Einführung in die Praktische Informatik Wintersemester 2022/2023,

Durchgeführt von *Christian Schulz*
Institut für Informatik
Universität Heidelberg
Im Neuenheimer Feld 205, 69120 Heidelberg
christian.schulz@informatik.uni-heidelberg.de

Hauptautor: *Peter Bastian*
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

basierend auf den Beiträgen von

Filip Sadlo
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

und

Nicolas Neuß
Universität Erlangen-Nürnberg

Version 2.2

Erstellt: 6. Oktober 2022

URL zur Vorlesung (enthält u.a. die Beispielprogramme):
https://ae.ifi.uni-heidelberg.de/IPI_WS23.html/

Inhaltsverzeichnis

1 Grundbegriffe

Diese Vorlesung: Ein kleiner Ausflug in die Theoretische Informatik.

Diese untersucht (neben anderen Dingen) die Frage ,,Was ist berechenbar"?

Dafür ist eine Formalisierung des Rechenbegriffes notwendig.

Inhalt:

- Textersetzungssysteme
- Graphen und Bäume
- Turingmaschine

1.1 Formale Systeme: MIU

Im folgenden betrachten wir Zeichenketten über einem Alphabet.

Ein Alphabet $\mathcal{A}$ ist eine endliche, nichtleere Menge (manchmal verlangt man zusätzlich, dass die Menge geordnet ist). Die Elemente von $\mathcal{A}$ nennen wir Zeichen (oder Symbole).

Eine endliche Folge nicht notwendigerweise verschiedener Zeichen aus $\mathcal{A}$ nennt man ein Wort. Das leere Wort ϵ besteht aus keinem einzigen Zeichen. Es ist ein Symbol für „Nichts".

Die Menge aller möglichen Wörter inklusive dem leeren Wort wird als freies Monoid $\mathcal{A}^*$ bezeichnet.

Beispiel: $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

Formale Systeme dienen der Beschreibung interessanter Teilmengen von $\mathcal{A}^*$ (auch Sprachen genannt) .

Definition: Ein formales System (oder Semi-Thue-System, oder Wortersetzungssystem) ist ein System von Wörtern und Regeln. Die Regeln sind Vorschriften für die Umwandlung eines Wortes in ein anderes.

Mathematisch: $F = (\mathcal{A}, \mathcal{B}, \mathcal{X}, \mathcal{R})$, wobei

- $\mathcal{A}$ das Alphabet,
- $\mathcal{B} \subseteq \mathcal{A}^*$ die Menge der wohlgebildeten Worte,
- $\mathcal{X} \subset \mathcal{B}$ die Menge der Axiome und
- $\mathcal{R}$ die Menge der Produktionsregeln

sind. Ausgehend von $\mathcal{X}$ werden durch Anwendung von Regeln aus $\mathcal{R}$ alle wohlgeformten Wörter $\mathcal{B}$ erzeugt.

Formale Systeme entstanden Anfang des 20. Jahrhunderts im Rahmen der Formalisierung der Mathematik. Ziel war es ein System zu schaffen mit dem alle mathematischen Sätze (wahre Aussagen über einen mathematischen Sachverhalt, möglicherweise in Teilgebieten der Mathematik) aus einem kleinen Satz von Axiomen mittels Regeln hergeleitet werden können (Hilbertprogramm[1]).

[1] David Hilbert, dt. Mathematiker, 1862–1943.

Ein Traum, der sich letztlich aufgrund der Arbeiten von Kurt Gödel[2] als undurchführbar erwiesen hat.

Formale Sprachen erweitern das Konzept der formalen Systeme durch Unterscheidung von Terminalsymbolen und Nichtterminalsymbolen.

Beispiel: MIU-System (aus [Hofstadter[3], 2007])

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

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

$$\mathcal{B} = \{\text{MI}, \text{MIU}, \text{MIUUI}, \ldots\}.$$

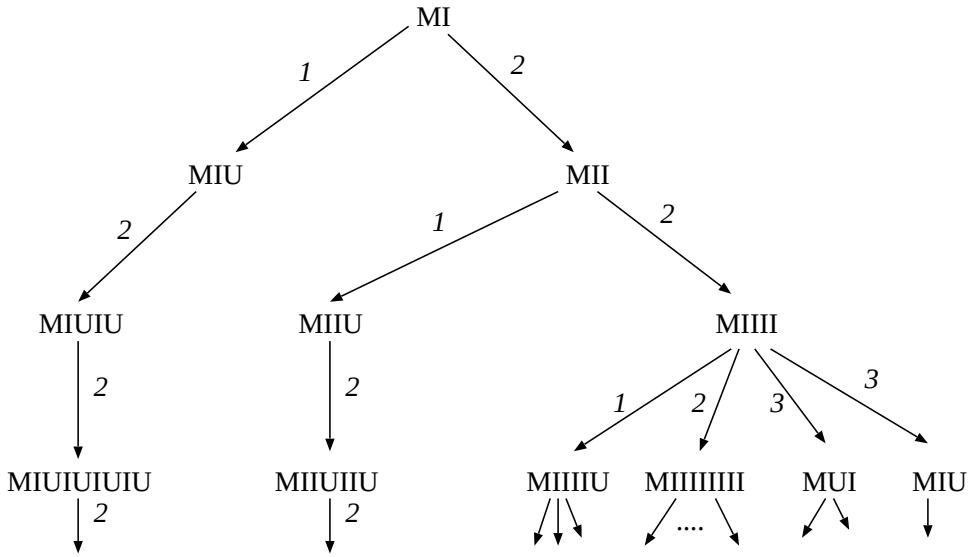
Beobachtung: B_{MIU} enthält offenbar unendlich viele Worte.

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

Systematische Erzeugung aller Worte des MIU-Systems

Dies führt auf folgende Baumstruktur:

[2]Kurt Gödel, österreichisch/amerikanischer Mathematiker und Logiker, 1906–1978.
[3]Douglas R. Hofstadter, US-amerk. Physiker, Informatiker und Kognitionswissenschaftler, geb. 1945.



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

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

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

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

Frage: Wie löst man nun das MU-Rätsel?

Lösung des MU-Rätsels

Zur Lösung muss man Eigenschaften der Wörter in $\mathcal{B}_{\mathrm{MIU}}$ analysieren.

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

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

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

Beweis: Ersieht man leicht aus den Regeln, sei anzahli(n) die Anzahl der I nach Anwendung von n Regeln, $n \in \mathbb{N}_0$. Dann gilt:

$$\text{anzahli}(n) = \begin{cases} 1 & n = 0, \text{Axiom}, \\ \text{anzahli}(n-1) & n > 0, \text{Regel 1, 4}, \\ \text{anzahli}(n-1) \cdot 2 & n > 0, \text{Regel 2}, \\ \text{anzahli}(n-1) - 3 & n > 0, \text{Regel 3} \end{cases}$$

Ist anzahli$(n-1) \bmod 3 \neq 0$, so gilt dies auch nach Anwendung einer beliebigen Regel.

Von Graphen und Bäumen

Der Baum ist eine sehr wichtige Struktur in der Informatik und ein Spezialfall eines Graphen.

Definition: Ein Graph $G = (V, E)$ besteht aus

- einer nichtleeren Menge V, der sogenannten Menge der Knoten, sowie
- der Menge der Kanten $E \subseteq V \times V$.

$V \times V = \{(v, w) \ : \ v, w \in V\}$ bezeichnet das kartesische Produkt.

Teilmengen von $V \times V$ bezeichnet man auch als Relationen.

Beispiel: Gleichheit als Relation. Sei V eine Menge (dies impliziert, dass alle Elemente verschieden sind). Setze

$$E_= = \{(v, w) \in V \times V \ : \ v = w\}.$$

Dann gilt $v = w \Leftrightarrow (v, w) \in E_=$.

Wichtige Spezialfälle von Graphen sind:

- Ungerichter Graph: $(v, w) \in E \Rightarrow (w, v) \in E$. Sonst heisst der Graph gerichtet.
- Verbundener Graph: Ein ungerichteter Graph heisst verbunden, falls jeder Knoten von jedem anderen Knoten über eine Folge von Kanten erreichbar ist. Bei einem gerichteten Graphen ergänze erst alle Kanten der Gegenrichtung und wende dann die Definition an.
- Zyklischer Graph: Es gibt, ausgehend von einem Knoten, eine Folge von Kanten mit der man wieder beim Ausgangsknoten landet.
- Gerichteter, azyklischer Graph (directed acyclic graph, DAG): Gerichteter Graph bei dem von keinem Knoten eine Folge von Knoten wieder zum Ausgangsknoten führt.

Definition: Wir definieren die Menge der Bäume rekursiv über die Anzahl der Knoten als Teilmenge aller möglicher Graphen. (Beachte die Ähnlichkeit zu formalen Systemen).

- $(\{v\}, \emptyset)$ ist ein Baum.

- Sei $B = (V, E)$ ein Baum, so ist $B' = (V', E')$ ebenfalls ein Baum, wenn

$$V' = V \cup \{v\}, \qquad v \notin V, \qquad E' = E \cup \{(w, v) \,:\, w \in V\}.$$

  Man hängt also einen neuen Knoten an genau einen Knoten des existierenden Baumes an. v heisst Kind und w wollen wir geschlechtsneutral als Elter von v bezeichnen.

Bemerkung: Auch andere Definitionen sind möglich, etwa als (ungerichteter), zyklenfreier, verbundener Graph.

Bezeichnung:

- Jeder Baum besitzt genau einen Knoten, der keine eingehenden Kanten hat. Dieser heisst Wurzel. (Dies ist der Knoten v aus der ersten Regel ($\{v\}$, $\emptyset$) oben).
- Knoten ohne ausgehende Kanten heissen Blätter, alle anderen Knoten heissen innere Knoten
- Ein Baum bei dem jeder innere Knoten höchstens zwei Kinder hat heisst Binärbaum.

Beobachtung: Ein Baum ist verbunden. Es gibt genau einen Weg von der Wurzel zu jedem Blatt, bzw. von jedem Knoten zu jedem anderen Knoten.

1.2 Turingmaschine

Als weiteres Beispiel für ein „Regelsystem“ betrachten wir die Turingmaschine (TM).

Diese wurde 1936 von Alan Turing[4] zum theoretischen Studium der Berechenbarkeit eingeführt.

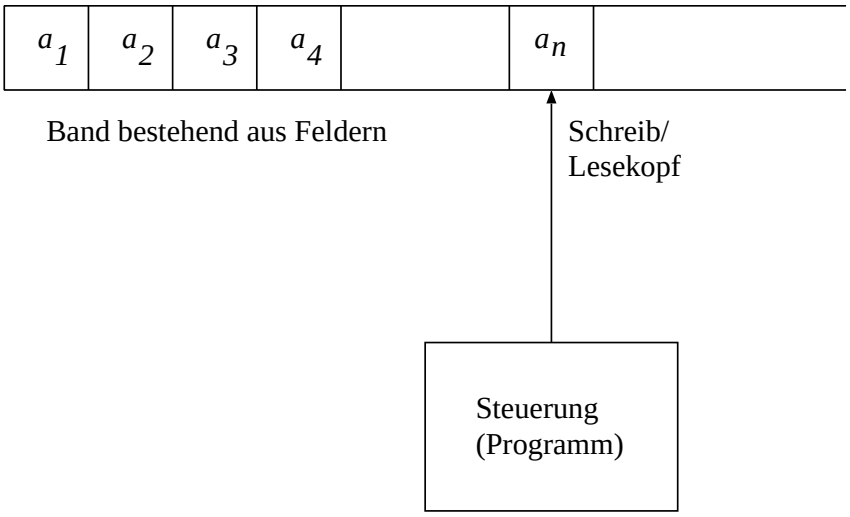
Wissen: Der sogenannte Turing-Preis (Turing Award) ist so etwas wie der „Nobelpreis der Informatik“.

Eine TM besteht aus einem festen Teil („Hardware“) und einem variablen Teil („Software“). TM bezeichnet somit nicht eine Maschine, die genau eine Sache tut, sondern ist ein allgemeines Konzept, welches eine ganze Menge von verschiedenen Maschinen definiert. Alle Maschinen sind aber nach einem festen Schema aufgebaut.

Die **Hardware** besteht aus einem einseitig unendlich langen 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.

Diese Beschreibung suggeriert, dass eine TM als eine Art primitiver Computer verstanden werden kann. Dies war aber nicht die Absicht von Alan Turing. Er verstand diese als Gedankenmodell um die Berechenbarkeit von Funktionen zu studieren.

[4] Alan Turing, brit. Mathematiker, 1912–1954.



Die Steuerung, der variable Teil der Maschine, befindet sich in einem von endlich vielen Zuständen und arbeitet wie folgt:

1. Am Anfang befindet sich die Maschine im sog. Startzustand, das Band ist mit einer Eingabe belegt und die Position des Schreib-/Lesekopfes ist festgelegt.
2. Lese das Zeichen unter dem Lesekopf vom Band.
3. Abhängig vom gelesenen Zeichen und dem aktuellen Zustand der Steuerung führe **alle** folgende Aktionen aus:
   - Schreibe ein Zeichen auf das Band,
   - bewege den Schreib-/Lesekopf um ein Feld nach links oder rechts,
   - überführe die Steuerung in einen neuen Zustand.
4. Wiederhole diese Schritte solange bis ein spezieller Endzustand erreicht wird.

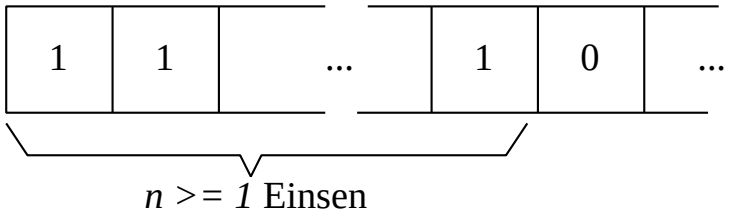
Die auszuführenden Aktionen kann man in einer Übergangstabelle notieren. Diese Tabelle nennt man auch Programm.

Beispiel:

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

Jede Zeile der Tabelle beschreibt die auszuführenden Aktionen für eine Eingabe/Zustand-Kombination. Links vom Doppelbalken stehen Eingabe und Zustand, rechts davon Ausgabe, Bewegungsrichtung und Folgezustand.

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



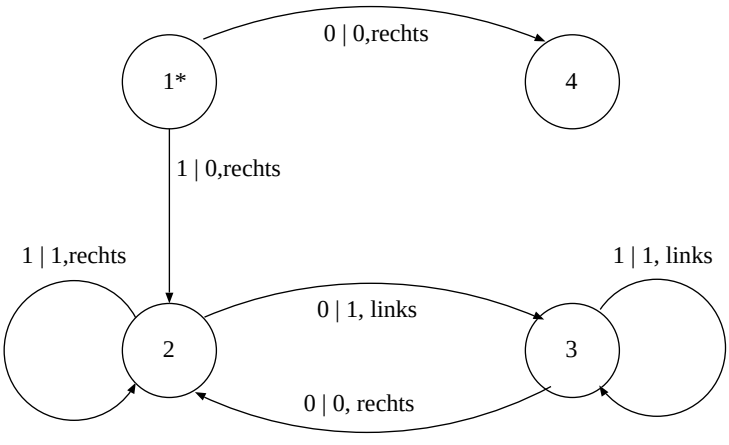
Der Kopf steht zu Beginn auf der Eins ganz links. Folgendes Programm mit zwei Zuständen löscht die Einserkette und stoppt:

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

Beispiel: Raten Sie was folgendes Programm macht:

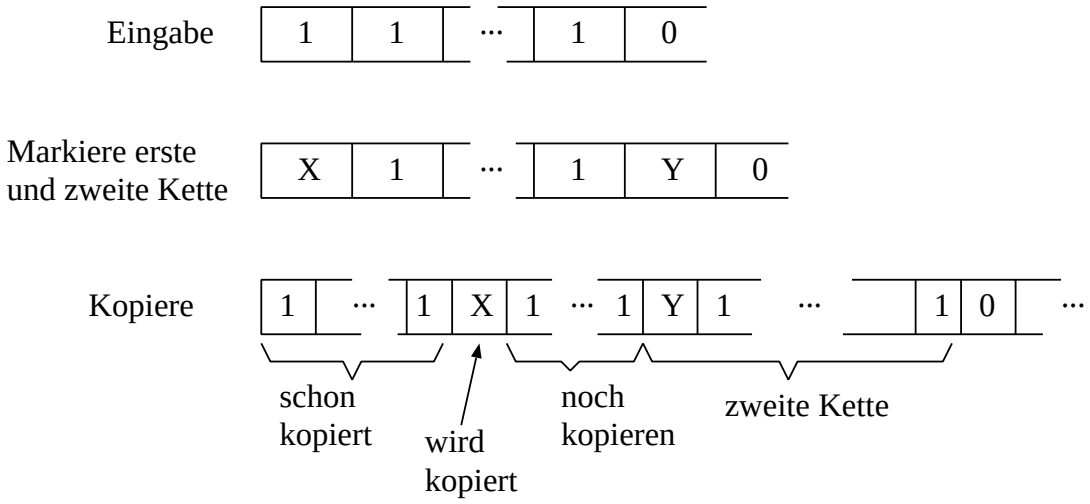
Zustand	Eingabe	Operation	Folgezustand	Bemerkung
1	1	0,rechts	2	Anfangszustand
	0	0,rechts	4	
2	1	1,rechts	2	
	0	1,links	3	
3	1	1,links	3	
	0	0,rechts	2	
4				Endzustand

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

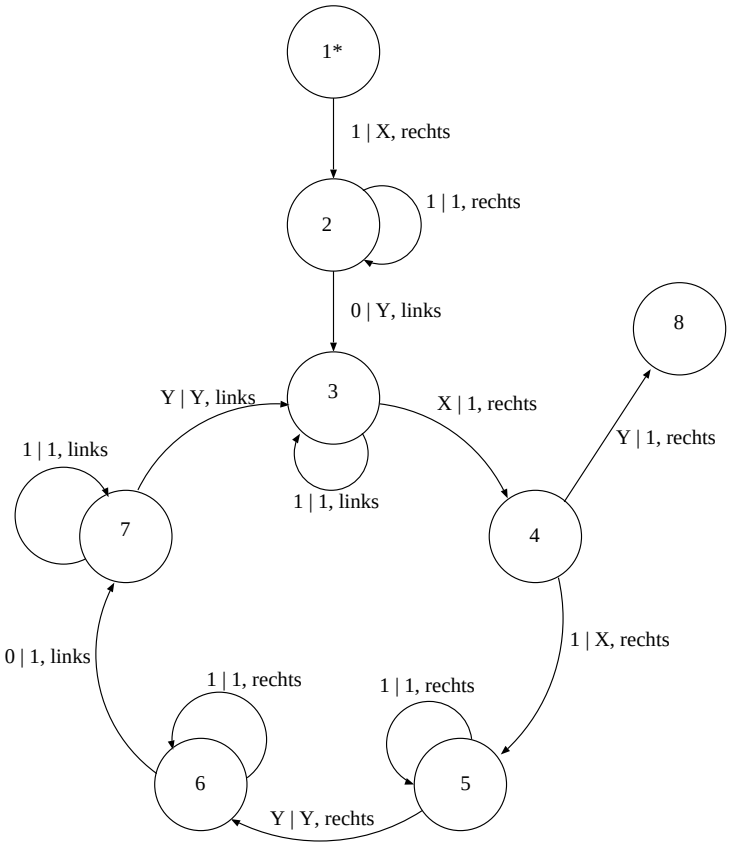


Beispiel: Verdoppeln einer Einserkette. Eingabe: n Einsen wie in Beispiel 1. Am Ende der Berechnung sollen ganz links $2n$ Einsen stehen, sonst nur Nullen.

Wie löst man das mit einer TM? Hier eine Idee:



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



Bemerkung: Wir erkennen die drei wesentlichen Komponenten von Berechnungsprozessen:

- Grundoperationen
- Selektion
- Wiederholung

1.3 Problem, Algorithmus, Programm

Definition: Ein Problem ist eine zu lösende Aufgabe. Wir sind daran interessiert Verfahren zu finden, die Aufgaben in einer Klasse von Problemen zu lösen. Das konkrete zu lösende Problem wird mittels Eingabeparameter ausgewählt.

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

Definition: Ein Algorithmus beschreibt, wie ein Problem einer Problemklasse mittels einer Abfolge bekannter Einzelschritte gelöst werden kann. Beispiele aus dem Alltag, wie Kochrezepte oder Aufbauanleitungen für Abholmöbel erinnern an Algorithmen, sind aber oft nicht allgemein und unpräzise.

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

Ein Algorithmus muss gewisse Eigenschaften erfüllen:

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

Bemerkung: Spezielle Algorithmen sind:

- Terminierende Algorithmen: Der Algorithmus stoppt für jede zulässige Eingabe nach endlicher Zeit.
- Deterministische Algorithmen: In jedem Schritt ist bekannt, welcher Schritt als nächstes ausgeführt wird.
- Determinierte Algorithmen: Algorithmus liefert bei gleicher Eingabe stets das gleiche Ergebnis. Ein terminierender, deterministischer Algorithmus ist immer determiniert. Terminierende, nichtdeterministische Algorithmen können determiniert sein oder nicht.

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

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

Wir haben also das Schema: Problem $\Longrightarrow$ Algorithmus $\Longrightarrow$ Programm.

Die Informatik beschäftigt sich damit algorithmische Problemlösungen systematisch zu finden:

- Zunächst muss das Problem analysiert und möglichst präzise formuliert werden. Dieser Schritt wird auch als Modellierung bezeichnet.

- Im folgenden entwirft man einen effizienten Algorithmus zur Lösung des Problems. Dieser Schritt ist von zentralem Interesse für die Informatik.
- Schließlich muss der Algorithmus als Computerprogramm formuliert werden, welches auf einer konkreten Maschine ausgeführt werden kann.

1.4 Berechenbarkeit und Turing-Äquivalenz

Es sei $\mathcal{A}$ das Bandalphabet einer TM. Wir können uns die Berechnung einer konkreten TM (d. h. gegebenes Programm) auch als Abbildung vorstellen:

$$f : \mathcal{A}^* \to \mathcal{A}^*.$$

Hält die TM für einen Eingabewert nicht an, so sei der Wert von f undefiniert.

Dies motiviert folgende allgemeine

Definition: Eine Funktion $f : E \to A$ heisst berechenbar, wenn es einen Algorithmus gibt, der für jede Eingabe $e \in E$, für die $f(e)$ definiert ist, terminiert und das Ergebnis $f(e) \in A$ liefert.

Welche Funktionen sind in diesem Sinne berechenbar?

Auf einem PC mit unendlich viel Speicher könnte man mit Leichtigkeit eine TM simulieren. Das bedeutet, dass man zu jeder TM ein äquivalentes PC-Programm erzeugen kann, welches das Verhalten der TM Schritt für Schritt nachvollzieht. Ein PC (mit unendlich viel Speicher) kann daher alles berechnen, was eine TM berechnen kann.

Interessanter ist aber, dass man zeigen kann, dass die TM trotz ihrer Einfachheit alle Berechnungen durchführen kann, zu denen der PC in der Lage ist. Zu einem PC mit gegebenem Programm kann man also eine TM angeben, die die Berechnung des PCs nachvollzieht! Computer und TM können dieselbe Klasse von Problemen berechnen!

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

Die Church'sche[5] These lautet daher:

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

Dabei heißt intuitiv berechenbar, dass man einen Algorithmus dafür angeben kann.

Mehr dazu in Theoretische Informatik.

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

[5] Alonzo Curch, US-amerikanischer Mathematiker, Logiker und Philosoph, 1903–1995

Bemerkung: Es gibt auch nicht berechenbare Probleme! So kann man z. B. keine TM angeben, die für jede gegebene TM entscheidet, ob diese den Endzustand erreicht oder nicht (Halteproblem).

Dieses Problem ist aber noch partiell-berechenbar, d. h. für jede terminierende TM erfährt man dies nach endlicher Zeit, für jede nicht-terminierende TM erfährt man aber kein Ergebnis.

1.5 Reale Computer

Algorithmen waren schon vor der Entwicklung unserer heutigen Computer bekannt, allerdings haperte es mit der Ausführung. Zunächst arbeiteten Menschen als „Computer“!

- Lewis Fry Richardson[6] schlägt in seinem Buch *Weather Prediction by Arithmetical Finite Differences* vor, das Wetter für den nächsten Tag mit 64000 (!) menschlichen Computern auszurechnen. Der Vorschlag wird als unpraktikabel verworfen.
- In Los Alamos werden Lochkartenmaschinen und menschliche Rechner für Berechnungen eingesetzt. Richard Feynman[7] organisierte sogar einen Wettbewerb zwischen beiden.

Der Startpunkt der Entwicklung realer Computer stimmt (zufällig?) relativ genau mit der Entwicklung theoretischer Berechenbarkeitskonzepte durch Church und Turing überein.

Dabei verstehen wir Computer bzw. (Universal-)Rechner als Maschinen zur Ausführung beliebiger Algorithmen in obigem Sinne (d. h. sie können nicht „nur“ rechnen im Sinne arithmetischer Operationen).

