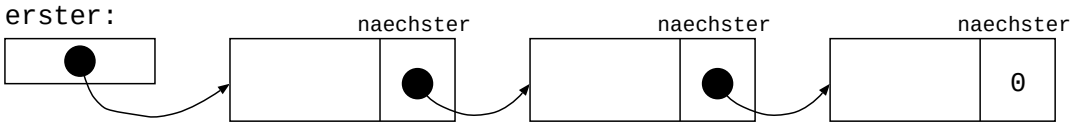
Die Knoten des Huffmanbaumes stellen wir mit einem zusammengesetzten Datentyp dar:

huffman-ds.cc

```
struct Knoten {
    char Zeichen;         // Blatt: zugehoeriges Zeichen
    int haeufigkeit;      // wie oft ist es in der Nachricht
    std::string code;     // dem Zeichen zugeordneter code
    Knoten* naechster;    // für Liste in Aufbauphase
    Knoten* wurzel;       // zur Wurzel dieses Knotens
    Knoten* links;        // linker Zweig
    Knoten* rechts;       // rechter Zweig
} ;
```

In der Aufbauphase müssen wir die Wurzeln der bisher gebildeten Teilbäume in einer Menge verwalten. Da jeweils die beiden Knoten mit der kleinsten Häufigkeiten entnommen werden müssen sollte die Menge aufsteigend sortiert werden (*priority queue*). Dies heben wir uns für später auf!

Stattdessen implementieren wir die Menge als *einfach verkettete Liste*:



mit folgenden Funktionen:

huffman-menge.cc

```
Knoten *erster=0;                       // Das ist unsere Menge

bool leer ()                            // liefert wahr falls Menge
{                                       // leer ist
    if (erster==0)
        return true;
    else
        return false;
}

void loesche ()                         // loesche Menge
{                                       // (etwas grob, zugegeben)
    erster=0;
}
```

```
void einfuegen (Knoten* el)                 // fuege ein Element ein
{
    el->naechster=erster;
    erster=el;
    return;
}

Knoten* entnehme_kleinstes ()               // Entferne Element mit
{                                           // kleinster Haeufigkeit
    Knoten* p;
    Knoten* r;

    // leere Menge ?
    if (leer()) return 0;

    // finde kleinste Haeufigkeit
    int min = 10000000;                     // SEHR grosse Zahl
    for (p=erster; p!=0; p=p->naechster)    // laufe durch Menge
        if (p->haeufigkeit<min)             // Kleineres gefunden?
            min=p->haeufigkeit;             // merke Minimum

    // entferne Element mit kleinster Haeufigkeit
    if (erster->haeufigkeit==min)           // erstes == kleinstes?
    {
        // entferne erstes Element
        p=erster;
        erster=p->naechster;
        return p;
    }
    else                                    // erstes != kleinstes
    {
        // Finde p: Nachfolger von p ist kleinstes Element
        p=erster;                           // p!=0 in jedem Fall
        while (p->naechster!=0)
        {
            if (p->naechster->haeufigkeit==min)
            {
                r = p->naechster;           // Entferne Nachfolger
                p->naechster=p->naechster->naechster;
                return r;
            }
            else p=p->naechster;
        }
        return 0; // sollte nicht passieren
    }
}
```

Hier kommt die Prozedur zum Aufbau des Huffmanbaumes. Sie beinhaltet die folgenden Phasen:

- Bestimmen der Häufigkeiten der Zeichen.
- Initialisieren der Knotenmenge.
- Verknüpfen der beiden Knoten mit kleinster Häufigkeit.

huffman-baum.cc

```
Knoten* erzeuge_huffman_baum (std::string nachricht)
{
    // bestimme Haeufigkeiten
    int h[128];                              // Abbildung: char -> int

    // Erstelle Haeufigkeitstabelle
    for (int i=0; i<128; i++) h[i] = 0;      // Loesche Tabelle
    for (unsigned int i=0; i<nachricht.size(); i=i+1)
        h[nachricht[i]] = h[nachricht[i]]+1;// Erhoehe um eins

    // Erzeuge Blätter, fülle Blätter in Liste
    Knoten* neu;
    loesche();                               // loesche globale Liste
    for (int i=0; i<128; i++)                // fuer alle
        if (h[i]>0)                          //   vorkommenden Zeichen
        {
            // erzeuge neuen Knoten
            neu = new Knoten;

            // initialisiere Knoten
            neu->Zeichen = i;
            neu->haeufigkeit = h[i];
            neu->wurzel = 0;
            neu->links = 0;
            neu->rechts = 0;

            // Knoten in Menge
            einfuegen(neu);
        }

    // Erzeuge Huffman-Baum
    Knoten* links;
    Knoten* rechts;
    while (!leer())
    {
        // entnehme die zwei kleinsten Elemente
        links = entnehme_kleinstes();      // es gibt immer mindestens
        rechts = entnehme_kleinstes();     // zwei Elemente in der Liste

        // fasse zwei Knoten zusammen
        neu = new Knoten;                  // erzeuge neuen Knoten

        // initialisiere Knoten
        neu->haeufigkeit = links->haeufigkeit+rechts->haeufigkeit;
```

```
        neu->wurzel = 0;
        neu->links = links;
        neu->rechts = rechts;
        links->wurzel = neu;
        rechts->wurzel = neu;

        if (leer())                          // gibts noch weitere Knoten?
        {
            return neu;                      // erzeuge codes
        }
        else
            einfuegen(neu);                  // ja, mache weiter
    }

    // Hier sollte man NIE hinkommen!
    return 0;
}
```

Wenn wir einen Huffmanbaum aufgebaut haben so enthalten die Blätter des Baumes die Kodes für die jeweiligen Zeichen. Um die Zeichen einer Nachricht schnell kodieren zu können erstellen wir eine Tabelle, die jedem Zeichen seinen Kode zuordnet.

huffman-code.cc

```
void ec (std::string *tabelle, Knoten *k)
{
    // Am Blatt drucke Code
    if (k->links==0)                         // Jeder Knoten hat
    {                                        // 0 oder 2 Nachfolger !
        cout << "Zeichen: '" << k->Zeichen
             << "', Code: "  << k->code << endl;
        tabelle[k->Zeichen] = k->code;       // Eintrag in Tabelle
        return;
    }

    // setze code in linkem Zweig
    k->links->code = k->code + "0";          // haenge 0 an
    ec(tabelle,k->links);                    // rekursiver Aufruf

    // setze code in rechtem Zweig
    k->rechts->code = k->code + "1";         // haenge 1 an
    ec(tabelle,k->rechts);                   // rekursiver Aufruf
}

void erzeuge_codes (std::string *tabelle, Knoten *wurzel)
{                                            // Aufruf mit Wurzel
    wurzel->code = "";                       // leerer code
    ec(tabelle,wurzel);                      // starte Rekursion
}
```

Nun ist die Eingabenachricht einfach zu kodieren:

huffman-kodieren.cc

```
void kodiere_nachricht (Knoten* wurzel, std::string nachricht)
{
    std::string tabelle[128];               // Zeichen -> Code
    int n=0;                                // Laenge kodierte Nachricht

    // Erzeuge code Tabelle
    erzeuge_codes(tabelle,wurzel);

    // arbeite Nachricht ab
    for (unsigned int i=0; i<nachricht.size(); i=i+1)
    {
        cout << tabelle[nachricht[i]];
        n = n + tabelle[nachricht[i]].size();
    }

    // statistik
    cout << endl << endl;
    cout << "Kodierte Nachricht hat " << n << " Bits" << endl;
}
```

Zum Schluss das Hauptprogramm. Dieses bindet die übrigen Datein mittels `#include` ein. Nur `huffman-main.cc` ist zu übersetzen!

huffman-main.cc

```
#include<iostream.h>
#include<string>
#include<fstream.h>

#include"huffman-ds.cc"
#include"huffman-menge.cc"
#include"huffman-baum.cc"
#include"huffman-code.cc"
#include"huffman-kodieren.cc"

int main ()
{
    std::string nachricht = "ABRACADABRASIMSALABIM";
    Knoten* wurzel = erzeuge_huffman_baum(nachricht);
    kodiere_nachricht(wurzel,nachricht);
}
```

Hier ist die Ausgabe des Programmes für unser Beispiel:

```
Zeichen: 'M', Code: 000
Zeichen: 'I', Code: 001
Zeichen: 'S', Code: 010
```

```
Zeichen: 'R', Code: 011
Zeichen: 'C', Code: 1000
Zeichen: 'L', Code: 10010
Zeichen: 'D', Code: 10011
Zeichen: 'B', Code: 101
Zeichen: 'A', Code: 11
11101011111000111001111101011110100010000101110010111010010000

Kodierte Nachricht hat 61 Bits
```

Kritik am obigen Programm

- Keine klaren Schnittstellen zum Huffmanbaum. Alle Prozeduren arbeiten mehr oder weniger direkt auf der Datenstruktur.
- Menge ist global sichtbar obwohl sie nur in der Aufbauphase gebraucht wird.
- Code des Zeichens könnte mit Bitoperationen effizienter implementiert werden.
- Liste und Binärbaum sind oft auftretende *Datenstrukturen*. Kann man sie so implementieren, dass sie wiederverwendbar sind?

Weitere Details zu Huffmanbäumen, siehe [Sed92].

TEIL III

OBJEKTORIENTIERTE PROGRAMMIERUNG

11 Klassen

11.1 Klassen aus Benutzersicht

Klassendefinition

Bis jetzt hatten wir

- Funktionen/Prozeduren als *aktive* Entitäten, und
- Daten als *passive* Entitäten.

(Prozedur: Funktion ohne Rückgabewert, Seiteneffekt ist das Wesentliche)

Wir denken: *Prozeduren operieren auf Daten.*

Denke etwa an folgende Realisierung eines Kontos:

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

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

Kritik an dieser Realisierung:

- 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.

Das Konzept der *Klasse* erlaubt es uns

> *Daten und Funktionen zu einer Einheit zu verbinden.*

Hier das Konto als Klasse:

```
class Konto {
public:
    int konto_stand ();
    int abheben (int betrag);
private:
    int k;        // das Konto
} ;
```

Syntax 11.1 (Klassendefinition) Die allgemeine Syntax der Klassendefinition lautet

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

Im Rumpf werden sowohl Variablen als auch Funktionen aufgeführt. Bei den Funktionen genügt der Kopf.

Die Funktionen einer Klasse heißen *Methoden*. Alle Komponenten (Daten und Methoden) heißen *Mitglieder*.

Die Klassendefinition

- beschreibt aus welchen Daten eine Klasse besteht
- und welche Operationen auf diesen Daten ausgeführt werden können.
- belegt (in diesem Fall) keinen Speicherplatz.

Objektdefinition

Die Klasse kann man sich als Bauplan vorstellen, der im Fall von `Konto` beschreibt aus was ein `Konto` besteht und was man damit anfangen kann.

Nach diesem Bauplan werden *Objekte* erstellt, die dann im Rechner existieren. Objekte heißen auch *Instanzen einer Klasse*.

Objektdefinitionen sehen aus wie Variablendefinitionen:

```
Konto k1;      // ein Konto
Konto *pk=&k1; // Zeiger auf das Konto
```

Die Klasse erscheint also wie ein neuer Datentyp.

Der Methodenaufruf wird folgendermaßen geschrieben:

```
k1.abheben(25);                    // Methodenaufruf
cout << pk->konto_stand() << endl; // Aufruf durch Zeiger
```

Methoden werden wie Komponenten eines zusammengesetzten Datentyps selektiert und mit Argumenten wie eine Funktion versehen.

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

Kapselung

Unser Konto:

```
class Konto {
public:
    int konto_stand ();
    int abheben (int betrag);
private:
    int k;      // das Konto
} ;
```

Der Rumpf einer Klassendefinition zerfällt in zwei Teile:

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

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.

```
void main ()
{
  Konto k1;

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

So ist ein Konto eben keine simple `int`-Variable mehr, die beliebig manipuliert werden kann. Der Zustand des Kontos kann nur noch durch die vorgegebenen Methoden manipuliert werden.

Diese Trennung nennt man Kapselung (*encapsulation*).

Der öffentliche Teil einer Klasse ist die Schnittstelle der Klasse zum restlichen Programm. Der private Teil der Klasse enthält Mitglieder die zur Implementierung der Schnittstelle benutzt werden.

Kapselung erlaubt uns das Prinzip der *versteckten Information* (*information hiding*) zu realisieren.

Parnas [CACM, 15(12): 1059-1062, 1972] hat dieses Grundprinzip (damals bei 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.*

Insbesondere sollte eine Klasse alle Implementierungsdetails verstecken, die sich möglicherweise in der Zukunft ändern werden.

Hierzu ein Zitat von *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.

Obwohl das Zitat reichlich alt ist, die Trennung von Algorithmen und Datenstrukturen wird in aktueller Software konsequent verfolgt (z. B. der *Standard Template Library*, STL).

Als wichtigste Regel sollte man sich also merken:

Halte Datenstrukturen geheim!

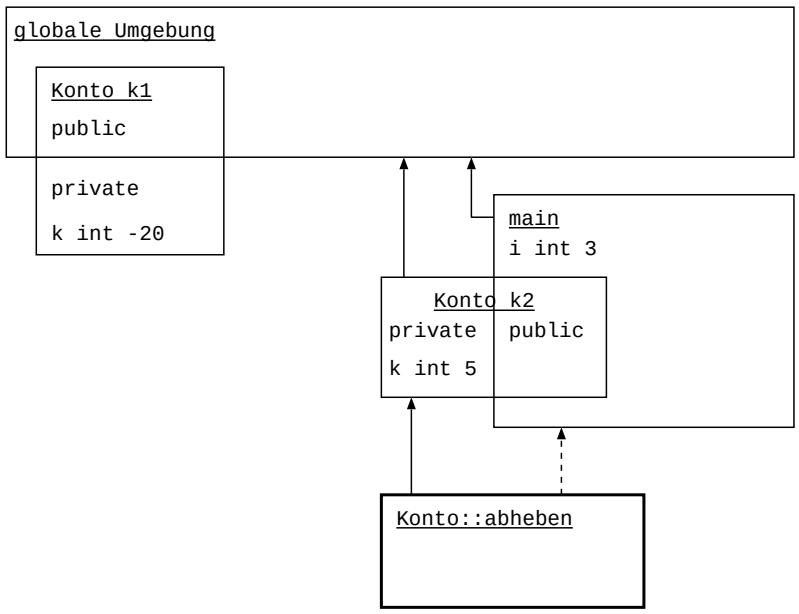
Eine Klasse versteckt also nicht primär Daten vor dem Benutzer einer Klasse, sondern vor allem *wie* diese Daten in den Objekten gespeichert werden.

Deshalb sollte eine Klasse normalerweise *keine* öffentlichen Datenmitglieder haben.

Klassen im Umgebungsmodell

```
class Konto; // wie oben

Konto k1;    // global

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

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

In `k2.abheben(-5)`



Jedes Objekt definiert eine eigene Umgebung.

Die öffentlichen Daten einer Objektumgebung überlappen mit der Umgebung in der das Objekt definiert ist. Diese Daten sind dort auch sichtbar.

Der Methodenaufruf erzeugt eine neue Umgebung unterhalb der Umgebung des zugehörigen Objektes

Also

- 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).

Die Lebensdauer von Objekten ist genauso geregelt wie die von Variablen:

- Globale Objekte exisitieren solange wie das Programm läuft.
- Lokale Objekte für die Dauer des Funktionsaufrufes.
- Dynamische Objekte solange bis `delete` aufgerufen wird.

11.2 Implementierung von Klassen

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 zwei Methoden besitzt:

- 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 Speicher, den das Objekt belegt, freigegeben wird. Der Destruktor kann entsprechende Aufräumarbeiten durchführen (Beispiele folgen).

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.

Eingebaute Typen und Zeigertypen besitzen keine Konstruktoren und werden somit nicht initialisiert [Stroustrup, p. 243].

Ein Beispiel für eine Klassendefinition mit Konstruktor und Destruktor:

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

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
```

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`:

`Konto.cc`

```
#include"iostream.h"

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

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

Konto::~Konto ()
{
    cout << "Konto mit " << k << " aufgeloest" << endl;
}

int Konto::konto_stand ()
{
    return k;
}

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

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

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

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.

Selbstreferenz

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

Somit wäre

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

eine umständliche, aber völlig gleichwertige Implementierung von `abheben`.

Will man innerhalb einer Methode eine Methode desselben Objektes aufrufen kann man schreiben:

```
int Konto::einzahlen (int betrag) // neue Methode
{
  abheben(-betrag); // Methode desselben Objektes aufrufen
  return k;         // neuer kontostand
}
```

was dasselbe ist wie

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

Allgemeiner: Alle Namen, die innerhalb einer Klasse definiert sind (später: nicht nur Methoden und Daten sondern auch andere Klassen, Aufzählungstypen usw.), sind innerhalb der Mitglieder einer Klasse (z. B. einer Methode) sichtbar (also ohne scope resolution `::`).

11.3 Beispiel: Monte-Carlo objektorientiert

Wir betrachten nochmal das Beispiel der Bestimmung von π mit Hilfe von Zufallszahlen (Seite 81).

Dort hatten wir drei Dinge

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

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

`Zufall.cc`

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

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

unsigned int Zufall::ziehe_zahl ()
{
    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.
}
```

Durch die Angabe des Konstruktors ist sichergestellt, dass der Generator initialisiert werden muss (Es gibt dann keinen Default-Konstruktor!).

Die Realisierung des Generators ist nach aussen nicht sichtbar (`x` private). Wir könnten leicht aus zwei alten Werten einen neuen berechnen.

Hier die Klasse für das Experiment: `Experiment.cc`

```
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,x2;

    x1 = zg.ziehe_zahl(); x2 = zg.ziehe_zahl();
    if (ggT(x1,x2)==1)
        return 1;
    else
        return 0;
}
```

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.

Schließlich das Hauptprogramm: `MonteCarlo.cc`

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

```
#include<math.h>

#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.precision(20);
    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 (hätte in C Funktionszeiger erfordert, die wir nicht eingeführt haben).

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.

Syntax 11.2 (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.

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 dann eben die Default-Version.

Dies nennt man hierarchische Konstruktion/Destruktion.

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

Anwendung auf unser Beispiel:

`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.

11.4 Überladen und Operatoren

Überladen von Funktionen und Methoden

C++ erlaubt es mehrere Funktionen *gleichen Namens* aber mit unterschiedlichen Argumenten zu definieren.

Hier ein Beispiel:

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

Dabei bestimmt der Übersetzer anhand der Typen der Argumente welche Funktion aufgerufen wird. Der Rückgabewert ist dabei unerheblich, d. h. für jeden Satz von Argumenten kann man nur eine Funktion definieren!

Diesen Mechanismus nennt man *Überladen* von Funktionen.

Leider macht die *automatische Konversion* eingebauter numerischer Typen den Prozess der Auswahl der richtigen Funktion in manchen Fällen etwas schwierig.

Der Übersetzer geht 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 beiden 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 dritten Beispiel oben wird `c` nach `int` konvertiert.
3. Versuche Standardkonversionen (Informationsverlust!) anzuwenden: `int` nach `double`, `double` nach `int` usw.

Verwende Überladen möglichst nur mit exakter Übereinstimmung!

Gibt es verschiedene Möglichkeiten auf *einer* Stufe so gibt das einen Fehler (viertes Beispiel oben).

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
```

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: Überladene Funktionen (Methoden) haben zwar den gleichen Namen, können aber völlig verschiedene Dinge tun! Um einen Benutzer des Programmes nicht zu verwirren sollte man das aber bleiben lassen!

Operatoren

Für zwei `int`-Variablen können wir (natürlich!)

`a+b`

schreiben um die Summe zu berechnen.

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

Dies ist insbesondere bei Klassen sinnvoll, die mathematische Konzepte realisieren, wie etwa rationale Zahlen (siehe unten), Vektoren, Polynome, Matrizen, gemischtzahlige Artihmetik, Arithmetik beliebiger Genauigkeit, usw.

Dieses Feature von C++ ist nicht unbedingt lebenswichtig, aber es ermöglicht einem Klassen zu schreiben die sich wie eingebaute Typen verwenden lassen. Insbesondere können auch eckige Klammern `[]`, Dereferenzierung `->`, Vergleichsoperatoren `<,>,==` und sogar die Zuweisung `=` (um-)definiert werden. Doch davon später.

Damit ein Code wie

```
class X { ... };

X a,b,c;
c = a+b;
```

Sinn macht, wird der Ausdruck `a+b` interpretiert als

```
a.operator+(b)
```

d. h. die Methode `operator+` des Objektes `a` (des linken Operanden) wird mit dem Argument `b` (rechter Operand) aufgerufen.

Hier die Definition der Klasse `X`:

```
class X {
public:
  X operator+ (X b);
};

X X::operator+ (X b)
{ .... }
```

Der `operator+` ist also ein ganz normaler Methodenname. Nur dass der Methodenaufruf aus der Infixschreibweise generiert wird.

So könnten wir etwa das Abheben in unserer Klasse `Konto` alternativ durch den `-`-Operator realisieren:

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

```
  void operator- (int betrag);
private:
  int k;     // Zustand
} ;

void Konto::operator- (int betrag)
{
  abheben(betrag)
}

Konto k1(100);
k1-35; // k1.abheben(35);
```

Allerdings ist das ein schlechtes Beispiel, denn man sollte Operatoren nur einsetzen wo eingebaute Typen auch einen solchen Operator besitzen. Bei den Operatoren `+`, `-`, `*`, `/` sind das eben Klassen, die mathematische Objekte mit solchen Operatoren realisieren.

Beispiel: Rationale Zahlen objektorientiert

Als Anwendung betrachten wir nochmal die rationalen Zahlen. So könnte eine simple Klasse für rationale Zahlen aussehen:

`Rational.cc`

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

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

    // Ausgabe
    void print ();

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

Hier die Implementierung der Methoden `RationalImp.cc`

```
// Zugriff auf Komponenten von aussen
int Rational::nom ()
{
    return n;
}

int Rational::denom ()
{
    return d;
}

// Darstellung (geht eleganter)
void Rational::print ()
{
    cout << n << "/" << d << endl;
}

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

// Konstruktoren
Rational::Rational (int nom, int denom)
{
    int t=ggT(nom,denom);
    if (t!=0)
    {
        n=nom/t;
        d=denom/t;
    }
    else
    {
        n = nom;
        d = denom;
    }
}

Rational::Rational (int nom)
{
    n=nom;
    d=1;
}

Rational::Rational ()
{
    n=0;
    d=1;
```

```
}

// Operatoren
Rational Rational::operator+ (Rational q)
{
    return Rational(n*q.denom()+q.nom()*d,d*q.denom());
}

Rational Rational::operator- (Rational q)
{
    return Rational(n*q.denom()-q.nom()*d,d*q.denom());
}

Rational Rational::operator* (Rational q)
{
    return Rational(n*q.nom(),d*q.denom());
}

Rational Rational::operator/ (Rational q)
{
    return Rational(n*q.denom(),d*q.nom());
}
```

Und hier ein lauffähiges Beispiel `UseRational.cc`

```
#include <iostream.h>

#include "Rational.cc"
#include "RationalImp.cc"

void main ()
{
    Rational p(3,4),q(5,3),r;

    p.print(); q.print();
    r = (p+q*p)*p*p;
    r.print();

}
```

Die Implementierung einer leistungsfähigen gemischtzahligen Arithmetik (inklusive langer Zahlen) ist eine spannende Aufgabe, hier jedoch zu komplex.

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

Beginnen wir mit dem Band. Hier eine Klasse, die das Wesentliche des Bandes erfasst: Band.h

```
// Eine Klasse, die das Band einer Turingmaschine realisiert
// Bandalphabet: char

class Band {
public:
    enum Groesse {N=1000};              // definiere Groesse
    Band (char *name, char init);       // lese Datei, initialer Bandinh.
    char lese ();                       // liefere aktuelles Zeichen
    void schreibe_links (char symbol);  // schreibe und gehe links
    void schreibe_rechts (char symbol); // schreibe und gehe rechts
    void drucke ();                     // drucke den Bandinhalt
private:
    char band[N];                       // realisiert als statisches Feld
    int pos;                            // aktuelle Position
    int maxpos;                         // maximal beschriebene Position
} ;
```

Die Programmtabelle könnte durch die folgende Klasse realisiert werden:

Programm.h

```
// Eine Klasse, die das Programm einer Turingmaschine realisiert
// Zustaende sind vom Typ int
// Bandalphabet ist der Typ char

class Programm {
public:
    enum G {N=1000};                    // definiere Groesse der Tabelle
    enum R {links,rechts};              // Namen fuer links und rechts gehen

    Programm (char *name);              // lese Tabelle von Datei
                                        // q ist ein Zustand
    char Ausgabe (int q, char symbol);  // Abb (q,sym) -> Ausgabe
    R Richtung (int q, char symbol);    // Abb (q,sym) -> Richtung
    int Folgezustand (int q, char symbol); // Abb (q,sym) -> Folgezustand

    int Anfangszustand ();              // Anfang des Programmes
    int Endzustand ();                  // Ende des Programmes

    void drucke ();                     // Drucke gelesenes Programm

private:
    bool FindeZeile(int q, char symbol);// table lookup
```

```
    int zeilen;                         // Anzahl der gelesenen Zeilen
    int Qaktuell[N];                    // Die Programmtabelle
    char eingabe[N];                    // als statisches Feld
    char ausgabe[N];
    R richtung[N];
    int Qfolge[N];

    int letztesQ;                       // Cache fuer zuletzt
    int letzteEingabe;                  // zugegriffene Zeile in der
    int letzteZeile;                    // Programmtabelle
} ;
```

Und schließlich die Klasse für die Turingmaschine: TM.h

```
// Klasse, die eine Turingmaschine realisiert

class TM {
public:
    TM (Programm& p, Band& b);          // braucht Programm und Band
    void Schritt ();                    // Mache einen Schritt
    bool Endzustand ();                 // Ist Maschine im Endzustand?
private:
    Programm& prog;                     // mein Programm
    Band& band;                         // mein Band
    int q;                              // aktueller Zustand
} ;
```

. . . und ihre Implementierung: 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;
}
```

Das Hauptprogramm, welches entsprechende Objekte erzeugt:

Turingmaschine.cc

```
#include"iostream.h"
#include"fstream.h"

#include"Band.h"
#include"Band.cc"
#include"Programm.h"
#include"Programm.cc"
#include"TM.h"
#include"TM.cc"

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.
}
```

Als Beispiel hier die Turingmaschine zum Verdoppeln einer Einserkette von Seite 7.

Das Programm lautet

```
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
```

und für die Eingabe

```
1111
```

liefert die Maschine folgende Ausgabe:

```
Band: [1]111
Band: X[1]11
Band: X1[1]1
Band: X11[1]
Band: X111[0]
Band: X11[1]Y
Band: X1[1]1Y
Band: X[1]11Y
Band: [X]111Y
Band: 1[1]11Y
Band: 1X[1]1Y
Band: 1X1[1]Y
Band: 1X11[Y]
Band: 1X11Y[0]
Band: 1X11[Y]1
Band: 1X1[1]Y1
Band: 1X[1]1Y1
Band: 1[X]11Y1
Band: 11[1]1Y1
Band: 11X[1]Y1
Band: 11X1[Y]1
Band: 11X1Y[1]
Band: 11X1Y1[0]
Band: 11X1Y[1]1
Band: 11X1[Y]11
Band: 11X[1]Y11
Band: 11[X]1Y11
Band: 111[1]Y11
Band: 111X[Y]11
Band: 111XY[1]1
Band: 111XY1[1]
Band: 111XY11[0]
Band: 111XY1[1]1
```

```
Band: 111XY[1]11
Band: 111X[Y]111
Band: 111[X]Y111
Band: 1111[Y]111
Band: 11111[1]11
```

Diese Realiserung könnte noch verbessert werden:

- Band könnte seine Größe dynamisch verändern.
- Statt eines einseitig unendlichen Bandes könnten wir auch ein zweiseitig unendliches Band realisieren.
- Die Notation der Programme könnte um die Möglichkeit von Unterprogrammen erweitert werden.
- Das Finden einer Tabellenzeile könnte durch bessere Datenstrukturen beschleunigt werden.

Besonders wichtig ist jedoch, dass alle diese Änderungen jeweils nur die Implementierung einer einzelnen Klasse betreffen. Solange die Schnittstelle nicht geändert wird sind die anderen Klassen nicht betroffen.