Einige der wichtigsten frühen Rechenmaschinen waren:

- Zuse Z3, Mai 1941, mechanisch, turing-vollständig (aber nicht als solcher konstruiert), binäre Gleitkommaarithmetik
- Atanasoff-Berry-Computer, Sommer 1941, elektronisch (Röhren), nicht turing-mächtig, gebaut zur Lösung linearer Gleichungssysteme (29×29)
- Colossus, 1943, elektronisch, nicht turing-mächtig, Kryptographie
- Mark 1, 1944, mechanisch, turing-vollständig, Ballisitik
- ENIAC, 1946, elektronisch, turing-vollständig, Ballistik
- EDVAC, 1949, elektronisch, turing-vollständig, Ballistik, erste „Von-Neumann-Architektur“

Praktische Computer basieren meist auf dem von John von Neumann 1945 im Rahmen der EDVAC-Entwicklung eingeführten Konzept. Es ist umstritten welche der Ideen tatsächlich genau von ihm sind.

Geschichte: John von Neumann[8] war einer der bedeutendsten Mathematiker. Von ihm stammt die Spieltheorie, die mathematische Begründung der Quantenmechanik, sowie wichtige Beiträge zu Informatik und Numerik.

[6] Lewis Fry Richardson, brit. Meteorologe, 1881–1953.
[7] Richard P. Feynman, US-amerik. Physiker, Nobelpreis 1965, 1918–1988.
[8] János Neumann Margittai, Mathematiker österreichisch-ungarischer Herkunft, 1903–1957.

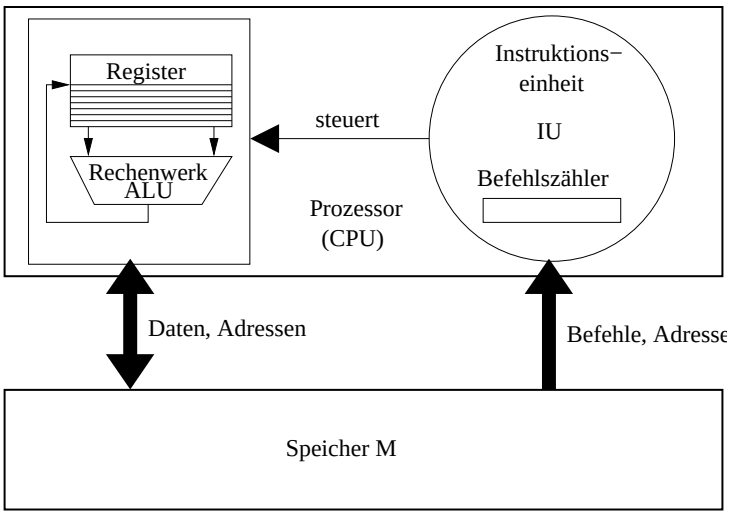


Abbildung 1: Grobe Struktur der von-Neumann-Architektur.

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).

Zum Zugriff auf den Speicher wird ein Index, auch Adresse genannt, verwendet, d. h. wir können den Speicher als Abbildung

$$M : A \to D$$

auffassen.

Für die Adressen gilt $A = [0, N-1] \subset \mathbb{N}_0$ wobei aufgrund der binären Organisation $N = 2^n$ gilt. n ist die Anzahl der erforderlichen Adressleitungen.

Für D gilt $D = [0, 2^m - 1]$ mit der Wortbreite m, die meistens ein Vielfaches von 8 ist. m ist die Anzahl der erforderlichen Datenleitungen.

Die Gesamtkapazität des Speichers ist demnach $m \cdot 2^n$ Bit. Jedes Bit kann zwei Werte annehmen, 0 oder 1. In der Praxis wird die Größe des Speichers in Byte angegeben, wobei ein Byte aus 8 Bit besteht. Damit enthält ein Speicher mit n Adressleitungen bei Wortbreite m genau $(m/8) \cdot 2^n$ Byte.

Gebräuchlich sind auch noch die Abkürzungen 1 Kilobyte = 2^{10} Byte = 1024 Byte, 1 Megabyte = 2^{20} Byte, 1 Gigabyte = 2^{30} Byte.

Der Speicher enthält sowohl Daten (das Band in der TM) als auch Programm (die Tabelle in der TM). Den einzelnen Zeilen der Programmtabelle der TM entsprechen beim von Neumannschen Rechner die Befehle. Die Vereinigung von Daten und Programm im Speicher (stored program computer) war der wesentliche Unterschied zu den früheren Ansätzen.

Befehle werden von der Instruktionseinheit (instruction unit, IU) gelesen und dekodiert.

Die Instruktionseinheit steuert das Rechenwerk, welches noch zusätzliche Daten aus dem Speicher liest bzw. Ergebnisse zurückschreibt.

Die Maschine arbeitet zyklisch die folgenden Aktionen ab:

- Befehl holen
- Befehl dekodieren
- Befehl ausführen

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

Bemerkung: Hier wurde insbesondere die Interaktion von Rechnern mit der Umwelt, die sog. Ein- und Ausgabe, in der Betrachtung vernachlässigt. Moderne Rechner haben insbesondere die Fähigkeit, auf äußere Einwirkungen hin (etwa Tastendruck) den Programmfluss zu unterbrechen und an anderer Stelle (Turingmaschine: in anderem Zustand) wieder aufzunehmen. Von Neumann hat die Ein-/Ausgabe im Design des EDVAC schon ausführlich beschrieben.

Bemerkung: Heutige Rechner sind wesentlich komplizierter als dieses einfache Modell. Insbesondere sind viele Möglichkeiten der **parallelen** Verarbeitung enthalten. Wichtige Konzepte in modernen Rechnern sind:

- Hierarchisch organisierter Speicher mit Caches
- Pipelining des Befehlsholzyklus
- SIMD Instruktionen, Superskalarität
- Multicorerechner

1.6 Programmiersprachen

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

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

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

Es benötigt weniger als 1KB an Speicher!

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

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

erstellen kann.

Wir lernen in dieser Vorlesung die Sprache C++. C++ ist eine Weiterentwicklung der Sprache C, die Ende der 1960er Jahre entwickelt wurde.

Programme in einer Hochsprache lassen sich *automatisch* in Programme der Maschinensprache übersetzen. Ein Programm, das dies tut, nennt man *Übersetzer* oder Compiler.

Ein Vorteil dieses Vorgehens ist auch, dass Programme der Hochsprache in verschiedene Maschinensprachen (*Portabilität*) übersetzt und andererseits verschiedene Hochsprachen auch in ein und dieselbe Maschinensprache übersetzt werden können (*Flexibilität*).

Es gibt auch sog. interpretierte Sprachen. Dort werden die Anweisungen der Hochsprache während der Ausführung „on the fly" in Maschinensprache übersetzt. Beispiele: Python, Shell, Basic.

Schließlich gibt es Mischformen, bei denen von der Hochsprache in eine Zwischensprache übersetzt wird, die dann interpretiert wird. Beispiel: Java.

Abbildung 2 zeigt die notwendigen Schritte bei der Programmerstellung im Überblick.

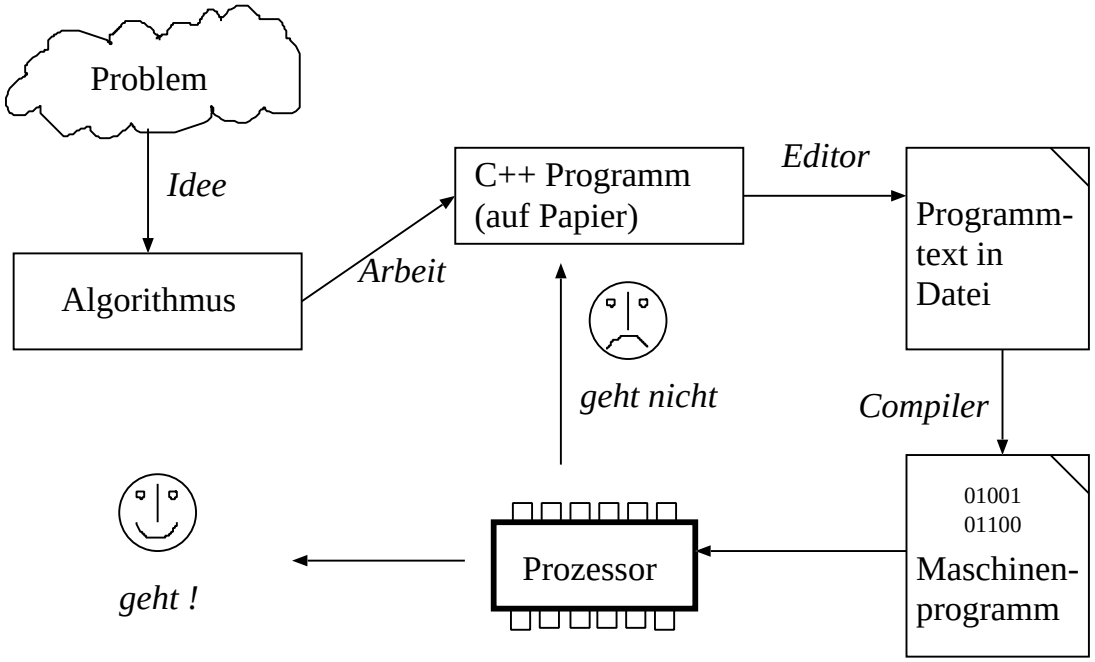


Abbildung 2: Workflow bei der Programmerstellung.

Warum gibt es verschiedene Programmiersprachen ? Wie bei der Umgangssprache: teils sind Unterschiede historisch gewachsen, teils sind die Sprachen wie Fachsprachen auf verschiedene Problemstellungen hin optimiert.

Andererseits sind die Grundkonzepte in vielen prozeduralen bzw. objektorientierten Sprachen sehr ähnlich. Eine neue Sprache in dieser Klasse kann relativ leicht erlernt werden.

1.7 Komplexität von Programmen

Die Leistungsfähigkeit von Computern wächst schnell.

Wissen: (Moore'sches[9] „Gesetz“)
Die Anzahl der Transistoren pro Flächeneinheit auf einem Halbleiterchip verdoppelt sich etwa alle 18–24 Monate.

Beispiel: Entwicklung von Taktgeschwindigkeit, Speichergröße und Größe des Linux-Kernel.

Zeit	Proz	Takt	RAM	Disk	Linux Kernel (.tar.gz)
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)
2004	P4 (Prescott)	3.8 GHz	2048 MB	250 GB	36 MB (2.4.26)
2010	i7 (Westmere)	3.5 GHz	8196 MB	1024 GB	84 MB (2.6.37.7)
2019	Xeon W	2.6 GHz	32 GB	4 TB	102 MB (5.2)

Bis 2001 exponentielles Wachstum. Prozessortaktfrequenz stagniert seit 2004. Wachstum des Linux-Kernel ist auch abgeflacht.

Grenzen der Hardwareentwicklung:

- Je höher die Takfrquenz desto höher die Wärmeentwicklung.
- Mehr an Transistoren wird in viele unabhängige Cores gesteckt.
- Erfordert parallele Programmierung.

Grenzen der Softwareentwicklung:

- Die benötigte Zeit zum Erstellen großer Programme skaliert mehr als linear, d. h. zum Erstellen eines doppelt so großen Programmes braucht man mehr als doppelt so lange.
- Verbesserte Programmiertechnik, Sprachen und Softwareentwurfsprozesse. Einen wesentlichen Beitrag leistet hier die objektorientierte Programmierung, die wir in dieser Vorlesung am Beispiel von C++ erlernen werden.

2 Funktionale Programmierung

2.1 Auswertung von Ausdrücken

Arithmetische Ausdrücke

Beispiel: Auswertung von:

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

[9] Gordon E. Moore, US-amerk. Unternehmer (Mitbegründer der F. Intel), geb. 1929.

Programm: (Erste Schritte [erstes.cc])

```
#include "fcpp.hh"

int main ()
{
  return print( (3+(5*8))-(16*(7+9)) );
}
```

Übersetzen (in Unix-Shell):

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

Ausführung:

```
> ./erstes
-213
```

Bemerkung:

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

Wie wertet der Rechner so einen Ausdruck aus?

Die Auswertung eines zusammengesetzten Ausdruckes lässt sich auf die Auswertung der vier elementaren Rechenoperationen $+, -, *$ und $/$ zurückführen.

Dazu fassen wir die Grundoperationen als zweistellige Funktionen auf:

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

Jeden Ausdruck können wir dann äquivalent umformen:

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

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

Bemerkung: Die Infix-Schreibweise ist für arithmetische Ausdrücke bei Hinzunahme von Präzedenzregeln wie „Punkt vor Strich“ und dem Ausnutzen des Assoziativgesetzes kürzer (da Klammern wegelassen werden können) und leichter lesbar als die Präfix-Schreibweise.

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

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

Der Compiler übersetzt den Ausdruck aus der Infix-Schreibweise in die äquivalente Präfixschreibweise. Die Auswertung des Ausdrucks, d. h. die Berechnung der Funktionen, erfolgt dann von innen nach aussen:

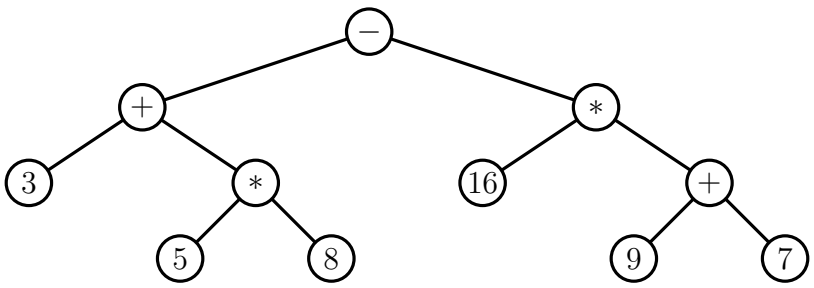
```
  -(+(3,*(5,8)),*(16,+(7,9)))
= -(+(3,  40  ),*(16,+(7,9)))
= -(    43     ,*(16,+(7,9)))
= -(    43     ,*(16,  16  ))
= -(    43     ,    256     )
=            -213
```

Bemerkung: Dies ist nicht die einzig mögliche Reihenfolge der Auswertung der Teiloperationen, alle Reihenfolgen führen jedoch zum gleichen Ergebnis! (Zumindest bei nicht zu großen, ganzen Zahlen).

Bemerkung: C++ kennt die Punkt-vor-Strich-Regel und das Assoziativgesetz. Überflüssige Klammern können also weggelassen werden.

Ausdrücke als Bäume

Jeder arithmetische Ausdruck kann als binärer Baum dargestellt werden. Die Auswertung des Ausdruckes erfolgt dann von den Blättern zur Wurzel. In dieser Darstellung erkennt man welche Ausführungsreihenfolgen möglich sind bzw. welche Teilausdrück gleichzeitig ausgewertet werden können (Datenflussgraph).



2.2 Funktionen

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

Beispiel: Eine einstellige Funktion:

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

Die erste Zeile (Funktionskopf) vereinbart, dass die neue Funktion namens `quadrat` als Argument eine Zahl mit Namen `x` vom Typ `int` als Eingabe bekommt und einen Wert vom Typ `int` als Ergebnis liefert.

Der Funktionsrumpf (*body*) zwischen geschweiften Klammern sagt, was die Funktion tut. Der Ausdruck nach return ist der Rückgabewert.

Wir werden uns zunächst auf einen sehr kleinen Teil des Sprachumfangs von C/C++ beschränken. Dort besteht der Funktionsrumpf nur aus dem Wort return gefolgt von einem Ausdruck gefolgt von einem Semikolon.

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

Dies ist in völliger Analogie zur Mathematik:

$$x \in \mathbb{Z}, \qquad f : \mathbb{N} \to \mathbb{N}.$$

Bemerkung: Der Typ `int` entspricht dabei (kleinen) ganzen Zahlen. Andere Typen sind `float`, `double`, `char`, `bool`. Später werden wir sehen, dass man auch neue Typen hinzufügen kann.

Programm: (Verwendung [quadrat.cc])

```
#include "fcpp.hh"

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

int main ()
{
  return print( quadrat(3)+quadrat(4+4) );
}
```

Bemerkung: Damit können wir die Bedeutung aller Elemente des Programmes verstehen.

- Neue Funktionen kann man (in C) nur in Präfix-Schreibweise verwenden.
- `main` ist eine Funktion ohne Argumente und mit Rückgabetyp `int`.
- #include "fcpp.hh" ist ein sogenannter Include-Befehl. Er sorgt dafür, dass die in der Datei fcpp.hh enthaltenen Erweiterungen von C++, etwa zusätzliche Funktionen, verwendet werden können. fcpp.hh ist nicht Teil des C++ Systems, sondern

wird von uns für die Vorlesung zur Verfügung gestellt (erhältlich auf der Webseite). Achtung: Die Datei muss sich im selben Verzeichnis befinden wie das zu übersetzende Programm damit der Compiler diese finden kann.

- `print` ist eine Funktion mit Rückgabewert 0 (unabhängig vom Argument), welche den Wert des Arguments auf der Konsole ausdruckt (Seiteneffekt). Die Definition dieser Funktion ist in der Datei `fcpp.hh` enthalten.
- Die Programmausführung beginnt immer mit der Funktion `main` (sozusagen das Startsymbol).

2.3 Selektion

Fehlt noch: Steuerung des Programmverlaufs in Abhängigkeit von Daten.

Beispiel: Betragsfunktion

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

Um dies ausdrücken zu können, führen wir eine spezielle dreistellige Funktion `cond` ein:

Programm: **(Absolutwert [absolut.cc])**

```
#include "fcpp.hh"

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

int main ()
{
  return print( absolut(-3) );
}
```

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

Bemerkung: `cond` kann keine einfache *Funktion* sein:

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

Bemerkung: Damit haben wir bereits eine Menge von Konstrukten kennengelernt, die turing-äquivalent ist!

2.4 Syntaxbeschreibung mit Backus-Naur Form

Erweiterte Backus-Naur-Form (EBNF)

Die Regeln nach denen wohlgeformte Sätze einer Sprache erzeugt werden, nennt man Syntax.

Die Syntax von Programmiersprachen ist recht einfach. Zur Definition verwendet man eine spezielle Schreibweise, die erweiterte Backus[10]-Naur[11] Form (EBNF):

Man unterscheidet in der EBNF folgende Zeichen bzw. Zeichenketten:

- Unterstrichene Zeichen oder Zeichenketten sind Teil der zu bildenden, wohlgeformten Zeichenkette. Sie werden nicht mehr durch andere Zeichen ersetzt, deshalb nennt man sie terminale Zeichen.
- Zeichenketten in spitzen Klammern, wie etwa $< \mathrm{Z} >$ oder $< \text{Ausdruck} >$ oder $< \text{Zahl} >$, 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. (Es sei denn sie sind unterstrichen und somit terminale Zeichen).
- (Alternativ findet man auch die Konvention terminale Symbole in Anführungszeichen zu setzen und die spitzen Klammern bei nichtterminalen wegzulassen).

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

Beispiel:

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

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

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

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

Kurzschreibweisen

Oder:

[10] John Backus, 1924–2007, US-amerik. Informatiker.
[11] Peter Naur, geb. 1928, dänischer Informatiker.

Das Zeichen „ | “ („oder“) erlaubt die Zusammenfassung mehrerer Regeln in einer Zeile. Beispiel: $<\text{A}> ::= \underline{\text{a}} <\text{A}> \underline{\text{b}} \mid <\epsilon>$

Option:

$<\text{A}> ::= [<\text{B}>]$ ist identisch zu $<\text{A}> ::= <\text{B}> \mid <\epsilon>$

Wiederholung mit $n \geq 0$:

$<\text{A}> ::= \{ <\text{B}> \}$ ist identisch mit $<\text{A}> ::= <\text{A}> <\text{B}> \mid <\epsilon>$

Wiederholung mit $n \geq 1$:

$<\text{A}> ::= \{ <\text{B}> \}^+$ ist identisch zu
$<\text{A}> ::= <\text{A}> <\text{B}> \mid <\text{B}>$

Syntaxbeschreibung für FC++

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

Syntax: (Zahl)

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

Syntax: (Ausdruck)

$<\text{Ausdruck}>$ ::= $<\text{Zahl}> \mid [\underline{-}] <\text{Bezeichner}> \mid$
$\underline{(} <\text{Ausdruck}> <\text{Operator}> <\text{Ausdruck}> \underline{)} \mid$
$<\text{Bezeichner}> \underline{(} [<\text{Ausdruck}> \{ \underline{,} <\text{Ausdruck}> \}] \underline{)} \mid$
$<\text{Cond}>$

$<\text{Bezeichner}>$ ::= $<\text{Buchstabe}> \{ <\text{Buchstabe oder Zahl}> \}$

$<\text{Operator}>$ ::= $\underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/}$

Weggelassen: Regeln für $<\text{Buchstabe}>$ und $<\text{Buchstabe oder Zahl}>$.

Diese einfache Definition für Ausdrücke enthält weder Punkt-vor-Strich noch das Weglassen von Klammern aufgrund des Assoziativgesetzes!

Hier die Syntax einer Funktionsdefinition in EBNF:

Syntax: (Funktionsdefinition)

$<\text{Funktion}>$ ::= $<\text{Typ}> <\text{Name}> \underline{(} <\text{formale Parameter}> \underline{)}$
$\underline{\{} <\text{Funktionsrumpf}> \underline{\}}$

$<\text{Typ}>$ ::= $<\text{Bezeichner}>$

$<\text{Name}>$ ::= $<\text{Bezeichner}>$

$<\text{formale Parameter}>$ ::= $[<\text{Typ}> <\text{Name}> \{ \underline{,} <\text{Typ}> <\text{Name}> \}]$

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.

Regel für den Funktionsrumpf:

$$<\text{Funktionsrumpf}> ::= \underline{\text{return}} <\text{Ausdruck}> \underline{;}$$

Hier ist noch die Syntax für die Selektion:

Syntax: **(Cond)**

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

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

Syntax: (FC++ Programm)

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

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

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

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

- Kein Funktionsname darf doppelt vorkommen.
- Genau eine Funktion muss `main` heissen.
- Namen müssen an der Stelle bekannt sein wo sie vorkommen.

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

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  (auch über
    mehrere Zeilen)
*/
```

2.5 Das Substitutionsmodell

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

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

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

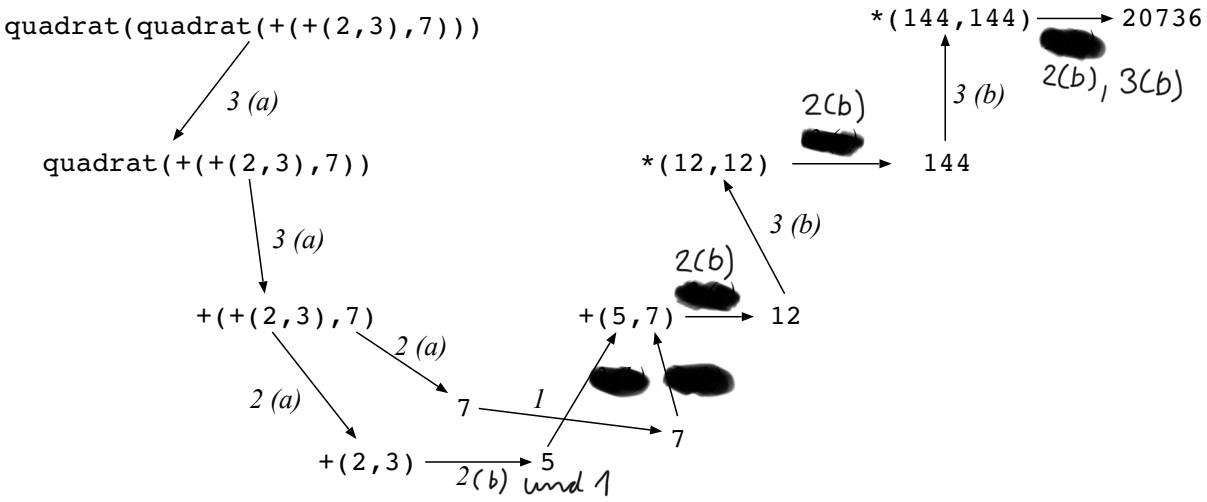
Bemerkung: Die Namen der formalen Parameter sind egal, sie entsprechen sogenannten gebundenen Variablen in logischen Ausdrücken.

Beispiel:

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

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
```



2.6 Linear-rekursive Prozesse

Beispiel: (Fakultätsfunktion) Sei $n \in \mathbb{N}$. Dann gilt

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

Oder rekursiv:

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

Programm: **(Rekursive Berechnung der Fakultät [fakultaet.cc])**

```
#include "fcpp.hh"

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

```
int main ()
{
  return print(fakultaet(5));
}
```

Die Auswertung kann mithilfe des Substitutionsmodells wie folgt geschehen:

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

Definition: Dies bezeichnen wir als linear rekursiven Prozess (die Zahl der *verzögerten* Operationen wächst linear in n). Die Aufrufe formen eine lineare Kette von Funktionsaufrufen.

2.7 Linear-iterative Prozesse

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

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

Idee: Führe das Produkt als zusätzliches Argument mit.

Programm: **(Iterative Fakultätsberechnung [fakultaetiter.cc])**

```
#include "fcpp.hh"

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 ()
{
  return print(fakultaet(5));
}
```

Die Analyse mit Hilfe des Substitutionsprinzips liefert:

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

Hier wird allerdings von folgender Optimierung ausgegangen: In fakIter wird das Ergebnis des rekursiven Aufrufes von fakIter ohne weitere Verarbeitung zurückgegeben. In diesem Fall muss keine Kette verzögerter Aufrufe aufgebaut werden, das Endergebnis entspricht dem Wert der innersten Funktionsauswertung. Diese Optimierung kann vom Compiler durchgeführt werden (tail recursion).

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

Bemerkung:

- Von einem Zustand kann man ohne Kenntnis der Vorgeschichte aus weiterrechnen. Der Zustand fasst alle bis zu diesem Punkt im Programm durchgeführten Berechnungen zusammen.
- Die Zahl der durchlaufenen Zustände ist proportional zu n.
- Die Informationsmenge zur Darstellung des Zustandes ist konstant.
- Bei geeigneter Implementierung ist der Speicherplatzbedarf konstant.
- Beim Lisp-Dialekt Scheme wird diese Optimierung von am Ende aufgerufenen Funktionen (*tail-call position*) im Sprachstandard verlangt.
- Bei anderen Sprachen (auch C++) ist diese Optimierung oft durch Compilereinstellungen erreichbar (nicht automatisch, weil das Debuggen erschwert wird), ist aber nicht Teil des Standards.
- Beide Arten von Prozessen werden durch rekursive Funktionen beschrieben!

2.8 Baumrekursion

Beispiel: (Fibonacci-Zahlen)

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

Die Folge der Fibonacci Zahlen modelliert (unter anderem) das Wachstum einer Kaninchenpopulation unter vereinfachten Annahmen. Sie ist benannt nach Leonardo di Pisa.[12]

Programm: **(Fibonacci rekursiv [fibonacci.cc])**

```
#include "fcpp.hh"

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

int main (int argc, char** argv)
{
  return print(fib(readarg_int(argc,argv,1)));
}
```

`readarg_int(argc,argv,1)` liest das erste Argument auf der Kommandozeile in das Programm. Also:

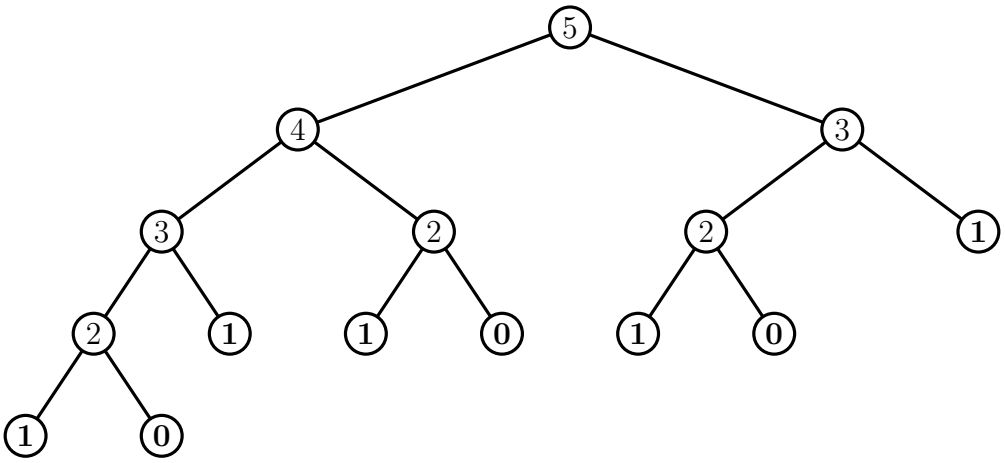
```
> g++ -o fibonacci fibonacci.cc
> ./fibonacci 5
```

$\rightsquigarrow$ Berechnung von `fib(5)`. Auswertung von `fib(5)` nach dem Substitutionsmodell:

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

Graphische Darstellung des Aufrufbaumes

[12]Leonardo di Pisa (auch Fibonacci), etwa 1180–1241, ital. Rechenmeister in Pisa.



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

Bezeichnung: Der Rekursionsprozess bei der Fibonaccifunktion heißt daher baumrekursiv.

Frage:

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

Antwort: (Wachstum von fib). $F_n := \text{fib}(n)$ erfüllt die lineare 3-Term-Rekursion

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

Die Lösungen dieser Gleichung sind von der Form $a\lambda_1^n + b\lambda_2^n$, wobei $\lambda_{1/2}$ die Lösungen der quadratischen Gleichung $\lambda^2 = \lambda + 1$ sind, also $\lambda_{1/2} = \frac{1\pm\sqrt{5}}{2}$. Die Konstanten a und b werden durch die Anfangsbedingungen $F_0 = 0, F_1 = 1$ festgelegt und damit ergibt sich

$$F_n = \underbrace{\frac{1}{\sqrt{5}}}_{a}\left(\frac{1+\sqrt{5}}{2}\right)^n \underbrace{-\frac{1}{\sqrt{5}}}_{b}\left(\frac{1-\sqrt{5}}{2}\right)^n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

für große n, da $|\lambda_2| < 1$.

Bemerkung: $\lambda_1 \approx 1.61803$ ist der *goldene Schnitt*.

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

- Der Gesamtaufwand A_n zur Auswertung von fib (n) ist *größer gleich* einer Konstante c_1 multipliziert mit der Zahl B_n der Blätter im Berechnungsbaum:

  $$A_n \geq c_1 B_n.$$

  Die Zahl der Blätter B_n erfüllt die Rekursion:

  $$B_0 = 1\,,\ B_1 = 1\,,\ B_n = B_{n-1} + B_{n-2}\,, \quad n > 1$$

woraus man

$$B_n = \text{fib}(n+1) \geq \frac{\lambda_1}{\sqrt{5}}\lambda_1^n - \epsilon_1$$

ersieht (Beachte $B_0 = 1$!). Die Ungleichung gilt für $n \geq N_1(\epsilon_1)$.

- Der Gesamtaufwand A_n zur Auswertung von fib (n) ist *kleiner gleich* einer Konstante c_2 multipliziert mit der Anzahl G_n der Knoten im Baum:

  $$A_n \leq c_2 G_n.$$

  Diese erfüllt:

  $$G_0 = 1\,,\ G_1 = 1\,,\ G_n = G_{n-1} + G_{n-2} + 1\,, \quad n > 1.$$

  Durch die Transformation $G_n = G'_n - 1$ ist dies äquivalent zu

  $$G'_0 = 2\,,\ G'_1 = 2\,,\ G'_n = G'_{n-1} + G'_{n-2}\,, \quad n > 1.$$

  Mit den Methoden von oben erhält man

  $$G'_n = \left(1 + \frac{1}{\sqrt{5}}\right)\lambda_1^n + \left(1 - \frac{1}{\sqrt{5}}\right)\lambda_2^n \leq \left(1 + \frac{1}{\sqrt{5}}\right)\lambda_1^n + \epsilon_2$$

  für $n \geq N_2(\epsilon_2)$.

Damit erhalten wir also zusammengefasst:

$$c_1\frac{\lambda_1}{\sqrt{5}}\lambda_1^n - c_1\epsilon_1 \leq A_n \leq c_2\left(1 + \frac{1}{\sqrt{5}}\right)\lambda_1^n + c_2\epsilon_2$$

für $n \geq \max(N_1(\epsilon_1), N_2(\epsilon_2))$.

Bemerkung:

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

Auch die Fibonaccizahlen kann man iterativ berechnen indem man die aktuelle Summe mitführt:

Programm: **(Fibonacci iterativ [fibiter.cc])**

```
#include "fcpp.hh"

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

int fib (int n)
```

```
12  {
      return fibIter(1,0,n);
    }
15
    int main (int argc, char** argv)
    {
18    return print(fib(readarg_int(argc,argv,1)));
    }
```

Hier liefert das Substitutionsmodell:

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

Bemerkung:

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

2.9 Größenordnung

Es gibt eine formale Ausdrucksweise für Komplexitätsaussagen wie „der Aufwand zur Berechnung von `fib(n)` wächst exponentiell“.

Sei n ein Parameter der Berechnung, z. B.

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

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

- Rechenzeit
- Anzahl auszuführender Operationen
- Speicherbedarf

Definition:

- $R(n) = \Omega(f(n))$, falls es von n unabhängige Konstanten c_1, n_1 gibt mit

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

- $R(n) = O(f(n))$, falls es von n unabhängige Konstanten c_2, n_2 gibt mit

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

- $R(n) = \Theta(f(n))$, falls $R(n) = \Omega(f(n)) \wedge R(n) = O(f(n))$.

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

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

Bezeichnung:

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

Beispiel 1: Telefonbuch

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

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

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

$$A_1(n) = C_1 n$$

Algorithmus: (A2) Rekursives Halbieren.

1. Setze $[a_1 = 1, b_1 = n]$, $i = 1$;
2. Ist $a_i = b_i$ durchsuche Seite a_i; Fertig;
3. Setze $m = (a_i + b_i)/2$ (ganzzahlige Division);
4. Falls Name vor Seite m
   setze $[a_{i+1} = a_i, b_{i+1} = m]$, $i = i + 1$, gehe zu 2.;
5. Falls Name nach Seite m
   setze $[a_{i+1} = m, b_{i+1} = b_i]$, $i = i + 1$, gehe zu 2.;
6. Durchsuche Seite m; Fertig;

Satz: Sei $C_1 > 0$ die (maximale) Zeit, die das Durchsuchen einer Seite benötigt, und $C_2 > 0$ die (maximale) Zeit für die Schritte 3–5. Der maximale Zeitaufwand $A_2 = A_2(n)$ für Algorithmus A2 ist dann abschätzbar durch

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

Man ist vor allem an der Lösung großer Probleme interessiert. Daher interessiert der Aufwand $A(n)$ für große n.

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

Beweis:

$$\frac{A_1(n)}{A_2(n)} = \frac{C_1 n}{C_1 + C_2 \log_2 n} = \frac{n}{1 + \frac{C_2}{C_1} \log_2 n} \to +\infty$$

Beobachtung:

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

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

2.10 Wechselgeld

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

Beachte: Die Reihenfolge in der wir die Münzen verwenden ist egal.

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

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

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

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

Dies ist ein Beispiel für eine Rekursion in zwei Argumenten.

Bemerkung: Es gilt auch:

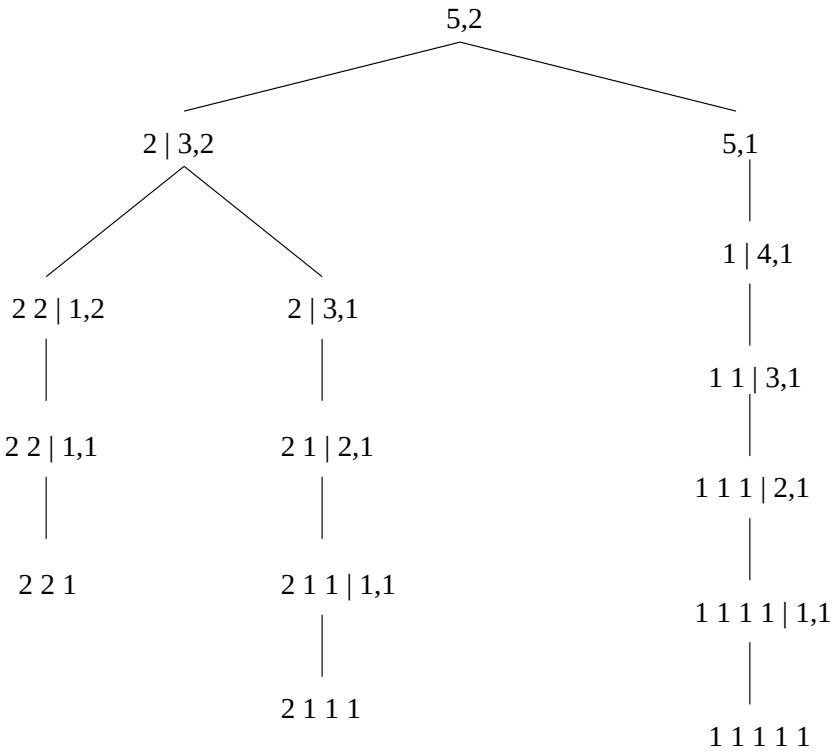


Abbildung 3: Aufrufbaum im Wechselgeld Beispiel.

- $A(0, n) = 1$ für alle $n \geq 0$. Wenn der Betrag a den Wert 0 erreicht hat haben wir den ursprünglichen Betrag gewechselt. ($A(0, 0)$ kann nicht vorkommen).
- $A(a, n) = 0$ falls $a > 0$ und $n = 0$. Der Betrag kann nicht gewechselt werden.
- $A(a, n) = 0$ falls $a < 0$. Der Betrag kann nicht gewechselt werden.

Das Wechseln von 5 Cent in 1 und 2 Centstücke zeigt Abbildung 3.

Bemerkung: Dies ist wieder ein baumrekursiver Prozess.

Programm: **(Wechselgeld zählen [wechselgeld.cc])**

```
#include "fcpp.hh"

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

int wg (int betrag, int muenzarten)
```

```
15  {
      return cond(betrag==0, 1,
                  cond(betrag<0 || muenzarten==0, 0,
18                 wg(betrag,muenzarten-1) +
                   wg(betrag-nennwert(muenzarten),muenzarten)));
    }
21
    int wechselgeld (int betrag)
    {
24    return wg(betrag,6);
    }

27  int main (int argc, char** argv) {
      return print(wechselgeld(readarg_int(argc,argv,1)));
    }
```

Hier einige Resultate:

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

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

2.11 Der größte gemeinsame Teiler

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

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

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

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

Effizienter: Euklidscher[13] Algorithmus. Dieser basiert auf folgenden Überlegungen:

Bezeichnung: Seien $a, b \in \mathbb{N}$ (ohne Null). Dann gibt es Zahlen q, r so dass $a = q \cdot b + r$ mit $q \in \mathbb{N}_0$ und $0 \leq r < b$. Wir schreiben a mod b für den Rest r. Wenn $r = 0$, so schreiben wir $b|a$.

Bemerkung:

[13]Euklid von Alexandria, ca. 360–280 v. Chr., bedeutender griechischer Mathematiker.

1. Wir verlangen $a + b > 0$.
2. Falls $b = 0$ und $a > 0$, so ist $\mathrm{ggT}(a, b) = a$. (Jedes $n \in \mathbb{N}$ teilt 0).
3. Für jeden Teiler s von a und b gilt $\frac{a}{s} = q\frac{b}{s} + \frac{r}{s} \in \mathbb{N}$. Wegen $\frac{a}{s} \in \mathbb{N}$ und $\frac{b}{s} \in \mathbb{N}$ nach Voraussetzung muss auch $\frac{r}{s} \in \mathbb{N}$ gelten, d. h. s ist auch Teiler von r!

   Somit gilt $\mathrm{ggT}(a, b) = \mathrm{ggT}(b, r)$.

Somit haben wir folgende Rekursion bewiesen:

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

Programm: **(Größter gemeinsamer Teiler [ggT.cc])**

```
#include "fcpp.hh"

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


int main (int argc, char** argv)
{
  return print(ggT(readarg_int(argc,argv,1),
                   readarg_int(argc,argv,2)));
}
```

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

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

- Terminiert das Verfahren immer?
- Wie schnell terminiert es?

Bemerkung:

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

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

Beweis: Ausgehend von der Eingabe $a_0 = a, b_0 = b$, $a, b \in \mathbb{N}_0$, $a + b > 0$, erzeugt der Euklidsche Algorithmus eine Folge von Paaren

$$(a_i, b_i), \qquad i \in \mathbb{N}_0.$$

Dabei gilt nach Konstruktion

$$a_{i+1} = b_i, \qquad b_{i+1} = a_i \bmod b_i.$$

Wir beweisen nun einige Eigenschaften dieser Folge.

1. Es gilt $b_i < a_i$ für alle $i \geq 1$. Wir zeigen dies in zwei Schritten.

   α. Sei bereits $b_i < a_i$, dann gilt

   $$a_i = q_i b_i + r_i \quad \text{mit } 0 \leq r_i < b_i.$$

   Da $a_{i+1} = b_i$ und $b_{i+1} = r_i$ gilt offensichtlich

   $$b_{i+1} = r_i < b_i = a_{i+1}.$$

   β. Ist $b_0 < a_0$ dann gilt wegen α. auch $b_1 < a_1$. Bleiben also die Fälle $b_0 = a_0$ und $b_0 > a_0$:

   $$\begin{aligned} b_0 = a_0 \qquad & \Rightarrow a_0 = 1 \cdot b_0 + 0 \Rightarrow b_1 = 0 < b_0 = a_1. \\ b_0 > a_0 \qquad & \Rightarrow a_0 = 0 \cdot b_0 + a_0 \Rightarrow b_1 = a_0 < b_0 = a_1. \end{aligned}$$

2. Nun können wir bereits zeigen, dass der Algorithmus terminieren muss. Wegen 1. gilt

   $$b_{i+1} < a_{i+1} = b_i < a_i, \quad \text{für } i \geq 1,$$

   mithin ist also die Folge der b_i (und auch der a_i) streng monoton fallend. Wegen $b_i \in \mathbb{N}_0$ impliziert $b_{i+1} < b_i$ dass $b_{i+1} \leq b_i - 1$.

   Andererseits ist $b_i \geq 0$ für alle $i \geq 0$ da $b_0 \geq 0$ und $b_{i+1} = a_i \bmod b_i$. Somit muss irgendwann $b_i = 0$ gelten und der Algorithmus terminiert.

3. Sei $b_i < a_i$. Dann gilt $b_{i+2} < a_{i+2} < a_i/2$. Dies ist also eine Behauptung über die Konvergenzgeschwindigkeit. Wir unterscheiden zwei Fälle.

   I. Sei $b_i \leq a_i/2$. Dann gilt $a_i = q_i \cdot b_i + r_i$ mit $0 \leq r_i < b_i \leq a_i/2$, also $b_{i+1} = r_i < b_i = a_{i+1} \leq a_i/2$.
   Im nächsten Schritt gilt dann $a_{i+1} = q_{i+1} \cdot b_{i+1} + r_{i+1}$ mit

   $$b_{i+2} = r_{i+1} < b_{i+1} = a_{i+2} < b_i \leq a_i/2.$$

   Somit gilt $b_{i+2} < a_{i+2} < a_i/2$.

   II. Sei $b_i > a_i/2$. Dann gilt $a_i = 1 \cdot b_i + (a_i - b_i)$, also $q_i = 1$, $r_i = a_i - b_i$. Damit gilt $b_{i+1} = r_i = a_i - b_i < a_i/2$ und $a_{i+1} = b_i > a_i/2$ (nach Vor.). Im nächsten Schritt gilt nun wieder $a_{i+1} = q_{i+1} \cdot b_{i+1} + r_{i+1}$ mit

   $$b_{i+2} = r_{i+1} < b_{i+1} = a_{i+2} < a_i/2,$$

   also ebenfalls $b_{i+2} < a_{i+2} < a_i/2$.

Damit ist gezeigt, dass a_i und b_i nach zwei Schritten noch höchstens halb so groß sind. Da $a_i, b_i \in \mathbb{N}_0$ sind höchstens $2 \log_2(\min(a_0, b_0))$ Halbierungen möglich bis b_i den Wert 0 erreicht.

2.12 Zahlendarstellung im Rechner

In der Mathematik gibt es verschiedene Zahlenmengen:

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

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 $\beta \geq 2$ 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 $\mathbb{N}_0$ darstellbar:

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

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

Umgang mit negativen Zahlen In der Mathematik ist das Vorzeichen eine separate Information welche 1 Bit zur Darstellung benötigt.

Im Rechner wird bei ganzen Zahlen zur Basis $\beta = 2$ eine andere Darstellung gewählt, die Zweierkomplementdarstellung.

Früher war auch das Einerkomplement gebräuchlich.

Einer- und Zweierkomplement **Definition**: Sei $(a_{n-1}a_{n-2}\ldots a_1a_0)_2$ die Binärdarstellung von $a \in [0, 2^n - 1]$. Dann heisst

$$e_n(a) = e_n((a_{n-1}a_{n-2}\ldots a_1a_0)_2) = (\bar{a}_{n-1}\bar{a}_{n-2}\ldots \bar{a}_1\bar{a}_0)_2$$

das Einerkomplement von a, wobei $\bar{a}_i = 1 - a_i$.

Definition: Sei $a \in [0, 2^n - 1]$. Dann heisst

$$z_n(a) = 2^n - a$$

das Zweierkomplement von a.

Es gilt: $e_n(e_n(a)) = a$ und $z_n(z_n(a)) = a$!

Das Zweierkomplement einer Zahl kann sehr einfach und effizient berechnet werden.

Sei $a \in [0, 2^n - 1]$. Dann folgt aus der Identität

$$a + e_n(a) = 2^n - 1$$

die Formel

$$z_n(a) = 2^n - a = e_n(a) + 1.$$

Somit wird keine Subtraktion benötigt sondern es genügt das Einerkomplement und eine Addition von 1. (Und das kann noch weiter vereinfacht werden).

Definition: Die Zweierkomplementdarstellung einer Zahl ist eine bijektive Abbildung

$$d_n : [-2^{n-1}, 2^{n-1} - 1] \to [0, 2^n - 1]$$

welche definiert ist als

$$d_n(a) = \begin{cases} a & 0 \leq a < 2^{n-1} \\ 2^n - |a| & -2^{n-1} \leq a < 0 \end{cases} .$$

Die negativen Zahlen $[-2^{n-1}, -1]$ werden damit auf den Bereich $[2^{n-1}, 2^n - 1]$ positiver Zahlen abgebildet.

Sei $d_n(a) = (x_{n-1}x_{n-2}\dots x_1x_0)_2$ dann ist a positiv falls $x_{n-1} = 0$ und a negativ falls $x_{n-1} = 1$.

Es gilt $d_n(-1) = 2^n - 1 = (1, \dots, 1)_2$ und $d_n(-2^{n-1}) = 2^n - 2^{n-1} = 2^{n-1}(2-1) = 2^{n-1} = (1, 0, \dots, 0)_2$.

Operationen mit der Zweierkomplementdarstellung Die Ein-/Ausgabe von ganzen Zahlen erfolgt (1) im Zehnersystem und (2) mittels separatem Vorzeichen.

Bei der Eingabe einer ganzen Zahl wird diese in das Zweierkomplement umgewandelt:

- Lese Betrag und Vorzeichen ein und teste auf erlaubten Bereich
- Wandle Betrag in das Zweiersystem
- Für negative Zahlen berechne das Zweierkomplement

Bei der Ausgabe gehe umgekehrt vor.

Im folgenden benötigen wir noch eine weitere Operation.

Definition: Sei $x = (x_{m-1}x_{m-2}\dots x_1x_0)_2$ eine m-stellige Binärzahl. Dann ist

$$s_n((x_{m-1}x_{m-2}\dots x_1x_0)_2) = \begin{cases} (x_{m-1}x_{m-2}\dots x_1x_0)_2 & m \leq n \\ (x_{n-1}x_{n-2}\dots x_1x_0)_2 & m > n \end{cases}$$

die Beschneidung auf n-stellige Binärzahlen.

Die Addition von Zahlen in der Zweierkomplementdarstellung gelingt mit

Satz: Sei $n \in \mathbb{N}$ und $a, b, a+b \in [-2^{n-1}, 2^{n-1} - 1]$. Dann gilt

$$d_n(a+b) = s_n(d_n(a) + d_n(b)).$$

Es genügt eine einfache Addition. Beachtung der Vorzeichen entfällt!

Beweis. Wir unterscheiden drei Fälle (mittlerer Fall steht für zwei).

$\underline{a, b \geq 0.}$ Damit ist auch $a + b \geq 0$. Also

$$s_n(d_n(a) + d_n(b)) = s_n(a+b) = a + b = d_n(a+b).$$

$\underline{a < 0, b \geq 0.}$ $a + b$ kann positiv oder negativ sein. Mit $a = -|a|$:

$$s_n(d_n(a) + d_n(b)) = s_n(2^n - |a| + b) = s_n(2^n + (a + b))$$
$$= \begin{cases} a + b & 0 \leq a + b < 2^{n-1} \\ 2^n - |a + b| & -2^{n-1} \leq a + b < 0 \end{cases}.$$

$\underline{a, b < 0.}$ Damit ist auch $a + b < 0$ und $2^n - |a + b| < 2^n$:

$$s_n(d_n(a) + d_n(b)) = s_n(2^n - |a| + 2^n - |b|) = s_n(2^n + 2^n + a + b)$$
$$= s_n(2^n + 2^n - |a + b|) = 2^n - |a + b|.$$

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

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

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

3	→	011
-1	→	111
2	→	[1]010

Die Negation einer Zahl in Zweierkomplementdarstellung. Basis für Subtraktion.

Satz: Sei $n \in \mathbb{N}$ und $a \in [-2^{n-1} + 1, 2^{n-1} - 1]$. Dann gilt

$$d_n(-a) = s_n(2^n - d_n(a)) = s_n(e_n(d_n(a)) + 1).$$

Beweis. Nutze Identität $a + e_n(a) = 2^n - 1 \Leftrightarrow e_n(a) + 1 = 2^n - a$.

$\underline{a = 0.}$ $d_n(-0) = d_n(0) = s_n(2^n - d_n(0))$. Nur dieser Fall braucht die Anwendung von s_n.

$\underline{0 < a < 2^{n-1}.}$ Also ist $-a < 0$ und somit

$$d_n(-a) = 2^n - |a| = 2^n - a = s_n(2 - a) = s_n(2^n - d_n(a)) = s_n(e_n(d_n(a)) + 1)$$

$\underline{-2^{n-1} < a < 0.}$ Also ist $-a > 0$ und somit

$$d_n(-a) = -a = s_n(-a) = s_n(2^n - (2^n + a)) = s_n(2^n - d_n(a)) = s_n(e_n(d_n(a)) + 1)$$

Schließlich behandeln wir noch die Subtraktion.

Diese wird auf die Addition zurückgeführt:

$d_n(a - b) = d_n(a + (-b))$	$a - b = a + (-b)$
$= s_n(d_n(a) + d_n(-b))$	Satz über Addition
$= s_n(d_n(a) + s_n(2^n - d_n(b)))$	Satz über Negation.

Natürlich vorausgesetzt, dass alles im erlaubten Bereich ist.

Gebräuchliche Zahlenbereiche in C++ $\beta = 2$ und $n = 8, 16, 32$:

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

Bemerkung: Die genaue Größe legt der C++ Standard nicht fest, diese kann aber abgefragt werden.

Bemerkung: Achtung: bei Berechnungen mit ganzen Zahlen führt Überlauf, d. h. das Verlassen des darstellbaren Zahlenbereichs üblicherweise nicht zu Fehlermeldungen. Es liegt in der Verantwortung des Programmierers solche Fehler zu vermeiden.

2.13 Darstellung reeller Zahlen

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

Festkommazahlen

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

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

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

Bemerkung:

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

Fließkommaarithmetik

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

$$\pm \left(\sum_{i=0}^{n-1} a_i \beta^{-i} \right) \beta^e = \pm \left(a_0 + a_1 \beta^{-1} + \ldots + a_{n-1} \beta^{-(n-1)} \right) \beta^e$$

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

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

dezimalen Nachkommastellen in der Mantisse.

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

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

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

Bemerkung:

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

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

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

Mit Rundung der Zwischenergebnisse ergibt sich dagegen

$$\ldots 11.2 - 11.1 = 0.1$$

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

Typkonversion

Im Ausdruck `5/3` ist „/“ die ganzzahlige Division ohne Rest. Bei `5.0/3.0` oder `5/3.0` oder `5.0/3` wird hingegen eine Fließkommadivision durchgeführt.

Will man eine gewisse Operation erzwingen, kann man eine explizite Typkonversion einbauen:

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

2.14 Wurzelberechnung mit dem Newtonverfahren

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

$$f(x) = a$$

lösen.

Beispiel: $f(x) = x^2 \rightsquigarrow$ Berechnung von Quadratwurzeln.

Mathematik: $\sqrt{a}$ ist die positive Lösung von $x^2 = a$.

Informatik: Will Algorithmus zur Berechnung des Zahlenwerts von $\sqrt{a}$.

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

Definition: (Taylorreihe)

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

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

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

und somit zur Iterationsvorschrift

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

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

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

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

Programm: **(Quadratwurzelberechnung [newton.cc])**