Das Essentielle am Programmieren ist somit der Entwurf der Schnittstellen!

11.5 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.

Das Konzept des ADT ist unabhängig von einer Programmiersprache, die Beschreibung kann in natürlicher Sprache abgefasst werden.

Der ADT beschreibt *was* die Operationen tun, aber nicht *wie* sie das tun. Getreu dem Prinzip der versteckten Information ist die Realisierung nicht Teil des ADT.

Vom Standpunkt des Abstraktionsgedankens aus ist der ADT ein mächtigeres Konzept als die Funktion (Kapitel 2).

Die Klasse ist der Mechanismus zur Konstruktion von abstrakten Datentypen in C++. Allerdings fehlt dort die Beschreibung der Semantik der Operationen! Diese wollen wir – so oft es geht – als Kommentar *unter* jede Funktion schreiben.

Beispiel 1: Positive m-Bit-Zahlen im Computer

Wir wollen genau beschreiben wie sich positive ganze Zahlen im Computer verhalten:

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 $q * b \leq a$.

Die Definition des ADT stützt sich auf die Definition der natürlichen Zahlen.

In C++ (auf einer 32-Bit Maschine) entsprechen `unsigned char`, `unsigned short`, `unsigned int` den Werten $m = 8, 16, 32$.

Beispiel 2: Der Stack

Ein Stack ist über einer Menge X erklärt.

Der ADT Stack ist folgendermaßen definiert:

- 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. $top(push(S, x)) = x$
3. $pop(push(S, x)) = S$

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:

$$top(pop(push(push(push(empty(), x_1), x_2), x_3))) =$$

$$top(push(push(empty(), x_1), x_2)) = x_2$$

(Eliminiere alle $pop(push(\ldots))$ Paare nach Regel 3 und wende Regel 2 an)

Auch nicht gültige Folgen lassen sich erkennen:

$$pop(pop(push(empty(), x_1))) = pop(empty())$$

und dafür gibt es keine Regel!

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), Priority Queue, usw.

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 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 *numIndices* liefert $|I|$.
- Operation $isIndex : \mathbb{N} \to \{w, f\}$. $isIndex(i)$ liefert wahr falls $i \in I$.

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.
- 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 $i \neq j$.

In unserer Definition darf $I \subset \mathbb{N}$ beliebig gewählt werden. Es sind also auch nichtzusammenhängende Indexmengen erlaubt.

Eine Variante würde auch die Manipulation der Indexmenge erlauben (die Indexmenge sollte dann als weiterer ADT definiert werden).

Zusammenfassung

- Klassen verbinden Daten und Funktionen (Methoden genannt) zu einer logischen Einheit
- Klassen können benutzt werden um abstrakte Datentypen zu realisieren.
- Der öffentliche Teil einer Klasse stellt die Schnittstelle zur Benutzung des Datentyps bereit
- Der private Teil einer Klasse erlaubt es, den Zugriff auf implementierungspezifische Mitglieder zu verhindern (information hiding).
- Konstruktoren verhindern die Konstruktion von nicht sauber initialisierten Objekten.
- Operatoren ermöglichen es Klassen zu schreiben, die sich wie eingebaute Datentypen verwenden lassen.

12 Klassen und dynamische Speicherverwaltung

In diesem Abschnitt wollen wir Klassen mit dynamisch allokierten Mitgliedern betrachten. Als konkrete Anwendung wollen eine Klasse vorstellen, die das Konzept eines Feldes realisiert.

12.1 Nachteile eingebauter Felder

Es gibt ja schon eingebaute Felder in C/C++, die aber eine Reihe von Nachteilen haben:

- Ein eingebautes Feld kennt seine Größe nicht, diese muss immer extra mitgeführt werden, was ein Konsistenzproblem mit sich bringt.
- Die Größe von eingebauten Feldern muss zur Übersetzungszeit bekannt sein, ansonsten müssen sie dynamisch allokiert werden.
- Bei dynamischen Feldern ist der Programmierer für die Freigabe des Speicherplatzes verantowrtlich.
- Eingebaute Felder haben eine zusammenhängende Indexmenge, die immer bei 0 (Null) beginnt. Was wenn $I = \{110, 1000, 100000\}$, sog. *dünnbesetzte* Indexmenge?
- Felder können nur bei *by value* an eine Prozedur übergeben werden wenn aktueller und formaler Parameter die Größe angeben:

```
void g (int f[5])
{ }

int main ()
{
  int a[5];
  int b[6];

  g(a); // a wird kopiert
  g(b); // Fehler
}
```

- Ansonsten werden eingebaute Felder immer *by reference* übergeben. In diesem Fall wird die Äquivalenz von Feldern und Zeigern ausgenutzt.
- Eingebaute Felder prüfen nicht ob der Index im erlaubten Bereich liegt.

Einige dieser Nachteile werden wir mit der folgenden Klasse beheben.

Nichts ändern werden wir bei dieser ersten Version an der Indexmenge. Diese bleibt $I = [0, n-1]$.

12.2 Klassendefinition

Unsere Feldklasse soll Elemente des Grundtyps `float` aufnehmen. Hier ist die Klassendefinition:

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

    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
} ;
```

Der private Teil offenbart die zur Implementierung notwendigen Mitglieder:

`n` wird die Größe des Feldes sein.

p ist ein Zeiger auf ein eingebautes Feld, d. h. unsere Implementierung wird das eingebaute Feld nutzen.

12.3 Konstruktoren, Destruktoren und Indexmenge

Hier ein Teil der Funktionen `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)
{
    if (i>=0 && i<n) return true;
    else return false;
}
```

In realen Programmen treten manchmal Ereignisse ein, die einen normalen Programmverlauf unmöglich machen.

Dabei gilt es zwei Dinge zu trennen:

- Erkennen des Ereignisses.
- Angemessene Reaktion auf das Ereignis.

Oft weiss man an der Stelle des Erkennens des Ereignisses nicht wie man darauf reagieren soll. Insbesondere dann, wenn das Erkennen innerhalb einer Funktion erfolgt.

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.

Fehlerbehandlung (Reaktionen auf Ausnahmen) ist in einem großen, professionellen Programm extrem wichtig!

Wir gehen hier auf dieses Thema (sträflicherweise!) nicht weiter ein und begnügen uns mit einer Ausgabe auf `cout`.

12.4 Indizierter Zugriff

Der ADT Feld beschreibt die zwei Operationen *read* und *write* zum Auslesen und manipulieren der Abbildung.

Bei eingebauten Feldern werden beide Operatoren durch Indizierung und Zuweisung realisiert:

```
x = 3*a[i]+17.5; // Werte a am Index i aus
a[i] = 3*x+17.5; // Manipuliere Abbildung am Index i
```

Unsere neue Klasse soll sich in dieser Beziehung wie ein eingebautes Feld verhalten. Dies gelingt durch die Definition eines Operators `operator[]`:

`SimpleFloatArrayIndex.cc`

```
float& SimpleFloatArray::operator[] (int i)
{
    return p[i];
}
```

`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.

12.5 Copy–Konstruktor

Schließlich ist zu klären was beim Kopieren von Feldern passieren soll. Hier gilt es zwei Situationen zu unterscheiden:

1. Es wird ein *neues, nicht initialisiertes* 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;
  ```

- Exceptions (werden wir nicht behandeln)

2. Kopieren eines Objektes auf ein bereits initialisiertes 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.

Hier unser Copy-Konstruktor: `SimpleFloatArrayCopyCons.cc`

```
SimpleFloatArray::SimpleFloatArray (const SimpleFloatArray& a)
{
    // Erzeuge Feld mit selber Groesse wie a
    n = a.n;
    try {
        p = new float[n];
    }
    catch (std::bad_alloc) {
        cerr << "nicht genug Speicher!" << endl;
        return;
    }

    // und kopiere Elemente
    for (int i=0; i<n; i=i+1) p[i]=a.p[i];
}
```

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, da es kein anderes Objekt mit demselben Zeiger geben kann.

```
int f ()
{
  SimpleFloatArray a(100,0.0); // Feld mit 100 Elementen
  SimpleFloatArray b=a;        // Copy-Konstruktor wird aufgerufen

  ... // mach etwas schlaues
} // Destruktor von a und b rufen delete[] ihres eingeb. Feldes auf
```

Wir erkennen, dass wir mit der dynamischen Speicherverwaltung der eingebauten Felder nichts mehr zu tun haben, so 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. Dann müsste man zählen wieviele Referenzen auf das dynamisch allokierte Unterobjekt zeigen. Immer wenn eine Referenz gelöscht wird, wird der Zähler um eins dekrementiert. Ist der Zähler 0 dann kann das dynamisch allokierte Objekt auch gelöscht werden.

Dies nennt man *reference counted object pointers*.

12.6 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 Objekt rechts auf.

Hier unser Zuweisungsoperator: `SimpleFloatArrayAssign.cc`

```
SimpleFloatArray& SimpleFloatArray::operator= (const SimpleFloatArray& a)
{
    // Wird fuer ein bereits initialisiertes Feld aufgerufen.

    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;
            try {
                p = new float[n];
            }
            catch (std::bad_alloc) {
                cerr << "nicht genug Speicher!" << endl;
                return *this;
```

```
            }
        }
        for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }
    return *this; // Gebe Referenz zurueck damit a=b=c; klappt
}
```

Haben beide Felder unterschiedliche Größe, so wird für das Feld links vom Zuweisungszeichen ein neues eingebautes Feld der korrekten Größe erzeugt.

Alle Feldelemente werden kopiert.

Beispiel

Hier ein Beispiel zu Illustration `UseSimpleFloatArray.cc`

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

#include "SimpleFloatArray.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

void show (SimpleFloatArray f)
{
    cout << f.maxIndex() << endl;

    for (int i=f.minIndex(); i<=f.maxIndex(); i++)
        cout << f[i] << 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
}
```

Die Funktion `show` arbeitet auf Feldern beliebiger Größe. Jeder Aufruf der Funktion `show` kopiert das Argument mittels des Copy-Konstruktors.

Die Zuweisung von Feldern arbeitet dank dem Zuweisungsoperator auch bei Feldern verschiedener Größe.

Der Benutzer wird nicht mehr mit dynamischer Speicherverwaltung konfrontiert.

12.7 Default-Methoden

Für bestimmte Methoden einer Klasse `T` erzeugt der Übersetzer automatisch Default-Methoden, sofern man keine eigenen definiert.

Dies sind

- Argumentloser Konstruktor `T ();`
  Wird erzeugt wenn man keinen anderen Konstruktor (ausser dem Copy-Konstruktor) angibt.

  Hinweis: Alle Konstruktoren, ob default oder selbst programmiert, rufen immer rekursiv zuerst Konstruktoren der Unterobjekte auf bevor der Rumpf ausgeführt wird. Unterobjekte von eingebauten Typen bleiben uninitialisiert.

- Copy-Konstruktor `T (const T&);`
  Kopiert alle Mitglieder in das neue Objekt (*memberwise copy*).

- Destruktor `~T ();`
  Ruft rekursiv die Destruktoren der Unterobjekte auf.

- Zuweisungsoperator `T& operator= (const T&);`
  Kopiert alle Mitglieder des Quellobjektes auf das Zielobjekt (*memberwise copy*).

- Adress-of-Operator (`&`) mit Standardbedeutung.

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.

13 Vererbung von Schnittstelle und Implementierung

13.1 Motivation: Polynome

Angenommen wir hätten die Aufgabe eine Klasse für Polynome

$$p(x) = \sum_{i=0}^{n} p_i x^i, \quad p_i \in \mathbb{R}$$

zu entwickeln.

Polynome zeichnen sich aus durch

- Grad n und
- Koeffizienten $p_0, \ldots, p_n$.
- Raum aus dem die Koeffizienten zu wählen sind, hier $\mathbb{R}$.
- Wieviele Variablen, hier eine, das x.

Auf Polynomen braucht man folgende Operationen (Vorschlag):

- 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

$$\begin{aligned} r(x) = p(x) * q(x) &= \left(\sum_{i=0}^{n} p_i x^i\right)\left(\sum_{j=0}^{m} q_j x^j\right) \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} p_i q_j x^{i+j} \\ &= \sum_{k=0}^{m+n} \underbrace{\left(\sum_{\{(i,j)|i+j=k\}} p_i q_j\right)}_{r_k} x^k \end{aligned}$$

Wie implementieren wir eine entsprechende Klasse `Polynomial`?

Für den Koeffizientenvektor $p_0, \ldots, p_n$ wäre offensichtlich ein Feld der adäquate Datentyp. Damit wären auch die Methoden Konstruktion und Manipulation erledigt.

Die Methoden für Auswertung, Addition und Multiplikation könnten aufbauend auf den Methoden des Feldes realisiert werden.

Idee: *Kopiere* Klasse `SimpleFloatArray`, nenne alles in `Polynomial` um und füge die zusätzlichen Methoden hinzu.

Problem: Fällt uns eine Verbesserung von `SimpleFloatArray` ein, so ist alles in `Polynomial` nachzuziehen. Bauen beide Implementierungen auf gemeinsamen Vereinbarungen auf, so können sich durch Inkonsistenzen Fehler ergeben.

Daher:

Vermeide das Vervielfältigen von Programmteilen!

13.2 Öffentliche Vererbung

C++ bietet einen Mechanismus der zur Vermeidung von Codevervielfältigung eingesetzt werden kann: Die *Vererbung*.

(Wir werden später weitere Anwendungen kennenlernen.)

Wir definieren die Klasse `Polynomial` mittels Vererbung: `Polynomial.cc`

```
class Polynomial :
    public SimpleFloatArray {
public:
    Polynomial (int n);
```

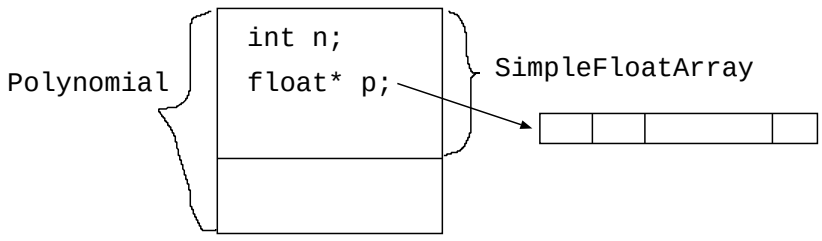
```
    // konstruiere Polynom vom Grad n

    // Default-Destruktor ist ok

    Polynomial (const Polynomial&);
    // Copy-Konstruktor

    Polynomial& operator= (const Polynomial&);
    // Zuweisung von Polynomen

    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
} ;
```

Man sagt die Klasse `Polynomial` wird *öffentlich* von `SimpleFloatArray` abgeleitet (wegen des Schlüsselwortes `public` nach dem ersten Doppelpunkt).

Dies bedeutet:

- `Polynomial` enthält ein Objekt der Klasse `SimpleFloatArray` als Unterobjekt.
- `Polynomial` besitzt alle öffentlichen Methoden von `SimpleFloatArray` inklusive deren Implementierung. Diese Methoden operieren auf dem Unterobjekt vom Typ `SimpleFloatArray` und müssen in `Polynomial` nicht aufgeführt werden. *Jedoch gilt folgende Ausnahme*: Konstruktoren, Destruktor und Zuweisungsoperatoren werden *nicht* vererbt.

  Graphisch stellen wir uns das so vor:

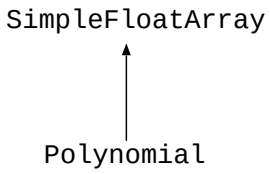


- Zusätzlich kann `Polynomial` noch weitere Mitglieder (Methoden, Daten, öffentlich und privat) haben.

Zusätzlich sagt man

- `Polynomial` ist eine *abgeleitete Klasse*.
- `SimpleFloatArray` ist eine *öffentliche Basisklasse* von `Polynomial` (eine Klasse kann mehrere Basisklassen haben, diesen Fall behandeln wir aber nicht).
- Da `SimpleFloatArray` seine komplette Implementierung an `Polynomial` vererbt spricht man auch von einer *Implementierungsbasisklasse*.

`SimpleFloatArray` und `Polynomial` stehen in einer hierarchischen Beziehung. Dies drückt man in *Klassendiagrammen* aus:



Es gibt eine ausgefeilte Notation für Klassendiagramme, die Teil der *Unified Modelling Language* (UML) ist.

Syntax 13.1 (Öffentliche Vererbung) Die Syntax der öffentlichen Vererbung lautet:

<ÖAbleitung> ::= class <Klasse2> : public <Klasse1> {
<Rumpf>
} ;

Alle öffentlichen Mitglieder der Klasse 1 (Basisklasse) mit Ausnahme von Konstruktoren, Destruktor und Zuweisungsoperatoren sind damit auch öffentliche Mitglieder der Klasse 2 (abgeleitete Klasse).

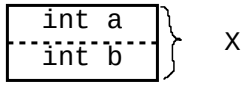
Im Rumpf kann Klasse 2 weitere Mitglieder vereinbaren.

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.

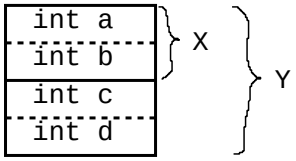
Beispiel zu public/private und öffentlicher Vererbung

```
class X {
public:
  int a;
  void A();
private:
  int b;
  void B();
} ;
```




```
X x;

x.a = 5;    // OK
x.b = 10;   // Fehler, b private

void X::A ()
{
  B();      // OK
  b = 3;    // OK
}
```

```
class Y : public X {
public:
  int c;
  void C();
private:
  int d;
  void D();
} ;
```




```
Y y;
y.a = 1; // OK
y.c = 2; // OK
y.b = 4; // Fehler, b kein Mitgl. !
y.d = 8; // Fehler, d privat

void Y::C() {
  d = 8; // OK
  b = 4; // Fehler, b kein Mitgl. !
  A();   // OK, manipuliert b
  B();   // Fehler, B() kein Mitgl!
}
```

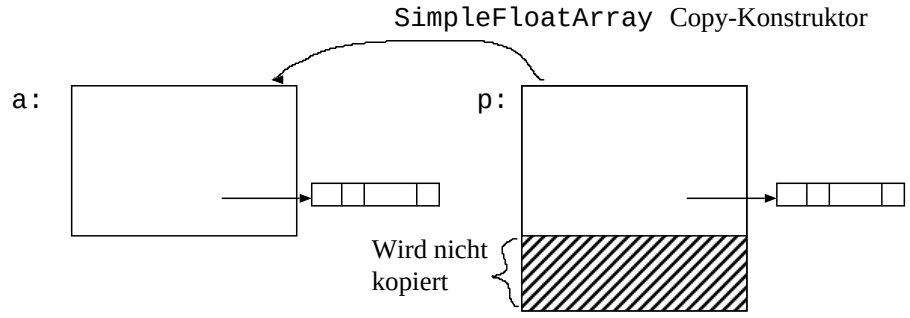
13.3 Ist-ein-Beziehung

Ein Objekt einer abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.

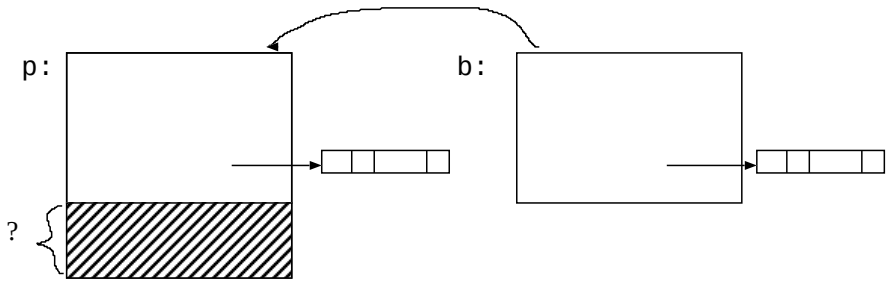
Daher darf ein Objekt der abgeleiteten Klasse für ein Objekt der Basisklasse eingesetzt werden. Allerdings sind dann nur Methoden der Basisklasse für das Objekt aufrufbar. Betrachte

```
void g (SimpleFloatArray a)
{
  a[3] = 1.0;
  ...
}
...
Polynomial p(10);
SimpleFloatArray b(100,0.0);

g(p);  // (1) OK, aber: slicing
p = b; // (2) Fehler
```

Im Fall (1) wird bei Aufruf von `g(p)` der Copy-Konstruktor des formalen Parameters `a`, also `SimpleFloatArray`, benutzt um das `Polynomial p` auf den formalen Parameter `a` vom Typ `SimpleFloatArray` zu kopieren. Dabei wird aber nur das `SimpleFloatArray`-Unterobjekt kopiert (*slicing*):



Im Fall (2) soll einem Objekt der abgeleiteten Klasse ein Objekt der Basisklasse zugewiesen werden. Dies ist nicht möglich, da nicht klar ist welchen Wert die zusätzlichen Mitglieder der abgeleiteten Klasse bekommen sollen:



Um diese Zuweisung zu ermöglichen müsste man einen Zuweisungsoperator

```
Polynomial& operator= (const SimpleFloatArray&);
```

in die Klasse `Polynomial` aufnehmen.

Natürlich kann man auch Referenzen (oder Zeiger) übergeben

```
void g (SimpleFloatArray& a)
{
  a[3] = 1.0;
  ...
}

...
Polynomial p(10);

g(p);  // (1) OK, kein slicing
```

Merke: Objekte einer öffentlich abgeleiteten Klasse können für ein Objekt der Basisklasse eingesetzt werden.

Will man Slicing vermeiden muss man Referenzen oder Zeigern verwenden.

13.4 Konstruktoren, Destruktor und Zuweisungsoperatoren

Wir kommen nun zur Implementierung der Methoden von `Polynomial`.

Konstruktoren, Destruktor und Zuweisungsoperatoren werden *nicht* vererbt. Man muss sie für jede abgeleitete Klasse neu schreiben oder es werden die auf Seite 142 beschriebenen Default-Versionen erzeugt.

Hier sind die Konstruktoren von Polynomial: `PolynomialKons.cc`

```
// Konstruktor, Nullpolynom, Grad n hat n+1 Koeffizienten
Polynomial::Polynomial (int n) : SimpleFloatArray(n+1,0.0)
{}

// Copy-Konstruktor
Polynomial::Polynomial (const Polynomial& rhs) : SimpleFloatArray(rhs)
{}
```

Beide Konstruktoren rufen nur den entsprechenden Konstruktor der Basisklasse auf, da `Polynomial` keine Datenmitglieder hat.

Die syntaktische Form entspricht der Initialisierung von Unterobjekten wie auf Seite 119 beschrieben.

Auch der Zuweisungsoperator kann aus der Basisklasse übernommen werden:

`PolynomialAssign.cc`

```
// Zuweisung
Polynomial& Polynomial::operator= (const Polynomial& rhs)
{
    // rufe Zuweisung in der Basisklasse
    SimpleFloatArray::operator=(rhs);
    return *this;
}
```

Der Zuweisungsoperator der Basisklasse wird explizit aufgerufen, wir nutzen, dass ein `Polynomial` für ein `SimpleFloatArray` eingesetzt werden kann.

Auf die Angabe eines Destruktors kann verzichtet werden, da der Default-Destruktor automatisch den Destruktor der Basisklasse aufruft.

13.5 Weitere Methoden von `Polynomial`

Die Auswertung des Polynoms wird mit dem *Hornerschema* vorgenommen:

$$p_0 + p_1 x + p_2 x^2 + \ldots + p_n x^n = p_0 + x(p_1 + x(p_2 + \ldots x(p_{n-1} + x p_n) \ldots).$$

Dies ist numerisch günstiger, da nur Zahlen vergleichbarer Größenordnung addiert werden (Auslöschung!).

`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;
}
```

Beachte auch wie der `operator[]` explizit aufgerufen wird, alternativ könnte man auch `(*this)[i]` schreiben.

Hier Methoden für den Grad, Addition, Multiplikation und Ausdrucken:

`PolynomialImp.cc`

```
// 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

    if (q.degree()>degree())
    {
        for (int i=0; i<=degree(); i=i+1)
            r[i] = (*this)[i]+q[i];
        for (int i=degree()+1; i<=q.degree(); i=i+1)
            r[i] = q[i];
    }
```

```
    else
    {
        for (int i=0; i<=q.degree(); i=i+1)
            r[i] = (*this)[i]+q[i];
        for (int i=q.degree()+1; i<=degree(); i=i+1)
            r[i] = (*this)[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 ()
{
    cout << (*this)[0];

    for (int i=1; i<=maxIndex(); i=i+1)
        cout << "+" << (*this)[i] << "*x^" << i;

    cout << endl;
}
```

(eigentlich könnten wir auch mal `cout` überladen ...)

Alle Methoden benutzen die öffentliche Schnittstelle von `SimpleFloatArray` um auf die Koeffizienten zuzugreifen.

Auf das eingebaute Feld in `SimpleFloatArray` kann nicht zugegriffen werden, da dies ein privates Mitglied ist! Damit ist `Polynomial` auch unabhängig von der Datenstruktur für die Koeffizienten.

13.6 Gleichheit

Betrachte

```
Polynomial p(10), q(20);
```

```
Polynomial* z1 = &p;
Polynomial* z2 = &p;
Polynomial* z3 = &q;

if (z1==z2) ... // ist wahr
if (z1==z3) ... // ist falsch
```

Die Gleichheit von Zeigern charakterisiert die Objektidentität. Zwei Zeiger sind gleich wenn sie auf *dasselbe* Objekt zeigen.

Der Gleichheitsoperator auf Zeigern ist immer definiert, egal von welchem Typ sie sind.

Dagegen meint der Ausdruck

`(p==q)`

ob die Polynome `p` und `q` gleich sind obwohl es unterschiedliche Objekte sind.

Eine Klasse hat *keinen* Default-Gleichheitsoperator (z. B. Gleichheit aller Datenmitglieder). Dieser muss extra definiert werden. Hier der von `Polynomial`:

`PolynomialEqual.cc`

```
// Gleichheit von Polynomen
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;
}
```

Benutzung von `Polynomial`

Folgendes Beispiel definiert das Polynom

$$p = 1 + x$$

und druckt p, p^2 und p^3.

UsePolynomial.cc

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

// 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 "PolynomialAssign.cc"
#include "PolynomialEqual.cc"
#include "PolynomialEval.cc"

int main ()
{
    Polynomial p(1),q(0);

    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 Vererbung der Implementierung

14.1 Motivation: Polynominterpolation

Mit einem Polynom $p(x)$ sollen $n+1$ Datenpunkte

$$(x_i, y_i) \quad i = 0, \ldots, n, \quad x_i \neq y_j \; \forall i \neq j$$

interpoliert werden, d. h. es soll gelten

$$p(x_i) = y_i \quad \forall i = 0, 1, \ldots, n. \tag{1}$$

Dazu benötigt man ein Polynom $p(x) = \sum_{j=0}^{n} p_j x^j$ vom Grad n (mit $n+1$ Koeffizienten).

Einsetzen des Polynoms in (1) liefert für jeden Datenpunkt i eine Bedingung der Form

$$p_0 + p_1 x_i + p_2 x_i^2 + \ldots + p_n x_i^n = y_i.$$

und damit das *lineare Gleichungssystem*

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

zur Bestimmung der Koeffizienten p_i.

Die zu entwickelnde Klasse `PolynomialFit` soll folgende Methoden besitzen:

- Konstruktor, der aus den Datenpunkten das Interpolationspolynom konstruiert.
- Methode zum Auswerten des Polynoms.

Offensichtlich ist `PolynomialFit` ein spezielles Polynom und wir können zu dessen Realisierung die bereits exisitierende Klasse `Polynomial` nutzen. Aber wie?

Bei öffentlicher Ableitung in der Form

```
class PolynomialFit : public Polynomial ...;
```

würde `PolynomialFit` *alle* öffentlichen Methoden von `Polynomial` erben. Das wollen wir aber gerade nicht, so soll `PolynomialFit` explizit *keine* Möglichkeit zur Manipulation der Koeffizienten besitzen.

Andere Möglichkeit: Wir geben `PolynomialFit` als privates Datenmitglied ein Objekt vom Typ `Polynomial`, oder ...

14.2 Private Vererbung

Wer definieren `PolynomialFit` mittels *privater* Vererbung:

`PolynomialFit.cc`