```
#include "fcpp.hh"

bool gut_genug (double xn, double a) {
  return fabs(print(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(1.0,a);
}

int main (int argc, char** argv)
{
  return print(wurzel(readarg_double(argc,argv,1)));
}
```

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

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

Bemerkung:

- Die print-Funktion sorgt dafür, dass 16 Stellen bei Fließkommazahlen ausgegeben werden.
- Unter gewissen Voraussetzungen an f kann man zeigen, dass sich die Zahl der gültigen Ziffern mit jedem Schritt verdoppelt.

2.15 Fortgeschrittenere funktionale Programmierung

Funktionen in der Mathematik

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

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

Beispiele:

- Ableitung: Funktionen → Funktionen; $X = C^1([a, b])$, $Y = C^0([a, b])$
- Stammfunktion: Funktionen → Funktionen
- Integraler Mittelwert: Funktionen → Zahlen

In C und C++ ist es ebenfalls möglich Funktionen als Argumente an Funktionen zu übergeben. Dazu stehen zwei Möglichkeiten zur Verfügung:

- Funktionszeiger (behandeln wir nicht)
- Funktoren (behandeln wir im Rahmen der Objektorientierung)

Bei kompilierten Sprachen werden alle Funktionen zur Übersetzungszeit erzeugt (aber arbeiten auf zur Laufzeit erzeugten Daten).

Interpretierte Sprachen erlauben auch die Erzeugung des Codes selbst zur Laufzeit.

Beispiel: Beispiel mit sog. Lambdaausdruck aus C++ 11:

```
#include "fcpp.hh"

typedef int (*F)( int n ); // Datentyp Funktionszeiger

int apply( F f, int arg )  // apply wendet f auf arg an
{
  return f(arg);
}

int main( int argc, char *argv[] )
{
  return print( apply( [](int n){ return n+5; }, // anonyme Fkt.
                       readarg_int( argc, argv, 1 ) ) );
}
```

Warum funktionale Programmierung?

Mathematisch am besten verstanden.

Vorteilhaft, wenn Korrektheit von Programmen gezeigt werden soll.

Die funktionale Programmierung kommt mit wenigen Konzepten aus C++ aus: Auswertung von Ausdrücken, Funktionsdefinition, cond-Funktion.

Bestimmte Probleme lassen sich mit rekursiv formulierten Algorithmen (und nur mit diesen) sehr elegant lösen.

Funktionales Programmieren ist nicht für alle Situationen die beste Wahl! Zum Beispiel legt die Interaktion mit der Außenwelt oder ihre *effiziente* Nachbildung oft andere Paradigmen nahe.

Ungeschickte rekursive Formulierungen können zu sehr langen Laufzeiten führen. Geeignete Formulierung und Endrekursion können dies vermeiden.

3 Prozedurale Programmierung

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 zeigen, dass dieses Berechnungsmodell äquivalent zur Turingmaschine ist.

In der Praxis erweist es sich 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!

3.1 Lokale Variablen und die Zuweisung

Konstanten

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

Beispiel:

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

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

Bemerkung:

- Wir können uns vorstellen, dass einem Namen ein Wert zugeordnet wird.
- Einem formalen Parameter wird der Wert bei Auswertung der Funktion zugeordnet.
- Einer Konstanten kann nur einmal ein Wert zugeordnet werden.
- Die Auswertung solcher Funktionsrümpfe erfordert eine Erweiterung des Substitutionsmodells:
  - Ersetze formale Parameter durch aktuelle Parameter.
  - Erzeuge (der Reihe nach !) die durch die Zuweisungen gegebenen Name-Wert Zuordnungen und ersetze neue Namen im Rest des Rumpfes durch den Wert.

Variablen

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

Beispiel: (Variablen)

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

Bemerkung:

- Zeile 1 im Funktionsrumpf definiert eine Variable `y`, die Werte vom Typ `int` annehmen kann und initialisiert diese mit dem Wert des Ausdrucks `x*x`.
- Zeile 2 nennt man eine Zuweisung. Die links des `=` stehende Variable erhält den Wert des rechts stehenden Ausdrucks als neuen Wert.
- Beachte, dass der boolsche Operator „ist gleich“ (also die Abfrage nach Gleichheit) in C++ durch == notiert wird!
- Der Wert von `y` wird erst geändert, nachdem der Ausdruck rechts ausgewertet wurde.
- Der Typ einer Variablen kann aber nicht geändert werden!

Problematik der Zuweisung

Beispiel:

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

```
  return x1*x2;          // Zeile 5
}
```

Bemerkung:

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

Umgebungsmodell

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

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

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

Name	Typ	Wert
`x`	`int`	4
`y`	`int`	5
`x1`	`int`	12
`x2`	`int`	20

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

Bemerkung:

- Ein Ausdruck wird in Zukunft immer relativ zu einer Umgebung ausgewertet, d. h. nur Ausdruck und Umgebung zusammen erlauben die Berechnung des Wertes eines Ausdruckes.
- Die Zuweisung können wir nun als Modifikation der Bindungstabelle verstehen: nach der Ausführung von `y=5` gilt $w(\texttt{y}) = 5$.
- Die Bindungstabelle ändert sich dynamisch während der Programmausführung. Um herauszufinden „was ein Programm tut“ muss sich der Programmierer die fortwährende Entwicklung der Bindungstabelle vorstellen.

3.2 Syntax von Variablendefinition und Zuweisung

Syntax:

$$
\begin{array}{lcl}
\langle \text{Def} \rangle & ::= & \langle \text{ConstDef} \rangle \mid \langle \text{VarDef} \rangle \\
\langle \text{ConstDef} \rangle & ::= & \underline{\text{const}} \ \langle \text{Typ} \rangle \ \langle \text{Name} \rangle \equiv \langle \text{Ausdruck} \rangle \\
\langle \text{VarDef} \rangle & ::= & \langle \text{Typ} \rangle \ \langle \text{Name} \rangle \ [\equiv \langle \text{Ausdruck} \rangle \ ]
\end{array}
$$

Syntax:

$$
\langle \text{Zuweisung} \rangle \quad ::= \quad \langle \text{Name} \rangle \equiv \langle \text{Ausdruck} \rangle
$$

Bemerkung:

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

Lokale Umgebung

Wie sieht die Umgebung im Kontext mehrerer Funktionen aus?

Programm:

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

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

Es gilt folgendes:

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

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

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

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

vor die Funktion `g`.

3.3 Anweisungsfolgen (Sequenz)

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

Erinnerung: Wir kennen schon eine Reihe wichtiger Anweisungen:

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

Bemerkung:

- Jede Anweisung endet mit einem Semikolon.
- Überall wo eine Anweisung stehen darf, kann auch eine durch geschweifte Klammern eingerahmte Folge (*Sequenz*) von Anweisungen stehen.
- Auch die leere Anweisung ist möglich indem man einfach ein Semikolon einfügt.
- Anweisungen werden der Reihe nach abgearbeitet.

Syntax: (Anweisung)

$$
\begin{array}{lll}
\langle \text{Anweisung} \rangle & ::= & \langle \text{EinfacheAnw} \rangle \mid \underline{\{} \ \{ \ \langle \text{EinfacheAnw} \rangle \ \}^{+} \ \underline{\}} \\
\langle \text{EinfacheAnw} \rangle & ::= & \langle \text{VarDef} \rangle \ \underline{;} \mid \langle \text{Zuweisung} \rangle \ \underline{;} \mid \\
 & & \langle \text{Selektion} \rangle \mid \ldots
\end{array}
$$

Beispiel

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

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

```
  int t;

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

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

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

nicht gleichzeitig durchgeführt werden können.

Bemerkung: Man beachte, dass die Reihenfolge in

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

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

3.4 Bedingte Anweisung (Selektion)

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

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

$$\begin{array}{rcl} \text{<Selektion>} & ::= & \underline{\text{if}}\ (\ \text{<BoolAusdr>}\ )\ \text{<Anweisung>} \\ & & [\ \underline{\text{else}}\ \text{<Anweisung>}\ ] \end{array}$$

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

Genauer bezeichnet man die Variante **ohne** `else` als bedingte Anweisung, die Variante **mit** `else` als Selektion.

Beispiel: Die funktionale Form

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

ist äquivalent zu

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

3.5 While-Schleife

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

Programm: **(Fakultät mit While-Schleife [fakwhile.cc])**

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

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

Syntax: **(While-Schleife)**

$$<\text{WhileSchleife}> \quad ::= \quad \underline{\text{while}}\ \underline{(}\ <\text{BoolAusdr}>\ \underline{)}\ <\text{Anweisung}>$$

Die Anweisung wird solange ausgeführt wie die Bedingung erfüllt ist.

Wir überlegen informell warum das Beispiel funktioniert.

- Ist $n = 0$ oder $n = 1$, also $n < 2$ (andere Zahlen sind nicht erlaubt), so ist die Schleifenbedingung nie erfüllt und das Ergebnis ist 1, was korrekt ist.
- Ist $n \geq 2$ so wird die Schleife mindestens einmal durchlaufen. In jedem Durchlauf wird der zaehler dranmultipliziert und dann erhöht. Es werden also sukzessive die Zahlen $2, 3, \ldots$ an den aktuellen Wert multipliziert. Irgendwann erreicht zaehler den Wert n und damit ergebnis den Wert $n!$. Da zaehler nun den Wert $n + 1$ hat, wird die Schleife verlassen.

Später werden wir eine formale Methode kennenlernen, mit der man beweisen kann, dass das Programm korrekt funktioniert.

3.6 For-Schleife

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

Syntax: **(For-Schleife)**

<ForSchleife> ::= 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.

Programm: **(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;
}
```

Bemerkung:

- Eine `For`-Schleife kann direkt in eine `While`-Schleife transformiert werden. Dabei wird die Laufvariable vor der Schleife definiert.
- Der *Gültigkeitsbereich* von `zaehler` erstreckt sich nur über die `for`-Schleife (ansonsten hätte man es wie `ergebnis` außerhalb der Schleife definieren müssen). Wir werden später sehen wie man den Gültigkeitsbereich gezielt mit neuen Umgebungen kontrollieren kann.
- Die Initialisierungsanweisung enthält Variablendefinition und Initialisierung.
- Wie beim Fakultätsprogramm mit `while` 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.

Programm: **(Fibonacci mit For-Schleife [fibfor.cc])**

```
int fib (int n)
{
```

```
3       int a=0;
        int b=1;
        for (int i=0; i<n; i=i+1)
6       {
            int t=a+b; a = b; b = t;
        }
9       return a;
    }
```

Programm: **(Newton mit While-Schleife [newtonwhile.cc])**

```
    #include "fcpp.hh"

3   double wurzel (double a)
      {
          double x=1.0;
6
          while (fabs(x*x-a)>1E-12)
              x = 0.5*(x+a/x);
9         return x;
      }

12  int main ()
    {
      return print(wurzel(2.0));
15  }
```

3.7 Goto

Neben den oben eingeführten Schleifen gibt es eine alternative Möglichkeit die Wiederholung zu formulieren. Wir betrachten nochmal die Berechnung der Fakultät mittels einer `while`-Schleife:

```
int t = 1;
int i = 2;
3
while ( i <= n )
{
6   t = t*i;
    i = i+1;
}
```

Mit der `goto`-Anweisung kann man den Programmverlauf an einer anderen, vorher markierten Stelle fortsetzen:

```
int t = 1; int i = 2;
anfang: if ( i > n ) return t;
3 t = t*i;
i = i+1;
goto anfang;
```

- anfang nennt man eine Sprungmarke (engl.: label). Jede Anweisung kann mit einer Sprungmarke versehen werden.
- Der Sprung kann nur innerhalb einer Funktion erfolgen.
- `While`- und `For`-Schleife können mittels `goto` und Selektion realisiert werden.

In einem berühmten Letter to the Editor [*Go To Statement Considered Harmful, Communications of the ACM, Vol. II, Number 3, 1968*] hat Edsger W. Dikstra[14] dargelegt, dass `goto` zu sehr unübersichtlichen Programmen führt und nicht verwendet werden sollte.

Man kann zeigen, dass `goto` nicht notwendig ist und man mit den obigen Schleifenkonstrukten auskommen kann. Dies nennt man strukturierte Programmierung. Die Verwendung von `goto` in C/C++ Programmen gilt daher als verpönt und schlechter Programmierstil!

Eine abgemilderte Form des `goto` stellen die **break**- und **continue**-Anweisung dar. Diese erhöhen, mit Vorsicht eingesetzt, die Übersichtlichkeit von Programmen.

Auch die Behandlung von Ausnahmen (exceptions) mittels **throw** Anweisung und **try/catch** Blöcken stellt eine Art goto dar.

Regeln guter Programmierung

1. Einrückung sollte verwendet werden um Schachtelung von Schleifen bzw. `if`-Anweisungen anzuzeigen:

```
if ( x >= 0 )
{
  if ( y <= x )
    b = x - y; // b ist groesser 0
  else
  {
    while ( y > x )
      y = y - 1;
    b = x + y;
  }
  i = f(b);
}
```

2. Verwende möglichst sprechende Variablennamen! Kurze Variablennamen wie `i`, `j`, `k` sollten nur innerhalb (kurzer) Schleifen verwendet werden (oder wenn sie der mathematischen Notation entsprechen).
3. Beschränke die Gültigkeit von Konstanten und Variablen auf den kleinsten möglichen Bereich.
4. Nicht mit Kommentaren sparen! Wichtige Anweisungen oder Programmzweige sollten dokumentiert werden. Beim „Programmieren im Großen“ ist die Programmdokumentation natürlich ein wesentlicher Bestandteil der Programmierung.

[14] Edsger Wybe Dijkstra, 1930–2002, niederländischer Informatiker.

5. Verletzung dieser Regeln werden wir in den Übungen ab sofort mit Punktabzug belegen!
6. To be continued ...

3.8 Formale Programmverifikation

Das Verständnis selbst einfacher imperativer Programme bereitet einige Mühe. Übung und Erfahrung helfen hier zwar, aber trotzdem bleibt der Wunsch formal beweisen zu können, dass ein Programm „funktioniert". Dies gilt insbesondere für sicherheitsrelevante Programme.

Eine solche „formale Programmverifikation" erfordert folgende Schritte:

1. Eine formale Beschreibung dessen was das Programm leisten soll. Dies bezeichnet man als Spezifikation.
2. Einen Beweis dass das Programm die Spezifikation erfüllt.
3. Dies erfordert eine formale Definition der Semantik der Programmiersprache.

Beginnen wir mit dem letzten Punkt. Hier haben sich unterschiedliche Vorgehensweisen herausgebildet, die wir kurz beschreiben wollen:

- Operationelle Semantik. Definiere eine Maschine, die direkt die Anweisungen der Programmiersprache verarbeitet.
- Denotationelle Semantik. Beschreibe Wirkung der Anweisungen der Programmiersprache als Zustandsänderung auf den Variablen:
  - $v_1, \ldots, v_m$ seien die Variablen im Programm. $W(v_i)$ der Wertebereich von v_i.
  - $Z = W(v_1) \times \ldots \times W(v_m)$ ist die Menge aller möglichen Zustände.
  - Sei a eine Anweisung, dann beschreibt $F_a : Z \to Z$ die Wirkung der Anweisung.
- Axiomatische Semantik. Beschreibe Wirkung der Anweisungen mittels prädikatenlogischer Formeln. Man schreibt
  $$P \ \{a\} \ Q$$
  wobei P, Q Abbildungen in die Menge $\{\text{wahr}, \text{falsch}\}$, sog. Prädikate, und a eine Anweisung ist.

  $P \ \{a\} \ Q$ bedeutet dann:
  - Wenn P **vor** der Ausführung von a wahr ist, dann gilt Q **nach** der Ausführung von a (P impliziert Q).
  - P heißt auch **Vorbedingung** und Q **Nachbedingung**.

  Beispiel:
  $$-1000 < x \leq 0 \ \{x = x - 1\} \ -1000 \leq x < 0$$

Der oben beschriebene Formalismus der axiomatischen Semantik heisst auch Hoare[15]-Kalkül. Für die gängigen Konstrukte imperativer Programmiersprachen, wie Zuweisung,

[15] Sir Charles Anthony Richard Hoare, geb. 1934, brit. Informatiker.

Sequenz und Selektion, lassen sich Zusammenhänge zwischen Vor- und Nachbedingung herleiten.

Sei nun S ein Programmfragment oder gar das ganze Programm. Dann lassen sich mit dem Hoare-Kalkül entsprechende P und Q finden so dass

$$P\ \{S\}\ Q.$$

Schließlich lässt sich die Spezifikation des Programms ebenfalls mittels prädikatenlogischer Formeln ausdrücken. P_{SPEC} bezeichnet entsprechend die Vorbedingung (Bedingung an die Eingabe) unter der das Ergebnis Q_{SPEC} berechnet wird.

Für ein gegebenes Programm S, welches die Spezifikation implementieren soll, sei nun $P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}$ gezeigt. Der formale Prozess der Programmverifikation besteht dann im folgenden Nachweis:

$$(P_{\text{SPEC}} \Rightarrow P_{\text{PROG}}) \wedge (P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}) \wedge (Q_{\text{PROG}} \Rightarrow Q_{\text{SPEC}}). \tag{1}$$

Dabei kann z. B. P_{SPEC} eine stärkere Bedingung als P_{PROG} sein. Beispiel:

$$-1000 < x \leq 0\ \Rightarrow\ x \leq 0.$$

Der Nachweis von (1) liefert erst die **partielle Korrektheit**. Getrennt davon ist zu zeigen, dass das Programm terminiert. Kann man dies nachweisen ist der Beweis der **totalen Korrektheit** erbracht.

Der Nachweis von $P_{\text{PROG}}\ \{S\}\ Q_{\text{PROG}}$ kann durch automatische Theorembeweiser unterstützt werden.

Korrektheit von Schleifen mittels Schleifeninvariante Wir betrachten nun eine Variante des Hoare-Kalküls, mit der sich die Korrektheit von Schleifenkonstrukten nachweisen lässt. Dazu betrachten wir eine `while`-Schleife in der kanonischen Form

while (B(v)) { v = H(v); }

mit

- $v = (v_1, \ldots, v_m)$ dem Vektor von Variablen, die im Rumpf modifiziert werden,
- $B(v)$, der Schleifenbedingung und
- $H(v) = (H_1(v_1, \ldots, v_m), \ldots, H_m(v_1, \ldots, v_m))$ dem **Schleifentransformator**.

Als Beispiel dient die Berechnung der Fakultät. Dort lautet die Schleife:

while (i <= n) { t = t∗i; i = i+1; }

Also

$$v = (t, i) \qquad B(v) \equiv i \leq n \qquad H(v) = (t * i, i + 1).$$

Zusätzlich definieren wir die Abkürzung

$$H^j(v) = \underbrace{H(H(\ldots H(}_{j \text{ mal}} v) \ldots)).$$

Definition: (Schleifeninvariante)
Sei $v^j = H^j(v^0)$, $j \in \mathbb{N}_0$, die Belegung der Variablen nach j-maligem Durchlaufen der Schleife. Eine Schleifeninvariante INV(v) erfüllt:

1. INV(v^0) ist wahr.
2. INV(v^j) $\wedge$ $B(v^j) \Rightarrow$ INV(v^{j+1}).

Gilt die Invariante vor Ausführung der Schleife und ist die Schleifenbedingung erfüllt, dann gilt die Invariante nach Ausführung des Schleifenrumpfes.

Angenommen, die Schleife wird nach k-maligem Durchlaufen verlassen, d. h. es gilt $\neg B(v^k)$. Ziel ist es nun zu zeigen, dass

$$\text{INV}(v^k) \wedge \neg B(v^k) \Rightarrow Q(v^k)$$

wobei $Q(v^k)$ die geforderte Nachbedingung ist.

Beispiel: Betrachte das Programm zur Berechnung der Fakultät von n:

```
t = 1; i = 2;
while ( i <= n ) { t = t*i; i = i+1; }
```

Behauptung: Sei $n \geq 1$, dann lautet die Schleifeninvariante:

$$\text{INV}(t, i) \;\equiv\; t = (i-1)! \wedge i - 1 \leq n.$$

1. Für $v^0 = (t^0, i^0) = (1, 2)$ gilt INV$(1, 2) \equiv 1 = (2-1)! \wedge (2-1) \leq n$. Wegen $n \geq 1$ ist das immer wahr.
2. Es gelte nun INV$(v^j) \;\equiv\; t^j = (i^j - 1)! \wedge i^j - 1 \leq n$ sowie $B(v^j) = i^j \leq n$. (Vorsicht v^j bedeutet **nicht** v hoch j!). Dann gilt
   - $t^{j+1} = t^j \cdot i^j = (i^j - 1)! \cdot i^j = i^j!$
   - $i^{j+1} = i^j + 1$, somit gilt wegen $i^j = i^{j+1} - 1$ auch $t^{j+1} = (i^{j+1} - 1)!$.
   - Schließlich folgt aus $B(i^j, t^j) \equiv i^j = i^{j+1} - 1 \leq n$ dass INV(i^{j+1}, t^{j+1}) wahr.
3. Am Schleifenende gilt $\neg(i \leq n)$, also $i > n$, also $i = n + 1$ da i immer um 1 erhöht wird. Damit gilt dann also

$$\begin{aligned} & \text{INV}(i, t) \wedge \neg B(i, t) \\ \Leftrightarrow \quad & t = (i-1)! \wedge i - 1 \leq n \wedge i = n + 1 \\ \Leftrightarrow \quad & t = (i-1)! \wedge i - 1 = n \\ \Rightarrow \quad & t = n! \equiv Q(n) \end{aligned}$$

Für den Fall $n \geq 0$ muss man den Fall $0! = 1$ als Sonderfall hinzunehmen. Das Programm ist auch für diesen Fall korrekt und die Schleifeninvariante lautet INV$(i, t) \equiv (t = (i - 1)! \wedge i - 1 \leq n) \vee (n = 0 \wedge t = 1 \wedge i = 2)$.

3.9 Prozeduren und Funktionen

In der Mathematik ist eine Funktion eine Abbildung $f : X \to Y$. C++ erlaubt entsprechend die Definition n-stelliger Funktionen

```
int f( int x1, int x2 ) { return x1*x1 + x2; }
...
int y = f( 2, 3 );
```

In der Funktionalen Programmierung ist das einzig interessante an einer Funktion ihr Rückgabewert. Seiteneffekte spielen keine Rolle. In der Praxis ist das jedoch anders. Betrachte

```
void drucke( int x )
{
  int i = print( x ); // print aus fcpp.hh
}
...
drucke( 3 );
...
```

Es macht durchaus Sinn eine Funktion definieren zu können, deren einziger Zweck das Ausdrucken des Arguments ist. So eine Funktion hat keinen sinnvollen Rückgabewert, ihr einziger Zweck ist der Seiteneffekt.

C++ erlaubt dafür den Rückgabetyp **void** (nichts). Der Funktionsrumpf darf dann keine **return**-Anweisung enthalten, welche einen Wert zurückgibt.

Der Funktionsaufruf ist eine gültige Anweisung, allerdings ist dann keine Zuweisung des Rückgabewerts erlaubt (die Funktion gibt keinen Wert zurück).

Funktionen, die keine Werte zurückliefern heißen Prozeduren. In C++ werden Prozeduren durch den Rückgabetyp **void** gekennzeichnet.

C++ erlaubt auch die Verwendung von Funktionen als Prozeduren, d. h. man verwendet einfach den Rückgabewert **nicht**.

4 Benutzerdefinierte Datentypen

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

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

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

4.1 Aufzählungstyp

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

Beispiel:

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

Syntax: **(Aufzählungstyp)**

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

Programm: **(Vollständiges Beispiel [enum.cc])**

```
#include "fcpp.hh"

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

void druckeZustand (Zustand x) {
  if (x==neu)       print("neu");
  if (x==gebraucht) print("gebraucht");
  if (x==alt)       print("alt");
  if (x==kaputt)    print("kaputt");
}

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

4.2 Felder

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

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

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

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

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

dargestellt. Auf die Komponenten greift man wie folgt zu:

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

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

Syntax: **(Felddefinition)**

$$<\text{FeldDef}> \quad ::= \quad <\text{Typ}> <\text{Name: }> \, \underline{[} \, <\text{Anzahl}> \, \underline{]}$$

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

Bemerkung: Eine Felddefinition darf wie eine Variablendefinition verwendet werden.

Achtung: Bei der hier beschriebenen Felddefinition muss die Größe des Feldes zur **Übersetzungszeit bekannt sein**! Folgendes geht also **nicht**:

```
void f( int n )
{
  char s[n];
  ...
}
```

aber immerhin

```
const int n = 8;
char s[ 3*(n+1) ];
```

Vorsicht: Der GNU-C-Compiler erlaubt Felder variabler Größe als Spracherweiterung! Sie müssen die Optionen `-ansi -pedantic` verwenden, um für obiges Programm eine Fehlermeldung zu erhalten.

Sieb des Eratosthenes

Als Anwendung des Feldes betrachten wir eine Methode zur Erzeugung einer Liste von Primzahlen, die Sieb des Eratosthenes[16] genannt wird.

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

Bemerkung:

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

Programm: **(Sieb des Eratosthenes [eratosthenes.cc])**

[16] Eratosthenes von Kyrene, ca. 276–194 v. Chr., außergewöhnlich vielseitiger griechischer Gelehrter. Methode war damals schon bekannt, Eratosthenes hat nur den Begriff „Sieb“ geprägt.

```
#include "fcpp.hh"

    const int n = 50000000;
    bool prim[n]; // prim[0],...,prim[499999]

int main ()
{

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

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

    // Ausgabe
    int m=0;
    for (int i=0; i<n; i=i+1)
      if (prim[i])
        m = m+1;
    print("Anzahl Primzahlen kleiner als",n,0);
    print(m);

    return 0;
}
```

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

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

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

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

Bemerkung: Die Beziehung zur funktionalen Programmierung ist etwa folgende:

- Mit großen Feldern operierende Algorithmen kann man nur schlecht rein funktional darstellen.
- Dies ist vor allem eine Effizienzfrage, weil oft kleine Veränderungen großer Felder verlangt werden. Bei funktionaler Programmierung müsste man ein neues Feld erzeugen und als Rückgabewert verwenden.

- Algorithmen wie das Sieb des Eratosthenes formuliert man daher funktional auf andere Weise (Datenströme, Streams), was interessant ist, allerdings manchmal auch recht komplex wird. (Allerdings ist dieser Programmierstil auf neuen Prozessoren wie Grafikkarten interessant).

4.3 Zeichen und Zeichenketten

Datentyp `char`

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

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

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

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

Normalerweise sollte man diese Eigenschaft aber nicht brauchen!

ASCII

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

Programm: **(ASCII.cc)**

```
#include "fcpp.hh"

int main ()
{
    for (int i=32; i<=127; i=i+1)
      print(i,(char) i,0);
}
```

Bemerkung:

- Das dritte Argument von `print` ist der (ignorierte) Rückgabewert.
- Die Zeichen $0, \ldots, 31$ dienen Steuerzwecken wie Zeilenende, Papiervorschub, Piepston, etc.
- Für die negativen Werte $-128, \ldots, -1$ (entspricht $128, \ldots, 255$ bei vorzeichenlosen Zahlen) gibt es verschiedene Belegungstabellen (ISO 8859-n), mit denen man Zeichensätze und Sonderzeichen anderer Sprachen abdeckt.

- Noch komplizierter wird die Situation, wenn man *Zeichensätze* für Sprachen mit sehr vielen Zeichen (Chinesisch, Japanisch, etc) benötigt, oder wenn man mehrere Sprachen gleichzeitig behandeln will.
  Stichwort: Unicode.

Zeichenketten

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

Beispiel: Initialisierung eines `char`-Felds:

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

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

Programm: **(Zeichenketten im C-Stil [Cstring.cc])**

```
#include "fcpp.hh"

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

  for (int i=0; name[i]!=0; i=i+1)
    print(name[i]);       // einzelne Zeichen
  print(name);            // ganze Zeichenkette
}
```

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

Programm: **(Zeichenketten im C++-Stil [CCstring.cc])**

```
#include "fcpp.hh"
#include <string>

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

Dies erfordert das einbinden der Header-Datei `string` mit dem **#include** Befehl.

4.4 Typedef

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

Beispiel:

```
typedef int MyInteger;
```

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

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

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

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

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

Beispiel: Auch Feldtypen kann man einen neuen Namen geben:

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

Dann kann man bequem schreiben:

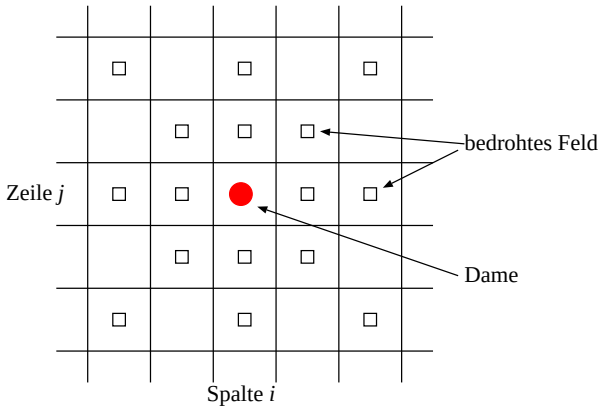
```
Punkt3d a, b;
a[0] = 0.0; a[1] = 1.0; a[2] = 2.0;
b[0] = 0.0; b[1] = 1.0; b[2] = 2.0;
```

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

4.5 Das Acht-Damen-Problem

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

Zugmöglichkeiten der Dame: horizontal, vertikal, diagonal



Bemerkung:

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

Idee:

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

Programm: **(Acht-Damen-Problem [queens.cc])**

```
#include "fcpp.hh"
#include <string>

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

bool good_position (int new_row, columns queen_cols, int new_col)
{
  for (int row=0; row<new_row; row=row+1)
    if ((queen_cols[row] == new_col) ||
    (new_row-row == abs(queen_cols[row]-new_col)))
      return false;
  return true;
}

void display_board (columns queen_cols)
{
  for (int i=0; i<board_size; i=i+1)
    {
      std::string s("");
      for (int j=0; j<board_size; j=j+1)
```

```
          if (j!= queen_cols[i])
            s = s + ".";
          else
            s = s + "D";
        print(s);
      }
    print("␣");
  }

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

  int main ()
  {
    columns queen_cols;
    print("Anzahl␣Loesungen");
    print(queen_configs(0,queen_cols));
    return 0;
  }
```

Bemerkung: Dieses Programm benutzt ein weiteres neues Element:

- Es wurde eine globale Konstante `board_size` verwendet. Diese kann innerhalb aller Funktionen benutzt werden.
- Auch die Typdefinition kann innerhalb aller Funktionen benutzt werden.
- Ein Feld wird als Argument an eine Funktion übergeben. Solche Argumente werden **nicht** kopiert (!). Dazu später mehr. Solange man Argumente nur liest oder nur neue Feldelemente beschreibt (wie hier) kann man sich auch vorstellen das Argument würde kopiert werden.
- Daher bis auf weiteres: Vorsicht bei der Benutzung von Feldern als Argumente von Funktionen!

Bemerkung:

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

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

4.6 Zusammengesetzte Datentypen

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

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

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

Dieser Datentyp kann nun wie folgt verwendet werden

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

Syntax: **(Zusammengesetzter Datentyp)**

$$\begin{array}{lcl} \langle \text{StructDef} \rangle & ::= & \underline{\text{struct}}\ \langle \text{Name} \rangle\ \underline{\{}\ \{\ \langle \text{Komponente} \rangle\ \underline{;}\ \}^{+}\ \underline{\}}\ \underline{;} \\ \langle \text{Komponente} \rangle & ::= & \langle \text{VarDef} \rangle \mid \langle \text{FeldDef} \rangle \mid \ldots \end{array}$$

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

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

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

Anwendung: Rationale Zahlen **Programm: (Rationale Zahlen, die erste [Rational2.cc])**

```
#include "fcpp.hh"

struct Rational {
  int zaehler;
  int nenner;
};

// Abstraktion: Konstruktor und Selektoren

Rational erzeuge_rat (int z, int n)
{
  Rational t;
  t.zaehler = z;
  t.nenner = n;
  return t;
}

int zaehler (Rational q)
{
  return q.zaehler;
}

int nenner (Rational q)
{
  return q.nenner;
}

// Arithmetische Operationen

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

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

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

```
48  }

    Rational div_rat (Rational p, Rational q)
51  {
      return erzeuge_rat(zaehler(p)*nenner(q),
                         nenner(p)*zaehler(q));
54  }

    void drucke_rat (Rational p)
57  {
      print(zaehler(p),"/",nenner(p),0);
    }
60
    int main ()
    {
63    Rational p = erzeuge_rat(3,4);
      Rational q = erzeuge_rat(5,3);
      drucke_rat(p); drucke_rat(q);
66
      // p*q+p-p*p
      Rational r = sub_rat(add_rat(mul_rat(p,q), p),
69                         mul_rat(p,p));
      drucke_rat(r);
      return 0;
72  }
```

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

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

Abhilfe: Normalisierung im Konstruktor:

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

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

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

Komplexe Zahlen Analog lassen sich komplexe Zahlen einführen:

Programm: **(Komplexe Zahlen, Version 1 [Complex2.cc])**

```
#include "fcpp.hh"

struct Complex {
  float real;
  float imag;
};

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

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

// etc

void drucke_complex (Complex p)
{
  print(real(p),"+i*",imag(p),0);
}

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

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

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

Antwort: Eine Möglichkeit, die bereits die Sprache C bietet, ist die folgende:

1. Führe eine sogenannte variante Struktur (Schlüsselwort `union`) ein, die entweder eine rationale oder eine komplexe Zahl enthalten kann ⇝ neuer Datentyp `Combination`
2. Füge auch eine Kennzeichnung hinzu, um was für eine Zahl (rational/komplex) es sich tatsächlich handelt ⇝ neuer Datentyp `Mixed`.
3. Funktionen wie `add_mixed` prüfen die Kennzeichnung, konvertieren bei Bedarf und rufen dann `add_rat` bzw. `add_complex` auf.

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

struct Complex { float re; float im; };

union Combination  // entweder oder!
{
  Rational p;
  Complex c;
};

enum Kind { rational, complex };

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

Bemerkung: Diese Lösung hat etliche Probleme:

- Umständlich und unsicher (überschreiben)
- Das Hinzufügen weiterer Zahlentypen macht eine Änderung von bestehenden Funktionen nötig
- Typprüfungen zur Laufzeit → keine optimale Effizienz
- Speicherplatzbedarf der größten Komponente
- Hätten gerne: Infix-Notation mit unseren Zahlen

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

5 Globale Variablen und das Umgebungsmodell

5.1 Globale Variablen

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

Grund:

- Berechnung einer Funktion hängt nur von ihren Parametern ab.
- Die lokale Umgebung bleibt zwischen Funktionsaufrufen nicht erhalten.

Das werden wir jetzt ändern!

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

Programm: **(Konto [konto1.cc])**

```
#include "fcpp.hh"

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);
  print(abheben(25));
  print(abheben(25));
  print(abheben(25));
}
```

Bemerkung:

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

- Die Funktionen `einrichten`, `kontostand` und `abheben` stellen die **Schnittstelle** zur Bearbeitung des Kontos dar.

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

5.2 Das Umgebungsmodell

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

Definition: **(Umgebung)**

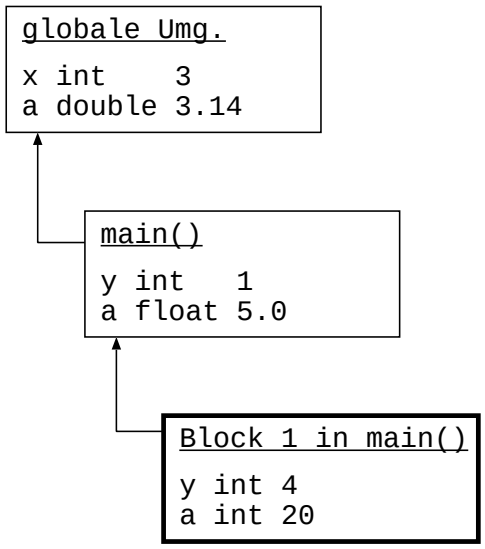
- Eine Umgebung enthält eine Bindungstabelle, d. h. eine Zuordnung von Namen zu Werten.
- Es kann beliebig viele Umgebungen geben. Umgebungen werden während des Programmlaufes implizit (automatisch) oder explizit (bewusst) erzeugt bzw. zerstört.
- Die Menge der Umgebungen bildet eine Baumstruktur. Die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

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

Beispiel:

```
int x = 3;
double a = 4.3;      // 1
int 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 Marke 5:



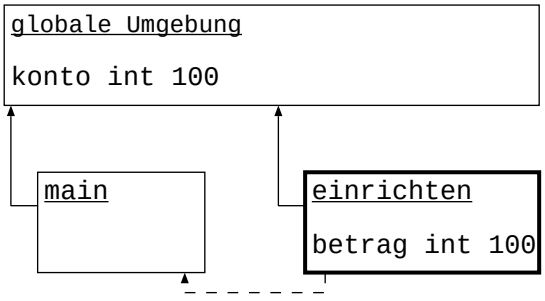
Eigenschaften:

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

Beispiel: (Funktionsaufruf)

```
int konto;

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

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

Nach Marke 2:



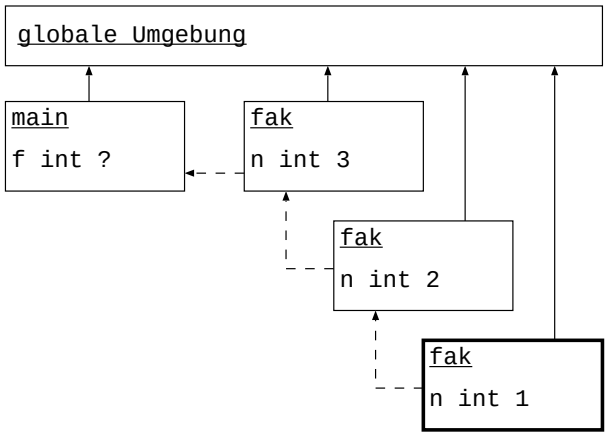
Bemerkung:

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

Beispiel: (Rekursiver Aufruf)

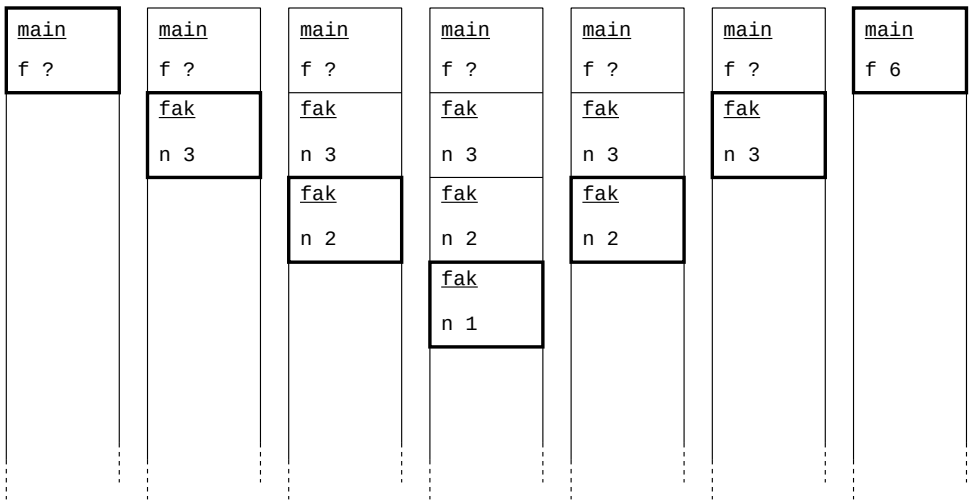
```
int fak( int n )
{
  if ( n == 1 )
    return 1;
  else
    return n * fak( n-1 );
}

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

Während Auswertung von `fak(3)`:



Bemerkung: Im obigen Beispiel gibt es zusätzlich noch eine „versteckte“ Variable für den Rückgabewert einer Funktion. Return kann als **Zuweisung über Umgebungsgrenzen** hinweg verstanden werden.

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



5.3 Stapel

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

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

- Erzeugen eines leeren Stapels (*create*)
- Ablegen eines neuen Elements auf dem Stapel (*push*)
- Test, ob Stapel leer (*empty*)

- Holen des zuletzt abgelegten Elements vom Stapel (*pop*)

Programm: **(Stapel [stack1.cc])**

```
#include "fcpp.hh"

typedef int element_type;    // Integer-Stack

// START stack-library ...

const int stack_size = 1000;

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

// Stack-Operationen

void push (element_type e)
{
  stack[top] = e;
  top = top+1;
}

bool empty ()
{
  return top==0;
}

element_type pop ()
{
  top = top-1;
  return stack[top];
}

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

Bemerkung: Die Stapel-Struktur kann man verwenden, um rekursive in nicht rekursive Programme zu transformieren (wen es interessiert, findet unten eine nichtrekursive Variante für das Wechselgeld Beispiel). Dies ist aber normalerweise nicht von Vorteil, da der für Rekursion verwendete Stapel höchstwahrscheinlich effizienter verwaltet wird.

Programm: **(Wechselgeld nichtrekursiv [wg−stack.cc])**

```
#include "fcpp.hh"

int nennwert (int nr) {// Muenzart -> 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 {         // 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;   // das Ergebnis
  int b,m;        // Hilfsvariablen in Schleife

  stapel[i].betrag = betrag; // initialisiere St.
  stapel[i].muenzarten = 5;  // Startwert
  i = i+1;                   // ein Element mehr

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

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

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

  return anzahl; // Stapel ist jetzt leer
}
```

```
51
    int main () {
      print(wechselgeld2(300));
54  }
```

5.4 Monte-Carlo Methode zur Bestimmung von π

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

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

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

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

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

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

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

Einfachste Methode: (Linear Congruential Method) Ausgehend von einem x_0 verlangt man für $x_1, x_2, \ldots$ die Iterationsvorschrift

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

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

Im Folgenden wird die einfache Vorschrift mit $c = 0$ verwendet:

$$x_{n+1} = ax_n \bmod m$$

mit $a = 7^5 = 16807$ und $m = 2^{31} - 1 = 2147483647$ verwendet [17]

In der Implementierung entsteht das Problem, dass das größte auftretende Produkt $a(m-1)$ nicht in 32 Bit dargestellt werden kann. Mittels eines Tricks, Darstellung von $m = aq+r$ kann man das umgehen.

Programm: **(π mit Monte Carlo Methode [montecarlo1.cc])**

```
#include "fcpp.hh"

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;        // keine lange Arithmetik
  return x;                 // s. 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)
{
```

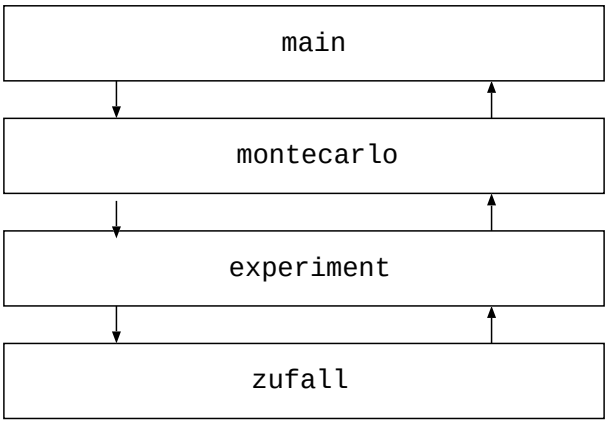
[17] https://www.unf.edu/~cwinton/html/cop4300/s09/class.notes/LCGinfo.pdf

```
36    int erfolgreich=0;

      for (int i=0; i<N; i=i+1)
39      erfolgreich = erfolgreich+experiment();

      return ((double)erfolgreich)/((double)N);
42  }

    int main (int argc, char** argv) {
45    print(sqrt(6.0/montecarlo(readarg_int(argc,argv,1))));
    }
```

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

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

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



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

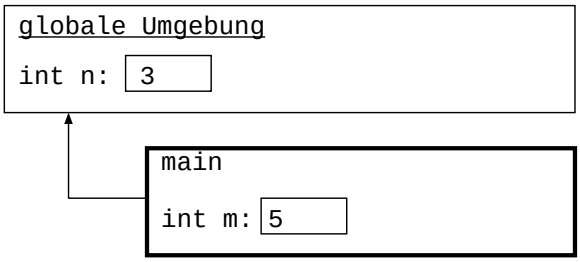
Beobachtung: In dieser Situation entstünde durch Beharren auf einem funktionalen Stil zwar kein Effizienzproblem, die Struktur des Programms würde aber deutlich komplizierter.

6 Zeiger und dynamische Datenstrukturen

6.1 Zeiger

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

```
int n = 3;

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



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

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

Idee: Man führt das Konzept Zeiger (Pointer) ein, der auf Variable (Schubladen) zeigen kann. Variablen, die Zeiger als Werte haben, heißen Zeigervariablen.

Zeiger haben auch einen Typ, der aus dem Typ abgeleitet wird auf den sie zeigen.

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

Notation: Die Definition „**int**∗ x;“ vereinbart, dass `x` auf Variablen (Schubladen) vom Typ `int` zeigen kann. Man sagt `x` habe den Typ „**int**∗“.

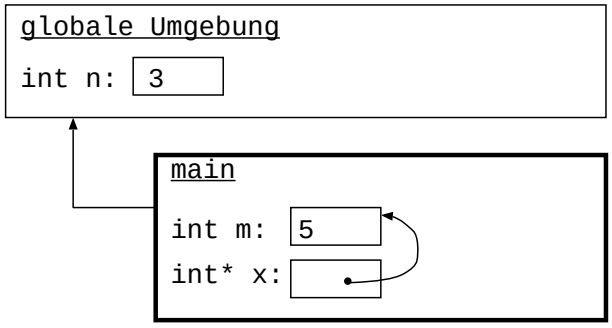
Die Zuweisung „x = &n;“ lässt `x` auf den Ort zeigen, an dem der Wert von `n` steht. & heisst Adressoperator.

Die Zuweisung „∗x = 4;“ verändert den Wert der Schublade, „auf die `x` zeigt“. ∗ heisst Dereferenzierungsoperator.