```
class PolynomialFit :
    private Polynomial {
public:
    PolynomialFit (SimpleFloatArray& x, SimpleFloatArray& y);
    // konstruiere Interpolationspolynom uas Datenpunkten

    float eval (float x);
    // Werte Interpolation aus
} ;
```

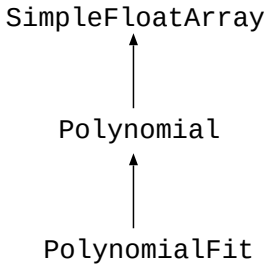
Private Vererbung bedeutet:

- 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!

`PolynomialFit` hat nicht die öffentliche Schnittstelle von `Polynomial`, sondern eine völlig neue.

In der Implementierung der Klassenmethoden können wir die Methoden von `Polynomial` nutzen. Das ist was wir wollen.

Das Klassendiagramm sieht nun folgendermaßen aus:



14.3 Konstruktion des Interpolationspolynoms

Sie brauchen das nicht zu verstehen. Ich habe den Konstruktor ausprogrammiert, damit sie eine funktionsfähige Klasse haben.

Das Gleichungssystem wird mittels Gauß-Elimination gelöst.

PolynomialFitKons.cc

```
float abs (float x)
{
    if (x<0.0) return -x;
    else return x;
}

PolynomialFit::PolynomialFit (SimpleFloatArray& x, SimpleFloatArray& y)
    : Polynomial(x.maxIndex())
{
    // Teste ob beide Felder bei 0 beginnen
    if (x.minIndex()!=0) {cerr << "falsche Eingabe" << endl; return;}
    if (y.minIndex()!=0) {cerr << "falsche Eingabe" << endl; return;}

    int n=x.maxIndex()+1; // Anzahl Koeffizienten

    // Teste ob beide Felder gleich gross sind
    if (y.maxIndex()+1!=n) {cerr << "falsche Eingabe" << endl; return;}

    // allokiere Speicher fuer Gleichungssystem
    // keine Fehlerabfrage hier !
    float* A = new float[n*n]; // Matrix
    float* p = new float[n];   // Koeffizientenvektor fuer Polynom
    float* b = new float[n];   // rechte Seite

    // Stelle lineares Gleichungssystem auf
    for (int i=0; i<n; i=i+1) // alle Zeilen
    {
        // Bedingung: p(x_i) = y_i
        float prod = 1.0; // p_0*1 + p_1*x_i + p_2*x_i^2 ...
        for (int j=0; j<n; j=j+1) // alle Spalten von Zeile i
        {
            A[i*n+j] = prod;
            prod = prod*x[i];
        }

        // rechte Seite
        b[i] = y[i];
    }

    // Loese lineares Gleichungssystem
    // Gauss Elimination mit maximalem Spaltenpivot
    for (int k=0; k<n-1; k=k+1)
```

```
{
    // finde Pivot
    float pivot = abs(A[k*n+k]);
    int pivoti = k;
    for (int i=k+1; i<n; i=i+1)
        if (abs(A[i*n+k])>pivot)
        {
            pivot = abs(A[i*n+k]);
            pivoti = i;
        }
    // Zeilenvertauschung
    if (pivoti!=k)
    {
        float t;

        for (int j=k; j<n; j=j+1)
        {
            t=A[k*n+j];
            A[k*n+j] = A[pivoti*n+j];
            A[pivoti*n+j] = t;
        }
        t = b[k];
        b[k] = b[pivoti];
        b[pivoti] = t;
    }

    // Eliminiere Rest der Spalte k
    pivot = A[k*n+k];
    for (int i=k+1; i<n; i=i+1)
    {
        for (int j=k+1; j<n; j=j+1)
            A[i*n+j] = A[i*n+j] - A[i*n+k]*A[k*n+j]/pivot;
        b[i] = b[i]-A[i*n+k]*b[k]/pivot;
    }
}

// Rueckwaerts Einsetzen
for (int i=n-1; i>=0; i=i-1)
{
    for (int j=i+1; j<n; j=j+1)
        b[i] = b[i]-A[i*n+j]*p[j];
    p[i] = b[i]/A[i*n+i];
}

// Fuelle Polynomkoeffizienten
for (int i=0; i<n; i=i+1)
    (*this)[i] = p[i];

// Deallokiere Speicher
delete[] A;
delete[] b;
delete[] p;
```

```
}
```

14.4 Beispiel für `PolynomialFit`

Die Auswertemethode nutzt die Basisklasse: `PolynomialFitEval.cc`

```
float PolynomialFit::eval (float x)
{
    return Polynomial::eval(x);
}
```

... und schließlich das Hauptprogramm: `UsePolynomialFit.cc`

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

// 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 "PolynomialAssign.cc"
#include "PolynomialEqual.cc"
#include "PolynomialEval.cc"

// PolynomialFit
#include "PolynomialFit.cc"
#include "PolynomialFitKons.cc"
#include "PolynomialFitEval.cc"

int main ()
{
    SimpleFloatArray x(6,0), y(6,0);

    x[0] = 0.10; y[0]=0.1;
    x[1] = 0.25; y[1]=0.8;
    x[2] = 0.40; y[2]=0.6;
    x[3] = 0.50; y[3]=0.0;
    x[4] = 0.80; y[4]=0.0;
    x[5] = 0.90; y[5]=0.2;

    PolynomialFit fit(x,y);

    for (float xc=0.0; xc<=1.000001; xc=xc+0.01)
```

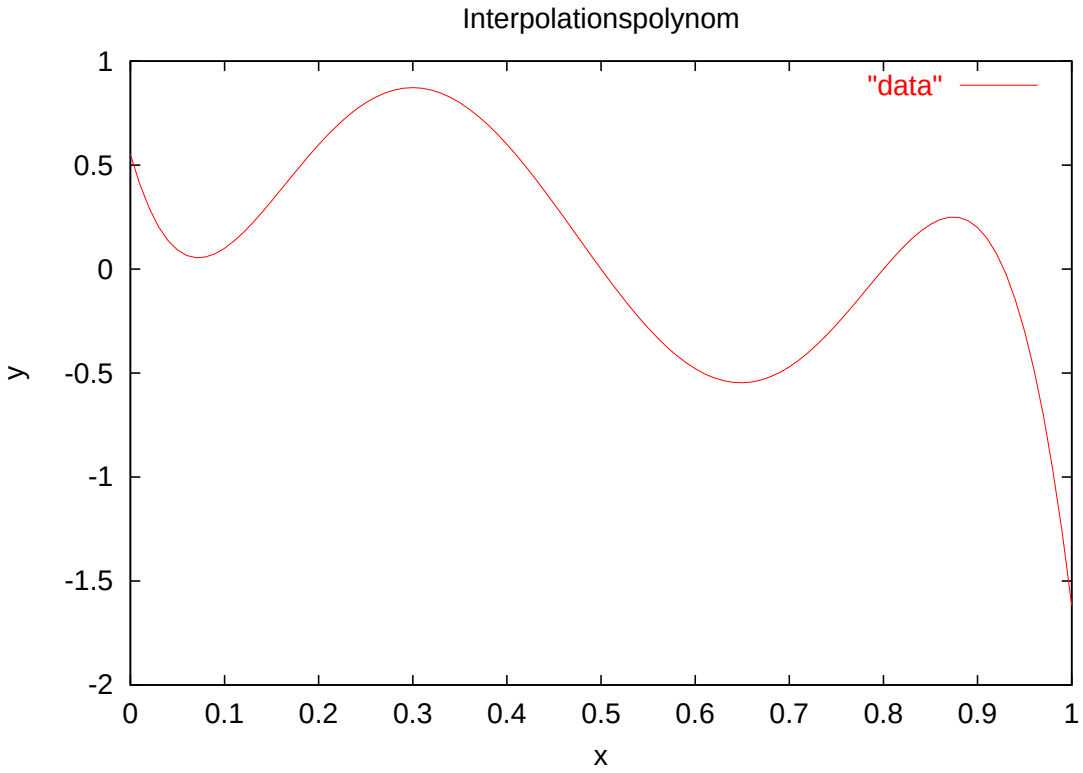
```
        cout << xc << " " << fit.eval(xc) << endl;
}
```

Wenn Sie die Ausgabe des Programmes in eine Datei umlenken

```
> g++ -Wall UsePolynomialFit.cc
> a.out > data
```

können Sie diese mit dem Programm *gnuplot* visualisieren:

```
> gnuplot
gnuplot> plot "data" with lines
```



Viel Spaß beim experimentieren!

15 Methodenauswahl und virtuelle Funktionen

15.1 Motivation: Feld mit Bereichsprüfung

Wir kommen zurück zur Klasse `SimpleFloatArray`.

Die dort implementierte Methode `operator[]` prüft nicht, ob der Index im erlaubten Bereich liegt.

Zumindest in der Entwicklungsphase eines Programmes wäre es nützlich ein Feld mit Indexüberprüfung zu haben.

Da wir keinen Code vervielfältigen wollen leiten wir das neue Feld von dem bereits vorhandenen `SimpleFloatArray` ab:

`CheckedSimpleFloatArray.cc`

```
class CheckedSimpleFloatArray :
    public SimpleFloatArray {
public:
    CheckedSimpleFloatArray (int s, float f);
    // Erzeuge ein neues Feld mit s Elementen, I=[0,s-1]

    CheckedSimpleFloatArray (const CheckedSimpleFloatArray&);
    // Copy-Konstruktor

    CheckedSimpleFloatArray& operator= (const CheckedSimpleFloatArray&);
    // Zuweisung von Feldern

    // Default-Destruktor ist OK

    float& operator[](int i);
    // Indizierter Zugriff auf Feldelemente
    // mit Überprüfung der Indizes
} ;
```

und schreiben einen neuen `operator[]` `CheckedSimpleFloatArrayIndex.cc`

```
float& CheckedSimpleFloatArray::operator[] (int i)
{
    if (i<minIndex()||i>maxIndex())
        cerr << "Index nicht erlaubt !" << endl;
    else
        return SimpleFloatArray::operator[](i);
}
```

Diese Implementierung *ersetzt* die von `SimpleFloatArray` vererbte Implementierung.

Die Implementierung der übrigen Methoden ruft wie bei `Polynomial` nur die entsprechende Methode der Basisklasse auf. Sehen Sie sich doch den Code für `CheckedSimpleFloatArrayImp.cc` selbst an.

Das `CheckedSimpleFloatArray` funktioniert wie erwartet:

```
CheckedSimpleFloatArray f(10,0.0);

f[1] = 2.5;  // Index OK
f[11] = 0.4; // Index ausserhalb -> Fehlermeldung
```

In einer praktischen Situation haben wir aber normalerweise schon Code der mit `SimpleFloatArray` arbeitet und wir möchten gerne zu Testzwecken alle Felder von diesem Typ durch welche der neuen Variante mit Bereichstest, also `CheckedSimpleFloatArray`, ersetzen.

Dies ist ohne Problem möglich: `UseCheckedSimpleFloatArray.cc`

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

// alles zum SimpleFloatArray
#include "SimpleFloatArray.cc"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// Array mit Index Check
#include "CheckedSimpleFloatArray.cc"
#include "CheckedSimpleFloatArrayImp.cc"
#include "CheckedSimpleFloatArrayIndex.cc"

void g (SimpleFloatArray& f) {
    float a = f[1];  // Hier wird SimpleFloatArray::operator[]
    float b = f[11]; // verwendet! Kein Bereichstest
}

int main () {
    CheckedSimpleFloatArray a(10,0);
    g(a);
}
```

Leider verhält sich dieses Programm nicht so wie erwartet. Der formale Parameter der Funktion `g()` hat den Typ einer Referenz auf `SimpleFloatArray`. In C++ bestimmt der Typ einer Referenz (oder Zeiger oder Variable) welche Methode verwendet wird. Der Übersetzer verwendet somit den `operator[]` der Klasse `SimpleFloatArray`. Der Grund ist, dass der Typ bereits zur Übersetzungszeit des Programmes bekannt ist und man daher effizienteren Code erzeugen kann. Zur Übersetzungszeit ist im allgemeinen nicht bekannt, dass die Funktion `g()` mit einer Referenz auf ein Objekt einer öffentlich abgeleiteten Klasse aufgerufen wird.

Die Auswahl der Methode wird also vom Typ der Referenz und nicht vom Typ des Objektes bestimmt!

15.2 Virtuelle Funktionen

Mit den nun einzuführenden *virtuellen Funktionen* erreicht man den Effekt, dass die Methode durch das tatsächlich zur Laufzeit übergebene Objekt bestimmt wird.

Dazu ist die Klasse `SimpleFloatArray` (also die Basisklasse!) zu modifizieren und dem `operator[]` das Schlüsselwort `virtual` voranzustellen:

`SimpleFloatArrayV.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
} ;
```

Die Einführung einer virtuellen Funktion erfordert hier also Änderungen in bereits existierendem Code!

Die Implementierung der Methoden bleibt unverändert.

Auch in `CheckedSimpleFloatArray` stellen wir dem `operator[]` ebenfalls ein `virtual`[1] voran:

`CheckedSimpleFloatArrayV.cc`

```
class CheckedSimpleFloatArray :
    public SimpleFloatArray {
public:
    CheckedSimpleFloatArray (int s, float f);
    // Erzeuge ein neues Feld mit s Elementen, I=[0,s-1]

    CheckedSimpleFloatArray (const CheckedSimpleFloatArray&);
    // Copy-Konstruktor

    CheckedSimpleFloatArray& operator= (const CheckedSimpleFloatArray&);
    // Zuweisung von Feldern

    // Default-Destruktor ist OK

    virtual float& operator[](int i);
    // Indizierter Zugriff auf Feldelemente
    // mit Überprüfung der Indizes
} ;
```

Nun funktioniert das Beispiel: `UseCheckedSimpleFloatArrayV.cc`

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

// alles zum SimpleFloatArray
#include "SimpleFloatArrayV.cc"         // virtual operator[] !
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// Array mit Index Check
#include "CheckedSimpleFloatArrayV.cc"  // virtual operator[] !
#include "CheckedSimpleFloatArrayImp.cc"
#include "CheckedSimpleFloatArrayIndex.cc"

void g (SimpleFloatArray& f) {
    float a = f[1];  // Hier wird der operator[] des
    float b = f[11]; // Objektes f verwendet !
}                    // also mit Bereichstest

int main () {
    CheckedSimpleFloatArray a(10,0);
    g(a);
}
```

[1]Dies ist nicht zwingend notwendig, siehe Regeln unten

Wird `g()` mit einem `CheckedSimpleFloatArray` aufgerufen wird der Bereichstest durchgeführt.

Wird `g()` mit einem `SimpleFloatArray` aufgerufen wird der Bereichstest nicht durchgeführt.

Die Auswahl der Methode hängt also vom eingesetzten Objekt ab.

Dies wird dadurch erreicht, dass jedes Objekt eine Tabelle für alle als virtuell definierten Funktionen mitführt.

Hier einige Regeln für die Verwendung virtueller Funktionen:

- 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 exakt mit der Definition in der Basisklasse übereinstimmen, sonst erfolgt überladen!
- Das Schlüsselwort `virtual` muss in der abgeleiteten Klasse nicht wiederholt werden, es ist aber guter Stil dies zu tun. Virtuelle Funktionen bleiben auch bei wiederholter Ableitung virtuell.
- Das Schlüsselwort `virtual` bezieht sich immer nur auf die Methode der es vorangestellt wird, alle anderen Funktionen sind nicht virtuell. Natürlich können beliebig viele Funktionen virtuell sein.
- Die Eigenschaften virtueller Funktionen lassen sich nur nutzen wenn auf das Objekt über Referenzen oder Zeiger zugegriffen wird. Betrachte:

  ```
  void g (SimpleFloatArray f) // call by value
  {
    float a = f[1];  // Verwendet wieder
    float b = f[11]; // SimpleFloatArray::operator[] !
  }
  ```

  Durch call-by-value wird mittels dem Copy-Konstruktor ein Objekt `f` vom Typ `SimpleFloatArray` erzeugt (slicing) und innerhalb `g()` entsprechend dessen `operator[]` verwendet!
- Virtuelle Funktionen ermöglichen Polymorphismus, was wir mal als „eine Schnittstelle — viele Methoden“ erklärt hatten.

Zusammenfassung

Wir haben in den letzten Kapiteln mehrere Methoden vorgestellt, wie man das Vervielfältigen von Programmteilen vermeiden, also gemeinsame Implementierung nutzen kann. Dies waren:

Öffentliche Ableitung

Die abgeleitete Klasse erbt alle öffentlichen Mitglieder der Basisklasse und es können weitere hinzugefügt werden.

Die abgeleitete Klasse besitzt im Vergleich mit der Basisklasse eine erweiterte Schnittstelle.

Private Ableitung

Die abgeleitete Klasse erbt alle öffentlichen Mitgleider der Basisklasse als private Mitglieder.

Die abgeleitete Klasse besitzt (im allgemeinen) eine andere Schnittstelle als die Basisklasse.

Wir nutzen dies um die Schnittstelle der abgeleiteten Klasse mit Hilfe der Schnittstelle der Basisklasse zu implementieren.

Virtuelle Funktionen

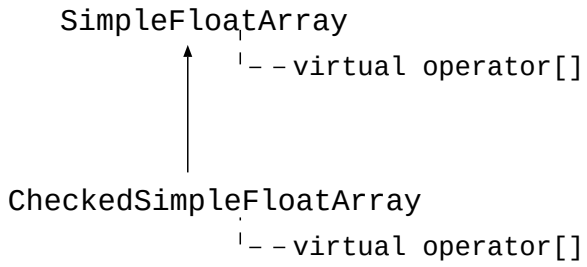
erlauben es zusammen mit öffentlicher Ableitung, dass in der abgeleiteten Klasse eine Methode der Basisklasse durch eine Variante ersetzt wird.

Die neue Methode wird auch benutzt, wenn sie durch die Schnittstelle der Basisklasse aufgerufen wird.

Allerdings müssen die Objekte über Zeiger oder Referenzen benutzt werden (da sonst kopiert wird).

16 Abstrakte Klassen

Im letzten Abschnitt hatten wir folgende Situation:



Beide Klassen besitzen dieselben Methoden und unterscheiden sich nur in der Implementierung von `operator[]`.

Wir könnten ebenso `Simple...` von `Checked...` ableiten. Das Klassendiagramm drückt diese Symmetrie aber nicht aus.

Dies liegt daran, dass `SimpleFloatArray` zwei verschiedene Dinge vereint:

- Die Definition der Schnittstelle eines ADT Feld.
- Die Implementierung der Methoden der Schnittstelle.

Wir lernen in diesem Abschnitt, wie man diese beiden Aspekte voneinander trennen kann.

Die Notwendigkeit dessen wird klar, wenn wir uns vorstellen, dass es zwei völlig verschiedene Implementierungen einer Schnittstelle geben kann.

16.1 Motivation: Integration

Angenommen wir benötigen eine Funktion, die ein bestimmtes eindimensionales Integral $\int_a^b f(x)\ dx$ einer Funktion f numerisch berechnet. Da wir verschiedene Integranden f integrieren wollen, soll die Integrationsfunktion den Integranden als Parameter erhalten.

Mit unseren bisherigen Techniken könnten wir folgende Lösung entwerfen:

`Integration.cc`

```
#include"iostream.h"
#include"math.h"

// Eine Basisklasse fuer Integranden
class Integrand {
public:
    virtual double func (double x);
};

double Integrand::func (double x)
```

```
{
    return 0; // eine dummy Funktion, eigentlich ist hier
}             // noch kein Integrand bekannt

// Die Integrationsfunktion
double integrate (double a, double b, Integrand& f)
{
    const int n=100;
    const double h = (b-a)/n;

    // Trapezregel
    double sum=0.0;
    for (int i=0; i<n; i=i+1)
        sum = sum + 0.5*h*(f.func(h*i)+f.func(h*(i+1)));
    return sum;
}

// hier ein paar spezielle Integranden
class Cos : public Integrand {
public:
    virtual double func (double x);
};

double Cos::func (double x)
{
    return cos(x);
}

class Exp : public Integrand {
public:
    virtual double func (double x);
};

double Exp::func (double x)
{
    return exp(x);
}

// und eine Anwendung
int main ()
{
    Cos f1;
    cout << integrate(0,3.1415,f1) << endl;
    Exp f2;
    cout << integrate(0,2.71,f2) << endl;
}
```

Die Basisklasse `Integrand` dient als Schnittstelle, die den Zugriff der Integrationsfunktion auf den Integranden regelt. Der eigentliche Integrand ist die virtuelle Methode `func`.

Um verschiedene Funktionen zu integrieren leiten wir von der Klasse `Integrand` ab und implementieren in der Methode `func` jeweils den gewünschten Integranden.

Auch in der Basisklasse `Integrand` müssen wir eine Implementierung der Methode `func` bereitstellen, obwohl dort noch kein Integrand bekannt ist. Wir sind aber nur an den Implementierungen der abgeleiteten Klassen interessiert. Die gemeinsame Basisklasse beschreibt nur, wie die Integrationsfunktion den Integranden benutzen kann.

16.2 Schnittstellenbasisklassen

Will man für eine virtuelle Methode in einer Basisklasse keine Implementierung angeben, so muss man diese mit dem Zusatz `= 0` am Ende kennzeichnen. Etwa:

```
class Integrand {
public:
    virtual double func (double x) = 0;
};
```

Solche Funktionen bezeichnet man als *rein virtuelle* (*engl.: pure virtual*) Funktionen.

Klassen, die mindestens eine rein virtuelle Funktion enthalten, nennt man *abstrakt*. Das Gegenteil ist eine *konkrete* Klasse.

Man kann keine Objekte von abstrakten Klassen instanzieren. Sehr wohl kann man aber Zeiger und Referenzen dieses Typs haben, die dann aber auf Objekte abgeleiteter Klassen zeigen.

Aus diesem Grund haben abstrakte Klassen auch keine Konstruktoren.

Abstrakte Klassen bei denen alle Funktionen rein virtuell sind bezeichnet man sie als *Schnittstellenbasisklasse*. Sie dienen dazu eine Schnittstelle zu vereinbaren, wie man eine bestimmte Funktionalität nutzen kann.

Die Implementierung der Schnittstelle erfolgt in abgeleiteten Klassen.

Schnittstellenbasisklassen enthalten üblicherweise auch keine Datenmitglieder.

16.3 Beispiel: Exotische Felder

Betrachten wir noch ein weiteres, etwas komplexeres Beispiel, das auf unseren Feldklassen aufbaut.

Ein dynamisches Feld

Der ADT Feld realisiert die Abbildung von einer Indexmenge $I \subset \mathbb{Z}$ in eine Wertemenge W.

In der Realisierung von `SimpleFloatArray` ist $I = [0, n-1]$ für ein fest gewähltes n. Bei einem einmal existierenden Objekt kann die Indexmenge nicht mehr geändert werden.

Wir wollen nun den Fall einer beliebigen, variablen Indexmenge betrachten.

Dabei treffen wir folgende Vereinfachung: Die zu erstellende Klasse soll der Darstellung von Polynomen dienen. In diesem Fall gilt:

$$i \in I \wedge p_i = 0 \;\equiv\; i \in I,$$

d. h. Abbildung eines Index auf 0 ist äquivalent dazu, dass der Index nicht in der Indexmenge ist.

Damit können wir uns in der Implementierung entscheiden, ob wir die Indexmenge exakt darstellen wollen oder ob überzählige Indizes einfach auf 0 abgebildet werden.

Die erste Implementierung der variablen Indexmenge soll ein eingebautes Feld nutzen:

- Der Konstruktor allokiert dynamisch ein eingebautes Feld mit einem Element, das den Wert 0 hat.
- `operator[]` prüft ob $i \in I$ gilt, wenn nein, so wird ein entsprechend großes Feld allokiert und die Werte aus dem alten Feld kopiert.
- Unsere Implementierung verwendet eine Indexmenge $I' = [o, o+n-1]$. Für die darzustellende Indexmenge I gilt $I \subseteq I'$. Für alle überzähligen Indizes $i \in I' \setminus I$ gilt der Wert 0.

Hier ist die entsprechende Klassendefinition `DynamicFloatArray.cc`

```
class DynamicFloatArray { // Feld mit Indexmenge [o,o+n-1]
public:
    // Konstruktoren
    DynamicFloatArray ();        // 1 Element 0.0

    // Destruktor
    ~DynamicFloatArray();

    // Indizierung
    float& operator[](int i);    // automatische Groessenanpassung
```

```
                                    // neue Elemente sind 0.0

    // allgemeine Indexmenge
    int numIndices ();
    int minIndex ();
    int maxIndex ();
    bool isMember (int i);

private:
    int n;      // Anzahl Elemente
    int o;      // Offset
    float *p;   // Zeiger auf built-in array

    // Copy-Konstruktor und Zuweisung privat: siehe Text
    DynamicFloatArray (const DynamicFloatArray&);
    DynamicFloatArray& operator= (const DynamicFloatArray&);
} ;

// Dummy Implementierung
DynamicFloatArray::DynamicFloatArray (const DynamicFloatArray& a) {}
DynamicFloatArray&
    DynamicFloatArray::operator= (const DynamicFloatArray& a) {}
```

Beachte: Die öffentliche Schnittstelle ist exakt die von `SimpleFloatArray` aber die Semantik der Methoden ist eine andere! Zumindest Kommentare in der Schnittstellendefinition sind also wichtig!

Der Einfachheit halber wollen wir Copy-Konstruktor und Zuweisungsoperatoren weglassen. Dies kann so wie bei `SimpleFloatArray` realisiert werden.

Damit niemand aus Versehen eine Kopie eines Objektes anlegen kann definieren wir diese Methoden als privat und geben eine Dummy-Implementierung.

Hier die Implementierung der Methoden: `DynamicFloatArrayImp.cc`

```
DynamicFloatArray::DynamicFloatArray ()
{
    o = 0;
    n = 1;
    try {
        p = new float[n];
    }
    catch (std::bad_alloc) {
        cerr << "nicht genug Speicher!" << endl;
        return;
    }
    for (int i=0; i<n; i=i+1) p[i]=0.0;
}

DynamicFloatArray::~DynamicFloatArray ()
{
```

```
    delete[] p;
}

float& DynamicFloatArray::operator[] (int i)
{
    if (i<o || i>=o+n) // resize
    {
        int new_o, new_n;
        float *q;

        if (i<o)
        {
            new_o = i;
            new_n = n+o-i;
        }
        else
        {
            new_o = o;
            new_n = i-o+1;
        }
        try {
            q = new float[new_n];
        }
        catch (std::bad_alloc) {
            cerr << "nicht genug Speicher!" << endl;
            return p[o]; // dummy
        }
        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];
}

int DynamicFloatArray::numIndices ()
{
    return n;
}

int DynamicFloatArray::minIndex ()
{
    return o;
}

int DynamicFloatArray::maxIndex ()
{
    return o+n-1;
}
```

```
bool DynamicFloatArray::isMember (int i)
{
    if (i>=o && i<o+n)
        return true;
    else
        return false;
}
```

Als Anwendung programmieren wir die Multiplikation von Polynomen:

UseDynamicFloatArray.cc

```
#include <iostream.h>
#include <new.h>      // fuer exception-handling

#include "DynamicFloatArray.cc"
#include "DynamicFloatArrayImp.cc"

void show (DynamicFloatArray& f)
{
    for (int i=f.minIndex(); i<=f.maxIndex(); i=i+1)
        if (f.isMember(i))
            cout << "+" << f[i] << "*x^" << i;
    cout << endl;
}

void polymul (DynamicFloatArray& a,
              DynamicFloatArray& b, DynamicFloatArray& 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 ()
{
    DynamicFloatArray f,g;

    f[0] = 1.0; f[1] = 1.0;
    polymul(g,f,f);
    polymul(f,g,g);
    polymul(g,f,f);
```

```
    polymul(f,g,g); // f = (1+x)^16

    show(f);
}
```

```
+1*x^0+16*x^1+120*x^2+560*x^3+1820*x^4+4368*x^5+8008*x^6+11440*x^7+12870*x^8
+11440*x^9+8008*x^10+4368*x^11+1820*x^12+560*x^13+120*x^14+16*x^15+1*x^16
```

Ein listenbasiertes Feld

Angenommen wir multiplizieren das Polynom

$$p(x) = x^{100} + 1$$

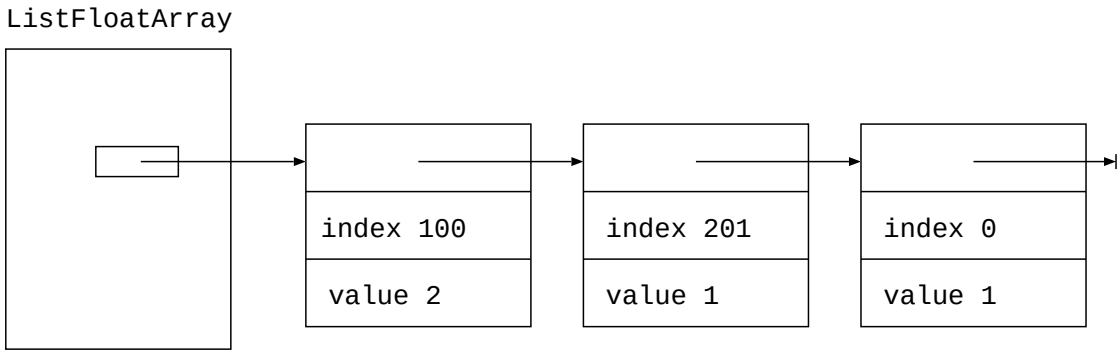
mit sich selbst, also

$$q(x) = p^2(x) = x^{200} + 2x^{100} + 1.$$

In diesem Fall ist `DynamicFloatArray` sehr ineffizient, da statt 3 eben 201 Koeffizienten gespeichert werden.

In diesem Fall wäre es effizienter die Koeffizienten nicht zu speichern, die sicher 0 sind.

Unsere zweite Implementierung speichert deshalb die Elemente des Feldes als eine *Liste* von Index–Wert–Paaren:



Die Liste ist einfach verkettet.

Die Liste ist unsortiert.

Hier die Klassendefinition: `ListFloatArray.cc`

```
class ListFloatArray { // Feld mit allgemeiner Indexmenge
public:
    // Konstruktoren
```

```
    ListFloatArray ();                      // leeres Feld

    // Destruktor
    ~ListFloatArray();

    // Indizierung
    float& operator[](int i);

    // allgemeine Indexmenge
    int numIndices ();
    int minIndex ();
    int maxIndex ();
    bool isMember (int i);

private:
    // lokal in der Klasse benutzte Datenstruktur
    struct FloatListElem {
        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); // Fuege in Liste ein
    FloatListElem* find (int i);  // finde Index
} ;
```

`ListFloatArray` und `DynamicFloatArray` haben die gleiche öffentliche Schnittstelle.

Für die Index–Wert–Paare wird innerhalb der Klassendefinition der zusammengesetzte Datentyp `FloatListElem` definiert (*nested struct*).

Dieser Datentyp ist ausserhalb der Klassendefinition als

`ListFloatArray::FloatListElem`

verwendbar.

Die privaten Methoden dienen der Manipulation der Liste und werden in der Implementierung der öffentlichen Methoden verwendet.

Hier die Implementierung der Methoden. Sehen Sie diese zuhause in Ruhe durch.

`ListFloatArrayImp.cc`

```
// 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)
{
    FloatListElem* r=find(i);

    if (r!=0)
        return true;
    else
        return false;
}
```

Trennung von Schnittstelle und Implementierung

Nun wollen wir erreichen, dass die Funktion `polymul` sowohl mit den dynamischen als auch mit den listenbasierten Feldern arbeiten kann. Wir wollen ausdrücken, dass `DynamicFloatArray` und `ListFloatArray` dieselbe öffentliche Schnittstelle besitzen.

Dazu führen wir eine neue Klasse `FloatArray` ein, die alle Methoden dieser Schnittstelle definiert:

`FloatArray.cc`

```
class FloatArray { // Interface fuer Feld
                   // als abstrakte Basisklasse
public:
    // KEINE Konstruktoren

    // virtual Destruktor
    virtual ~FloatArray();

    // Indizierung
    virtual float& operator[](int i) = 0;
```

```
    // allgemeine Indexmenge
    virtual int numIndices ()     = 0;
    virtual int minIndex ()       = 0;
    virtual int maxIndex ()       = 0;
    virtual bool isMember (int i) = 0;
} ;

FloatArray::~FloatArray () {}
```

Alle Methoden sind rein virtuell. `FloatArray` ist eine Schnittstellenbasisklasse.

Wir leiten `DynamicFloatArray` von `FloatArray` öffentlich ab:

`DynamicFloatArrayDerived.cc`

```
class DynamicFloatArray :
    public FloatArray {               // Ableitung
public:
    // Konstruktoren
    DynamicFloatArray ();             // 1 Element 0.0

    // Destruktoren
    virtual ~DynamicFloatArray();     // ersetzt ~FloatArray !!