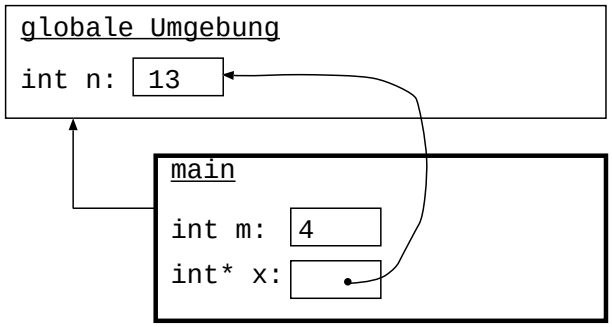
Beispiel:

```
int n = 3;

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

Nach (2)



Nach (5)



6.2 Zeiger im Umgebungsmodell

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

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

Mathematisch entspricht das einer Abbildung w, die die *Symbole* `n`, `m` auf die Wertemenge abbildet:

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

Die Zuweisung „`n = 3;`“ („=“ *ist Zuweisung*) manipuliert die Bindungstabelle so, dass nach der Zuweisung $w(\texttt{n}) = 3$ („=“ *ist Gleichheit*) gilt.

Wenn auf der rechten Seite der Zuweisung auch ein Name steht, etwa „`n = m+1;`“, dann gilt nach der Zuweisung $w(\texttt{n}) = w(\texttt{m}) + 1$. Auf beide Namen wird also w angewandt.

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

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

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

Verwirklichung:

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

Bemerkung:

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

Bemerkung: Auch eine Zeigervariable `x` kann wieder von einer anderen Zeigervariablen referenziert werden.

Diese hat dann den Typ „**int**$**$“ oder „Zeiger auf eine **int**$*$-Variable“.

Bemerkung: In C wird tendenziell die Notation „**int** $*$x;“ verwendet, wohingegen in C++ die Notation „**int**$*$ x;“ empfohlen wird.

Allerdings impliziert die Notation „**int**$*$ x, y;“ dass „`x` und `y` vom Typ **int**$*$“ sind, was aber **nicht** zutrifft! (`y` ist vom Typ **int**)

Man liest „**int** $*$x“ als „`x` dereferenziert ist vom Typ **int**“ (in anderen Worten: „`x` zeigt auf **int**“).

Beispiel:

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

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

Damit können wir schreiben

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

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

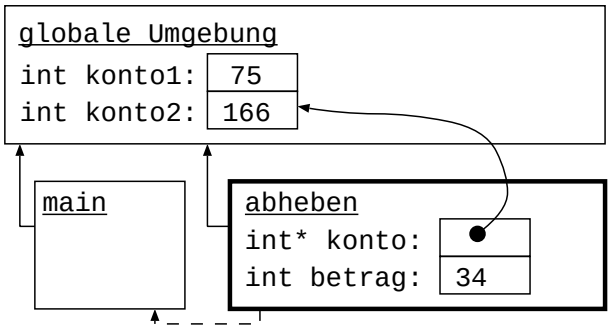
```
y = &n;   // geht nicht, da n nicht vom Typ int*
y = &&n;  // geht auch nicht, da & nur
          // einmal angewandt werden kann
```

6.3 Call by reference

Programm: Die Realisation der Kontoverwaltung könnte wie folgt aussehen:

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

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

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

Nach Marke 1, im zweiten Aufruf von `abheben`:



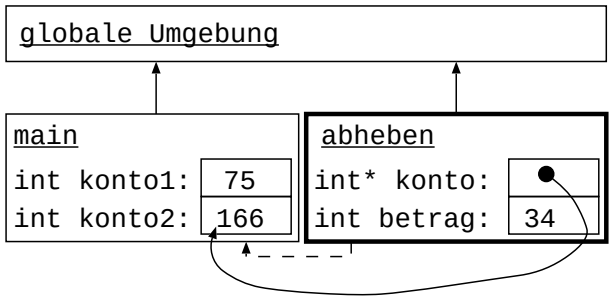
Definition: In der Funktion `abheben` nennt man `betrag` einen *call by value* Parameter und `konto` einen *call by reference* Parameter.

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

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

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

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

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

Nach (1), zweiter Aufruf von `abheben`



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

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

- Der Seiteneffekt ist explizit erwünscht wie in `abheben` (→ Objektorientierung).
- Man möchte das Kopieren großer Objekte sparen (→ `const` Zeiger).

Referenzen in C++

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

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

Programm: **(Konto mit Referenzen)**

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

int main()
{
```

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

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

Bemerkung: Referenzen können nicht nur als Funktionsargumente benutzt werden:

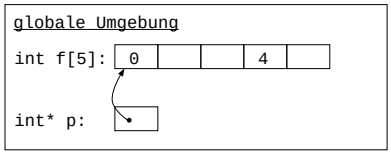
```
    int  n = 3;
    int& r = n; // independent reference
3   r = 5;      // selber Effekt n = 5;
```

6.4 Zeiger und Felder

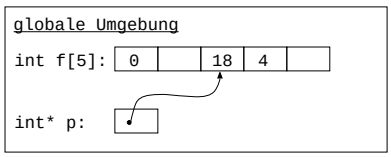
Beispiel: Zeiger und (eingebaute) Felder sind in C/C++ synonym:

```
    int f[5];
    int* p = f; // f hat Typ int*
3
    ...

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

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

Nach Marke 1:



Am Ende



Bemerkung:

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

6.5 Kommandozeilenargumente in C/C++

Um Kommandozeilenargumente zu übergeben, schreibt man die main-Funktion mit zwei Argumenten: **int** main(**int** argc, **char**** argv). Dabei speichert

- `argc` die Anzahl der Kommandozeilenargumente (inkl. Name des Programms).

- `argv` den Inhalt der Kommandozeilenargumente. `argv` ist vom Typ **char**$**$, also ein Pointer auf Pointer (Feld von Pointern).

Beispiel: Führe Programm aus: ./exampleKommandozeile 1 xz IPIWS19 xa8
Wie lauten `argc` und `argv`?

`argc = 5;`	Anzahl der Argumente inkl. Programmname gleich 5
`char[0]`	beinhaltet Name des Programms! Hier `./exampleKommandozeile`
`char[1]`	ist vom Typ **char**$*$. Zeigt auf das erste Argument, hier `1`.
`char[4]`	zeigt auf das letzte Argument, hier `xa8`.
`char[5]`	Wert des letzten Zeigers ist ein Nullpointer.

6.6 Felder als Argumente von Funktionen

```
void f( int a[10] )
{
  a[3] = 17;
}
```

ist äquivalent zu

```
void f( int* a )
{
  a[3] = 17;
}
```

(Eingebaute) Felder werden also immer by reference übergeben!

Es findet in keinem Fall eine Bereichsprüfung statt und die Länge ist in f unbekannt.

6.7 Zeiger auf zusammengesetzte Datentypen

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

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

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

6.8 Problematik von Zeigern

Beispiel: Betrachte folgendes Programm:

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

int main()
{
  char* c = alphabet();
  print(c);
}
```

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

Bemerkung:

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

6.9 Dynamische Speicherverwaltung

Bisher: Zwei Sorten von Variablen:

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

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

Beispiel:

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

Bemerkung:

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

Probleme bei dynamischen Variablen

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

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

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

```
int f()
{
  rational* p = new rational;
```

```
  p->n = 50;
  return p->n;  // Ooops, einziger Zeiger verloren
}
```

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

Abhilfe:

- Manipulation der Variablen und Zeiger in Funktionen (später: Klassen) verpacken, die eine konsistente Behandlung sicherstellen.
- Benutzung spezieller Zeigerklassen (*smart pointers*).
- Die für den Programmierer angenehmste Möglichkeit ist die Verwendung von Garbage collection (= Sammeln von nicht mehr referenziertem Speicher).

6.10 Die einfach verkettete Liste

Zeiger und dynamische Speicherverwaltung benötigt man zur Erzeugung *dynamischer Datenstrukturen*.

Dies illustrieren wir am Beispiel der einfach verketteten Liste. Das komplette Programm befindet sich in der Datei `intlist.cc`.

Eine Liste natürlicher Zahlen

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

zeichnet sich dadurch aus, dass

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

Zur Verwaltung von Listen wollen wir folgende Operationen vorsehen

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

Bemerkung: Der Hauptvorteil gegenüber dem Feld ist, dass das Einfügen und Löschen von Elementen schneller geschehen kann (es ist eine O(1)-Operation, wenn die Stelle nicht gesucht werden muss).

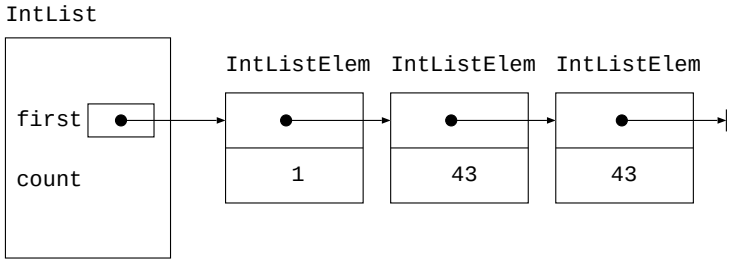
Eine übliche Methode zur Speicherung von Listen (natürlicher Zahlen) besteht darin ein *Listenelement* zu definieren, das ein Element der Liste sowie einen Zeiger auf das nächste Listenelement enthält:

```
struct IntListElem
{
  IntListElem* next;  // Zeiger auf naechstes 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 1. Element der Liste
};
```

Das sieht also so aus:



Das Ende der Liste wird durch einen Zeiger mit dem Wert 0 gekennzeichnet.

Das klappt deswegen, weil 0 kein erlaubter Ort eines Listenelementes (irgendeiner Variable) ist.

Bemerkung: Die Bedeutung von 0 ist in C/C++ mehrfach überladen. In manchen Zusammenhängen bezeichnet es die Zahl 0, an anderen Stellen einen speziellen Zeiger. Auch der bool-Wert `false` ist synonym zu 0. In C++11 gibt es nun das Schlüsselwort nullptr.

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

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

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

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

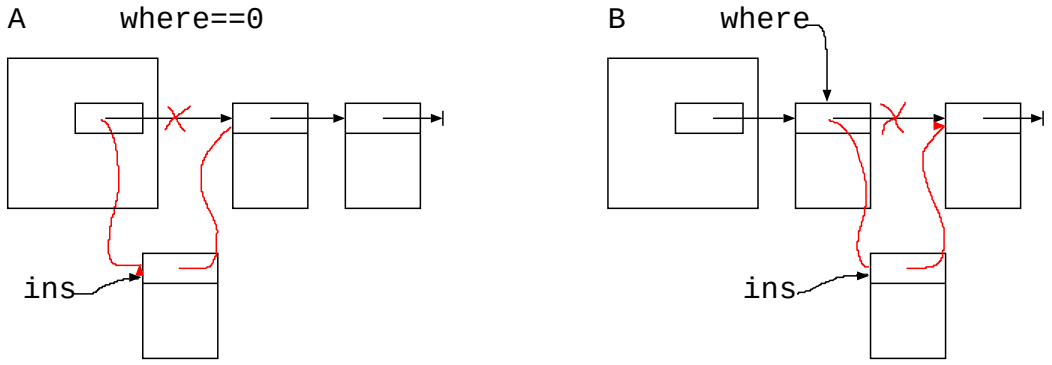
```
IntListElem* find_first_x( IntList l, int x )
{
  for ( IntListElem* p=l.first; p!=0; p=p->next )
    if ( p->value == x ) return p;
  return 0;
}
```

Einfügen Beim Einfügen von Elementen unterscheiden wir zwei Fälle:

A Am Anfang der Liste einfügen.

B *Nach* einem gegebenem Element einfügen.

Nur für diese beiden Fälle ist eine effiziente Realisierung der Einfügeoperation möglich. Es sind folgende Manipulationen der Zeiger erforderlich:



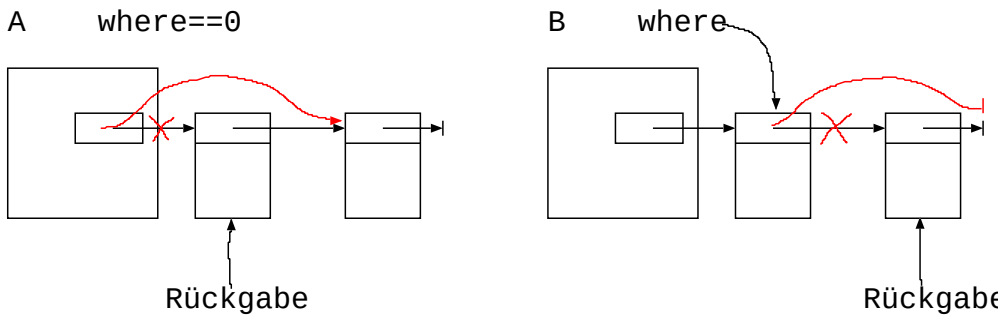
Programm: Folgende Funktion behandelt beide Fälle:

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

Entfernen Auch beim Entfernen von Elementen unterscheiden wir wieder ob

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

entsprechend graphisch:



Programm: Beide Fälle behandelt folgende Funktion:

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

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

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

Bemerkung:

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

Kritik am Programmdesign

- Ob die Anzahl der Elemente überhaupt benötigt wird, hängt von der konkreten Anwendung ab. Man hätte die Liste auch einfach als Zeiger auf Listenelemente definieren können:

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

- Man wird auch Listen anderer Typen brauchen. Dies ist erst mit den später behandelten Werkzeugen wirklich befriedigend zu erreichen (Templates). Etwas mehr Flexibilität erhielte man aber über:

```
typedef int list_element_type;
```

  und Verwendung dieses Datentyps später.

- Die Liste ist ein Spezialfall eines **Containers**. Mit der *Standard Template Library (STL)* bietet C++ eine leistungsfähige Implementierung von Containern unterschiedlicher Funktionalität.

Bemerkung: Lässt man nur das Einfügen und Löschen am **Listenanfang** zu, so implementiert die Liste das Verhalten eines Stacks mit dem Vorteil, dass man die maximale Zahl von Elementen nicht im Voraus kennen muss.

Listenvarianten

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

6.11 Endliche Menge

Im Gegensatz zu einer Liste kommt es bei einer endlichen Menge

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

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

Schnittstelle Als Operationen auf einer Menge benötigen wir

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

In der nachfolgenden Implementierung einer Menge von **int**-Zahlen enthalten die genannten Funktionen auch die Speicherverwaltung!

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

Datentyp und Initialisierung **Datentyp**: (Menge von Integer-Zahlen)

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

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

Programm: (Leere Menge)

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

Test auf Mitgliedschaft **Programm**:

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

Bemerkung:

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

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

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

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

Ausgabe **Programm**: (Ausgabe der Menge)

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

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

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

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

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

```
      IntListElem* q = remove_from_list( &s->list , p );
      delete q;
      return;
    }
}
```

Vollständiges Programm **Programm**: (`useintset.cc`)

```
#include "fcpp.hh"
#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 );
}
```

7 Klassen

7.1 Motivation

Bisher:

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

Beispiel:

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

Kritik:

- Auf welchen Daten operiert `abheben`? Es könnte mit jeder **int**-Variablen arbeiten.
- Wir könnten `konto1` auch ohne die Funktion `abheben` manipulieren.

- Nirgends ist der Zusammenhang zwischen den globalen Variablen `konto1`, `konto2` und der Funktion `abheben` erkennbar.

Idee: Verbinde Daten und Funktionen zu einer Einheit!

7.2 Klassendefinition

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

Beispiel: Klasse für das Konto:

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

Sieht einer Definition eines zusammengesetzten Datentyps sehr ähnlich.

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

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

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

Bemerkung:

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

7.3 Objektdefinition

Die Klasse kann man sich als Bauplan vorstellen. Nach diesem Bauplan werden Objekte (*objects*) erstellt, die dann im Rechner existieren. Objekte heißen auch Instanzen (*instances*) einer Klasse.

Objektdefinitionen sehen aus wie Variablendefinitionen, wobei die Klasse wie ein neuer Datentyp erscheint. Methoden werden wie Komponenten eines zusammengesetzten Datentyps selektiert und mit Argumenten wie eine Funktion versehen.

Beispiel:

```
Konto k1;
k1.abheben( 25 );

Konto* pk = &k1;
print( pk->kontostand() );
```

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

7.4 Kapselung

Der Rumpf einer Klassendefinition zerfällt in zwei Teile:

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

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

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

Bemerkung:

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

Beispiel:

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

Bemerkung: Kapselung erlaubt uns, das Prinzip der versteckten Information (*information hiding*) zu realisieren. David L. Parnas[18] [CACM, 15(12): 1059–1062, 1972] hat dieses Grundprinzip im Zusammenhang mit der modularen Programmierung so ausgedrückt:

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

[18]David Lorge Parnas, geb. 1941, kanadischer Informatiker.

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

Zitat: Brooks[19] [The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, 1975, page 102]:

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

Regel: *Halte Datenstrukturen geheim!*

Bemerkung:

- Die „Geheimhaltung“ durch die Trennung **public**/**private** ist kein perfektes Verstecken der Implementation, weil der Benutzer der Klasse ja die Klassendefinition einsehen kann/muss.
- Sie erzwingt jedoch bei gutwilligen Benutzern ein regelkonformes Verwenden der Bibliothek.
- Andererseits schützt sie nicht gegenüber böswilligen Benutzern! (z. B. sollte man nicht erwarten, dass ein Benutzer der Bibliothek ein **private**-Feld `password` nicht auslesen kann!)

7.5 Konstruktoren und Destruktoren

Objekte werden – wie jede Variable – erzeugt und zerstört, sei es automatisch oder unter Programmiererkontrolle.

Diese Momente erfordern oft spezielle Beachtung, so dass *jede* Klasse die folgenden Operationen zur Verfügung stellt:

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

Bemerkung:

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

[19] Fred Brooks, geb. 1931, amerik. Informatiker.

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

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

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

```
Konto k1( 1000 );  // Argumente des Konstruktors nach Objektname
Konto k2;  // Fehler! Klasse hat keinen argumentlosen Konstruktor
```

7.6 Implementierung der Klassenmethoden

Bisher haben wir noch nicht gezeigt, wie die Klassenmethoden implementiert werden. Dies ist Absicht, denn wir wollten deutlich machen, dass man nur die Definition einer Klasse *und die Semantik ihrer Methoden* wissen muss, um sie zu verwenden.

Nun wechseln wir auf die Seite des Implementierers einer Klasse. Hier nun ein vollständiges Programm mit Klassendefinition und Implementierung der Klasse `Konto`:

Programm: **(Konto.cc)**

```
#include "fcpp.hh"

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

Konto::Konto (int startkapital)
{
  bilanz = startkapital;
  print("Konto mit ",bilanz," eingerichtet",0);
}

Konto::~Konto ()
{
```

```
21      print("Konto mit ",bilanz," öaufgelst",0);
    }

24  int Konto::kontostand () {
      return bilanz;
    }
27
    int Konto::abheben (int betrag)
    {
30    bilanz = bilanz - betrag;
      return bilanz;
    }
33
    int main ()
    {
36    Konto k1(100), k2(200);

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

Bemerkung:

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

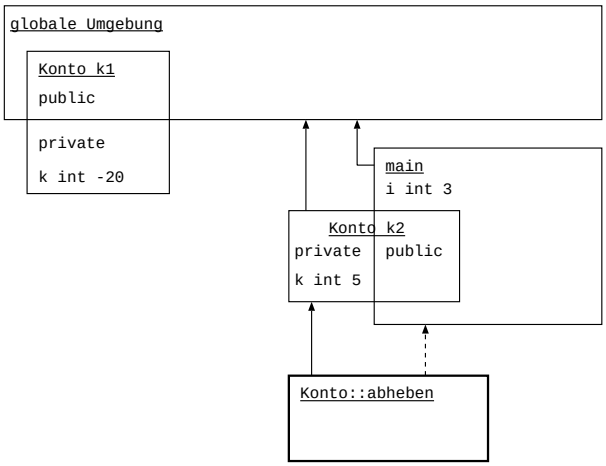
  <Klassenname>::<Methodenname>

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

7.7 Klassen im Umgebungsmodell

```
  class Konto; // wie oben

3 Konto k1( 0 );

  void main()
6 {
    int i = 3;
    Konto k2( 0 );
9
    k1.abheben( 20 );
    k2.abheben( -5 );
12 }
```

In `k2.abheben(-5)`



Bemerkung:

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

Folgerung:

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

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

7.8 Beispiel: Monte-Carlo objektorientiert

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

Bestandteile:

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

Zufallsgenerator

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

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

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

```
}

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

Vorteile:

- Durch die Angabe des Konstruktors ist sichergestellt, dass der Zufallsgenerator initialisiert werden muss. Beachte: Wenn ein Konstruktor angegeben ist, so gibt es keinen Default-Konstruktor!
- Die Realisierung des Zufallsgenerators ist nach außen nicht sichtbar (`x` ist **private**). Beispielsweise könnte man nun problemlos die Implementation so abändern, dass man intern mit längeren Zahlen arbeitet.

Klasse für das Experiment

Programm:

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

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

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

int Experiment::durchfuehren ()
```

```
  {
21  unsigned int x1 = zg.ziehe_zahl();
    unsigned int x2 = zg.ziehe_zahl();
    if (ggT(x1,x2)==1)
24    return 1;
    else
      return 0;
27  }
```

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

Monte-Carlo-Funktion und Hauptprogramm

Programm:

```
  #include "fcpp.hh"        // fuer print
  #include "Zufall.cc"      // Code fuer die beiden
3 #include "Experiment.cc" // Klassen hereinziehen

  double montecarlo (Experiment& e, int N)
6 {
    int erfolgreich=0;

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

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

15 int main (int argc, char** argv)
  {
    Zufall z(93267); // ein Zufallsgenerator
18  Experiment e(z); // ein Experiment

    print(sqrt(6.0/montecarlo(e,readarg_int(argc,argv,1))));
21 }
```

Diskussion:

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

7.9 Initialisierung von Unterobjekten

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

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

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

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

Eigenschaften:

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

Erinnerung: Eingebaute Datentypen und Zeiger haben keine Konstruktoren und werden nicht initialisiert (es sei denn man initialisiert sie explizit).

Anwendung: `Experiment` enthält eine Referenz als Unterobjekt. Mit einer Instanz der Klasse `Experiment` wird auch diese Referenz erzeugt. Referenzen müssen aber *immer* initialisiert werden, daher muss die erweiterte Form des Konstruktors benutzt werden.

Es ist in diesem Fall nicht möglich, die Referenz im Rumpf des Konstruktors zu initialisieren.

Frage: Was würde sich ändern, wenn man ein `Zufall`-Objekt statt der Referenz speichern würde?

7.10 Selbstreferenz

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

Beispiel: Folgendes Programmfragment zeigt eine gleichwertige Implementierung von `abheben`:

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

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

7.11 Überladen von Funktionen und Methoden

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

Beispiel:

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

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

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

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

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

Automatische Konversion

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

1. Versuche passende Funktion ohne Konversion oder mit trivialen Konversionen (z. B. Feldname nach Zeiger) zu finden. Man spricht von exakter Übereinstimmung. Dies sind die ersten vier Versionen oben.
2. Versuche innerhalb einer Familie von Typen ohne Informationsverlust zu konvertieren und so eine passende Funktion zu finden. Z. B. ist erlaubt, **bool** nach **int**, **short** nach **int**, **int** nach **long**, **float** nach **double**, etc. Im obigen Beispiel wird `c` in Version 5 nach **int** konvertiert.
3. Versuche Standardkonversionen (Informationsverlust!) anzuwenden: **int** nach **double**, **double** nach **int** usw.
4. Gibt es verschiedene Möglichkeiten auf *einer* der vorigen Stufen, so wird ein Fehler gemeldet.

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

Überladen von Methoden

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

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

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

Jetzt können wir ein Konto auf zwei Arten erzeugen:

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

Bemerkung:

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

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

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

7.12 Objektorientierte und funktionale Programmierung

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

Programm: (lokal erzeugte Funktion in Scheme)

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

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

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

Programm: **(Inkrementierer.cc)**

```
#include "fcpp.hh"        // fuer print

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

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

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

Bemerkung:

- Man beachte die Definition der Methoden innerhalb der Klasse. Dies ist zwar kürzer, legt aber die Implementation der Schnittstelle offen.

- Die innerhalb einer Klasse definierten Methoden werden „inline“ (d. h. ohne Funktionsaufruf) übersetzt. Bei Änderungen solcher Methoden muss daher aufrufender Code neu übersetzt werden!
- Man sollte dieses Feature daher nur mit Vorsicht verwenden (z. B. bei nur lokal verwendeten Klassen oder wenn das Inlining gewünscht wird).
- Eine erweiterte Schnittstelle zur Simulation funktionaler Programme erhält man in der STL (*Standard Template Library*) mit **#include** <functional>.