    // Indizierung
    virtual float& operator[](int i); // automatische Groessenanpassung
                                      // neue Elemente sind 0.0

    // allgemeine Indexmenge
    virtual int numIndices ();
    virtual int minIndex ();
    virtual int maxIndex ();
    virtual bool isMember (int i);

private:
    int n;      // Anzahl Elemente
    int o;      // Offset
    float *p;   // Zeiger auf built-in array
} ;
```

Alle Methoden der öffentlichen Schnittstelle sind nun virtuell.

Die Implementierung der Methoden bleibt ungeändert.

Wir leiten `ListFloatArray` von `FloatArray` öffentlich ab:

`ListFloatArrayDerived.cc`

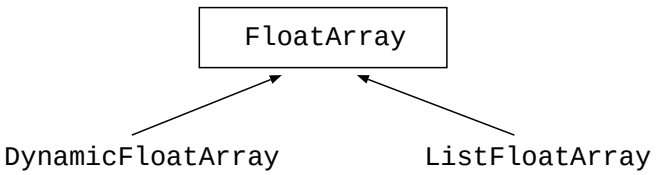
```
class ListFloatArray :
```

```
    public FloatArray {     // Ableitung
public:
    // Konstruktoren
    ListFloatArray ();      // leeres Feld

    // Destruktor
    ~ListFloatArray();      // ersetzt ~FloatArray !!

    // Indizierung
    virtual float& operator[](int i);

    // allgemeine Indexmenge
    virtual int numIndices ();
    virtual int minIndex ();
    virtual int maxIndex ();
    virtual bool isMember (int i);

    // lokal in der Klasse benutzte Datenstruktur
    struct FloatListElem {
        struct FloatListElem *next; // naechstes Element
        int index;                  // der Index
        float value;                // der Wert
    };

private:
    int n;              // Anzahl Elemente
    FloatListElem *p;   // einfach verkettete Liste

    FloatListElem* insert (int i, float v); // Fuege in Liste ein
    FloatListElem* find (int i);  // finde Index
} ;
```

Alle Methoden der öffentlichen Schnittstelle sind nun virtuell.

Die Implementierung der Methoden bleibt ungeändert.

Wir erhalten das folgende Klassendiagram:



Im Hauptprgramm ersetzen wir in `show` und `polymul` das `DynamicFloatArray` durch `FloatArray`.

`UseFloatArray.cc`

```
#include <iostream.h>
#include <new.h>      // fuer exception-handling
```

```
#include "FloatArray.cc"
#include "DynamicFloatArrayDerived.cc"
#include "ListFloatArrayDerived.cc"

// Benutze ungeaenderte Implementierung !
#include "DynamicFloatArrayImp.cc"
#include "ListFloatArrayImp.cc"

void show (FloatArray& f) {
    for (int i=f.minIndex(); i<=f.maxIndex(); i=i+1)
        if (f.isMember(i))
            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 ()
{
    ListFloatArray f,g; // geht mit Dynamic oder List

    f[0] = 1.0; f[1000] = 1.0;

    polymul(g,f,f); polymul(f,g,g);
    polymul(g,f,f); polymul(f,g,g); // f=(1+x^100)^16

    show(f);
}
```

und wir können nun beide Datentypen wahlweise (auch gemischt!) verwenden.

```
+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
```

16.4 Virtueller Destruktor

Eine Schnittstellenbasisklasse sollte einen virtuellen Destruktor

```
virtual ~FloatArray();
```

mit einer Dummy-Implementierung

```
FloatArray::~FloatArray () {}
```

besitzen.

Damit kann man dynamisch erzeugte Objekte abgeleiter Klassen durch die Schnittstelle der Basisklasse löschen:

```
void g (FloatArray* p)
{
  delete p;
}
```

Da der Destruktor virtuell ist wird der Destruktor der abgeleiteten Klasse benutzt.

Da man bei der Definition einer Schnittstellenbasisklasse meist noch nicht weiss, ob eine abgeleitete Klasse einen Destruktor brauchen wird, ist es eine gute Idee immer einen virtuellen Destruktor vorzusehen.

Der Destruktor darf nicht rein virtuell sein, da der Destruktor abgeleiteter Klassen einen Destruktor der Basisklasse aufrufen will.

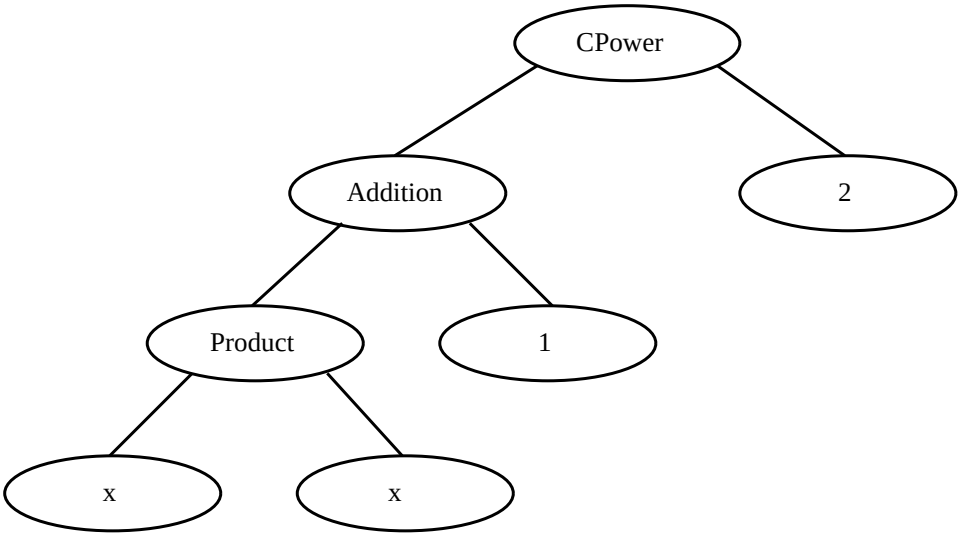
16.5 Beispiel: Symbolisches Differenzieren

Das Problem

Wir wollen in diesem Abschnitt ein Programm entwicklen, das Ausdrücke symbolisch differenzieren kann. Gefüttert etwa mit dem Ausdruck $(x^2 + 1)^2$ sollte das Programm mit $4(x^2 + 1)x$ antworten.

Das hier entwickelte Programm setzt alle bisher eingeführten Techniken im Konzert ein und illustriert insbesondere die Aspekte der Erweiterbarkeit und der Minimierung der Abhängigkeiten innerhalb eines Programmes.

Zunächst betrachten wir wie Ausdrücke im Rechner verarbeitet werden. Ausdrücke können, wie wir bereits wissen, als Bäume dargestellt werden. So hat der Ausdruck $(x^2 + 1)^2$ die Darstellung:



An den Blättern befinden sich entweder Konstanten oder Variablen. Wir erlauben Ausdrücke in mehreren Variablen. Jeder Operator entspricht einem inneren Knoten und verknüpft die Ausdrücke, die seine Kinder repräsentieren (hier werden nur binäre Operatoren verwendet). So steht „Addition“ und „Product“ für die Summe und das Produkt von zwei Ausdrücken. „CPower“ steht für Potenzen wobei der Exponent eine Konstante sein muss (offensichtlich kann x^2 auf verschiedene Arten dargestellt werden).

Bei der Differentiation von Ausdrücken mit den oben genannten Operatoren sind folgende Regeln zu berücksichtigen:

- $\frac{d}{dx}c = 0$
- $\frac{d}{dx}x = 1$
- $\frac{d}{dx}\left(f(x) + g(x)\right) = \frac{df}{dx}(x) + \frac{dg}{dx}(x)$
- $\frac{d}{dx}\left(f(x)g(x)\right) = \frac{df}{dx}(x)g(x) + f(x)\frac{dg}{dx}(x)$
- $\frac{d}{dx}\left(f(x)^p\right) = p\left(f(x)\right)^{(p-1)}\frac{df}{dx}(x)$

Man erkennt, dass das Differenzieren eines Ausdruckes rekursiv über die Baumdarstellung durchgeführt werden kann.

Idee: Entwerfe eine Schnittstellenbasisklasse für die Knoten in der Baumdarstellung eines Ausdruckes. Die speziellen Ausprägungen der Baumknoten (Konstante, Variable, Summe, . . .) sind dann in abgeleiteten konkreten Klassen realisiert. Hier die Basisklasse:

`Expr.h`

```
class Variable; // forward declaration

class Expr {                          // Schnittstellenbasisklasse
public:
    virtual ~Expr ();                 // virtueller Destruktor
    virtual double evaluate () = 0; // Auswerten. Variable sind belegt
    virtual Expr* differentiate       // Liefere Ableitung als einen
        (Variable* var) = 0;          // neuen Ausdrucke
    virtual void print () = 0;        // Drucke Ausdruck
    virtual Expr* clone () = 0;       // Erstelle eine Kopie des Ausdruckes
} ;

Expr::~Expr () {}                     // dummy Implementierung
```

Spezielles Augenmerk verdient die Speicherverwaltung für die einzelnen Objekte aus denen ein Ausdruck besteht. Da die Objekte dynamisch erzeugt werden und per `delete` wieder gelöscht werden sollen, ist sicherzustellen, dass jeweils nur ein Zeiger auf ein Objekt existiert. Falls notwendig, muss dies durch Anfertigung von Kopien mittels der Methode `clone` sichergestellt werden. Diese Methode arbeitet rekursiv und erstellt eine vollständige Kopie eines Ausdruckes.

Zur Illustration wie der oben benutzte Ausdruck erstellt und differenziert werden kann hier ein Hauptprogramm:

SymDiff.cc

```
#include"iostream.h"
#include"math.h"

#include"Expr.h"
#include"DoubleConstant.h"
#include"DoubleConstant.cc"
#include"Variable.h"
#include"Variable.cc"
#include"Addition.h"
#include"Addition.cc"
#include"Product.h"
#include"Product.cc"
#include"CPower.h"
#include"CPower.cc"

int main ()
{
    Variable* x = new Variable('x',2.71);      // die Variable x, Wert 2.71

    Expr* e = new CPower(                      // der Ausdruck (x*x+1)^2
        new Addition(
            new Product(x->clone(),x->clone()),// jeder Knoten ist ein neues
            new DoubleConstant(1.0)),          // Objekt
        2);                                    // Exponent ist ein double
    e->print(); cout << endl << endl;
```

```
    Expr* t = e->differentiate(x);             // Ableitung als neuer Ausdr.
    t->print(); cout << endl << endl;
}
```

Die Ausgabe ist

```
(((x*x)+1))^2

((2*(((x*x)+1))^1)*(x+x))
```

Man erkennt, dass das Ergebnis zwar korrekt aber nicht in einer möglichst einfachen Form ist. Dabei wurden während der Differentiation schon einige Vereinfachungen durchgeführt.

Hier die Definitionen der abgeleiteten Klassen für Konstante, Variable, Summe, Produkt und Potenz:

DoubleConstant.h

```
class DoubleConstant : public Expr { // die Konstante
public:
    DoubleConstant (double _value);  // Erzeugen mit Wert
    virtual ~DoubleConstant ();
    virtual double evaluate ();
    virtual Expr* differentiate (Variable* var);
    virtual void print ();
    virtual Expr* clone ();
    double get ();                   // Wert auslesen
    bool same_as (Expr *e);          // Teste ob Ausdruck eine Konstante
 private:                            // mit gleichem Wert ist.
    DoubleConstant ();               // zur internen Verwendung
    struct DC {                      // Eine per clone() erstellte
        int refcnt;                  // Kopie zeigt auf den selben Wert
        double value;                // DC Objekte werden mit reference
    };                               // counting verwaltet
    DC* p;                           // Zeiger auf Wert
} ;
```

Variable.h

```
class Variable : public Expr {       // die Variable
public:
    Variable (char _symbol, double _value);
    virtual ~Variable ();
    virtual double evaluate ();
    virtual Expr* differentiate (Variable* var);
    virtual void print ();
    virtual Expr* clone ();

    void set (double _value);        // Wert setzen fuer Auswertung
```

```
    double get ();
private:
    Variable ();                    // nur fuer interne Benutzung
    struct Var {                    // Alle Kopien einer Variablen
        int refcnt;                 // zeigen auf eine gemeinsame
        char symbol;                // Struktur, die Wert und Symbol
        double value;               // enthaelt. refcnt zaehlt Anzahl
    } ;                             // der erstellten Kopien
    Var* p;
} ;
```

Addition.h

```
class Addition : public Expr {      // die Addition
public:
    Addition (Expr* _left, Expr* _right); // verknuepfe Ausdruecke
    virtual ~Addition ();
    virtual double evaluate ();
    virtual Expr* differentiate (Variable* var);
    virtual void print ();
    virtual Expr* clone ();
private:
    Expr* left;
    Expr* right;
} ;
```

Product.h

```
class Product : public Expr {       // das Produkt
public:
    Product (Expr* _left, Expr* _right); // verknuepfe zwei Ausdruecke
    virtual ~Product ();
    virtual double evaluate ();
    virtual Expr* differentiate (Variable* var);
    virtual void print ();
    virtual Expr* clone ();
private:
    Expr* left;                     // hier die Referenzen auf die
    Expr* right;                    // beiden Ausdruecke
} ;
```

CPower.h

```
class CPower : public Expr {        // Potenz
public:
    CPower (Expr* _expr, double _power);
    virtual ~CPower ();
    virtual double evaluate ();
    virtual Expr* differentiate (Variable* var);
    virtual void print ();
    virtual Expr* clone ();
```

```
private:
    Expr* expr;
    double power;
} ;
```

Bei der Implementierung betrachten wir nur die Addition näher:

Addition.cc

```
Addition::Addition (Expr* _left, Expr* _right)
{
    left = _left;
    right = _right;
}

Addition::~Addition ()
{
    delete left;
    delete right;
}

double Addition::evaluate ()
{
    return left->evaluate() + right->evaluate();
}

Expr* Addition::differentiate (Variable* var)
{
    Expr* dleft = left->differentiate(var);
    Expr* dright = right->differentiate(var);

    if (ZERO.same_as(dleft)) {delete dleft; return dright;}
    if (ZERO.same_as(dright)){delete dright; return dleft;}
    return new Addition(dleft,dright);
}

void Addition::print ()
{
    cout << "(";
    left->print();
    cout << "+";
    right->print();
    cout << ")";
}

Expr* Addition::clone ()
{
    return new Addition(left->clone(),right->clone());
}
```

In der Methode `differentiate` werden die Ableitungen von linkem und rech-

tem Summanden berechnet. Zur Vereinfachung der Ausdrücke wird dann geprüft ob der linke Ausdruck Null ist (dann ist der rechte das Ergebnis) oder rechte Ausdruck Null ist (dann ist der linke das Ergebnis). Nur wenn die Ableitungen beider Summanden von Null verschieden sind dann wird eine Summe zurückgegeben.

Natürlich kann das Programm durch Iteration auch höhere Ableitungen berechnen. Hier die ersten vier Ableitungen des Ausdruckes $1/\sqrt{x^2+y^2}$ nach der Variablen x:

```
(((x*x)+(y*y)))^(-0.5)

(((-0.5)*(((x*x)+(y*y)))^(-1.5))*(x+x))

((((-0.5)*(((-1.5)*(((x*x)+(y*y)))^(-2.5))*(x+x)))*(x+x))
+(((-0.5)*(((x*x)+(y*y)))^(-1.5))*(1+1)))

(((((-0.5)*((((-1.5)*(((-2.5)*(((x*x)+(y*y)))^(-3.5))*(x+x)))*(x+x))
+(((-1.5)*(((x*x)+(y*y)))^(-2.5))*(1+1))))*(x+x))+(((-0.5)*
(((-1.5)*(((x*x)+(y*y)))^(-2.5))*(x+x)))*(1+1)))+(((-0.5)*
(((-1.5)*(((x*x)+(y*y)))^(-2.5))*(x+x)))*(1+1)))

((((((-0.5)*(((((-1.5)*((((-2.5)*(((-3.5)*(((x*x)+(y*y)))^(-4.5))*(x+x)))
*(x+x))+(((-2.5)*(((x*x)+(y*y)))^(-3.5))*(1+1))))*(x+x))+(((-1.5)*
(((-2.5)*(((x*x)+(y*y)))^(-3.5))*(x+x)))*(1+1)))+(((-1.5)*(((-2.5)*
(((x*x)+(y*y)))^(-3.5))*(x+x)))*(1+1))))*(x+x))+(((-0.5)*((((-1.5)*(((-2.5)
*(((x*x)+(y*y)))^(-3.5))*(x+x)))*(x+x))+(((-1.5)*(((x*x)+(y*y)))^(-2.5))
*(1+1))))*(1+1)))+(((-0.5)*((((-1.5)*(((-2.5)*(((x*x)+(y*y)))^(-3.5))
*(x+x)))*(x+x))+(((-1.5)*(((x*x)+(y*y)))^(-2.5))*(1+1))))*(1+1)))+(((-0.5)
*((((-1.5)*(((-2.5)*(((x*x)+(y*y)))^(-3.5))*(x+x)))*(x+x))+(((-1.5)
*(((x*x)+(y*y)))^(-2.5))*(1+1))))*(1+1)))
```

Die Vorteile dieses Programmes sind:

- Erweiterbarkeit: Neue Regeln wie z. B. Regeln für die Ableitung der Funktionen Funktionen log, e^x, sin, cos usw. können leicht dazugefügt werden.
- Lokalität: Diese Erweiterungen können hinzugefügt werden ohne die existierenden Programmteile modifizieren zu müssen.

Kritik:

- Die Speicherverwaltung wurde offengelegt. Der Benutzer (und der Implementierer) der Klassen muss explizit mittels der Methode `clone` dafür sorgen, dass jedes Objekt nur einmal referenziert wird. Diese liese sich durch ein etwas aufwendigeres Design noch vermeiden.

Zusammenfassung

In diesem Abschnitt haben wir gezeigt wie man mit Hilfe von Schnittstellenbasisklassen eine Trennung von

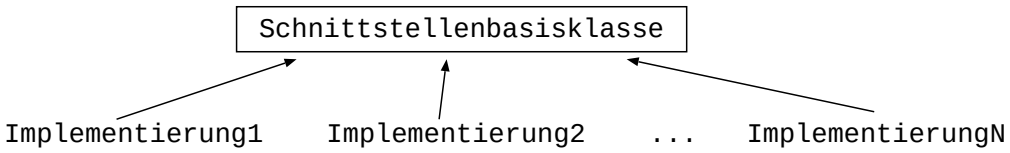
- Schnittstellendefinition und
- Implementierung

erreicht.

Dies gelingt durch

- rein virtuelle Funktionen in Verbindung mit
- Vererbung.

Typischerweise erhalt man Klassendiagramme der Form:



Man *erzeugt* Objekte konkreter (abgeleiteter) Klassen und *benutzt* diese Objekte durch die Schnittstellenbasisklasse:

Create objects, use interfaces!

17 Generische Programmierung

Klassenschablonen

Denken wir zurück an die Klasse `SimpleFloatArray` von Seite 136.

Was wäre, wenn wir eine Klasse `SimpleIntArray` haben wollten, die identisch zu `SimpleFloatArray` sein soll, nur eben mit `int` statt `float`?

Der Vererbungsmechanismus hilft an dieser Stelle nicht weiter, da wir den Grundtyp des Feldes in `SimpleFloatArray` dadurch nicht ändern können.

Eine Lösung: Kopiere `SimpleFloatArray` (inklusive aller Methoden) und ersetze überall `float` durch `int`. Sie ahnen schon, dass wir das nicht tun wollen ...

Außerdem wären sicher auch Felder über den Grundtypen `char`, `short`, `double`, ..., praktisch.

Die C++-Lösung für dieses Problem sind *parametrisierte Klassen*, die auch *Klassenschablonen* (*class templates*) genannt werden.

Man schreibt

```
SimpleArray<int> a(10,0);
```

Dabei ist `SimpleArray` eine (noch zu schreibende) Klassenschablone, die den Datentyp `int` als Parameter erhält.

In der Definition der Klassenschablone, die folgende Form hat:

```
template<class T> class SimpleArray {...};
```

ist der Datentyp `T` im Rumpf ein Platzhalter für einen konkreten Datentyp.

`SimpleArray<int>` ist ein neuer Datentyp, d. h. Sie können Objekte dieses Typs erzeugen, oder ihn als Parameter/Rückgabewert einer Funktion verwenden.

`SimpleArray` alleine ist kein Datentyp!

Der Mechanismus der Parametrisierung von Klassen arbeitet 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 *template-Instanzierung*.

Der Name Schablone (englisch: template) kommt daher, dass man sich die parametrisierte Klasse als Schablone vorstellt, die zur Anfertigung konkreter Varianten benutzt wird.

Daher ist die Schablone selbst auch kein Datentyp.

Datentypen sind nur konkrete Instanzen der Schablone, wie

`SimpleArray<int>, SimpleArray<Rational>, ...`

Hier die komplette Definition der Klassenschablone: `SimpleArray.cc`

```
template <class T>
class SimpleArray {
public:
    SimpleArray (int s, T f);
    // Erzeuge ein neues Feld mit s Elementen, I=[0,s-1]

    SimpleArray (const SimpleArray<T>&);
    // Copy-Konstruktor

    SimpleArray<T>& operator= (const SimpleArray<T>&);
    // Zuweisung von Feldern

    ~SimpleArray();
    // Destruktor: Gebe Speicher frei

    T& 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
    T *p;   // Zeiger auf built-in array
} ;
```

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; }
```

Die Implementierung aller Methoden finden Sie hier ...

```
// Destruktor
template <class T>
inline SimpleArray<T>::~SimpleArray () { delete[] p; }

// Konstruktor
template <class T>
inline SimpleArray<T>::SimpleArray (int s, T v)
{
    n = s;
    try {
        p = new T[n];
    }
    catch (std::bad_alloc) {
        cerr << "nicht genug Speicher!" << endl;
        return;
    }
    for (int i=0; i<n; i=i+1) p[i]=v;
}

// Zuweisungsoperator
template <class T>
inline SimpleArray<T>& SimpleArray<T>::operator= (const SimpleArray<T>& a)
{
    // Wird fuer ein bereits initialisiertes Feld aufgerufen.

    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;
            try {
                p = new T[n];
            }
            catch (std::bad_alloc) {
                cerr << "nicht genug Speicher!" << endl;
                return *this;
            }
        }
        for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }
    return *this; // Gebe Referenz zurueck damit a=b=c; klappt
}

template <class T>
inline SimpleArray<T>::SimpleArray (const SimpleArray<T>& a)
{
    // Erzeuge Feld mit selber Groesse wie a
    n = a.n;
```

```
    try {
        p = new T[n];
    }
    catch (std::bad_alloc) {
        cerr << "nicht genug Speicher!" << endl;
        return;
    }

    // und kopiere Elemente
    for (int i=0; i<n; i=i+1) p[i]=a.p[i];
}

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)
{
    if (i>=0 && i<n) return true;
    else return false;
}

template <class T>
ostream& operator<< (ostream& s, SimpleArray<T>& a)
{
    for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1)
        s << i << " " << a[i] << endl;
    return s;
}
```

Hier ein Beispiel für die Verwendung der Klassenschablone:

UseSimpleArray.cc

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

#include "SimpleArray.cc"
#include "SimpleArrayImp.cc"

int main ()
{
    SimpleArray<float> a(10,0.0); // erzeuge Felder
    SimpleArray<int> b(25,5.0);

    for (int i=a.minIndex(); i<=a.maxIndex(); i++)
        a[i] = i;
    for (int i=b.minIndex(); i<=b.maxIndex(); i++)
        b[i] = i;

    // hier wird der Destruktor gerufen
}
```

Hier generiert der Übersetzer zwei verschiedene Instanzen der Klassenschablone, nämlich

- `SimpleArray<float>`
- `SimpleArray<int>`

Als Schablonenparameter sind nicht nur Klassennamen sondern z. B. Konstanten von eingebauten Typen erlaubt und natürlich kann eine Klasse auch mehr als einen Schablonenparameter haben.

Dies verwenden wir nun um eine Variante des `SimpleArray` zu definieren in der das eingebaute Feld nicht dynamisch sondern zur Übersetzungszeit mit fester Größe angegeben wird.

```
template <class T, int m>
class SimpleArrayCS {
public:
    SimpleArrayCS (T f);
    // Erzeuge ein neues Feld mit m Elementen, I=[0,m-1]

    // Copy-Konstruktor, Zuweisung, Destruktor: Default ist OK!

    T& 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:
    T p[m]; // built-in array fester Groesse
} ;
```

Da nun keine Zeiger auf dynamisch allokierte Objekte verwendet werden sind für Copy-Konstruktor, Zuweisung und Destruktor die Defaultmethoden ausreichend.

Auch die Größe muss nicht mehr gespeichert werden, da sie zur Übersetzungszeit bekannt ist und in die Methoden eingebaut werden kann.

Aber: `SimpleArrayCS<int,5>` und `SimpleArrayCS<int,6>` sind unterschiedliche Klassen! D. h. Objekte beider Klassen können einander nicht zugewiesen werden.

Den Code für die Methoden finden Sie in `SimpleArrayCSImp.cc`.

Funktionenschablonen

Neben Klassenschablonen erlaubt C++ auch die Definition von Funktionenschablonen.

Die Definition einer Funktionsschablone sieht so aus:

```
template<class T> <Funktionsdefinition>
```

In der Funktionsdefinition verwendet man `T` wie einen vorhandenen Datentyp.

Hier ist ein Beispiel:

```
template<class T> void swap (T& a, T& b)
{
  T t;
  t = a;
  a = b;
  b = t;
}
```

Diese Funktion vertauscht den Inhalt der beiden Objekte, die per Referenz übergeben werden.

Bei der *Übersetzung* von

```
int a=10, b=20;
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).

Hier noch ein Beispiel:

```
template<class T> T max (T a, T b)
{
  if (a<b) return b; else return a;
}
```

Hier muss für den Typ `T` ein `operator<` definiert sein.

Dies wollen wir nun benutzen um eine generische Version einer Sortierfunktion zu schreiben.

Die Aufgabe: Ein Feld von Zahlen $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ ist zu sortieren. D. h. unsere Funktion soll ein Feld $a' = (a'_0, a'_1, a'_2, \ldots, a'_{n-1})$ liefern so dass

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$$

und das Feld a' ist eine Permutation von a.

Dabei wollen wir zum Sortieren das Feld a selbst benutzen, d. h. keinen zusätzlichen Speicher verbrauchen.

Eine Methode dies zu erreichen ist der Algorithmus *bubblesort*:

- 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.

Hier die nichtrekursive Implementierung dieser Idee: `bubblesort.cc`

```
template <class T> void swap (T& a, T&b) {
    T 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])       // operator< auf Feldelementen
                swap(a[j+1],a[j]); // generiert passende Version von swap!
}
```

Die Funktion nimmt an, dass auf den *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!

Hier ein Beispiel, das alle unsere Feldklassen benutzt: `Usebubblesort.cc`

```
#include<iostream.h>
#include<new.h> // fuer bad_alloc

// 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"

// generischer bubblesort
#include "bubblesort.cc"

// Zufallsgenerator
#include "Zufall.cc"

int main ()
{
    Zufall z(93576);
    SimpleArrayCS<float,2000> a(0.0);        // template feste Groesse
    SimpleArray<float> b(2000,0.0);          // template dynamisch allokiert
    SimpleFloatArray c(2000,0.0);            // virtuelle Fkt operator[]
    ListFloatArray d; d[0]=0.0; d[1999]=0.0; // Listenstruktur

    for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1) a[i] = z.ziehe_zahl();
    for (int i=b.minIndex(); i<=b.maxIndex(); i=i+1) b[i] = z.ziehe_zahl();
    for (int i=c.minIndex(); i<=c.maxIndex(); i=i+1) c[i] = z.ziehe_zahl();
    for (int i=d.minIndex(); i<=d.maxIndex(); i=i+1) d[i] = z.ziehe_zahl();

    bubblesort(a);
    bubblesort(b);
    bubblesort(c);
    bubblesort(d);
}
```

Hier werden vier Varianten von `bubblesort` und eine Variante von `swap` (für `float` Argumente) generiert!

Generische Programmierung und Effizienz

Sehen wir uns an welche Laufzeit (in Sekunden) die Funktion `bubblesort` für unsere verschiedenen Felder und Längen hat:

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	—	—	—	—

Bubblesort hat eine asymptotische Laufzeit von $O(n^2)$ falls der Zugriff auf ein Feldelement `a[i]` $O(1)$ ist:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} t_{cs} = t_{cs}\sum_{i=0}^{n-1} i = t_{cs}\frac{(n-1)n}{2}.$$

t_{cs} ist die Zeit für einen Vergleich und einen swap.

Dies zeigen die ersten fünf Zeilen deutlich: Verdopplung von n bedeutet vierfache Laufzeit.

Eine Variante mit eingebautem Feld (nicht vorgestellt, ohne Klassen) ist am schnellsten, gefolgt von den zwei Varianten mit Klassenschablonen, die unwesentlich langsamer sind.

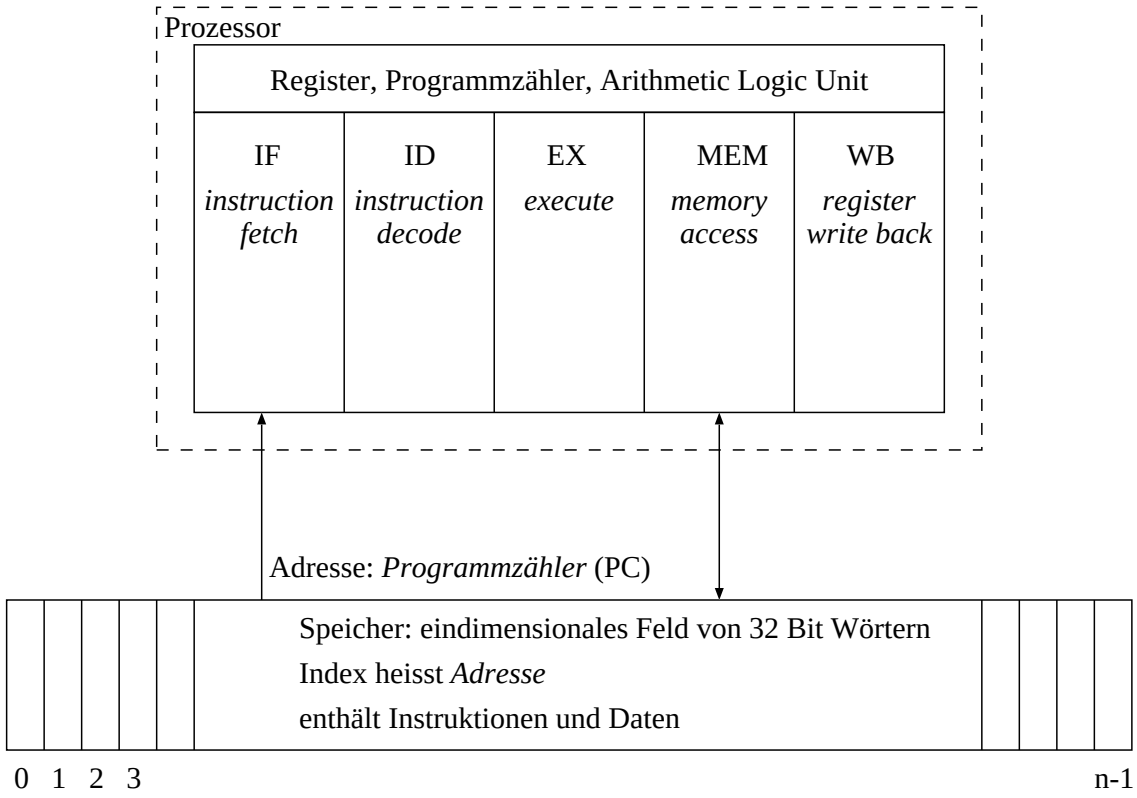
Die Zeilen fünf und vier zeigen die Laufzeit für die Variante mit einem virtuellem `operator[]` bzw. eine Version der Klassenschablone, bei der das Schlüsselwort `inline` vor der Methodendefinition des `operator[]` weggelassen wurde. Diese beiden Varianten sind etwa viermal langsamer als die vorherigen.

Schließlich ist `ListFloatArray` die listenbasierte Darstellung des Feldes mit Index-Wert-Paaren. Diese hat Komplexität $O(n^3)$ da nun jeder Zugriff auf ein Element des Feldes Komplexität $O(n)$ hat.

Um zu erklären warum die Varianten auf Schablonenbasis (mit inlining) schneller sind als die Variante mit virtueller Methode machen wir einen kleinen Ausflug in die Hardware des Rechners ...

Aufbau eines typischen RISC mit Pipelining

(*RISC = reduced instruction set computer*)



Speicher enthält Instruktionen (Befehle) und Daten.

Die Bearbeitung einer Instruktion im Prozessor gliedert sich in fünf Abschnitte:

IF Holen des nächsten Befehls aus dem Speicher. Ort: Programmzähler.

ID Dekodieren des Befehls, Auslesen der beteiligten Register.

EX Eigentliche Berechnung (z. B. Addieren zweier Zahlen).

MEM Speicherzugriff (entweder lesen oder schreiben).

WB Rückschreiben der Ergebnisse in Register.

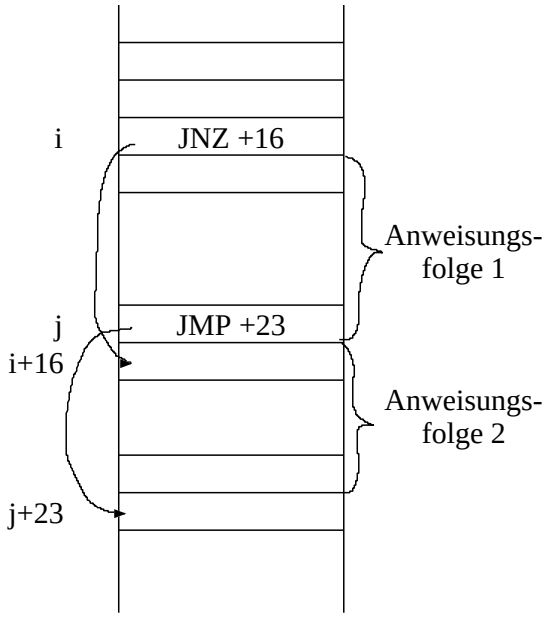
Pipelining: Überlappende Verarbeitung mehrerer Instruktionen

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

Probleme mit Pipelining

Sehen wir uns an wie eine `if`-Anweisung mit Hilfe sogenannter *Sprungbefehle* 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 Programmzähler addiert werden muss.

Welche Befehle sollen bis zu diesem Punkt weiter angefangen werden?

- 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.

Selbst das Ziel des unbedingten Sprungbefehls steht erst nach der Stufe EX zur Verfügung, da der offset auf den aktuellen Programmzähler addiert werden muss (oder man hat extra Hardware dafür).

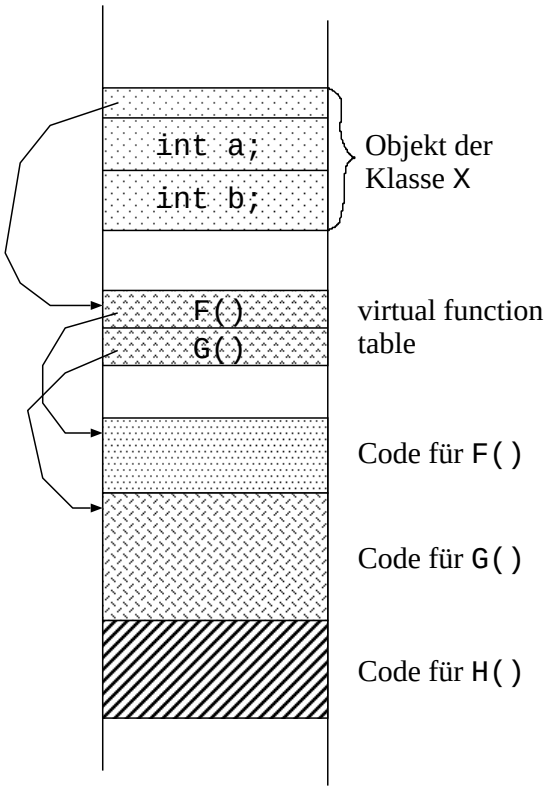
Ein Funktionsaufruf (Methodenaufruf) ist

- ein unbedingter Sprungbefehl,
- zusätzlich wird die Rücksprungadresse auf einem Stack gesichert
- und eventuell werden Register gesichert.

Ein Funktionsaufruf ist also in jedem Fall mit einigem Aufwand verbunden.

Realisierung virtueller Funktionen

```
class X {
public:
  int a;
  int b;
  virtual void F();
  virtual void G();
  void H();
} ;
```



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 den VFT der zugehörigen Klasse.

Beim Aufruf einer virtuellen Methode genereriert der Übersetzer Code der dem VFT des Objektes die Adresse der aufzurufenden Methode entnimmt und dann den Funktionsaufruf durchführt. Welcher Eintrag dem VFT zu entnehmen ist, ist zur Übersetzungszeit bekannt.

Der Aufruf *nichtvirtueller* Funktionen geschieht ohne VFT. Klassen (und ihre zugehörigen Objekte) ohne virtuelle Funktionen besitzen keinen (Zeiger auf den) VFT.

Für den Aufruf virtueller Funktionen ist immer ein Funktionsaufruf notwendig, da erst zur Laufzeit bekannt ist welche Methode auszuführen ist (dynamisches Binden).

Inlining

Der Funktionsaufruf stellt dann ein Problem dar wenn die Funktion sehr kurz ist, wie etwa in:

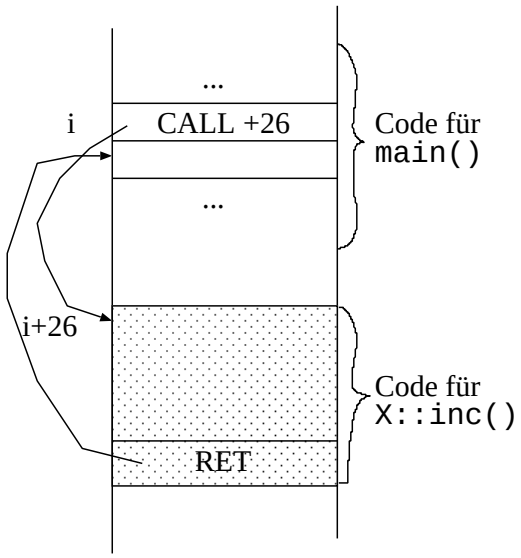
```
class X {
public:
  void inc();
private:
  int k;
} ;
```

```
inline void X::inc ()
{
  k = k+1;
}
```

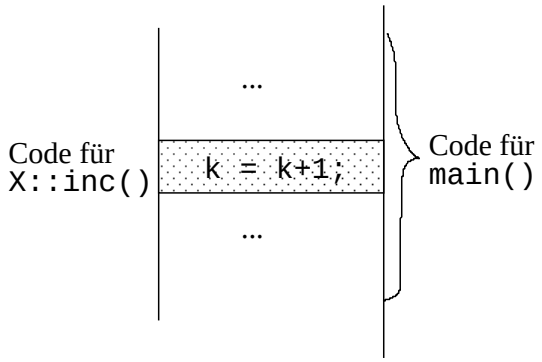
An einer Stelle im Programm schreiben wir

```
void main ()
{
  X x;

  x.inc();  // enthält nur einen Befehl!
}
```

Ohne dem Schlüsselwort `inline` in der Methodendefinition generiert der Übersetzer einen Funktionsaufruf für die Methode `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:



Inlining ändert nichts an der Semantik des Programmes.

Das Schlüsselwort `inline` ist nur ein *Vorschlag* an den Compiler. Z. B. werden rekursive Funktionen nicht inline ausgeführt.

Virtuelle Funktionen können nicht inline ausgeführt werden, da die auszuführende Methode zur Übersetzungszeit nicht bekannt ist.

Templates und function inlining erlauben Flexibilität und Effizienz!

Zusammenfassung

Klassenschablonen erlauben die Definition parametrisierter Datentypen und sind daher besonders geeignet um allgemein verwendbare Konzepte (ADT) zu implementieren. Dies nutzen wir im nächsten Kapitel aus.

Funktionenschablonen erlauben entsprechend die Definition parametrisierter Funktionen, die auf verschiedenen Datentypen mit gleicher Schnittstelle operieren können.

In beiden Fällen werden konkrete Varianten der Klassen/Funktionen zur Übersetzungszeit erzeugt und übersetzt. Deshalb spricht man von generischer Programmierung.

Nachteile der generischen Programmierung

Es wird viel Code erzeugt.

Es ist keine getrennte Übersetzung möglich. Der Übersetzer muss die Definition aller vorkommenden Schablonen kennen. Selbiges gilt für inline-Funktionen.

Das Finden von Fehlern in Klassen/Funktionenschablonen ist erschwert, da der Code für eine konkrete Variante nirgends existiert. Es ist eine gute Idee, eine kompliziertere Klassenschablone erst für einen konkreten Datentyp zu schreiben und hinterher eine Klassenschablone daraus zu machen.

18 Containerklassen

Klassen, die eine Menge anderer Objekte verwalten (man sagt *aggregieren*) nennt man *Containerklassen*.

Beispiele hatten wir schon: Liste, Stack, Feld. Andere werden wir noch kennenlernen: binärer Baum, queue, dequeue, map, priority queue ...

Diese Strukturen treten sehr häufig als Komponenten in größeren Programmen auf. So werden wir am Ende dieses Kapitels das Beispiel der Huffmanbäume ausschließlich mit Containerklassen realisieren.

Hinter effizienten Implementierungen von map oder priority queue stehen nichttriviale Algorithmen, die wir im zweiten Teil der Vorlesung kennenlernen werden.

Ziel von Containerklassen ist es diese Bausteine in *wiederverwendbarer* Form zur Verfügung zu stellen. Dies nennt man *code reuse*.

Vorteile sind:

- Weniger Zeitaufwand in Entwicklung und Fehlersuche.
- Klarere Programmstruktur, da man auf einer höheren Abstraktionsebene arbeitet.

Das Werkzeug zur Realisierung effizienter und flexibler Container in C++ sind Klassenschablonen.

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), die Container in professioneller Qualität enthält. Diese haben eine ganz ähnliche Schnittstelle wie unsere Spielzeugcontainer.

Ziel: Sie sind am Ende dieses Kapitels motiviert die STL zu verwenden und können die Konzepte verstehen.

Iteratoren

Container verwalten eine Menge von Objekten.

Eine Grundoperation aller Container ist das Durchlaufen aller Objekte in dem Container.

Dies soll mit einer containerunabhängigen Schnittstelle möglich sein, d. h. egal ob es sich um ein Feld, eine Liste oder einen Baum handelt.

Diese Abstraktion realisiert man mit *Iteratoren*. Iteratoren sind zeigerähnliche Objekte, die auf ein Element des Containers zeigen (obwohl der Iterator nicht als Zeiger realisiert sein muss).

Iteratoren erlauben ausserdem das Referenzieren von Elementen des Containers ohne die Implementierung der Elemente offenzulegen.

Hier das Prinzip:

```
template <class T> class Container {
public:
    class Iterator {  // nested class definition
       ...
    public:
        Iterator();                  // Konstruktor
        bool operator!= (Iterator x);// Zeiger auf selbes
        bool operator== (Iterator x);// Element ?
        Iterator operator++ ();      // prefix increment
        Iterator operator++ (int);   // postfix increment
        T& operator* () const;       // Derefernzierung
        T* operator-> () const;      // Selektor
        friend class Container<T>;   // Erlaube Zugriff
    } ;
    Iterator begin () const;
    Iterator end () const;
    .... // Spezialitäten des Containers
} ;
```

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.

Von aussen muss man die Iteratorklasse ansprechen mittels

```
Container<int>::Iterator i;
```

Das Durchlaufen aller Elemente des Containers gelingt dann mittels

```
Container<int> c;

for (Container<int>::Iterator i=c.begin(); i!=c.end(); i++)
  cout << *i << endl;
```

- 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 schon 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.
- `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. Da das Ergebnis eine Referenz ist kann man sowohl

  ```
  x = *i;
  ```

  als auch

  ```
  *i = x;
  ```

  schreiben. Dann wird das Element im Container überschrieben.
- Ist das Objekt im Container von einem zusammengesetzten Datentyp (also `struct` oder `class`) so kann mittels

  ```
  x = i-><Komponente>; // meint: x = (i.operator->())-><Komponente>;
  i-><Komponente> = x; // meint: (i.operator->())-><Komponente> = x;
  ```

  eine Komponente selektiert werden.

Damit verhalten sich Iteratoren wie Zeiger auf Elemente des Containers (*smart pointers*).

Feld

Das Feld erlaubt Zugriff auf seine Elemente mittels Indizierung.

Der generische Container `Array` entspricht unserem alten `SimpleArray<T>` mit der zusätzlichen Iteratorschnittstelle :

`Array.cc`

```
template <class T> class Array {
public:
    class Iterator {  // Iteratorklasse zum
    private:          // Durchlaufen der Elemente des Containers
        T* p;         // Iterator ist Zeiger auf Feldelement
    public:           // Iterator ohne Bereichstest !
        Iterator();
        bool operator!= (Iterator x);
        bool operator== (Iterator x);
        Iterator operator++ ();   // prefix Stroustrup p. 292
        Iterator operator++ (int);// postfix
        T& operator* () const;
        T* operator-> () const; // Stroustrup p. 289
        friend class Array<T>;
    } ;
    Iterator begin () const;
    Iterator end () const;

    Array(int m);
    Array (const Array<T>&);
    ~Array();
    Array<T>& operator= (const Array<T>&);
    int size () const;
    T& operator[](int i);
    typedef T MemberType;  // Merke den Grundtyp ...
private:
    int n;  // Anzahl Elemente
    T *p;   // Zeiger auf built-in array
} ;
```

Der Iterator ist als Zeiger auf ein Feldelement realisiert.

Die Schleife

```
Array<int> a(100,0);
for (Array<int>::Iterator i=a.begin(); i!=a.end(); i++) ...
```

entspricht nach inlining der Methoden einfach

```
Array<int> a(100,0);
for (int* p=a.p; p!=&a[100]; p=p+1) ...
```

und ist somit nicht langsamer als handprogrammiert!

Hier die Realisierung der Iteratormethoden: `ArrayIterator.cc`

```
template<class T>
inline Array<T>::Iterator::Iterator ()
{
    p=0; // nicht initialisierter Iterator
}

template<class T>
inline bool Array<T>::Iterator::operator!=
         (Array<T>::Iterator x)
{
    return p != x.p;
}

template<class T>
inline bool Array<T>::Iterator::operator==
        (Array::Iterator x)
{
    return p == x.p;
}

template<class T>
inline Array<T>::Iterator Array<T>::Iterator::operator++ () // prefix
{
    p++;  // C Zeigerarithmetik: p zeigt auf naechstes Feldelement
    return *this;
}

template<class T>
inline Array<T>::Iterator Array<T>::Iterator::operator++ (int) // postfix
{
    Iterator tmp = *this;
    ++*this;
    return tmp;
}

template<class T>
inline T& Array<T>::Iterator::operator* () const
{
    return *p;
}

template<class T>
inline T* Array<T>::Iterator::operator-> () const
{
    return p;
}
```

Die weiteren Methoden von `Array<T>` finden Sie in `ArrayImp.cc`.

Doppelt verkettete Liste

Als zweites Beispiel für einen Container betrachten wir eine doppelt verkettete Liste mit folgender Schnittstelle:

- Iteratorklasse zum Vorwärts- *und* Rückwärtslaufen in der Liste. Zugriff auf Listenelemente erfolgt ausschließlich über Iteratoren.
- Methoden `begin()` und `rbegin()` die auf das erste bzw. letzte Element des Containers zeigen. Die Methoden `end()` bzw. `rend()` zeigen auf kein gültiges Listenelement.
  Rückwärts durchläuft man die Liste mittels

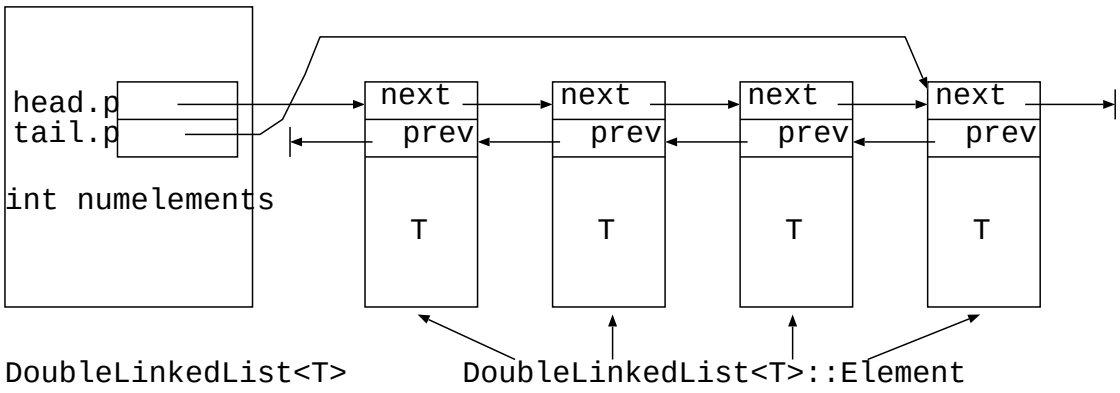
  ```
  DoubleLinkedList l;
  for (DoubleLinkedList<int>::Iterator i=l.rbegin(); i!=l.rend(); i--)
    ...
  ```

- Operationen zum Einfügen vor oder nach einem Element. Die Position wird durch einen Iterator angegeben.
- Entfernen eines Elementes. Das zu entfernende Element wird mit einem Iterator angegeben
- Methode zum Abfragen der Größe der Liste.

Hier die Klassendefinition: DoubleLinkedList.cc

```
template <class T> class DoubleLinkedList {
public:
    struct Element;   // Vorwaertsdeklaration fuer das Listenelement
    class Iterator {  // Iteratorklasse zum
    private:          // Durchlaufen der Elemente des Containers
        Element* p;   // Iterator ist ein Zeiger auf ein Listenelement
    public:
        Iterator();
        bool operator!= (Iterator x);
        bool operator== (Iterator x);
        Iterator operator++ ();   // prefix Stroustrup p. 292
        Iterator operator++ (int);// postfix
        Iterator operator-- ();   // prefix
        Iterator operator-- (int);// postfix
        T& operator* () const;
        T* operator-> () const; // Stroustrup p. 289
        friend class DoubleLinkedList<T>;
    } ;
    Iterator begin () const;
    Iterator end () const;
    Iterator rbegin () const;
    Iterator rend () const;

    DoubleLinkedList();
    DoubleLinkedList (const DoubleLinkedList<T>&);
    ~DoubleLinkedList();
    DoubleLinkedList<T>& operator= (const DoubleLinkedList<T>&);
    int size () const;
    Iterator insert_after (Iterator i, T t);
    Iterator insert_before (Iterator i, T t);
    void erase (Iterator i);

private:
    struct Element {      // Typ fuer das Listenelement
        Element* next;    // Nachfolger
        Element* prev;    // Vorgaenger
        T item;           // Datum
        Element ();       // setze next=prev=0
        Element (T &t);   // setze next=prev=0
    };

    Iterator head;        // erstes Element der Liste
    Iterator tail;        // letztes Element der Liste
    int numelements;      // Anzahl Elemente in der Liste
} ;
```

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 aus allen Datentypen einfach eine Liste machen ohne noch mit einem zusätzlichen Datentyp umgehen zu müssen (*non-intrusive container*).

Will man das Kopieren vermeiden so kann man eine Liste von Zeigern auf Objekte machen, z. B. `DoubleLinkedList<int *>`.

Manchmal möchte man allerdings direkte Kontrolle über die Speicherverwaltung haben, dies kann man etwa durch überladen der Methoden `new` und `delete` für den Grundtyp `T` erreichen. (Die STL bietet noch weitere Möglichkeiten).

Der Iterator ist ein Zeiger auf ein Listenelement. Der Iterator kapselt den Zugriff auf Objekte eines ausserhalb der Liste nicht bekannten Datentyps.