7.13 Operatoren

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

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

```
class X
{
public:
  X operator+( X b );
};

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

Bemerkung:

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

7.14 Anwendung: rationale Zahlen objektorientiert

Definition der Klasse (`Rational.hh`):

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

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

  // Ausgabe
  void print ();

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

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

```
int Rational::numerator () {
  return n;
}

int Rational::denominator () {
  return d;
}

void Rational::print () {
  ::print(n,"/",d,0);
}

// ggT zum kuerzen
int Rational::ggT (int a, int b) {
  return (b==0) ? a : ggT(b,a%b);
}

// Konstruktoren
Rational::Rational (int num, int denom)
{
  int t = ggT(num,denom);
  if (t!=0)
    {
      n=num/t;
      d=denom/t;
```

```
    }
  else
    {
      n = num;
      d = denom;
    }
}

Rational::Rational (int num) {
  n=num;
  d=1;
}

Rational::Rational () {
  n=0;
  d=1;
}

// Operatoren
Rational Rational::operator+ (Rational q) {
  return Rational(n*q.d+q.n*d,d*q.d);
}

Rational Rational::operator- (Rational q) {
  return Rational(n*q.d-q.n*d, d*q.d);
}

Rational Rational::operator* (Rational q) {
  return Rational(n*q.n, d*q.d);
}

Rational Rational::operator/ (Rational q) {
  return Rational(n*q.d,d*q.n);
}
```

Programm: Lauffähiges Beispiel (`UseRational.cc`):

```
#include "fcpp.hh"        // fuer print
#include "Rational.hh"
#include "Rational.cc"

int main () {
    Rational p(3,4), q(5,3), r;

    p.print(); q.print();
    r = (p+q*p)*p*p;
    r.print();

    return 0;
}
```

Bemerkung:

- Es ist eine gute Idee die Definition der Klasse (Schnittstelle) und die Implementierung der Methoden in getrennte Dateien zu schreiben. Dafür haben sich in C++ die Dateiendungen .hh („Headerdatei“) und .cc („Quelldatei“) eingebürgert. (Auch: .hpp, .hxx, .h, .cpp, .cxx).
- Später wird dies die sog. „getrennte Übersetzung“ ermöglichen.
- Wie schon früher erwähnt, ist die Implementierung einer leistungsfähigen gemischtzahligen Arithmetik eine hochkomplexe Aufgabe, für welche die Klasse `Rational` nur ein erster Ansatz sein kann.
- Sehr notwendig wäre auf jeden Fall die Verwendung von Ganzzahlen beliebiger Länge anstatt von **int** als Bausteine für `Rational`.

7.15 Beispiel: Turingmaschine

Ein großer Vorteil der objektorientierten Programmierung ist, dass man seine Programme sehr „problemnah“ formulieren kann. Als Beispiel zeigen wir, wie man eine Turingmaschine realisieren könnte. Diese besteht aus den drei Komponenten

- Band
- Programm
- eigentliche Turingmaschine

Es bietet sich daher an, diese Einheiten als Klassen zu definieren.

Band

Programm: **(Band.hh)**

```
// Klasse fuer ein linksseitig begrenztes Band
// einer Turingmaschine.
// Das Band wird durch eine Zeichenkette aus
// Elemente des Typs char realisiert
class Band {
public:
  // Initialisiere Band mit s, fuelle Rest
  // mit dem Zeichen init auf.
  // Setze aktuelle Bandposition auf linkes Ende.
  Band (std::string s, char init);

  // Lese Symbol unter dem Lesekopf
  char lese ();

  // Schreibe und gehe links
  void schreibe_links (char symbol);

  // Schreibe und gehe rechts
```

```
    void schreibe_rechts (char symbol);

    // Drucke aktuellen Bandinhalt bis zur
    // maximal benutzten Position
    void drucke ();
  private:
    enum {N=100000};  // maximal nutzbare Groesse
    char band[N];     // das Band
    int pos;          // aktuelle Position
    int benutzt;      // bisher beschriebener Teil
  };
```

TM-Programm

Programm: (Programm.hh)

```
  // Eine Klasse, die das Programm einer
  // Turingmaschine realisiert.
  // Zustaende sind vom Typ int
  // Bandalphabet ist der Typ char
  // Anfangszustand ist Zustand in der ersten Zeile
  // Endzustand ist Zustand in der letzten Zeile
  class Programm {
  public:
    // Symbole fuer links/rechts
    enum R {links,rechts};

    // Erzeuge leeres Programm
    Programm ();

    // definiere Zustandsuebergaenge
    // Mit Angabe des Endzustandes ist die
    // Programmierphase beendet
    void zeile (int q_ein, char s_ein,
                char s_aus, R richt, int q_aus);
    void zeile (int endzustand);

    // lese Zustandsuebergang in Abhaengigkeit
    // von akt. Zustand und gelesenem Symbol
    char Ausgabe (int zustand, char symbol);
    R Richtung (int zustand, char symbol);
    int Folgezustand (int zustand, char symbol);

    // Welcher Zustand ist Anfangszustand
    int Anfangszustand ();

    // Welcher Zustand ist Endzustand
    int Endzustand ();

```

```
  private:
    // Finde die Zeile zu geg. Zustand/Symbol
    // Liefere true, falls so eine Zeile gefunden
    // wird, sonst false
    bool FindeZeile(int zustand, char symbol);

    enum {N=1000};   // maximale Anzahl Uebergaenge
    int zeilen;      // Anzahl Zeilen in Tabelle
    bool fertig;     // Programmierphase beendet
    int Qaktuell[N]; // Eingabezustand
    char eingabe[N]; // Eingabesymbol
    char ausgabe[N]; // Ausgabesymbol
    R richtung[N];   // Ausgaberichtung
    int Qfolge[N];   // Folgezustand
    int letztesQ;    // Merke Eingabe und Zeilen-
    int letzteEingabe;// nummer des letzten Zu-
    int letzteZeile; // griffes.
  };
```

Bemerkung: Man beachte die Definition des lokalen Datentyps `R` durch `enum`. Andererseits wird eine Form von `enum`, bei der den Konstanten gleich Zahlen zugewiesen werden, verwendet, um die Konstante `N` innerhalb der Klasse `Programm` zur Verfügung zu stellen.

Turingmaschine

Programm: **(TM.hh)**

```
// Klasse, die eine Turingmaschine realisiert
class TM {
public:
  // Konstruiere Maschine mit Programm
  // und Band
  TM (Programm& p, Band& b);

  // Mache einen Schritt
  void Schritt ();

  // Liefere true falls sich Maschine im
  // Endzustand befindet
  bool Endzustand ();

private:
  Programm& prog; // Merke Programm
  Band& band;     // Merke Band
  int q;          // Merke akt. Zustand
};
```

Programm: **(TM.cc)**

```
// Konstruiere die TM mit Programm und Band
TM::TM (Programm& p, Band& b) : prog(p), band(b)
{
    q=p.Anfangszustand();
}

// einen Schritt machen
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);
}

// Ist Endzustand erreicht?
bool TM::Endzustand ()
{
    if (q==prog.Endzustand()) return true; else return false;
}
```

Turingmaschinen-Hauptprogramm

Programm: (Turingmaschine.cc)

```
#include "fcpp.hh" // fuer print

#include"Band.hh"  // Inkludiere Quelldateien
#include"Band.cc"
#include"Programm.hh"
#include"Programm.cc"
#include"TM.hh"
#include"TM.cc"

int main (int argc, char *argv[])
{
  // Initialisiere ein Band
  Band b("1111",'0');
  b.drucke();

  // Initialisiere ein Programm
  Programm p;
```

```
18      p.zeile(1,'1','X',Programm::rechts,2);
        p.zeile(2,'1','1',Programm::rechts,2);
        p.zeile(2,'0','Y',Programm::links,3);
21      p.zeile(3,'1','1',Programm::links,3);
        p.zeile(3,'X','1',Programm::rechts,4);
        p.zeile(4,'Y','1',Programm::rechts,8);
24      p.zeile(4,'1','X',Programm::rechts,5);
        p.zeile(5,'1','1',Programm::rechts,5);
        p.zeile(5,'Y','Y',Programm::rechts,6);
27      p.zeile(6,'1','1',Programm::rechts,6);
        p.zeile(6,'0','1',Programm::links,7);
        p.zeile(7,'1','1',Programm::links,7);
30      p.zeile(7,'Y','Y',Programm::links,3);
        p.zeile(8);

33      // Baue eine Turingmaschine
       TM tm(p,b);

36      // Simuliere Turingmaschine
        while (!tm.Endzustand()) { // Solange nicht Endzustand
            tm.Schritt() ;         // mache einen Schritt
39          b.drucke();            // und drucke Band
          }

42      return 0;                  // fertig.
    }
```

Die TM realisiert das Programm ,,Verdoppeln einer Einserkette" von Seite 14.

Experiment: Ausgabe des oben angegebenen Programms:

```
4  Symbole auf Band initialisiert
[1]111
Programm mit  14  Zeilen definiert
Anfangszustand  1
Endzustand  8
X[1]11
X1[1]1
X11[1]
X111[0]
X11[1]Y
X1[1]1Y
X[1]11Y
[X]111Y
1[1]11Y
1X[1]1Y
1X1[1]Y
1X11[Y]
1X11Y[0]
```

```
1X11[Y]1
1X1[1]Y1
1X[1]1Y1
1[X]11Y1
11[1]1Y1
11X[1]Y1
11X1[Y]1
11X1Y[1]
11X1Y1[0]
11X1Y[1]1
11X1[Y]11
11X[1]Y11
11[X]1Y11
111[1]Y11
111X[Y]11
111XY[1]1
111XY1[1]
111XY11[0]
111XY1[1]1
111XY[1]11
111X[Y]111
111[X]Y111
1111[Y]111
11111[1]11
```

Kritik:

- Das Band könnte seine Größe dynamisch verändern.
- Statt eines einseitig unendlichen Bandes könnten wir auch ein zweiseitig unendliches Band realisieren.
- Das Finden einer Tabellenzeile könnte durch bessere Datenstrukturen beschleunigt werden.
- Bei Fehlerzuständen bricht das Programm nicht ab. Fehlerbehandlung ist keine triviale Sache.

Aber: Diese Änderungen betreffen jeweils nur die *Implementierung* einer einzelnen Klasse (`Band` oder `Programm`) und beeinflussen die Implementierung anderer Klassen nicht!

7.16 Abstrakter Datentyp

Eng verknüpft mit dem Begriff der Schnittstelle ist das Konzept des abstrakten Datentyps (ADT). Ein ADT besteht aus

- einer Menge von Objekten, und
- einem Satz von Operationen auf dieser Menge, sowie

- einer genauen Beschreibung der Semantik der Operationen.

Bemerkung:

- Das Konzept des ADT ist unabhängig von einer Programmiersprache, die Beschreibung kann in natürlicher (oder mathematischer) Sprache abgefasst werden.
- Der ADT beschreibt, *was* die Operationen tun, aber nicht, *wie* sie das tun. Die Realisierung ist also nicht Teil des ADT!
- Die Klasse ist der Mechanismus zur Konstruktion von abstrakten Datentypen in C++. Allerdings fehlt dort die Beschreibung der Semantik der Operationen! Diese kann man als Kommentar über die Methoden schreiben.
- In manchen Sprachen (z. B. Eiffel, PLT Scheme) ist es möglich, die Semantik teilweise zu berücksichtigen (Design by Contract: zur Funktionsdefinition kann man Vorbedingungen und Nachbedingungen angeben).

Beispiel 1: Positive m-Bit-Zahlen im Computer

Der ADT „Positive m-Bit-Zahl“ besteht aus

- Der Teilmenge $P_m = \{0, 1, \ldots, 2^m - 1\}$ der natürlichen Zahlen.
- Der Operation $+_m$ so dass für $a, b \in P_m$: $a +_m b = (a + b) \bmod 2^m$.
- Der Operation $-_m$ so dass für $a, b \in P_m$: $a -_m b = ((a - b) + 2^m) \bmod 2^m$.
- Der Operation $*_m$ so dass für $a, b \in P_m$: $a *_m b = (a * b) \bmod 2^m$.
- Der Operation $/_m$ so dass für $a, b \in P_m$: $a/_m b = q$, q die größte Zahl in P_m so dass $q *_m b \leq a$.

Bemerkung:

- Die Definition dieses ADT stützt sich auf die Mathematik (natürliche Zahlen und Operationen darauf).
- In C++ (auf einer 32-Bit Maschine) entsprechen **unsigned char**, **unsigned short**, **unsigned int** den Werten $m = 8, 16, 32$.

Beispiel 2: ADT Stack

- Ein Stack S über X besteht aus einer geordneten Folge von n Elementen aus X: $S = \{s_1, s_2, \ldots, s_n\}$, $s_i \in X$. Die Menge aller Stacks $\mathcal{S}$ besteht aus *allen möglichen Folgen der Länge* $n \geq 0$.
- Operation $new : \emptyset \to \mathcal{S}$, die einen leeren Stack erzeugt.
- Operation $empty : \mathcal{S} \to \{w, f\}$, die prüft ob der Stack leer ist.
- Operation $push : \mathcal{S} \times X \to \mathcal{S}$ zum Einfügen von Elementen.
- Operation $pop : \mathcal{S} \to \mathcal{S}$ zum Entfernen von Elementen.
- Operation $top : \mathcal{S} \to X$ zum Lesen des obersten Elementes.

- Die Operationen erfüllen folgende Regeln:
  1. $empty(new()) = w$
  2. $empty(push(S, x)) = f$
  3. $top(push(S, x)) = x$
  4. $pop(push(S, x)) = S$

Bemerkung:

- Die einzige Möglichkeit einen Stack zu erzeugen ist die Operation *new*.
- Die Regeln erlauben uns formal zu zeigen, welches Element nach einer beliebigen Folge von *push* und *pop* Operationen zuoberst im Stack ist:

$$top(pop(push(push(push(new(), x_1), x_2), x_3))) =$$

$$top(push(push(new(), x_1), x_2)) = x_2$$

- Auch nicht gültige Folgen lassen sich erkennen:

$$pop(pop(push(new(), x_1))) = pop(new())$$

  und dafür gibt es keine Regel!

Bemerkung: Abstrakte Datentypen, wie Stack, die Elemente einer Menge X aufnehmen, heißen auch Container. Wir werden noch eine Reihe von Containern kennenlernen: Feld, Liste (in Varianten), Queue, usw.

Beispiel 3: Das Feld

Wie beim Stack wird das Feld über einer Grundmenge X erklärt. Auch das Feld ist ein Container.

Das charakteristische an einem Feld ist der indizierte Zugriff. Wir können das Feld daher als eine Abbildung einer Indexmenge $I \subset \mathbb{N}$ in die Grundmenge X auffassen.

Die *Indexmenge* $I \subseteq \mathbb{N}$ sei beliebig, aber im folgenden fest gewählt. Zur Abfrage der Indexmenge gebe es folgende Operationen:

- Operation *min* liefert kleinsten Index in I.
- Operation *max* liefert größten Index in I.
- Operation $isMember : \mathbb{N} \to \{w, f\}$. $isMember(i)$ liefert wahr falls $i \in I$, ansonsten falsch.

Den ADT Feld definieren wir folgendermaßen:

- Ein Feld f ist eine Abbildung der Indexmenge I in die Menge der möglichen Werte X, d. h. $f : I \to X$. Die Menge aller Felder $\mathcal{F}$ ist die Menge aller solcher Abbildungen.
- Operation $new : X \to \mathcal{F}$. $new(x)$ erzeugt neues Feld mit Indexmenge I (und initialisiert mit x, siehe unten).

- Operation $read : \mathcal{F} \times I \to X$ zum Auswerten der Abbildung.
- Operation $write : \mathcal{F} \times I \times X \to \mathcal{F}$ zum Manipulieren der Abbildung.
- Die Operationen erfüllen folgende Regeln:
  1. $read(new(x), i) = x$ für alle $i \in I$.
  2. $read(write(f, i, x), i) = x$.
  3. $read(write(f, i, x), j) = read(f, j)$ für $i \neq j$.

Bemerkung:

- In unserer Definition darf $I \subset \mathbb{N}$ beliebig aber fest gewählt werden. Es sind also auch nichtzusammenhängende Indexmengen erlaubt.
- Als Variante könnte man die Manipulation der Indexmenge erlauben (die Indexmenge sollte dann als weiterer ADT definiert werden).

8 Klassen und dynamische Speicherverwaltung

Erinnerung: Nachteile von eingebauten Feldern in C/C++:

- Ein eingebautes Feld kennt seine Größe nicht, diese muss immer extra mitgeführt werden, was ein Konsistenzproblem mit sich bringt.
- Bei dynamischen Feldern ist der Programmierer für die Freigabe des Speicherplatzes verantwortlich.
- Eingebaute Felder sind äquivalent zu Zeigern und können daher nur *by reference* übergeben werden.
- Eingebaute Felder prüfen nicht, ob der Index im erlaubten Bereich liegt.
- Manchmal bräuchte man Verallgemeinerungen, z. B. andere Indexmengen.

8.1 Klassendefinition

Unsere Feldklasse soll Elemente des Grundtyps **float** aufnehmen. Hier ist die Klassendefinition:

Programm: **(SimpleFloatArray.hh)**

```
class SimpleFloatArray {
public:
  // Neues Feld mit s Elementen, I=[0,s-1]
  SimpleFloatArray (int s, float f);

  // Copy-Konstruktor
  SimpleFloatArray (const SimpleFloatArray&);

  // Zuweisung von Feldern
```

```
    SimpleFloatArray& operator= (const SimpleFloatArray&);

    // Destruktor: Gebe Speicher frei
    ~SimpleFloatArray();

    // Indizierter Zugriff auf Feldelemente
    // keine Ueberpruefung ob Index erlaubt
    virtual float& operator[](int i);

    // Anzahl der Indizes in der Indexmenge
    int numIndices ();

    // kleinster Index
    int minIndex ();

    // ößgrter Index
    int maxIndex ();

    // Ist der Index in der Indexmenge?
    bool isMember (int i);

  private:
    int n;       // Anzahl Elemente
    float *p;    // Zeiger auf built-in array
};
```

Bemerkung: Man beachte, dass diese Implementierung das eingebaute Feld nutzt.

8.2 Konstruktor

Programm: **(SimpleFloatArrayImp.cc)**

```
SimpleFloatArray::SimpleFloatArray (int s,
                                    float v)
{
  n = s;
  try {
    p = new float[n];
  }
  catch (std::bad_alloc) {
    n = 0;
    throw;
  }
  for (int i=0; i<n; i=i+1) p[i]=v;
}

SimpleFloatArray::~SimpleFloatArray () { delete[] p; }

int SimpleFloatArray::numIndices () { return n; }
```

```
18
   int SimpleFloatArray::minIndex () { return 0; }

21 int SimpleFloatArray::maxIndex () { return n-1; }

   bool SimpleFloatArray::isMember (int i)
24 {
     return (i>=0 && i<n);
   }
```

Ausnahmen

Bemerkung:

- Oben kann in der Operation **new** das Ereignis eintreten, dass nicht genug Speicher vorhanden ist. Dann setzt diese Operation eine sogenannte Ausnahme (*exception*), die in der **catch**-Anweisung abgefangen wird.
- *Gute* Fehlerbehandlung (Reaktionen auf Ausnahmen) ist in einem großen, professionellen Programm sehr wichtig!
- *Die Schwierigkeit ist, dass man oft an der Stelle des Erkennens des Ereignisses nicht weiss, wie man darauf reagieren soll.*
- Hier wird in dem Objekt vermerkt, dass das Feld die Größe 0 hat und auf eine Fehlerbehandlung an anderer Stelle verwiesen.

8.3 Indizierter Zugriff

Erinnerung: Die Operationen *read* und *write* des ADT Feld werden bei eingebauten Feldern durch den Operator `[]` und die Zuweisung realisiert:

```
x = 3 * a[i] + 17.5;
a[i] = 3 * x + 17.5;
```

Unsere neue Klasse soll sich in dieser Beziehung wie ein eingebautes Feld verhalten. Dies gelingt durch die Definition eines Operators `operator[]`:

Programm: **(SimpleFloatArrayIndex.cc)**

```
   float& SimpleFloatArray::operator[] (int i)
   {
3    return p[i];
   }
```

Bemerkung:

- `a[i]` bedeutet, dass der `operator[]` von `a` mit dem Argument `i` aufgerufen wird.

- Der Rückgabewert von `operator[]` muss eine Referenz sein, damit `a[i]` auf der linken Seite der Zuweisung stehen kann. Wir wollen ja das i-te Element des Feldes verändern und keine Kopie davon.

8.4 Copy–Konstruktor

Schließlich ist zu klären, was beim Kopieren von Feldern passieren soll. Hier sind zwei Situationen zu unterscheiden:

1. Es wird ein *neues* Objekt erzeugt welches mit einem existierenden Objekt initialisiert wird. 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); SimpleFloatArray a = b;
     ```

2. Kopieren eines Objektes *auf ein bereits existierendes Objekt*, das ist die Zuweisung.

Im ersten Fall wird von C++ der sogenannte Copy-Konstruktor aufgerufen. Ein Copy-Konstruktor ist ein Konstruktor der Gestalt

<Klassenname> (const <Klassenname> &);

Als Argument wird also eine Referenz auf ein Objekt desselben Typs übergeben. Dabei bedeutet **const**, dass das Argumentobjekt nicht manipuliert werden darf.

Programm: **(SimpleFloatArrayCopyCons.cc)**

```
SimpleFloatArray::SimpleFloatArray (const SimpleFloatArray& a) {
  n = a.n;
  p = new float[n];
  for (int i=0; i<n; i=i+1)
    p[i]=a.p[i];
}
```

Bemerkung:

- Unser Copy-Konstruktor allokiert ein neues Feld und kopiert alle Elemente des Argumentfeldes.
- Damit gibt es *immer nur jeweils einen Zeiger auf ein dynamisch erzeugtes, eingebautes Feld*. Der Destruktor kann dieses eingebaute Feld gefahrlos löschen!

Beispiel:

```
int f()
{
  SimpleFloatArray a( 100, 0.0 ); // Feld mit 100 Elem.
  SimpleFloatArray b = a;         // Aufruf Copy-Konstr.
  ... // mach etwas schlaues
} // Destruktoren rufen delete[] ihres eingeb. Feldes auf
```

Bemerkung:

- Hier hat man mit der dynamischen Speicherverwaltung der eingebauten Felder nichts mehr zu tun, und es können auch keine Fehler passieren.
- Dieses Verhalten des Copy-Konstruktors nennt man *deep copy*.
- Alternativ könnte der Copy-Konstruktor nur den Zeiger in das neue Objekt kopieren (*shallow copy*). Hier dürfte der Destruktor das Feld aber nicht einfach freigeben, weil noch Referenzen bestehen könnten! (Abhilfen: *reference counting*, *garbage collection*)

8.5 Zuweisungsoperator

Bei einer Zuweisung `a = b` soll das Objekt rechts des =-Zeichens auf das *bereits initialisierte* Objekt links des =-Zeichens kopiert werden. In diesem Fall ruft C++ den `operator=` des links stehenden Objektes mit dem rechts stehenden Objekt als Argument auf.

Programm: **(SimpleFloatArrayAssign.cc)**

```
SimpleFloatArray& SimpleFloatArray::operator=
(const SimpleFloatArray& a)
{
  // nur bei verschiedenen Objekten ist was tun
  if (&a!=this)
    {
      if (n!=a.n) {
        // allokiere fuer this ein
        // Feld der Groesse a.n
        delete[] p; // altes Feld loeschen
        n = a.n;
        p = new float[n]; // keine Fehlerbeh.
      }
      for (int i=0; i<n; i=i+1) p[i]=a.p[i];
    }

  // Gebe Referenz zurueck damit a=b=c klappt
  return *this;
}
```

Bemerkung:

- Haben beide Felder unterschiedliche Größe, so wird für das Feld links vom Zuweisungszeichen ein neues eingebautes Feld der korrekten Größe erzeugt.

- Der Zuweisungsoperator ist in C/C++ so definiert, dass er gleichzeitig den zugewiesenen Wert hat. Somit werden Ausdrücke wie `a = b = 0` oder `return tabelle[i] = n` möglich.

8.6 Hauptprogramm

Programm: **(UseSimpleFloatArray.cc)**

```
#include <iostream>
#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

void show (SimpleFloatArray f) {
  std::cout << "#( ";
  for (int i=f.minIndex(); i<=f.maxIndex(); i++)
    std::cout << f[i] << " ";
  std::cout << ")" << std::endl;
}

int main () {
  SimpleFloatArray a(10,0.0); // erzeuge Felder
  SimpleFloatArray b(5,5.0);
  SimpleFloatArray c(b); // copy constructor

  for (int i=a.minIndex(); i<=a.maxIndex(); i++)
    a[i] = i*b[i];

  show(a); // call by value, ruft Copy-Konstruktor
  b = a;   // ruft operator= von b
  show(b);

  // hier wird der Destruktor beider Objekte gerufen
}
```

Bemerkung:

- Jeder Aufruf der Funktion `show` kopiert das Argument mittels des Copy-Konstruktors. (Für Demonstrationszwecke: eigentlich sollte man in `show` eine Referenz verwenden!)
- Entscheidend ist, dass der Benutzer gar nicht mehr mit dynamischer Speicherverwaltung konfrontiert wird.
- Hier wird erstmals „richtige“ C++ Ausgabe verwendet. Das werden wir noch behandeln.

8.7 Default-Methoden

Für folgende Methoden einer Klasse `T` erzeugt der Übersetzer automatisch Default-Methoden, sofern man keine eigenen definiert:

- Argumentloser Konstruktor `T();`
  Dieser wird erzeugt, wenn man keinen anderen Konstruktor außer dem Copy-Konstruktor angibt. Hierarchische Konstruktion von Unterobjekten.
- Copy-Konstruktor `T( const T& );`
  Kopiert alle Mitglieder in das neue Objekt (*memberwise copy*) unter Benutzung von deren Copy-Konstruktor.
- Destruktor `~T();` Hierarchische Destruktion von Unterobjekten.
- Zuweisungsoperator `T& operator=( const T& );`
  Kopiert alle Mitglieder des Quellobjektes auf das Zielobjekt unter Nutzung der jeweiligen Zuweisungsoperatoren.
- Adress-of-Operator (`&`) mit Standardbedeutung.

Bemerkung:

- Der Konstruktor (ob default oder selbstdefiniert) ruft rekursiv die Konstruktoren von selbstdefinierten Unterobjekten auf.
- Ebenso der Destruktor.
- Enthält ein Objekt Zeiger auf andere Objekte und ist für deren Speicherverwaltung verantwortlich, so wird man wahrscheinlich alle oben genannten Methoden speziell schreiben müssen (außer dem `&`-Operator). Die Klasse `SimpleFloatArray` illustriert dies.

8.8 C++ Ein- und Ausgabe

Eingabe von Daten in ein Programm sowie deren Ausgabe ist ein elementarer Aspekt von Programmen.

Wir haben diesen Aspekt bis jetzt verschoben nicht weil er unwichtig wäre, sondern weil sich die entsprechenden Konstrukte in C++ nur im Kontext von Klassen und Operatoren verstehen lassen.

Jedoch werden wir hier nur die Ein- und Ausgabe von Zeichen, insbesondere auf eine Konsole, betrachten. Dies lässt sich leicht auf Dateien erweitern.

Graphische Ein- und Ausgabe werden wir aus Zeitgründen nicht betrachten. Allerdings wurde die Programmierung von graphischen Benutzerschnittstellen durch objektorientierte Programmierung revolutioniert und C++ ist dafür gut geeignet.

Namensbereiche

Zuvor benötigen wir noch das für große Programme wichtige Konstrukt der Namensbereiche welches auch in der Standardbibliothek verwendet wird.

In der globalen Umgebung darf jeder Name höchstens einmal vorkommen. Dabei ist egal, ob es sich um Namen für Variablen, Klassen oder Funktionen handelt.

Damit ergibt sich insbesondere ein Problem, wenn zwei Bibliotheken die gleichen Namen verwenden.

Eine Bibliothek ist eine Sammlung von Klassen und/oder Funktionen, die einem bestimmten Zweck dienen und von einem Programmierer zur Verfügung gestellt werden. Eine Bibliothek enthält keine main-Funktion!

Mittels Namensbereichen lässt sich dieses Problem lösen.