Die Implementierung der Methoden gehen wir nicht im einzelnen durch, Sie können sie hier einsehen:

- `DoubleLinkedListIterator.cc`
- `DoubleLinkedListElement.cc`
- `DoubleLinkedListImp.cc`

Hier ein Beispiel: UseDoubleLinkedList.cc

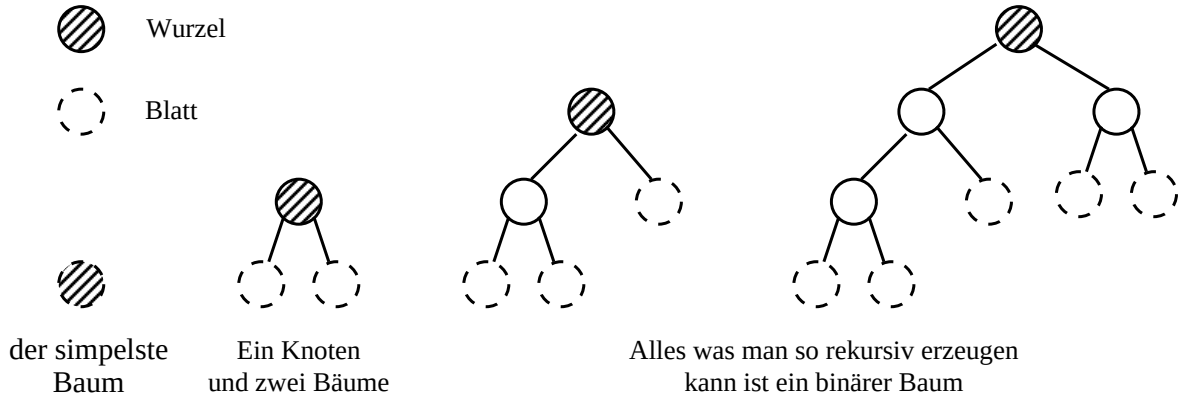
```
#include<iostream.h>

#include"DoubleLinkedList.cc"
#include"DoubleLinkedListIterator.cc"
#include"DoubleLinkedListElement.cc"
#include"DoubleLinkedListImp.cc"

#include"Zufall.cc"

int main ()
{
    Zufall z(87124);
    DoubleLinkedList<int> l1,l2,l3;

    // Erzeuge 3 Listen mit je 5 Zufallszahlen
    for (int i=0; i<5; i=i+1)
        l1.insert_after(l1.rbegin(), z.ziehe_zahl());
    for (int i=0; i<5; i=i+1)
        l2.insert_after(l2.rbegin(), z.ziehe_zahl());
    for (int i=0; i<5; i=i+1)
        l3.insert_after(l3.rbegin(), z.ziehe_zahl());

    // Loesche alle geraden in der ersten Liste
    DoubleLinkedList<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 ...
    DoubleLinkedList<DoubleLinkedList<int> > ll;
    ll.insert_after(ll.rbegin(),l1);
    ll.insert_after(ll.rbegin(),l2);
    ll.insert_after(ll.rbegin(),l3);
    cout << ll;
}
```

mit der Ausgabe:

```
dlist 3 elements = (
    dlist 2 elements = (
    768994395
    932209119
)

    dlist 5 elements = (
    1288811556
    1535758050
    879593057
    43083051
    394849118
)

    dlist 5 elements = (
    504656996
    1357209769
    53289149
    130046444
    1699715309
)

)
```

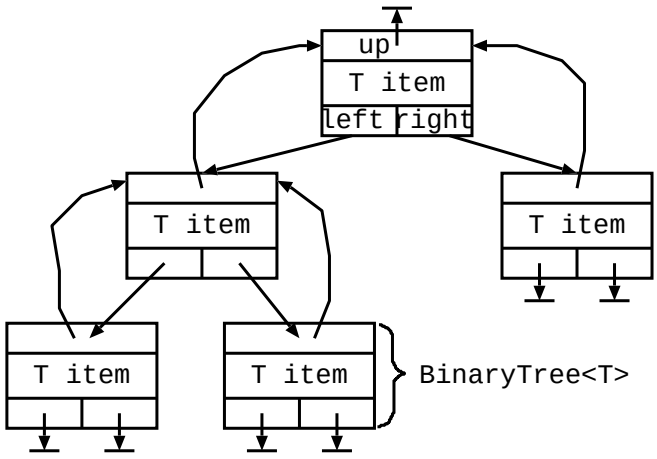
Binärer Baum

Wir hatten Bäume als rekursive Datenstruktur eingeführt:



Wir stellen nun einen rudimentären Container für binäre Bäume vor. Er hat folgende Eigenschaften:

- Jeder Knoten des Baumes enthält ein Objekt des Grundtyps `T`.
- Der Aufbau eines Baumes geschieht rekursiv von den Blättern zur Wurzel, wie in der Definition des Baumes (*bottom up*). Dazu werden zwei verschiedene Konstruktoren zur Verfügung gestellt: Einer für Blätter, einer für innere Knoten.
- Der Container übernimmt *nicht* die Speicherverwaltung für die einzelnen Knoten. Diese werden vom Benutzer der Klasse entweder dynamisch oder als lokale/globale Variablen erzeugt.
- Der Container enthält keine Möglichkeit zum entfernen von Knoten.
- Es gibt eine Iteratorklasse, die das Durchlaufen des ganzen Baumes ab der Wurzel, sowie die übliche Navigation links, rechts, hoch von jedem Baumknoten aus erlaubt.
- Hier die interne Datenstruktur des Containers:



Hier die Klassendefinition: BinaryTree.cc

```
template <class T>
class BinaryTree {
public:
    class Iterator {  // Iteratorklasse zum
    private:          // Durchlaufen der Elemente des Containers
        BinaryTree<T>* current;  // Iterator ist Zeiger auf Baumknoten
        BinaryTree<T>* last;     // woher komme ich gelaufen
    public:           // Iterator ohne Bereichstest !
        Iterator();
        bool operator!= (Iterator x);
        bool operator== (Iterator x);
        Iterator operator++ ();   // prefix
        Iterator operator++ (int);// postfix
        T& operator* () const;
        T* operator-> () const; // Stroustrup p. 289
        Iterator left () const;
        Iterator right () const;
        Iterator up () const;
        bool isleaf () const;
        friend class BinaryTree<T>;
    } ;
    friend class Iterator;

    Iterator begin (); // Wurzel des Baumes
    Iterator end ();   // Null

    BinaryTree (T t); // Ein Blatt
    BinaryTree (Iterator left, T t, Iterator right); // innerer Knoten

    void print ();
    void print (int n); // mit einruecken

private:
    T item;
    BinaryTree<T>* l;
    BinaryTree<T>* r;
    BinaryTree<T>* u;
} ;
```

Der Iterator bietet zusätzlich die Methoden `left()`, `right()` und `up()` um von einem Knoten aus zu den drei möglichen Nachbarknoten zu gelangen.

Die Methode `isleaf()` liefert `true` wenn der Iterator auf ein Blatt zeigt.

Die Methode `begin()` liefert einen Iterator der auf den Knoten selbst zeigt.

`print()` ist ein „pretty print“ mit Einrücken.

Hier ein Beispiel für die Benutzung: `UseBinaryTree.cc`

```
#include<iostream.h>

#include"BinaryTree.cc"
#include"BinaryTreeIterator.cc"
#include"BinaryTreeImp.cc"

int main ()
{
    BinaryTree<int> b1(1);
    BinaryTree<int> b2(2);
    BinaryTree<int> b3(b1.begin(),3,b2.begin());
    BinaryTree<int> b4(4);
    BinaryTree<int> b5(5);
    BinaryTree<int> b6(b4.begin(),6,b5.begin());
    BinaryTree<int> b7(b3.begin(),7,b6.begin());

    b7.print();

    for (BinaryTree<int>::Iterator i=b7.begin(); i!=b7.end(); i++)
    {
        cout << *i << endl;
        *i = 17;
    }
}
```

mit der Ausgabe:

```
    1
  3
    2
7
    4
  6
    5
7
3
1
2
6
4
5
```

Implementierung der Methoden in

- `BinaryTreeIterator.cc`
- `BinaryTreeImp.cc`

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 voll oder leer.

Ein Element kann in der Menge nur einmal enthalten sein.

Hier die Klassendefinition: `Set.cc`

```
template<class T>
class Set : private DoubleLinkedList<T> {
public :
    Set ();
    Set (const Set<T>&);
    Set<T>& operator= (const Set<T>&);

    bool isempty ();
    bool isfull ();
    bool ismember (T t);
    void insert (T t);
    void remove (T t);
} ;
```

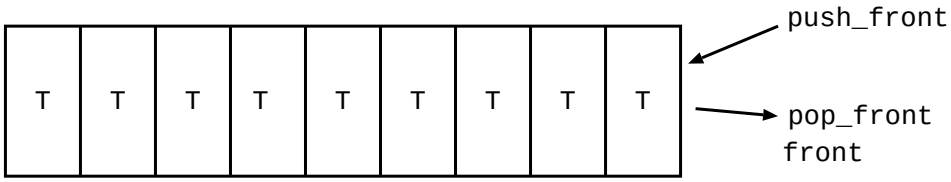
Die Implementierung hier basiert auf der doppelt verketteten Liste von oben (private Ableitung!).

Daher hat Einfügen, Suchen und Entfernen die Komplexität $O(n)$.

Wir lernen später Implementierungen kennen, die den Aufwand $O(\log n)$ für alle Operationen haben.

Stack

Ein Stack ist eine Struktur, die Einfügen und Entfernen nur an einem Ende erlaubt:



Damit wird immer das zuletzt eingefügte Element als erstes wieder gelöscht.

Unser Stack bietet die folgenden Funktionen:

- Konstruktion eines leeren Stack.
- Einfügen eines Elementes vom Typ `T` zuoberst.
- Entfernen des obersten Elementes.
- Inspektion des obersten Elementes.
- Test ob Stack voll oder leer.

Hier die Klassendefinition: `Stack.cc`

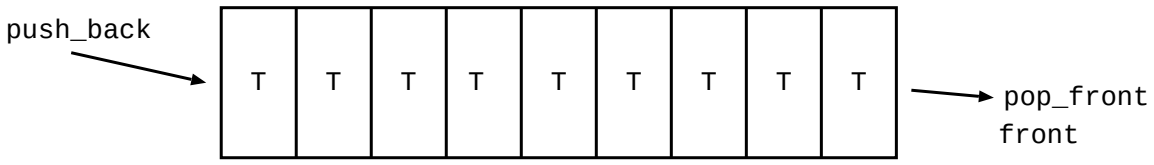
```
template<class T>
class Stack : private DoubleLinkedList<T> {
public :
    Stack ();
    Stack (const Stack<T>&);
    Stack<T>& operator= (const Stack<T>&);

    bool isempty ();       // Stack leer ?
    bool isfull ();        // Stack voll (nur bei vorgegebener Groesse)
    void push_front (T t); // Einfuegen eines Elementes
    T pop_front ();        // Entfernen eines Elementes
    T front ();            // Inspizieren des obersten Elementes
} ;
```

Der Stack wird hier mittels der doppelt verketteten Liste und privater Ableitung implementiert.

Die Implementierung ist in `StackImp.cc`

Queue

Eine Queue ist eine Struktur, die Einfügen an einem Ende und Entfernen nur am anderen Ende erlaubt:



Anwendung: Warteschlange.

Es wird immer das älteste Element als nächstes entfernt.

Unsere Queue bietet die folgenden Funktionen:

- 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 voll oder leer.

Hier die Klassendefinition: `Queue.cc`

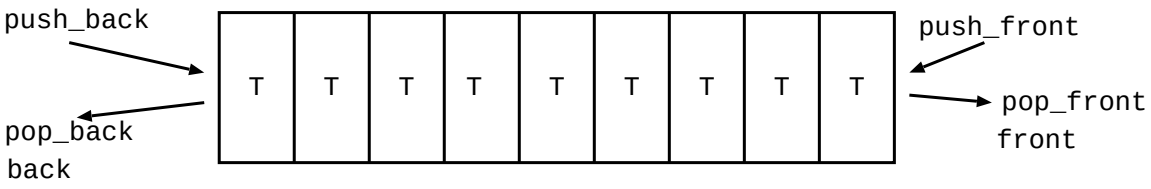
```
template<class T>
class Queue : private DoubleLinkedList<T> {
public :
    Queue ();
    Queue (const Queue<T>&);
    Queue<T>& operator= (const Queue<T>&);

    bool isempty ();
    bool isfull ();
    void push_back (T t);
    T pop_front ();
    T front ();
} ;
```

Die Queue wird hier mittels der doppelt verketteten Liste und privater Ableitung implementiert.

Die Implementierung ist in `QueueImp.cc`

DeQueue

Eine DeQueue (*double ended queue*) ist eine Struktur, die Einfügen und Entfernen an beiden Enden erlaubt:



Unsere DeQueue bietet die folgenden Funktionen:

- 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 voll oder leer.

Hier die Klassendefinition: `DeQueue.cc`

```
template<class T>
class DeQueue : private DoubleLinkedList<T> {
public :
    DeQueue ();
    DeQueue (const DeQueue<T>&);
    DeQueue<T>& operator= (const DeQueue<T>&);

    bool isempty ();
    bool isfull ();
    void push_front (T t);
    void push_back (T t);
    T pop_front ();
    T pop_back ();
    T front ();
    T back ();
} ;
```

Die DeQueue wird hier mittels der doppelt verketteten Liste und privater Ableitung implementiert.

Die Implementierung ist in `DeQueueImp.cc`

MinPriorityQueue/MaxPriorityQueue

Eine MinPriorityQueue ist eine Struktur in die man Objekte des Grundtyps `T` einfüllen kann und von der jeweils das *kleinste* der eingegebenen Elemente als nächstes entfernt werden kann.

Dazu muss auf dem Grundtyp `T` die Relation $<$ mittels dem `operator<` zur Verfügung stehen.

Eine MaxPriorityQueue erlaubt jeweils das Entfernen des *größten* der eingegebenen Elemente.

Eine MinPriorityQueue hat die folgenden Operationen:

- Konstruktion einer leeren MinPriorityQueue.
- Einfügen eines Elementes vom Typ `T`.
- Entfernen des kleinsten Elementes im Container.
- Inspektion des kleinsten Elementes im Container.
- Test ob MinPriorityQueue voll oder leer ist.

Hier die Klassendefinition: `MinPriorityQueue.cc`

```
template<class T>
class MinPriorityQueue : private DoubleLinkedList<T> {
public :
    MinPriorityQueue ();
    MinPriorityQueue (const MinPriorityQueue<T>&);
    MinPriorityQueue<T>& operator= (const MinPriorityQueue<T>&);

    bool isempty ();  // Container leer ?
    bool isfull ();   // Container voll ?
    void push (T t);  // Einfuegen
    T pop ();         // Entferne kleinstes
    T top ();         // Inspiziere kleinstes
} ;
```

Die Implementierung hier arbeitet mit einer einfach verketteten Liste. Das Einfügen hat Komplexität $O(1)$, das Entfernen/Inspizieren jedoch $O(n)$.

Es gibt Realisierungen, die beide Operationen in $O(\log n)$ Schritten schaffen.

Die Implementierung ist in `MinPriorityQueueImp.cc`

`MaxPriorityQueue.cc` bzw. `MaxPriorityQueueImp.cc` enthalten den Code für die MaxPriorityQueue.

```
#include<iostream.h>

#include"DoubleLinkedList.cc"
#include"DoubleLinkedListIterator.cc"
#include"DoubleLinkedListElement.cc"
#include"DoubleLinkedListImp.cc"

#include"Queue.cc"
#include"QueueImp.cc"
#include"DeQueue.cc"
#include"DeQueueImp.cc"
#include"Stack.cc"
#include"StackImp.cc"
#include"Set.cc"
#include"SetImp.cc"
#include"MaxPriorityQueue.cc"
#include"MaxPriorityQueueImp.cc"
#include"MinPriorityQueue.cc"
#include"MinPriorityQueueImp.cc"

int main ()
{
    Queue<int> q;

    q.push_back(1);
    q.push_back(10);
    q.push_back(100);
    cout << q.pop_front() << endl;
    cout << q.pop_front() << endl;
    cout << q.pop_front() << endl;
    cout << endl;

    DeQueue<int> dq;

    dq.push_back(1);
    dq.push_front(10);
    dq.push_back(100);
    cout << dq.pop_front() << endl;
    cout << dq.pop_front() << endl;
    cout << dq.pop_front() << endl;
    cout << endl;

    Stack<int> s;

    s.push_front(1);
    s.push_front(10);
    s.push_front(100);
    cout << s.pop_front() << endl;
    cout << s.pop_front() << endl;
    cout << s.pop_front() << endl;
```

```
    cout << endl;

    Set<int> m;

    m.insert(1);
    m.insert(10);
    m.insert(100);

    cout << m.ismember(1) << m.ismember(34) << endl << endl;

    MaxPriorityQueue<int> maxq;

    maxq.push(68);
    maxq.push(4582);
    maxq.push(9);
    cout << maxq.top() << endl << endl;

    MinPriorityQueue<int> minq;

    minq.push(68);
    minq.push(4582);
    minq.push(9);
    cout << minq.top() << endl << endl;
}
```

mit der Ausgabe:

```
1
10
100

10
1
100

100
10
1

10

4582

9
```

Map

Eine Map ist eine Struktur, die Objekten eines Typs `Key` Objekte eines Typs `T` zuordnet.

Man bezeichnet eine Map auch als *assoziatives Feld*.

Beispiel wäre ein Telephonbuch:

Meier	→	504423
Schmidt	→	162300
Müller	→	712364
Huber	→	8265498

Diese Zuordnung könnte man mittels

```
Map<string,int> telefonbuch;

telefonbuch["Meier"] = 504423;
...
```

realisieren.

In einer Map werden je zwei Objekte der Typen `Key` (*Schlüssel*) und `T` (*Wert*) zu einem Paar vom Typ `pair<Key,T>` kombiniert.

Der Zugriff erfolgt über ein Objekt des Typs `Key`. Das Ergebnis des Zugriffs ist das `T`-Objekt des entsprechenden Paares.

Ein Objekt vom Typ `Key` kann nur einmal in einem Paar vorkommen (Damit ist das Telephonbuch ein schlechtes Beispiel . . .).

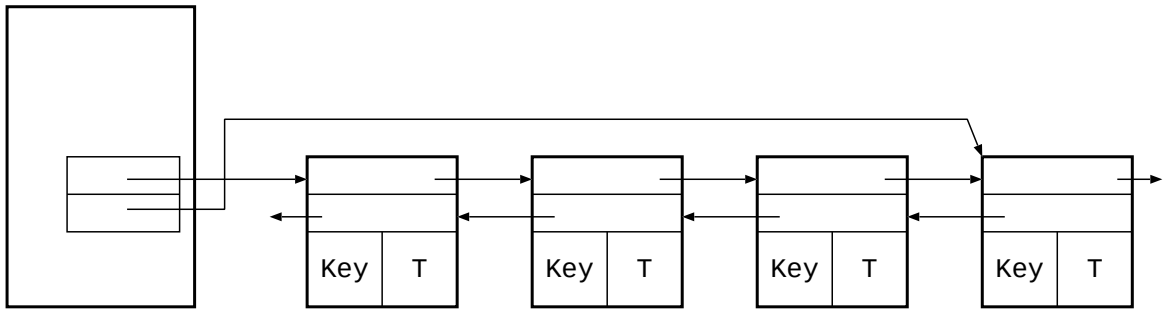
Eine Map bietet folgende Operationen:

- Iterator zum Durchlaufen aller Schlüssel-Wert-Paare im Container.
- `operator[](Key k)` für den Zugriff. Gibt es bereits ein Paar mit dem angegebenem Schlüssel, so wird eine Referenz auf den Wert zurückgegeben. Existiert der Schlüssel noch nicht so wird ein Paar in den Container aufgenommen und eine Referenz auf den Wert zurückgegeben (Initialisierung mit argumentlosem Konstruktor).
- `find(Key k)` liefert Zeiger auf den Wert, wenn der Schlüssel existiert, ansonsten 0.
- Auf dem Schlüssel muss der Gleichheitsoperator `operator==` definiert sein.

Hier die Klassendefinition: `Map.cc`

```
template<class Key, class T>
struct pair {
    Key key;
    T t;
} ;

template<class Key, class T>
class Map : private DoubleLinkedList<pair<Key,T> > {
public :
    Map ();
    Map (const Map<Key,T>&);
    Map<Key,T>& operator= (const Map<Key,T>&);

    T& operator[](const Key& k);
    T* find (const Key& k);

    typedef DoubleLinkedList<pair<Key,T> >::Iterator Iterator;
    Iterator begin () const;
    Iterator end () const;
} ;
```

Hier haben wir die Map als Liste von Schlüssel-Wert-Paaren realisiert:



Später werden wir Realisierungen einer Map angeben die Einfügen und Suchen in $O(\log n)$ Schritten realisiert (Dazu ist dann eine Relation $<$ auf den Schlüsseln erforderlich).

Die Implementierung der Methoden findet man in `MapImp.cc`

Ein Beispiel für die Anwendung einer Map findet sich in den nun folgenden Huffmanbäumen.

Huffmanbäume mit Containern

Dieses Beispiel behandelt nochmals die Huffmanbäume von Seite 100, nur erfolgt hier die Realisierung ausschließlich mit Containern.

Wir fassen die einzelnen Schritte zusammen, die Sie dann genau so in der Funktion `main` im folgenden Programm wiederfinden können:

- Zähle Häufigkeiten der zu kodierenden Zeichen in der vorliegenden Nachricht. Hierzu verwendet man eine `Map<char,int>`.
- In jedem Knoten des Huffmanbaumes brauchen wir
  - das zu kodierende Zeichen (eigentlich nur im Blatt) und
  - die Häufigkeit des Vorkommens.

  Dazu benutzen wir ein `pair<char,int>` dem wir mittels `typedef` zur Abkürzung den Namen `hdata` geben. Der Huffmanbaum selbst ist dann vom Typ `BinaryTree<hdata>`.
- Nächster Schritt ist das Erzeugen aller Blätter des Huffmanbaumes. Dies sind die vorkommenden Zeichen mit ihren jeweiligen Häufigkeiten. Iteratoren, die auf diese Blätter zeigen füllen wir in eine `MinPriorityQueue`, da wir in der Aufbauphase jeweils die zwei Knoten mit der kleinsten Häufigkeit suchen müssen.

  Für die `MinPriorityQueue` ist der `operator<` auf zwei Objekten des Typs `BinaryTree<hdata>::Iterator` zu definieren.
- Nun kann der Huffmanbaum aufgebaut werden:

  Solange die MinPriorityQueue nicht leer ist
  - Entnehme die beiden Teilbäume mit der kleinsten Häufigkeit.
  - Erzeuge neuen Baum aus beiden Teilbäumen, dessen Häufigkeit die Summe der Häufigkeiten der Teilbäume ist.
  - Ist die MinPriorityQueue leer, so ist man fertig.
- Schließlich kann mit dem Huffmanbaum der Code berechnet werden. Der Code wird in einer `Map<char,string>` gespeichert.

  Dazu läuft man den Baum rekursiv von der Wurzel aus ab (depth first). Für jeden Abstieg in einen linken Teilbaum hängt man eine „1“ an den bis dahin berechneten Code, bei jedem Abstieg in einen rechten Teilbaum eine „0“.

Hier das Programm: Huffman.cc

```
#include<iostream.h>
#include<string>
// Hier unsere Container ...
#include"DoubleLinkedList.cc"
#include"DoubleLinkedListIterator.cc"
#include"DoubleLinkedListElement.cc"
#include"DoubleLinkedListImp.cc"
#include"Map.cc"
#include"MapImp.cc"
#include"BinaryTree.cc"
#include"BinaryTreeIterator.cc"
#include"BinaryTreeImp.cc"
#include"MinPriorityQueue.cc"
#include"MinPriorityQueueImp.cc"

// Daten fuer einen Knoten im Huffmanbaum
typedef pair<char,int> hdata; // template pair aus Map.cc

// operator< auf Iteratoren der Knoten des Huffmanbaumes
// benutzt die MinPriorityQueue zum sortieren
bool operator< (BinaryTree<hdata>::Iterator i,
                BinaryTree<hdata>::Iterator j)
{
    return i->t < j->t;
}

// Rekursive Prozedur zum Aufbau des Codes aus dem Baum
void make_code (string s,
                BinaryTree<hdata>::Iterator i, Map<char,string>& code)
{
    if (i.isleaf()) { code[i->key]=s; return;}
    else {
        make_code(s+"1",i.left(),code);
        make_code(s+"0",i.right(),code);
    }
}

int main ()
{
    char* nachricht = "ABRACADABRASIMSALABIM";

    // Zaehle Haeufigkeiten
    Map<char,int> cmap;
    for (int i=0; nachricht[i]!='\0'; i=i+1)
        if (cmap.find(nachricht[i]))
            cmap[nachricht[i]]++;
        else
            cmap[nachricht[i]]=1;
    for (Map<char,int>::Iterator i=cmap.begin(); i!=cmap.end(); i++)
        cout << i->key << " " << i->t << endl;
```

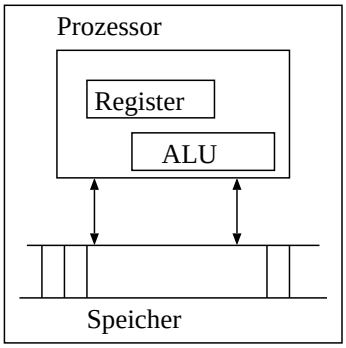
```
    // Erzeuge Blaetter mit ihren Haeufigkeiten
    MinPriorityQueue<BinaryTree<hdata>::Iterator> q;
    for (Map<char,int>::Iterator i=cmap.begin(); i!=cmap.end(); i++)
    {
        BinaryTree<hdata>* p = new BinaryTree<hdata>(*i);
        q.push(p->begin());
    }

    // Baue Huffmanbaum
    BinaryTree<hdata>* root;
    while (!q.isempty())
    {
        BinaryTree<hdata>::Iterator left = q.pop();  // kleinster
        BinaryTree<hdata>::Iterator right = q.pop(); // zweitkleinster
        hdata hd; hd.t = left->t+right->t;           // Summe Haeuf...
        root = new BinaryTree<hdata>(left,hd,right); // fasse zusammen
        if (!q.isempty()) q.push(root->begin());     // wieder in queue
    }

    // Erzeuge Code
    Map<char,string> code;
    make_code("",root->begin(),code);
    for (Map<char,string>::Iterator i=code.begin(); i!=code.end(); i++)
        cout << i->key << " " << i->t << endl;

    // codiere Nachricht
    for (int i=0; nachricht[i]!='\0'; i=i+1)
        cout << code[nachricht[i]];
    cout<<endl;
}
```

19 Beispiel: Logiksimulator

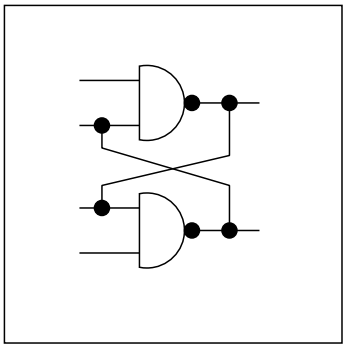
Zum Abschluss wollen wir ein größeres, zusammenhängendes Beispiel behandeln: Die Simulation von digitalen Schaltungen.

Simulation komplexer Systeme

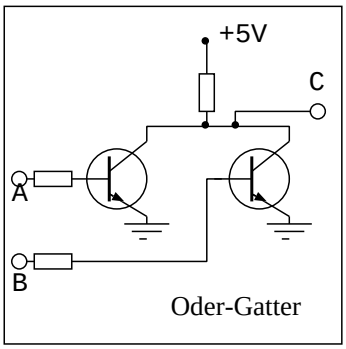
Ein Computer ist ein komplexes System, das auf verschiedenen Skalen modelliert werden kann.



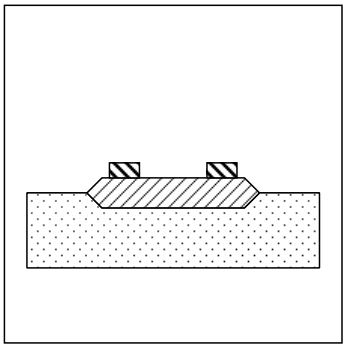
Aufbau aus Baugruppen: Prozessor, Speicher, Busse, Caches, Aufbau der Prozessoren aus Registern, ALU, usw.



Aufbau aus logischen Grundelementen: Und-Gatter, Oder-Gatter, Nand-Gatter, ...
Zu jeder Zeit gibt es zwei mögliche Zustände auf jeder Leitung: *high* und *low*. Modellierung durch boolsche Ausdrücke, zeitdiskrete, ereignisgesteuerte Simulation.



Die Gatter können wiederum aus Transistoren aufgebaut werden.
Spannungsverlauf an einem Punkt kontinuierlich. Modellierung durch Systeme gewöhnlicher Differentialgleichungen.



Aufbau eines Chips durch Schichten unterschiedlich dotierter Halbleiter.
Modellierung der Ladungsdichtenverteilung durch partielle Differentialgleichungen.

Wir beschäftigen uns hier mit der Modellierung und Simulation auf der Gatterebene.

Grundbausteine digitaler Schaltungen

Schaltungen auf der Logikebene sind aus digitalen Grundelementen aufgebaut.

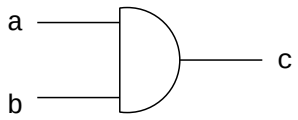
Die logischen Grundelemente besitzen einen oder mehrere Eingänge, sowie einen Ausgang.

Digital bedeutet, dass an einem Ein- bzw. Ausgang nur zwei verschiedene Spannungswerte vorkommen können: *high* bzw. 1 oder *low* bzw. 0.

In Abhängigkeit der Belegung der Eingänge liefert der Baustein einen bestimmten Ausgangswert.

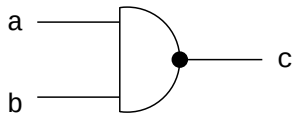
Hier sind die Grundschaltungen mit ihren Wahrheitstabellen:

And



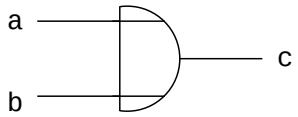
a	b	c
1	1	1
1	0	0
0	1	0
0	0	0

Nand



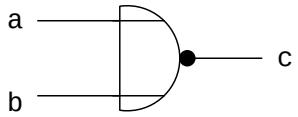
a	b	c
1	1	0
1	0	1
0	1	1
0	0	1

Or



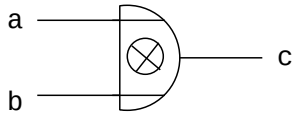
a	b	c
1	1	1
1	0	1
0	1	1
0	0	0

Nor



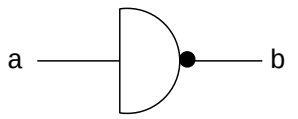
a	b	c
1	1	0
1	0	0
0	1	0
0	0	1

Exor



a	b	c
1	1	0
1	0	1
0	1	1
0	0	0

Inverter



a	b
1	0
0	1

Bemerkung: Durch Kombination mehrerer Nand-Gatter können alle anderen Gatter realisieriert werden.

Reale Gatter

Die digitalen Grundbausteine werden durch elektronische Schaltungen realisiert, die vollständig auf einem Halbleiterchip aufgebaut sind.

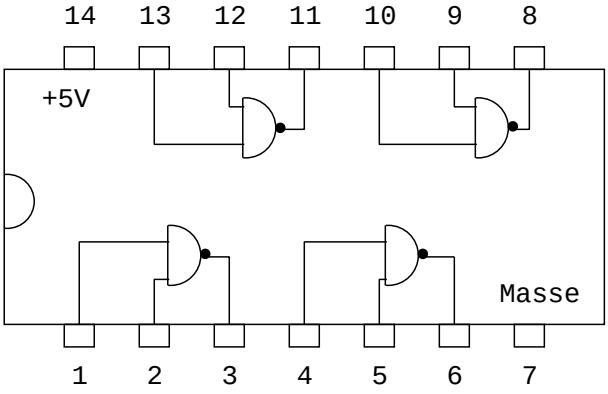
Im Prinzip besteht ein Mikroprozessor aus Millionen solcher Gatter auf einem einzigen Chip. Da sehr viele Gatter auf einem Chip sind spricht man von *very large scale integration* (VLSI).

Es gibt aber auch einzelne (oder wenige) Gatter auf einem Chip, man spricht von *small scale integration* (SSI).

Bei den realen Gattern unterscheidet man verschiedene *Schaltungsfamilien*. Bausteine gleicher Familien können untereinander beliebig verschaltet werden, bei Bausteinen unterschiedlicher Familien ist dies nicht unbedingt möglich.

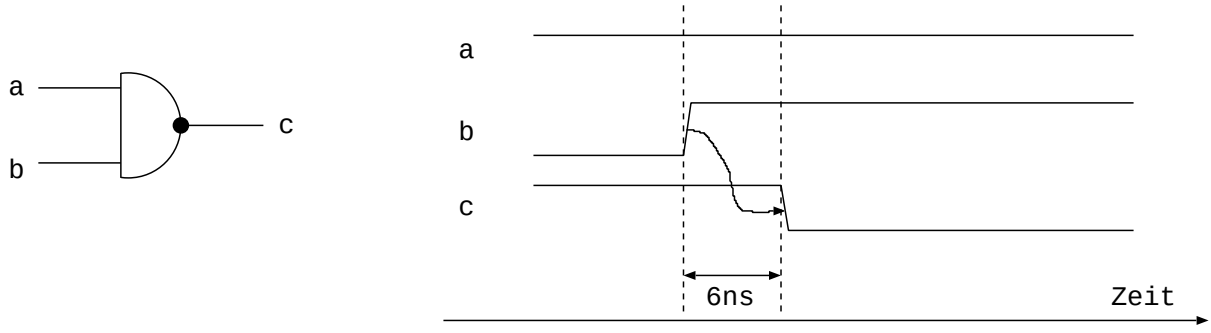
Eine der bekanntesten Familien digitaler Schaltglieder ist TTL (*transistor-transistor logic*). In einem Plastikgehäuse ist der Chip mit den Anschlüssen untergebracht.

So enthält der Baustein mit der Nummer 7400 vier Nand-Gatter:



Bei TTL entspricht der logische Wert *high* einer Spannung von $2 \ldots 5$ Volt, der Wert *low* $0 \ldots 0.4$ Volt.

Die Bausteine haben ausserdem ein *dynamisches Verhalten*, d. h. eine Änderung der Eingangsspannung macht sich erst mit einer gewissen Verzögerung am Ausgang bemerkbar:



Schaltnetze

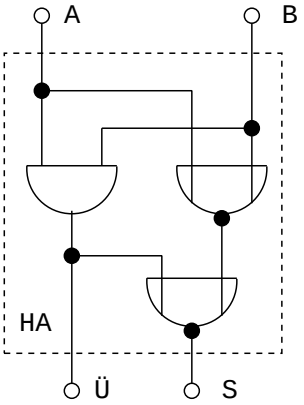
Ein *digitales Schaltnetz* ist eine Verknüpfung von logischen Grundelementen mit n Eingängen und m Ausgängen, wobei die Belegung der Ausgänge *nur* eine Funktion der Eingänge ist (abgesehen von der Verzögerung).

Ein Schaltnetz hat kein Gedächtnis, die Belegung der Ausgänge hängt nicht von früheren Belegungen der Eingänge ab.

Als Beispiel betrachten wir die Addition von Binärzahlen:

A3-A0	0	1	1	1	0
B3-B0	0	0	1	1	1
Übertrag	1	1	1		
Ergebnis	1	0	1	0	1

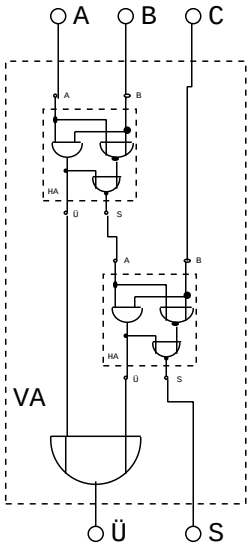
Halbaddierer: Addiert eine Stelle ohne Übertrag.



Wahrheits-tabelle

A	B	S	Ü
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Volladdierer: Addiert eine Stelle mit Übertrag.

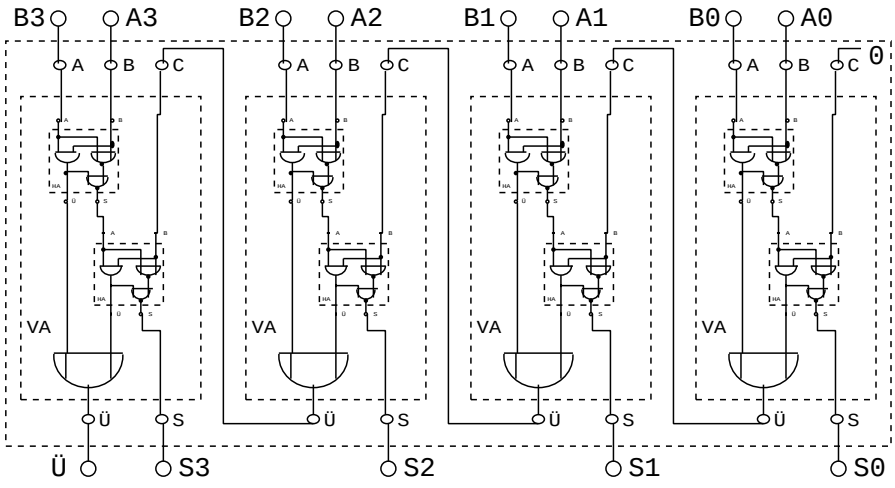


C	A	B	S	Ü
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

gerade Zahl von Einsen?

mehr als eine Eins?

4-Bit-Addierer mit durchlaufendem Übertrag

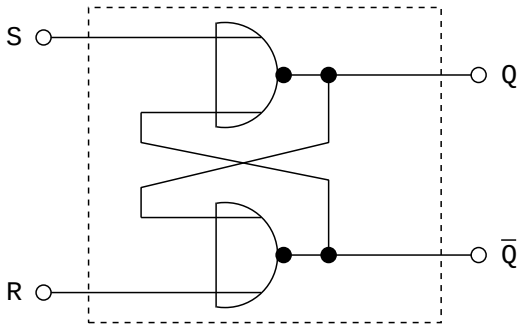


Schaltwerke

Ein *digitales Schaltwerk* ist eine Verknüpfung von Grundbausteinen mit n Eingängen und m Ausgängen, deren Belegung der Ausgänge von den Eingängen und internen Zuständen abhängt.

Die internen Zustände ergeben sich aus früheren Belegungen der Eingänge.

Wir betrachten als simpelstes Beispiel ein *SR-Flipflop* aus zwei Nor-Gattern:



Nor-Gatter

a	b	c
1	1	0
1	0	0
0	1	0
0	0	1

SR-Flipflop

S	R	Zust	Q	Q̄	Zust
1	1	-	0	0	G
0	1	G	1	0	I
0	0	I	1	0	I
1	0	I	0	1	II
1	0	G	0	1	II
0	0	II	0	1	II
0	1	II	1	0	I

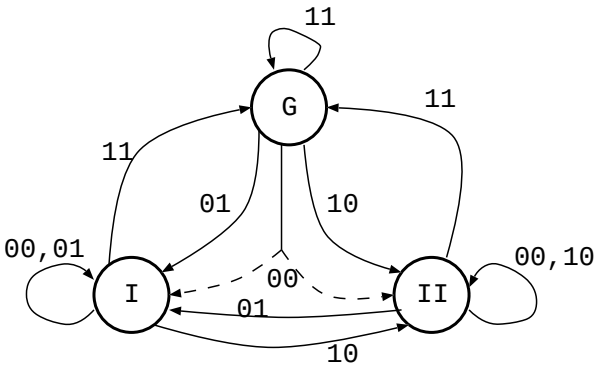
Die Eingangskombination $S = 0$, $R = 0$ hat verschiedene Ausgangsbelegungen zur Folge, je nach dem welche Signale vorher anlagen!

Bei zwei Ausgängen gibt es vier mögliche Kombinationen.

Die Kombination $Q = 1$, $\bar{Q} = 1$ ist nicht möglich, da eine 1 am Ausgang eines Nor-Gatters zweimal 0 am Eingang braucht, und das ist ein Widerspruch.

Die verbleibenden drei Zustände sind G ($Q = 0$, $\bar{Q} = 0$), I ($Q = 1$, $\bar{Q} = 0$) und II ($Q = 0$, $\bar{Q} = 1$) aus der Tabelle oben.

Aus der Tabelle oben erhalten wir das folgende Zustandsübergangsdiagramm:



Weitere Varianten von Flipflops können dazu benutzt werden um Speicher oder Zähler zu bauen.

Der Simulator

Die Zeit die eine bestimmte Operation im Rechner benötigt bestimmt sich aus den Gatterlaufzeiten der Schaltung die die Operation realisiert (siehe z. B. Addierer).

Der Schaltungsdesigner muss überprüfen ob die Schaltung in der Lage ist die geforderte Operation innerhalb einer bestimmten Zeit (Takt) zu berechnen.

Dazu benutzt er einen Logiksimulator, der die Schaltung im Rechner nachbildet und simuliert. Er kann damit die Logikpegel an jedem Punkt der Schaltung über die Zeit verfolgen.

Die offensichtlichen Objekte in unserem System sind:

- Logische Grundbausteine mit ihrem Ein-/Ausgabeverhalten und Verzögerungszeit.
- Drähte, die Aus- und Eingänge der Gatter miteinander verbinden.

Eine andere Sache ist nicht sofort offensichtlich:

- In der Realität arbeiten alle Gatter simultan und unabhängig voneinander.
- In einem C++ Programm wird zu einer Zeit die Methode genau eines Objektes (=Gatter) ausgeführt. In welcher Reihenfolge sollen dann die Objekte bearbeitet werden?
- Wenn sich am Eingang eines Gatters nichts ändert, so ändert sich auch am Ausgang nichts. Um Arbeit zu sparen sollte also nur dort gearbeitet werden „wo sich etwas tut“.

Die Problematik löst man dadurch, dass man *Ereignisse*, wie etwa

„Eingang 1 von Gatter x geht in den Zustand *high*“

einführt. Das Eintreten eines Ereignisses löst dann wieder weitere Ereignisse bei anderen Objekten (Drähten oder Gattern) aus.

Die Koordination der Ereignisse wird von einem Objekt der Klasse `Simulator` übernommen.

Diese Art der Simulation bezeichnet man auch als *ereignisgesteuerte Simulation* (*discrete event simulation*).

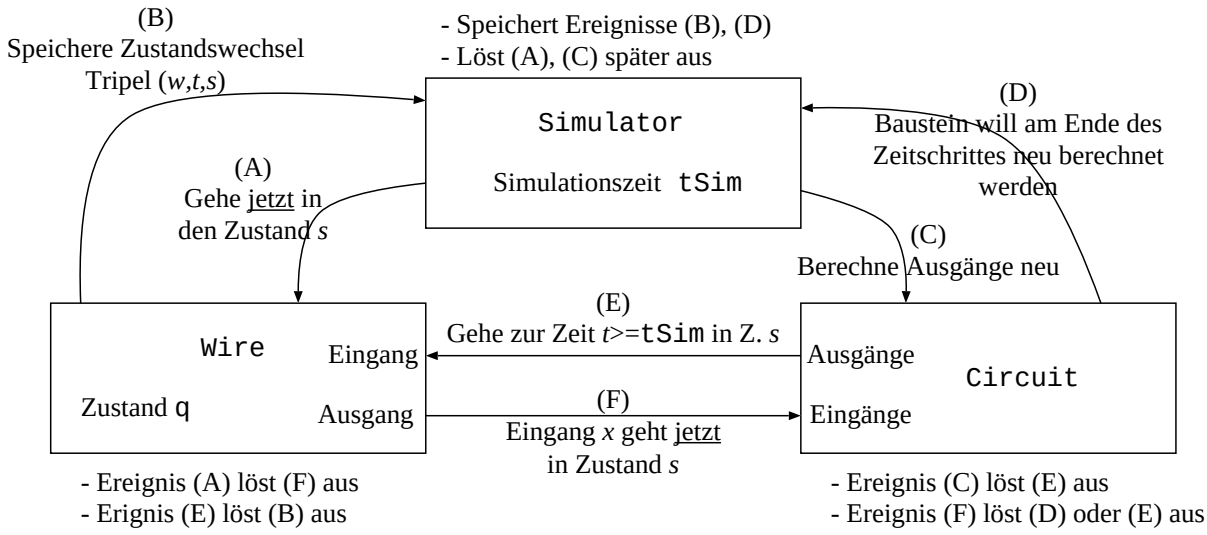
Die Simulation als Ganzes gliedert sich in zwei Abschnitte:

- Eine Aufbauphase in der wir die zu simulierende Logikschaltung definieren:
  - Es werden Bausteine erzeugt. Alle Bausteine werden von der Schnittstellenklasse `Circuit` abgeleitet.
  - Die Verbindungsdrähte sind vom Typ `Wire` (eine konkrete Klasse). Drähte sind Punkt-zu-Punkt Verbindungen. Der Eingang eines Drahtes wird an den Ausgang eines Bausteins und der Ausgang eines Drahtes an den Eingang eines Bausteins angeschloßen. Verzweigungen werden durch einen speziellen Baustein (mit einem Eingang und zwei Ausgängen) realisiert.
- Die eigentliche Simulation. In der Aufbauphase werden erste Ereignisse generiert, die dann die weiteren Ereignisse auslösen.

  Die Simulationsphase geht in diskreten *Zeitschritten* vor und wird von einem Objekt der Klasse `Simulator` koordiniert.

  Die Klasse `Simulator` definiert eine globale Zeit $t_{Sim} \in \mathbb{N}$. Die Simulation beginnt zum Zeitpunkt $t_{Sim} = 0$.

Betrachten wir die Ereignisse die zwischen den Objekten ausgetauscht werden:



$t_{Sim} = 0$

Solange $t_{Sim} \leq t_{End}$

- $\forall$ gespeicherten Zustandswechsel (w, t, s) mit $t = t_{Sim}$: löse (A) aus.
- $\forall$ Bausteine deren Eingang sich im Zeitschritt t_{Sim} geändert hat: Berechne Baustein neu.
- $t_{Sim} = t_{Sim} + 1$

Die Klassendefinition für den Draht: `Wire.cc`

```
// moegliche Zustaende
enum State {low, high, unknown};

class Wire {
public:
    Wire ();
    // Draht im Zustand unknown erzeugen

    State GetState ();
    // aktuellen Zustand auslesen

    void ChangeState (int t, State s);
    // (E): Zur Zeit t soll Zustand s werden

    void Action (State s);
    // (A): wechsle jetzt in neuen Zustand

    void ConnectInput (Circuit& cir, int i);
    // Eingang des Drahtes an Ausgang i des Bausteins c
    // anschliessen

    void ConnectOutput (Circuit& cir, int i);
    // Ausgang des Drahtes an Eingang i des Bausteins c
    // anschliessen

private:
    State q;      // der Zustand
    Circuit* c;   // Baustein am Ausgang des Drahtes
    int pin;      // pin des Bausteins
} ;
```

Der Zustand kann auch unbekannt sein, um feststellen zu können ob die Schaltung korrekt initialisiert wird.

Der Konstruktor erzeugt den Draht im Zustand `unknown`.

Die nächsten drei Methoden werden in der Simulationsphase verwendet.

Die letzten beiden Methoden werden in der Aufbauphase verwendet. Der Draht merkt sich an Baustein und Eingang des Bausteins an den er angeschlossen wurde.

```
Wire::Wire ()
{
    // Initialisiere mit unbekanntem Zustand
    q = unknown;
}

inline State Wire::GetState ()
{
    return q;
}


void Wire::ChangeState (int t, State s)
{
    Sim.StoreWireEvent(*this,t,s);
}

void Wire::Action (State s)
{
    if (s==q) return;      // nix zu tun
    q = s;                 // neuer Zustand
    c->ChangeInput(q,pin); // Nachricht an angeschlossen Baustein
}

void Wire::ConnectInput (Circuit& cir, int i)
{
    // Merke NICHT an wen ich angeschlossen bin
    // aber Baustein muss mich kennen.
    cir.ConnectOutput(*this,i);
}

void Wire::ConnectOutput (Circuit& cir, int i)
{
    // Merke Baustein, an den der Ausgang angeschlossen ist
    c = &cir;
    pin = i;
    // Rueckverbindung Baustein an Draht
    c->ConnectInput(*this,pin);
}
```

Das Simulatorobjekt `Sim` ist ein globales Objekt, das alle kennen.

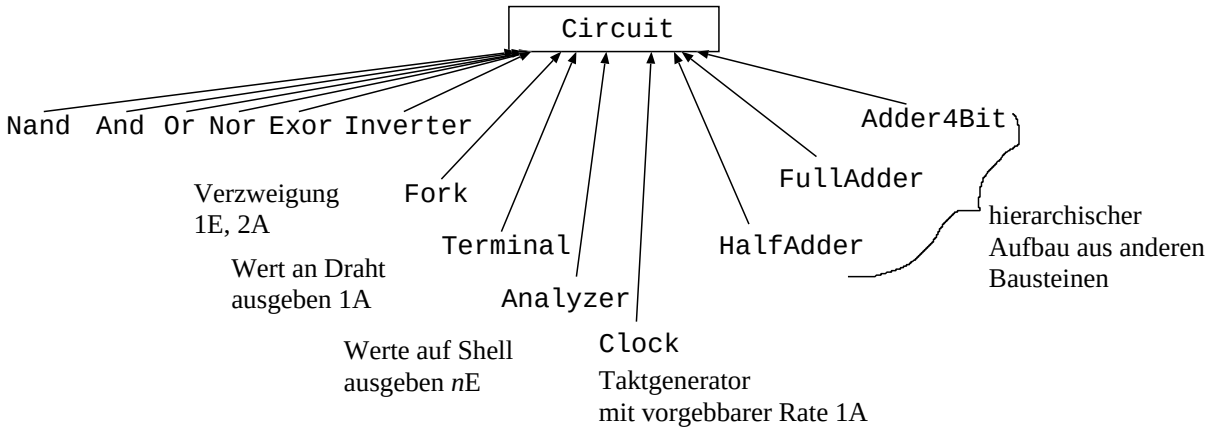
Es gibt nur ein Simulatorobjekt. Wenn es keinen Sinn macht mehr als ein Objekt von einer Klasse zu instanzieren, nennt man dies ein *Singleton*.

Die Schnittstellenbasisklasse für Bausteine: `Circuit.cc`

```
class Circuit {
public:
    virtual ~Circuit ();
    // virtual destructor

    virtual void ChangeInput (State s, int pin) = 0;
    // (F): Eingang wechselt Zustand

    virtual void Action () = 0;
    // (C): Ausgang neu berechnen

    virtual void ConnectInput (Wire& w, int pin) = 0;
    // verdrahte Eingang

    virtual void ConnectOutput (Wire& w, int pin) = 0;
    // verdrahte Ausgang
} ;

Circuit::~Circuit () {}
```

Die `Connect...`-Funktionen werden von der entsprechenden Funktion der Klasse `Wire` aufgerufen.

... und die zur Zeit existierenden abgeleiteten Klassen



Hier die abgeleitete Klasse für das Nand-Gatter: `Nand.cc`

```
class Nand : public Circuit {
public:
    Nand ();
    // Konstruktor

    ~Nand ();
    // default destructor ist OK

    virtual void ChangeInput (State s, int pin);
    // Eingang wechselt zur aktuellen Zeit den Zustand

    virtual void Action ();
    // berechne Gatter neu und benachrichtige Draht
    // am Ausgang

    virtual void ConnectInput (Wire& w, int pin);
    // verdrahte Eingang

    virtual void ConnectOutput (Wire& w, int pin);
    // verdrahte Ausgang

private:
    Wire* a;          // Eingang 1
    Wire* b;          // Eingang 2
    Wire* c;          // Ausgang
    bool actionFlag; // merke ob bereits aktiviert
} ;
```

Das Gatter merkt sich, welche Drähte an Ein- und Ausgängen angeschlossen sind.

```
Nand::Nand()
{
    a=b=c=0; // nix angschlossen
    actionFlag=false;
}

Nand::~Nand() {}

void Nand::ChangeInput (State s, int pin)
{
    // Sorge dafuer, dass Gatter neu berechnet wird
    if (!actionFlag)
    {
        Sim.StoreCircuitEvent(*this);
        actionFlag=true;
    }
}

void Nand::Action ()
{
    // Lese Eingangssignale
    State A = a->GetState();
    State B = b->GetState();
    State Output=unknown;

    // Wertetabelle
    if (A==high&& B==high) Output=low;
    if (A==low || B==low ) Output=high;

    // Setze Draht
    if (c!=0) c->ChangeState(Sim.GetTime()+3,Output);

    // erlaube neue Auswertung
    actionFlag=false;
}

void Nand::ConnectInput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    if (pin==0) a = &w;
    if (pin==1) b = &w;
}

void Nand::ConnectOutput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    c = &w;
}
```

Ein zweites Beispiel ist die Verzweigung `Fork`: `Fork.cc`

```
class Fork : public Circuit {
public:
    Fork ();
    // Konstruktor

    ~Fork ();
    // default destructor ist OK

    virtual void ChangeInput (State s, int pin);
    // Eingang wechselt zur aktuellen Zeit den Zustand

    virtual void Action ();
    // berechne Gatter neu und benachrichtige Draht
    // am Ausgang

    virtual void ConnectInput (Wire& w, int pin);
    // verdrahte Eingang

    virtual void ConnectOutput (Wire& w, int pin);
    // verdrahte Ausgang

private:
    Wire* a;          // Eingang
    Wire* b;          // Ausgang 1
    Wire* c;          // Ausgang 2
} ;
```

Hier werden sofort Ereignisse bei den Drähten am Ausgang ausgelöst:

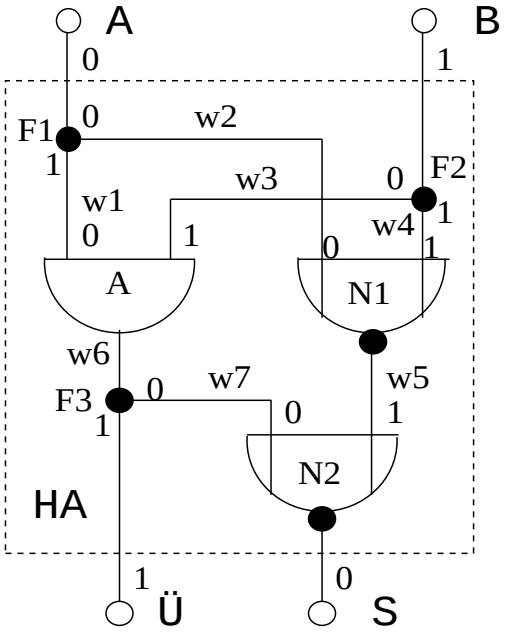
ForkImp.cc

```
Fork::Fork()
{
    a=b=c=0; // nix angschlossen
}

Fork::~Fork() {}

void Fork::ChangeInput (State s, int pin)
{
    // Leite Eingang SOFORT an beide Ausgaenge weiter
    if (b!=0) b->ChangeState(Sim.GetTime(),s);
    if (c!=0) c->ChangeState(Sim.GetTime(),s);
}

void Fork::Action ()
{
    // nix zu tun
}

void Fork::ConnectInput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    a = &w;
}

void Fork::ConnectOutput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    if (pin==0) b = &w;
    if (pin==1) c = &w;
}
```

`HalfAdder` ist ein Beispiel für einen zusammengesetzten Baustein:

`HalfAdder.cc`

```
class HalfAdder : public Circuit {
public:
    HalfAdder ();
    // Konstruktor

    ~HalfAdder ();
    // destruktor

    virtual void ChangeInput (State s, int pin);
    // Eingang wechselt zur aktuellen Zeit den Zustand

    virtual void Action ();
    // berechne Gatter neu und benachrichtige Draht
    // am Ausgang

    virtual void ConnectInput (Wire& w, int pin);
    // verdrahte Eingang

    virtual void ConnectOutput (Wire& w, int pin);
    // verdrahte Ausgang

private:
    Wire w1,w2,w3,w4,w5,w6,w7; // lokale Draehte
    And A;           // Und Gatter
    Nor N1,N2;       // sowie zwei Nor Gatter
    Fork F1,F2,F3;   // und drei Verzweigungen
} ;
```

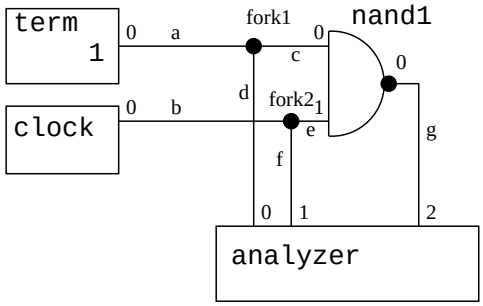


```
HalfAdder::HalfAdder()
{
    w1.ConnectInput(F1,1);
    w1.ConnectOutput(A,0);
    w2.ConnectInput(F1,0);
    w2.ConnectOutput(N1,0);
    w3.ConnectInput(F2,0);
    w3.ConnectOutput(A,1);
    w4.ConnectInput(F2,1);
    w4.ConnectOutput(N1,1);
    w5.ConnectInput(N1,0);
    w5.ConnectOutput(N2,1);
    w6.ConnectInput(A,0);
    w6.ConnectOutput(F3,0);
    w7.ConnectInput(F3,0);
    w7.ConnectOutput(N2,0);
}

HalfAdder::~HalfAdder() {}

void HalfAdder::ChangeInput (State s, int pin)
{
    if (pin==0) F1.ChangeInput(s,0);
    if (pin==1) F2.ChangeInput(s,0);
}

void HalfAdder::Action () {}

void HalfAdder::ConnectInput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    if (pin==0) F1.ConnectInput(w,0);
    if (pin==1) F2.ConnectInput(w,0);
}

void HalfAdder::ConnectOutput (Wire& w, int pin)
{
    // Wird von Connect-Funktion des Drahtes aufgerufen
    if (pin==0) N2.ConnectOutput(w,0);
    if (pin==1) F3.ConnectOutput(w,1);
}
```

```
// Simulator, Singleton
class Simulator {
public:
    Simulator ();
    // Konstruktor

    int GetTime ();
    // aktuelle Zeit auslesen

    void StoreWireEvent (Wire& w, int t, State s);
    // (B): Draht w wird zur Zeit t in Zustand s wechseln

    void StoreCircuitEvent (Circuit& c);
    // (D): Baustein c soll zur aktuellen Zeit neu berechnet werden

    void Simulate (int end);
    // Starte Simulation bei Zeit 0

private:
    struct WireEvent { // Eine lokale Struktur
        WireEvent ();  // fuer Ereignis "Zustandswechsel"
        WireEvent (Wire& W, int T, State S);
        Wire* w;
        int t;
        State s;
        bool operator< (WireEvent we);
    } ;
    int time;
    MinPriorityQueue<WireEvent> pq; // Fuer (B)-Ereignisse
    Queue<Circuit*> q;              // Fuer (D)-Ereignisse
} ;

// Globale Variable vom Typ Simulator (Singleton).
// Wird von allen Bausteinen und Draehten benutzt!
Simulator Sim;
```

```
// Methoden fuer die geschachtelte Klasse
Simulator::WireEvent::WireEvent () { w=0; t=0; s=unknown; }

Simulator::WireEvent::WireEvent (Wire& W, int T, State S) {w=&W; t=T; s=S;}

bool Simulator::WireEvent::operator< (WireEvent we)
{
    if (t<we.t) return true;
    if (t==we.t && ((unsigned int)w)<((unsigned int)we.w)) return true;
    return false;
}

// Konstruktor
Simulator::Simulator (){time = 0;}

int Simulator::GetTime (){return time;}

void Simulator::StoreWireEvent (Wire& w, int t, State s)
{
    pq.push(WireEvent(w,t,s));
}

void Simulator::StoreCircuitEvent (Circuit& c)
{
    q.push_back(&c);
}

void Simulator::Simulate (int end)
{
    WireEvent we;

    while (time<=end)
    {
        // Alle Draehte fuer die aktuelle Zeit
        while (!pq.isempty())
        {
            we = pq.top();        // kleinster Eintrag
            if (we.t>time) break; // alle Zustaende fuer Zeitschritt OK
            pq.pop();             // entferne Eintrag
            (we.w)->Action(we.s); // neuer Zustand
        }

        // Berechne Bausteine zur aktuellen Zeit neu
        while (!q.isempty())
            (q.pop_front())->Action();

        // Zeitschritt fertig
        time = time+1;
    }
}
```

Als Beispiel betrachten wir folgende Schaltung:



Dafür produziert der Simulator folgende Ausgabe:

```
 0  U  U  U
 1  U  1  1
 2  U  1  1
 3  U  1  1
 4  0  1  1
 5  0  1  1
 6  0  1  1
 7  0  1  1
 8  0  1  1
 9  0  1  1
10  0  1  1
11  0  0  1
12  0  0  1
13  0  0  1
14  1  0  1
15  1  0  1
16  1  0  1
17  1  0  1
18  1  0  1
19  1  0  1
20  1  0  1
21  1  1  1
22  1  1  1
23  1  1  1
24  0  1  1
25  0  1  1
26  0  1  1
27  0  1  1
28  0  1  1
29  0  1  1
30  0  1  1
31  0  0  1
32  0  0  1
33  0  0  1
```

Hier das zugehörige Hauptprogramm: `LSBeispiel.cc`

```
#include<iostream.h>
#include"DoubleLinkedList.cc"
#include"DoubleLinkedListIterator.cc"
#include"DoubleLinkedListElement.cc"
#include"DoubleLinkedListImp.cc"
#include"MinPriorityQueue.cc"
#include"MinPriorityQueueImp.cc"
#include"Queue.cc"
#include"QueueImp.cc"
class Simulator; // forward declaration
class Wire;      // forward declaration
class Circuit;   // forward declaration
#include"Wire.cc"
#include"Circuit.cc"
#include"Simulator.cc"
#include"SimulatorImp.cc"
#include"WireImp.cc"
#include"Nand.cc"
#include"NandImp.cc"
#include"Terminal.cc"
#include"TerminalImp.cc"
#include"Fork.cc"
#include"ForkImp.cc"
#include"Analyzer.cc"
#include"AnalyzerImp.cc"
#include"Clock.cc"
#include"ClockImp.cc"

int main ()
{
    Nand nand1; Analyzer analyzer(3);
    Fork fork1,fork2; Clock clock(10,high);
    Wire a,b,c,d,e,f,g; Terminal term(high);

    a.ConnectInput(term,0);   a.ConnectOutput(fork1,0);
    b.ConnectInput(clock,0);  b.ConnectOutput(fork2,0);
    c.ConnectInput(fork1,0);  c.ConnectOutput(nand1,0);
    d.ConnectInput(fork1,1);  d.ConnectOutput(analyzer,0);
    e.ConnectInput(fork2,0);  e.ConnectOutput(nand1,1);
    f.ConnectInput(fork2,1);  f.ConnectOutput(analyzer,1);
    g.ConnectInput(nand1,0);  g.ConnectOutput(analyzer,2);

    Sim.Simulate(33);
}
```

```
#include<iostream.h>

#include"DoubleLinkedList.cc"
#include"DoubleLinkedListIterator.cc"
#include"DoubleLinkedListElement.cc"
#include"DoubleLinkedListImp.cc"
#include"MinPriorityQueue.cc"
#include"MinPriorityQueueImp.cc"
#include"Queue.cc"
#include"QueueImp.cc"

class Simulator; // forward declaration
class Wire;      // forward declaration
class Circuit;   // forward declaration

#include"Wire.cc"
#include"Circuit.cc"
#include"Simulator.cc"
#include"SimulatorImp.cc"
#include"WireImp.cc"

#include"Nand.cc"
#include"NandImp.cc"
#include"And.cc"
#include"AndImp.cc"
#include"Nor.cc"
#include"NorImp.cc"
#include"Or.cc"
#include"OrImp.cc"
#include"Exor.cc"
#include"ExorImp.cc"
#include"Inverter.cc"
#include"InverterImp.cc"
#include"Fork.cc"
#include"ForkImp.cc"
#include"Terminal.cc"
#include"TerminalImp.cc"
#include"Analyzer.cc"
#include"AnalyzerImp.cc"
#include"Clock.cc"
#include"ClockImp.cc"
#include"HalfAdder.cc"
#include"HalfAdderImp.cc"
#include"FullAdder.cc"
#include"FullAdderImp.cc"
#include"Adder4Bit.cc"
#include"Adder4BitImp.cc"
```

```
int main ()
{
    Adder4Bit adder;
    Analyzer analyzer(5);
    Terminal a3(low ),a2(high),a1(high),a0(high);
    Terminal b3(high),b2(low ),b1(high),b0(low );
    Terminal c0(low);

    Wire wa0,wa1,wa2,wa3;
    Wire wb0,wb1,wb2,wb3;
    Wire ws0,ws1,ws2,ws3;
    Wire wc0,wc4;

    wc0.ConnectInput(c0,0);
    wc0.ConnectOutput(adder,8);

    wa0.ConnectInput(a0,0);
    wa1.ConnectInput(a1,0);
    wa2.ConnectInput(a2,0);
    wa3.ConnectInput(a3,0);
    wa0.ConnectOutput(adder,0);
    wa1.ConnectOutput(adder,1);
    wa2.ConnectOutput(adder,2);
    wa3.ConnectOutput(adder,3);

    wb0.ConnectInput(b0,0);
    wb1.ConnectInput(b1,0);
    wb2.ConnectInput(b2,0);
    wb3.ConnectInput(b3,0);
    wb0.ConnectOutput(adder,4);
    wb1.ConnectOutput(adder,5);
    wb2.ConnectOutput(adder,6);
    wb3.ConnectOutput(adder,7);

    ws0.ConnectInput(adder,0);
    ws1.ConnectInput(adder,1);
    ws2.ConnectInput(adder,2);
    ws3.ConnectInput(adder,3);
    ws0.ConnectOutput(analyzer,0);
    ws1.ConnectOutput(analyzer,1);
    ws2.ConnectOutput(analyzer,2);
    ws3.ConnectOutput(analyzer,3);

    wc4.ConnectInput(adder,4);
    wc4.ConnectOutput(analyzer,4);

    Sim.Simulate(40);
}
```

. . . und die Ausgabe:

```
 0  U  U  U  U  U
 1  U  U  U  U  U
 2  U  U  U  U  U
 3  U  U  U  U  U
 4  U  U  U  U  U
 5  U  U  U  U  U
 6  U  U  U  U  U
 7  U  U  U  U  U
 8  U  U  U  U  U
 9  U  U  U  U  U
10  U  U  U  U  U
11  U  U  U  U  U
12  U  U  U  U  U
13  U  U  U  U  1
14  U  U  U  U  1
15  U  U  U  U  1
16  U  U  U  U  1
17  U  U  U  U  1
18  U  U  U  U  1
19  U  U  U  0  1
20  U  U  U  0  1
21  U  U  U  0  1
22  U  U  U  0  1
23  U  U  U  0  1
24  U  U  U  0  1
25  U  U  0  0  1
26  U  U  0  0  1
27  U  U  0  0  1
28  U  U  0  0  1
29  U  U  0  0  1
30  U  U  0  0  1
31  1  0  0  0  1
32  1  0  0  0  1
33  1  0  0  0  1
34  1  0  0  0  1
35  1  0  0  0  1
36  1  0  0  0  1
37  1  0  0  0  1
38  1  0  0  0  1
39  1  0  0  0  1
40  1  0  0  0  1
```

Hier können Sie alle restlichen Programme herunterladen:

- `And.cc`, `AndImp.cc`
- `Nor.cc`, `NorImp.cc`
- `Or.cc`, `OrImp.cc`
- `Exor.cc`, `ExorImp.cc`
- `Inverter.cc`, `InverterImp.cc`
- `Terminal.cc`, `TerminalImp.cc`
- `Analyzer.cc`, `AnalyzerImp.cc`
- `Clock.cc`, `ClockImp.cc`
- `FullAdder.cc`, `FullAdderImp.cc`
- `Adder4Bit.cc`, `Adder4BitImp.cc`

20 Sortieren

Sortieren durch Auswahl: `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]);
    }
}
```

Bubble Sort für das parametrisierte Feld: `bubblesort2.cc`

```
template <class C>
void bubblesort (C& a)
{
    for (int i=a.size()-1; i>=0; i=i-1)
        for (int j=0; j<i; j=j+1)
            if (a[j+1]<a[j])       // operator< auf Feldelementen
                swap(a[j+1],a[j]); // generiert passende Version von swap!
}
```

Sortieren durch Einfügen `insertionsort.cc`

```
template <class C>
void insertionsort (C& a)
{
    int j;
    typename C::MemberType v;
    for (int i=1; i<a.size(); i=i+1) // i Elemente sind sortiert
    {
        j=i; v=a[i];
        while (j>0 && a[j-1]>v)
        {
            a[j] = a[j-1];
            j=j-1;
        }
        a[j] = v;
    }
}
```

```
template <class C>
void mergesort (C& a)
{
    // Rekursionsende
    if (a.size()<=1) return;

    // Teile Eingabefeld
    C a1(a.size()/2);
    C a2(a.size()-a1.size());
    for (int i=0; i<a1.size(); i=i+1) a1[i]=a[i];
    for (int i=0; i<a2.size(); i=i+1) a2[i]=a[i+a1.size()];

    // sortiere rekursiv
    mergesort(a1);
    mergesort(a2);

    // mischen
    int i1=0, i2=0;
    for (int k=0; k<a1.size()+a2.size(); k=k+1)
        if ( (i2>=a2.size()) || (i1<a1.size() && a1[i1]<=a2[i2]) )
        {
            a[k] = a1[i1]; i1 = i1+1;  // nehme aus Folge 1
        }
        else
        {
            a[k] = a2[i2]; i2 = i2+1;  // nehme aus Folge 2
        }
}
```

Sortieren durch mischen mit Speicheraufwand $2n$:

mergesort2.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);

    // mische
    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]; i1 = i1+1;  // nehme aus Folge 1
        }
        else
        {
            b[k] = a[i2]; i2 = i2+1;  // nehme aus Folge 2
        }

    // umkopieren
    for (int k=0; k<n; k=k+1) a[o+k] = b[k];
}

template <class C>
void mergesort2 (C& a)
{
    rec_merge_sort(a,0,a.size());
}
```

```
template <class C>
inline void heap_push (C& a, int n, typename C::MemberType x)
{
    int i=n; // Einfuegeposition

    a[i] = x;

    while (i>0 && a[i]>a[(i-1)/2])
    {
        swap(a[i],a[(i-1)/2]);
        i = (i-1)/2;
    }
}

template <class C>
inline void heap_reheap (C& a, int n, int i)
{
    // Im Feld a mit a.size()>=n bilden die ersten
    // n Elemente einen Heap.
    // Ab dem Knoten i wird reheap aufgerufen
    int l=2*i+1, r=l+1, k;
    while (l<n)    // mindestens ein Kind existiert
    {
        k = i;     // naechstes = i
        if (r<n) { // beide Kinder existieren
            if (a[l]>=a[r]) {
                if (a[l]>a[i]) {
                    swap(a[l],a[i]);
                    k = l;
                }
            }
            else { // rechtes ist groesser
                if (a[r]>a[i]) {
                    swap(a[r],a[i]);
                    k = r;
                }
            }
        }
        else // nur linkes Kind existiert, dies ist ein Blatt
            if (a[l]>a[i]) swap(a[l],a[i]);
        if (k==i) break;       // vorzeitiges Ende
        i = k; l=2*i+1; r=l+1; // naechster Knoten
    }
}

template <class C>
inline typename C::MemberType heap_pop (C& a, int n)
{
    typename C::MemberType t;
```

```
    t = a[0];              // das oberste Element
    a[0] = a[n-1];         // wird ersetzt durch das letzte
    heap_reheap(a,n-1,0); // neu organisieren
    return t;              // Rueckgabewert
}
```

Der generische Heapsort: heapsort.cc

```
template <class C>
void heapsort (C& a)
{
    // interpretiere Feld als unsortierten heap
    // rufe reheap levelweise von unten nach oben
    for (int i=a.size()-1; i>=0; i--)
        heap_reheap(a,a.size(),i);

    // vorderer Teil des Feldes: heap
    // hinterer Teil des Feldes: sortierte Folge
    for (int i=a.size()-1; i>0; i--)
        a[i] = heap_pop(a,i+1);
}
```

```
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);
}
```

Inhaltsverzeichnis

1 Einführung **2**

1.1 Formale Systeme: MIU 2

1.2 Turingmaschine 5

1.3 Problem, Algorithmus, Programm 9

1.4 Reale Computer 11

Von Neumann Rechner 11

Programmiersprachen 12

Warum gibt es verschiedene Programmiersprachen? 12

TEIL I FUNKTIONALE PROGRAMMIERUNG **14**

2 Auswertung von Ausdrücken **15**

2.1 Ausdrücke 15

2.2 Funktionen 16

2.3 Selektion 18

3 Syntaxbeschreibung mit Backus-Naur Form **19**

3.1 EBNF 19

3.2 Syntaxbeschreibung für FC++ 20

3.3 Kommentare 23

4 Das Substitutionsmodell **24**

5 Funktionen und Prozesse **26**

5.1 Lineare Rekursion und Iteration 26

5.2 Baumrekursion 29

Größenordnung 34

Wechselgeld 35

Der größte gemeinsame Teiler 37
5.3 Deklaration von Funktionen 39
5.4 Zahlendarstellung im Rechner 41
Wurzelberechnung mit dem Newtonverfahren 44

TEIL II PROZEDURALE PROGRAMMIERUNG 48

6 Kontrollfluß 49
6.1 Lokale Variablen und die Zuweisung 49
Was ist gleich? 52
Lokale Umgebung 52
6.2 Anweisungsfolgen (Sequenz) 53
Beispiel 53
6.3 Bedingte Anweisung (Selektion) 54
6.4 Schleifen 55
Beispiele 57

7 Benutzerdefinierte Datentypen 59
7.1 Aufzählungstyp 59
7.2 Felder 60
Sieb des Eratosthenes 61
7.3 Zeichen und Zeichenketten 62
7.4 Typedef 64
7.5 Das Acht-Damen-Problem 65
7.6 Zusammengesetzte Datentypen 68

8 Globale Variablen und das Umgebungsmodell 73
8.1 Globale Variablen 73
8.2 Das Umgebungsmodell 75

8.3 Beispiel: Monte-Carlo Methode zur Bestimmung von π 80

9 Zeiger und dynamische Datenstrukturen 84

9.1 Zeiger 84

9.2 Zeiger und die Bindungstabelle 85

9.3 Call by reference 88

Referenzen in C++ 89

9.4 Zeiger und Felder 90

9.5 Zeiger und zusammengesetzte Datentypen 90

9.6 Dynamische Speicherverwaltung 91

9.7 Die einfach verkettete Liste 93

9.8 Endliche Menge 97

10 Beispiel: Huffman Kodes 100

TEIL III OBJEKTORIENTIERTE PROGRAMMIERUNG 108

11 Klassen 109

11.1 Klassen aus Benutzersicht 109

11.2 Implementierung von Klassen 114

11.3 Beispiel: Monte-Carlo objektorientiert 117

11.4 Überladen und Operatoren 120

11.5 Abstrakter Datentyp 131

12 Klassen und dynamische Speicherverwaltung 135

12.1 Nachteile eingebauter Felder 135

12.2 Klassendefinition 136

12.3 Konstruktoren, Destruktoren und Indexmenge 137

12.4 Indizierter Zugriff 138

12.5 Copy–Konstruktor 138
12.6 Zuweisungsoperator 140
12.7 Default-Methoden 142

13 Vererbung von Schnittstelle und Implementierung 143

13.1 Motivation: Polynome 143
13.2 Öffentliche Vererbung 144
13.3 Ist-ein-Beziehung 147
13.4 Konstruktoren, Destruktor und Zuweisungsoperatoren 149
13.5 Weitere Methoden von `Polynomial` 150
13.6 Gleichheit 151

14 Vererbung der Implementierung 154

14.1 Motivation: Polynominterpolation 154
14.2 Private Vererbung 155
14.3 Konstruktion des Interpolationspolynoms 156
14.4 Beispiel für `PolynomialFit` 158

15 Methodenauswahl und virtuelle Funktionen 159

15.1 Motivation: Feld mit Bereichsprüfung 159
15.2 Virtuelle Funktionen 162

16 Abstrakte Klassen 165

16.1 Motivation: Integration 166
16.2 Schnittstellenbasisklassen 168
16.3 Beispiel: Exotische Felder 168
16.4 Virtueller Destruktor 180
16.5 Beispiel: Symbolisches Differenzieren 180

17 Generische Programmierung **188**

Klassenschablonen 188

Funktionenschablonen 195

Generische Programmierung und Effizienz 198

Zusammenfassung 203

18 Containerklassen **204**

Iteratoren 205

Feld 207

Doppelt verkettete Liste 209

Binärer Baum 214

Set 217

Stack 218

Queue 219

DeQueue 220

MinPriorityQueue/MaxPriorityQueue 221

Map 224

Huffmanbäume mit Containern 226

19 Beispiel: Logiksimulator **229**

Simulation komplexer Systeme 229

Grundbausteine digitaler Schaltungen 230

Reale Gatter 231

Schaltnetze 232

Schaltwerke 233

Der Simulator 234

20 Sortieren **253**

Liste der Programme und Klassen

`erstes.cc`, siehe Seite 15

`quadrat.cc`, siehe Seite 17

`absolut.cc`, siehe Seite 18

`fakultaet.cc`, siehe Seite 27

`fakultaetiter.cc`, siehe Seite 28

`fibonacci.cc`, siehe Seite 29

`fibiter.cc`, siehe Seite 32

`wechselgeld.cc`, siehe Seite 36

`ggT.cc`, siehe Seite 38

`newton.cc`, siehe Seite 45

`fakwhile.cc`, siehe Seite 55

`fibfor.cc`, siehe Seite 57

`newtonwhile.cc`, siehe Seite 57

`enum.cc`, siehe Seite 59

`eratosthenes.cc`, siehe Seite 61

`ASCII.cc`, siehe Seite 62

`Cstring.cc`, siehe Seite 63

`CCstring.cc`, siehe Seite 63

`achtdamen.cc`, siehe Seite 66

`Rational1.cc`, siehe Seite 69

`Mixed1.cc`, siehe Seite 71

`konto.cc`, siehe Seite 73

`wg-stack.cc`, siehe Seite 78

`montecarlo1.cc`, siehe Seite 81

`montecarlo2.cc`, siehe Seite 82

intlist.cc, siehe Seite 93

intset.cc, siehe Seite 97

huffman-ds.cc, siehe Seite 102

huffman-menge.cc, siehe Seite 102

huffman-code.cc, siehe Seite 105

huffman-baum.cc, siehe Seite 104

huffman-kodieren.cc, siehe Seite 106

huffman-main.cc, siehe Seite 106

Zufall.cc, siehe Seite 117

Experiment.cc, siehe Seite 118

MonteCarlo.cc, siehe Seite 118

Rational.cc, siehe Seite 124

SimpleFloatArray.cc, siehe Seite 136

UseSimpleFloatArray.cc, siehe Seite 141

Polynomial.cc, siehe Seite 144

PolynomialFit.cc, siehe Seite 155

UsePolynomialFit.cc, siehe Seite 158

CheckedSimpleFloatArray.cc, siehe Seite 160

SimpleFloatArrayV.cc, siehe Seite 162

CheckedSimpleFloatArrayV.cc, siehe Seite 163

DynamicFloatArray.cc, siehe Seite 169

ListFloatArray.cc, siehe Seite 173

FloatArray.cc, siehe Seite 176

SimpleArray.cc, siehe Seite 190

SimpleArrayCS.cc, siehe Seite 194

Usebubblesort.cc, siehe Seite 197

Array.cc, siehe Seite 207

DoubleLinkedList.cc, siehe Seite 210

BinaryTree.cc, siehe Seite 215

Set.cc, siehe Seite 217

Stack.cc, siehe Seite 218

Queue.cc, siehe Seite 219

DeQueue.cc, siehe Seite 220

MinPriorityQueue.cc, siehe Seite 221

MaxPriorityQueue.cc, siehe Seite 221

Map.cc, siehe Seite 225

Huffman.cc, siehe Seite 227

Wire.cc, siehe Seite 236

WireImp.cc, siehe Seite 237

Circuit.cc, siehe Seite 238

Nand.cc, siehe Seite 239

NandImp.cc, siehe Seite 240

Fork.cc, siehe Seite 241

ForkImp.cc, siehe Seite 242

HalfAdder.cc, siehe Seite 243

HalfAdderImp.cc, siehe Seite 244

Simulator.cc, siehe Seite 245

SimulatorImp.cc, siehe Seite 246

And.cc, siehe Seite 252

AndImp.cc, siehe Seite 252

Nor.cc, siehe Seite 252

NorImp.cc, siehe Seite 252

Or.cc, siehe Seite 252

OrImp.cc, siehe Seite 252

`Exor.cc`, siehe Seite 252

`ExorImp.cc`, siehe Seite 252

`Inverter.cc`, siehe Seite 252

`InverterImp.cc`, siehe Seite 252

`Terminal.cc`, siehe Seite 252

`TerminalImp.cc`, siehe Seite 252

`Analyzer.cc`, siehe Seite 252

`AnalyzerImp.cc`, siehe Seite 252

`Clock.cc`, siehe Seite 252

`ClockImp.cc`, siehe Seite 252

`FullAdder.cc`, siehe Seite 252

`FullAdderImp.cc`, siehe Seite 252

`Adder4Bit.cc`, siehe Seite 252

`Adder4BitImp.cc`, siehe Seite 252

Liste der Feldklassen

In der Vorlesung werden eine Reihe von Klassen eingeführt, die auf der Abstraktion eines Feldes aufbauen. Es folgt eine Liste dieser Klassen mit Kurzerklärung und ihrer gegenseitigen Beziehung.

Klasse	Kurzerklärung
`SimpleFloatArray`	Siehe Seite 136. Feste Indexmenge, ungeprüfter Indexzugriff, enthält `float` Elemente.
`Polynomial`	Siehe Seite 144. Öffentlich abgeleitet von `SimpleFloatArray`, zusätzlich Auswertung, Addition, Multiplikation
`PolynomialFit`	Siehe Seite 155. Klasse ist privat abgeleitet von `Polynomial`, nutzt dieses zur Interpolation von Datenpunkten.
`CheckedSimpleFloatArray`	Definition auf Seite 160. Ist öffentlich abgeleitet von `SimpleFloatArray`, die Methode `operator[]` wird ersetzt.
`SimpleFloatArrayV`	Siehe Seite 162. Variante mit virtueller Methode `operator[]`
`CheckedSimpleFloatArrayV`	Siehe Seite 163. Variante mit virtueller Methode `operator[]`
`DynamicFloatArray`	Seite 169. Feld mit variabler Indexmenge. Indexmenge ist zusammenhängend und wird bei Zugriff entsprechend vergrößert.
`ListFloatArray`	Siehe Seite 173. Feld für dünnbesetzte Indexmengen, bei denen Index-Wert-Paare in einer Liste gespeichert werden
`FloatArray`	Siehe Seite 176. Schnittstellenbasisklasse für die Felddatentypen
`SimpleArray<T>`	Siehe Seite 190. Klassenschablone mit selber Schnittstelle wie alle Feldklassen oben, Elementtyp als Schablonenparameter.

Liste der Konstruktionselemente

Ausdruck: Seite 21

Funktionsdefinition: Seite 21

Einfaches C++ Programm: Seite 22

Cond Funktion: Seite 22

Variablendefinition: Seite 51

Zuweisung: Seite 51

Bedingte Anweisung: Seite 54

Anweisungsfolge: Seite 53

While-Schleife: Seite 56

For-Schleife: Seite 56

Aufzählungstyp: Seite 60

Felddefinition: Seite 60

Zusammengesetzter Datentyp: Seite 68

Variante Struktur: Seite 71

Klasse: Seite 110

Konstruktor: Seite 119

Öffentliche Vererbung: Seite 146

Literatur

[ASS98] H. Abelson, G. J. Sussman, and J. Sussman. *Struktur und Interpretation von Computerprogrammen.* Springer, 1998.

[BN94] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++.* Addison-Wesley, 1994.

[Bud91] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[Gol91] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, March 1991.

[Heu00] V. Heun. *Grundlegende Algorithmen.* Vieweg, 2000.

[Hof79] D. R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid.* Basic Books, New York, 1979.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A quantitative approach.* Morgan Kaufmann, 1996.

[HSM95] E. Horowitz, S. Sahni, and D. Mehta. *Fundamentals of Data Structures in C++.* W. H. Freeman and Company, 1995.

[HU00] J. E. Hopcraft and J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Oldenbourg, München, 2000. 4. Auflage.

[Knu98] D. E. Knuth. *The Art of Computer Programming.* Addison–Wesley, 1998.

[Sed92] R. Sedgewick. *Algorithmen in C++.* Addison–Wesley, 1992.

[Str97] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[Wir83] N. Wirth. *Algorithmen und Datenstrukturen.* Teubner, 1983.