```
namespace A
{
  int n = 1;
  int m = n;
}

namespace B
{
  int n = 2;
  class X {};
}

int main()
{
  A::n = B::n + 3;
  return A::n;
}
```

Mittels **namespace** werden ähnlich einem Block Unterumgebungen innerhalb der globalen Umgebung geschaffen. Allerdings existieren diese Umgebungen gleichzeitig und sind auch nur innerhalb der globalen Umgebung möglich.

Namen innerhalb eines Namensbereiches werden von ausserhalb durch Voranstellen des Namens des Namensraumes und dem :: angesprochen (Qualifizierung).

Innerhalb des Namensraumes ist keine Qualifizierung erforderlich. Mittels **using** kann man sich die Qualifizierung innerhalb eines Blockes sparen.

Namensräume können wieder Namensräume enthalten.

Elemente eines Namensraumes können an verschiedenen Stellen, sogar in verschiedenen Dateien definiert werden. Ein Name darf aber innerhalb eines Namensraumes nur einmal vorkommen.

```
namespace C
{
  double x;
  int f( double x ) { return x; } // eine Funktion
  namespace D
  {
    double x, y; // x verdeckt das x in C
  }
```

```
9  }

   namespace C
12 { // fuege weitere Namen hinzu
     double y;
   }
15
   int main()
   {
18   C::x = 0.0; C::y = 1.0; C::D::y = 2.0;
     C::f( 2.0 );
     return 0;
21 }
```

Ein- und Ausgabe mit Streams

Für die Ein- und Ausgabe stellt C++ eine Reihe von Klassen und globale Variablen in der Standardbibliothek zur Verfügung. Ein- und Ausgabe ist also kein Teil der Sprache C++ selbst.

Alle Variablen, Funktionen und Klassen der C++-Standardbibliothek sind innerhalb des Namensraumes `std` definiert.

Grundlegend für die Ein- und Ausgabe in C++ ist die Idee eines Datenstromes. Dabei unterscheidet man Eingabe- und Ausgabeströme:

- Ein Ausgabestrom ist ein Objekt in welches man Datenelemente hineinsteckt. Diese werden dann an den gewünschten Ort weitergeleitet, etwa den Bildschirm oder eine Datei.
- Ein Eingabestrom ist ein Objekt aus welchem man Datenelemente herausholen kann. Diese kommen von einem gewünschten Ort, etwa der Tastatur oder einer Datei.

Datenströme werden mittels Klassen realisiert:

- `std::istream` realisiert Eingabeströme.
- `std::ostream` realisiert Ausgabeströme.

Um diese zu verwenden, muss der Header `iostream` eingebunden werden.

Die Ein-/Ausgabe wird mittels überladener Methoden realisiert:

- **operator**>> für die Eingabe.
- **operator**<< für die Ausgabe.

Zur Ein- und Ausgabe auf der Konsole sind globale Variablen vordefiniert:

- `std::cin` vom Typ `std::istream` für die Eingabe.
- `std::cout` vom Typ `std::ostream` für die reguläre Ausgabe eines Programmes.
- `std::cerr` vom Typ `std::ostream` für die Ausgabe von Fehlern.

Damit sind wir bereit für ein Beispiel.

Programm: **(iostreamexample.cc)**

```
#include <iostream>

int main ()
{
  int n;
  std::cin >> n;  // d.h. cin.operator>>(n);
  double x;
  std::cin >> x;  // d.h. cin.operator>>(x);
  std::cout << n; // d.h. cout.operator<<(n);
  std::cout << " ";
  std::cout << x;
  std::cout << std::endl; // neue Zeile
  std::cout << n << " " << x << std::endl;
  return 0;
}
```

Die Ausgabe mehrerer Objekte innerhalb einer Anweisung gelingt dadurch, dass die Methode **operator**<< den Stream, also sich selbst, als Ergebnis zurückliefert:

```
std::cout << n << std::endl;
```

ist dasselbe wie

```
( std::cout.operator<<( n ) ).operator<<( std::endl );
```

Die Methoden **operator**>> und **operator**<< sind für alle eingebauten Datentypen wie **int** oder **double** überladen.

Durch Überladen der Funktion

std::ostream& **operator**<<(std::ostream&, <Typ>);

kann man obige Form der Ausgabe für selbstgeschriebene Klassen ermöglichen.

Als Beispiel betrachten wir eine Ausgabefunktion für die Klasse Rational:

Programm: **(RationalOutput.cc)**

```
std::ostream&
operator<< (std::ostream& s, Rational q)
{
  s << q.numerator() << "/" << q.denominator();
  return s;
}
```

Beachte, dass das Streamargument zurückgegeben wird, um die Hintereinanderausführung zu ermöglichen.

In einer „richtigen“ Version würde man die rationale Zahl als **const** Rational& q übergeben um die Kopie zu sparen. Dies würde allerdings erfordern, die Methoden numerator und denominator als **const** zu deklarieren.

Schließlich können wir damit schreiben:

Programm: **(UseRationalOutput.cc)**

```
#include<iostream>
#include "fcpp.hh"          // fuer print
#include "Rational.hh"
#include "Rational.cc"
#include "RationalOutput.cc"

int main () {
  Rational p(3,4), q(5,3);

  std::cout << p << " " << q << std::endl;
  std::cout << (p+q*p)*p*p << std::endl;
  return 0;
}
```

9 Vererbung

9.1 Motivation: Polynome

Definition: Ein Polynom $p : \mathbb{R} \to \mathbb{R}$ ist eine Funktion der Form

$$p(x) = \sum_{i=0}^{n} p_i x^i,$$

Wir betrachten hier nur den Fall reellwertiger Koeffizienten $p_i \in \mathbb{R}$ und verlangen $p_n \neq 0$. n heißt dann Grad des Polynoms.

Operationen:

- Konstruktion.
- Manipulation der Koeffizienten.
- Auswerten des Polynoms an einer Stelle x.
- Addition zweier Polynome

$$p(x) = \sum_{i=0}^{n} p_i x^i, \quad q(x) = \sum_{j=0}^{m} q_j x^j$$

$$r(x) = p(x) + q(x) = \sum_{i=0}^{\max(n,m)} \underbrace{(p_i^* + q_i^*)}_{r_i} x^i$$

$$p_i^* = \begin{cases} p_i & i \leq n \\ 0 & \text{sonst} \end{cases}, \quad q_i^* = \begin{cases} q_i & i \leq m \\ 0 & \text{sonst} \end{cases}.$$

- Multiplikation zweier Polynome

$$\begin{aligned} r(x) = p(x) * q(x) &= \left(\sum_{i=0}^{n} p_i x^i\right)\left(\sum_{j=0}^{m} q_j x^j\right) \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} p_i q_j x^{i+j} \\ &= \sum_{k=0}^{m+n} \underbrace{\left(\sum_{\{(i,j)|i+j=k\}} p_i q_j\right)}_{r_k} x^k \end{aligned}$$

9.2 Implementation

In großen Programmen möchte man Codeduplizierung vermeiden und möglichst viel Code wiederverwenden.

Für den Koeffizientenvektor $p_0, \ldots, p_n$ wäre offensichtlich ein Feld der adäquate Datentyp. Wir wollen unser Feld `SimpleFloatArray` benutzen. Eine Möglichkeit:

Programm:

```
class Polynomial
{
private:
  SimpleFloatArray coefficients;
public:
   ...
};
```

Alternativ kann man Polynome als Exemplare von `SimpleFloatArray` mit zusätzlichen Eigenschaften ansehen, was im folgenden ausgeführt wird.

9.3 Öffentliche Vererbung

Programm: Definition der Klasse `Polynomial` mittels Vererbung:
(Polynomial.hh)

```
class Polynomial :
  public SimpleFloatArray {
public:
  // konstruiere Polynom vom Grad n
  Polynomial (int n);
```

```
6
    // Default-Destruktor ist ok
    // Default-Copy-Konstruktor ist ok
9   // Default-Zuweisung ist ok

    // Grad des Polynoms
12  int degree ();

    // Auswertung
15  float eval (float x);

    // Addition von Polynomen
18  Polynomial operator+ (Polynomial q);

    // Multiplikation von Polynomen
21  Polynomial operator* (Polynomial q);

    // Gleichheit
24  bool operator== (Polynomial q);

    // drucke Polynom
27  void print ();
};
```

Syntax: Die Syntax der *öffentlichen Vererbung* lautet:

<ÖAbleitung> ::= class <Klasse2> : public <Klasse1> {
<Rumpf>
};

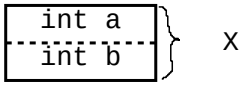
Bemerkung:

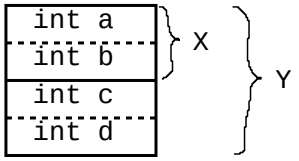
- Klasse 1 heißt Basisklasse, Klasse 2 heißt abgeleitete Klasse.
- Klasse 2 enthält ein Objekt von Klasse 1 als Unterobjekt.
- Alle *öffentlichen Mitglieder* der Basisklasse mit Ausnahme von Konstruktoren, Destruktor und Zuweisungsoperatoren sind auch öffentliche Mitglieder der abgeleiteten Klasse. Sie operieren auf dem Unterobjekt.
- Im Rumpf kann Klasse 2 weitere Mitglieder vereinbaren.
- Daher spricht man auch von einer Erweiterung einer Klasse durch *öffentliche Ableitung*.
- Alle privaten Mitglieder der Basisklasse sind *keine* Mitglieder der Klasse 2. Damit haben auch Methoden der abgeleiteten Klasse *keinen* Zugriff auf private Mitglieder der Basisklasse.
- Eine Klasse kann mehrere Basisklassen haben (Mehrfachvererbung), diesen Fall behandeln wir hier aber nicht.

9.4 Beispiel zu public/private und öffentlicher Vererbung

```
class X
{
public:
  int a;
  void A();
private:
  int b;
  void B();
};
```

```
class Y : public X
{
public:
  int c;
  void C();
private:
  int d;
  void D();
};
```






```
X x;

x.a = 5;     // OK
x.b = 10;    // Fehler

void X::A()
{
  B();       // OK
  b = 3;     // OK
}
```

```
Y y;
y.a = 1; // OK
y.c = 2; // OK
y.b = 4; // Fehler
y.d = 8; // Fehler

void Y::C() {
  d = 8; // OK
  b = 4; // Fehler
  A();   // OK
  B();   // Fehler
}
```

9.5 Ist-ein-Beziehung

Ein Objekt einer abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.

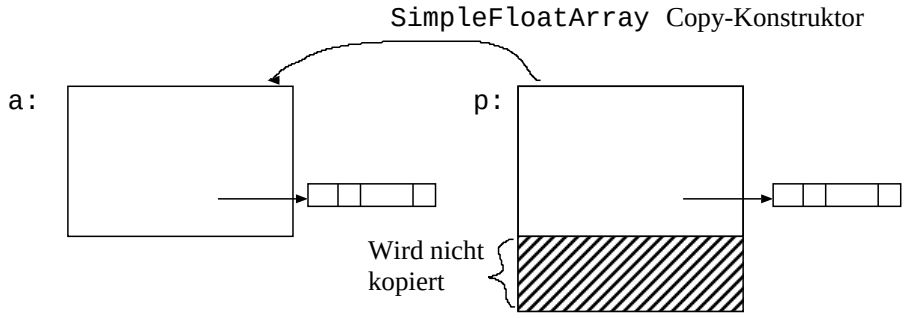
Daher darf ein Objekt der abgeleiteten Klasse für ein Objekt der Basisklasse eingesetzt werden. Allerdings sind dann nur Methoden der Basisklasse für das Objekt aufrufbar.

Beispiel:

```
void g( SimpleFloatArray a ) { a[3] = 1.0; }

Polynomial p( 10 );
SimpleFloatArray b( 100, 0.0 );
g( p );  // (1) OK
p = b;   // (2) Fehler
b = p;   // (3) OK
```

Bemerkung:

- Im Fall (1) wird bei Aufruf von `g(p)` der Copy-Konstruktor des formalen Parameters `a`, also `SimpleFloatArray`, benutzt, um das `SimpleFloatArray`-Unterobjekt von `p` auf den formalen Parameter `a` vom Typ `SimpleFloatArray` zu kopieren.
- Falls `Polynomial` weitere Datenmitglieder hätte, so würde die Situation so aussehen:



  In diesem Fall spricht man von *slicing*.
- Im Fall (2) soll einem Objekt der abgeleiteten Klasse ein Objekt der Basisklasse zugewiesen werden. Dies ist nicht erlaubt, da nicht klar ist, welchen Wert etwaige zusätzliche Datenmitglieder der abgeleiteten Klasse bekommen sollen.
- Fall (3) ist OK, der Zuweisungsoperator der Basisklasse wird aufgerufen und das Unterobjekt aus der abgeleiteten Klasse dem links stehenden Objekt der Basisklasse zugewiesen.

9.6 Konstruktoren, Destruktor und Zuweisungsoperatoren

Programm: **(PolynomialKons.cc)**

```
Polynomial::Polynomial (int n)
  : SimpleFloatArray(n+1,0.0) {}
```

Bemerkung:

- Die syntaktische Form entspricht der Initialisierung von Unterobjekten wie oben beschrieben.
- Die Implementierung des Copy-Konstruktors kann man sich sparen, da der Default-Copy-Konstruktor das Gewünschte leistet, dasselbe gilt für Zuweisungsoperator und Destruktor.

9.7 Auswertung

Programm: Auswertung mit Horner-Schema **(PolynomialEval.cc)**

```
// Auswertung
float Polynomial::eval (float x)
{
  float sum=0.0;
```

```
6       // Hornerschema
        for (int i=maxIndex(); i>=0; i=i-1)
          sum = sum*x + operator[](i);
9       return sum;
      }
```

Bemerkung: Statt **operator**[](i) könnte man (∗**this**)[i] schreiben.

9.8 Weitere Methoden

Programm: **(PolynomialImp.cc)**

```
      // Grad auswerten
      int Polynomial::degree ()
3     {
        return maxIndex();
      }
6
      // Addition von Polynomen
      Polynomial Polynomial::operator+ (Polynomial q)
9     {
        int nr=degree(); // mein grad

12      if (q.degree()>nr) nr=q.degree();

        Polynomial r(nr); // Ergebnispolynom
15
        for (int i=0; i<=nr; i=i+1)
          {
18          if (i<=degree())
              r[i] = r[i]+(*this)[i]; // add me to r
            if (i<=q.degree())
21            r[i] = r[i]+q[i];        // add q to r
          }

24      return r;
      }

27    // Multiplikation von Polynomen
      Polynomial Polynomial::operator* (Polynomial q)
      {
30      Polynomial r(degree()+q.degree()); // Ergebnispolynom

        for (int i=0; i<=degree(); i=i+1)
33        for (int j=0; j<=q.degree(); j=j+1)
            r[i+j] = r[i+j] + (*this)[i]*q[j];

36      return r;
```

```
  }

39  // Drucken
  void Polynomial::print ()
  {
42    if (degree()<0)
        std::cout << 0;
      else
45      std::cout << (*this)[0];

      for (int i=1; i<=maxIndex(); i=i+1)
48      std::cout << "+" << (*this)[i] << "*x^" << i;

      std::cout << std::endl;
51  }
```

9.9 Gleichheit

Gleichheit ist kein einfaches Konzept, wie man ja schon an Zahlen sieht: ist `0 == 0.0`? Oder `(int) 1000000000 == (short) 1000000000`? Gleichheit für selbstdefinierte Datentypen ist daher Sache des Programmierers:

Programm: **(PolynomialEqual.cc)**

```
  bool Polynomial::operator== (Polynomial q)
  {
3   if (q.degree()>degree())
      {
        for (int i=0; i<=degree(); i=i+1)
6         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;
9     }
    else
      {
12      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)
15        if ((*this)[i]!=0.0) return false;
      }

18  return true;
  }
```

Bemerkung: Im Gegensatz dazu ist Gleichheit von Zeigern immer definiert. Zwei Zeiger sind gleich, wenn sie auf *dasselbe* Objekt zeigen:

```
Polynomial p( 10 ), q( 10 );
```

```
Polynomial* z1 = &p;
Polynomial* z2 = &p;
Polynomial* z3 = &q;

if ( z1 == z2 ) ... // ist wahr
if ( z1 == z3 ) ... // ist falsch
```

9.10 Benutzung von `Polynomial`

Folgendes Beispiel definiert das Polynom

$$p = 1 + x$$

und druckt p, p^2 und p^3.

Programm: **(UsePolynomial.cc)**

```
#include<iostream>

// alles zum SimpleFloatArray
#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// Das Polynom
#include "Polynomial.hh"
#include "PolynomialImp.cc"
#include "PolynomialKons.cc"
#include "PolynomialEqual.cc"
#include "PolynomialEval.cc"

int main ()
{
  Polynomial p(2),q(10);

  p[0] = 1.0;
  p[1] = 1.0;
  p.print();

  q = p*p;
  q.print();

  q = p*p*p;
  q = q*q;
  q = q*q;
  q = q*q;
  q.print();
```

```
33    }
```

mit der Ausgabe:

```
1+1*x^1
1+2*x^1+1*x^2
1+3*x^1+3*x^2+1*x^3
```

9.11 Diskussion

- Diese Implementation hat die wesentliche Schwachstelle, dass der Grad bei führenden Nullen mathematisch nicht korrekt ist. Angenehmer wäre, wenn Konstanten den Grad 0 hätten, lineare Polynome den Grad 1 und der Koeffizientenvektor allgemein die Länge Grad+1 hätte.
- Die Abhilfe könnte darin bestehen, dafür zu sorgen, dass der Konstruktor nur Polynome mit korrektem Grad erzeugt. Allerdings können Polynome ja beliebig modifiziert werden, daher wäre eine andere Möglichkeit in der Methode `degree` den Grad jeweils aus den Koeffizienten zu bestimmen.

9.12 Private Vererbung

Wenn man nur die Implementierung von `SimpleFloatArray` nutzen will, ohne die Methoden öffentlich zu machen, so kann man dies durch private Vererbung erreichen:

Programm:

```
class Polynomial : private SimpleFloatArray
{
public:
  ...
};
```

Bemerkung: Hier müsste man dann die Schnittstelle zum Zugriff auf die Koeffizienten des Polynoms selbst Programmieren, oder zumindest die Initialisierung mittels

Polynomial::Polynomial(SimpleFloatArray& coeffs) { ... }

Eigenschaften der privaten Vererbung

Bemerkung: Private Vererbung bedeutet:

- Ein Objekt der abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.
- Alle *öffentlichen* Mitglieder der Basisklasse werden *private* Mitglieder der abgeleiteten Klasse.

- Alle privaten Mitglieder der Basisklasse sind keine Mitglieder der abgeleiteten Klasse.
- Ein Objekt der abgeleiteten Klasse kann *nicht* für ein Objekt der Basisklasse eingesetzt werden!

Zusammenfassung Wir haben somit drei verschiedene Möglichkeiten kennengelernt, um die Klasse `SimpleFloatArray` für `Polynomial` zu nutzen:

1. Als privates Datenmitglied
2. Mittels öffentlicher Vererbung
3. Mittels privater Vererbung

Bemerkung: Je nach Situation ist die eine oder andere Variante angemessener. Hier hängt viel vom guten Geschmack des Programmierers ab. In diesem speziellen Fall würde ich persönlich Möglichkeit 1 bevorzugen.

9.13 Methodenauswahl und virtuelle Funktionen

Motivation: Feld mit Bereichsprüfung

Problem: Die für die Klasse `SimpleFloatArray` implementierte Methode `operator[]` prüft nicht, ob der Index im erlaubten Bereich liegt. Zumindest in der Entwicklungsphase eines Programmes wäre es aber nützlich, ein Feld mit Indexüberprüfung zu haben.

Abhilfe: Ableitung einer Klasse `CheckedSimpleFloatArray`, bei der sich `operator[]` anders verhält.

Programm: Klassendefinition **(CheckedSimpleFloatArray.hh)**:

```
class CheckedSimpleFloatArray :
  public SimpleFloatArray {
public:
  CheckedSimpleFloatArray (int s, float f);

  // Default-Versionen von copy Konstruktor, Zuweisungsoperator
  // und Destruktor sind OK

  // Indizierter Zugriff mit üIndexprfung
  float& operator[](int i);
};
```

Methodendefinition **(CheckedSimpleFloatArrayImp.cc)**:

```
CheckedSimpleFloatArray::CheckedSimpleFloatArray (int s, float f)
  : SimpleFloatArray (s,f)
{}
```

```
float& CheckedSimpleFloatArray::operator[] (int i)
{
  assert (i>=minIndex() && i<=maxIndex());
  return SimpleFloatArray::operator[](i);
}
```

Verwendung **(UseCheckedSimpleFloatArray.cc)**:

```
#include<iostream>
#include<cassert>

#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

#include "CheckedSimpleFloatArray.hh"
#include "CheckedSimpleFloatArrayImp.cc"

void g (SimpleFloatArray a) {
  std::cout << "beginn in g: " << std::endl;
  std::cout << "access: " << a[1] << "  " << a[10] << std::endl;
  std::cout << "ende in g: " << std::endl;
}

int main () {
  CheckedSimpleFloatArray a(10,3.1415);
  g(a);
  std::cout << "beginn in main: " << std::endl;
  std::cout << "zugriff in main: " << a[10] << std::endl;
  std::cout << "ende in main: " << std::endl;
}
```

mit der Ausgabe:

```
beginn in g:
access: 0  1.85018e-40
ende in g:
beginn in main:
UseCheckedSimpleFloatArray: CheckedSimpleFloatArrayImp.cc:8:
  float& CheckedSimpleFloatArray::operator[](int):
  Assertion `i>=minIndex() && i<=maxIndex()' failed.
Aborted (core dumped)
```

Bemerkung:

- In der Funktion `main` funktioniert die Bereichsprüfung dann wie erwartet.
- In der Funktion `g` wird hingegen keine Bereichsprüfung durchgeführt, *auch wenn sie mit einem Objekt vom Typ* ***`CheckedSimpleFloatArray`*** *aufgerufen wird*!

- Warum ist das so?
- In der Funktion `g` betrachtet der Compiler die übergebene Referenz als Objekt vom Typ SimpleFloatArray und dies hat keine Bereichsprüfung.
- Der Funktionsaufruf mit dem Objekt der öffentlich abgeleiteten Klasse ändert daran nichts! Dies ist eine Konsequenz der Realisierung der Ist-ein-Beziehung.
- Die Auswahl der Methode hängt vom angegeben Typ ab und nicht vom konkreten Objekt welches übergeben wird.
- Meistens ist dies aber *nicht das gewünschte Verhalten* (vgl. dazu auch die späteren Beispiele).

Virtuelle Funktionen

Idee: Gib dem Compiler genügend Information, so dass er schon bei der Übersetzung von `SimpleFloatArray`-Methoden ein flexibles Verhalten von `[]` möglich macht. In C++ geschieht dies, indem man Methoden *in der Basisklasse* als virtuell (*virtual*) kennzeichnet.

Programm:

```
class SimpleFloatArray
{
public:
  ...
  virtual float& operator[]( int i );
  ...
private:
  ...
};
```

Beobachtung: Mit dieser Änderung funktioniert die Bereichsprüfung auch in der Funktion `g` in `UseCheckedSimpleFloatArray.cc`: wird sie mit einer Referenz auf ein `CheckedSimpleFloatArra` Objekt aufgerufen, so wird der Bereichstest durchgeführt, bei Aufruf mit einer Referenz auf ein `SimpleFloatArray`-Objekt aber nicht.

Bemerkung:

- Die Einführung einer virtuellen Funktion erfordert also Änderungen in bereits existierendem Code, nämlich der Definition der Basisklasse!
- Die Implementierung der Methoden bleibt jedoch unverändert.

Implementation: Diese Auswahl der Methode in Abhängigkeit vom tatsächlichen Typ des Objekts kann man dadurch erreichen, dass jedes Objekt entweder Typinformation oder einen Zeiger auf eine Tabelle mit den für seine Klasse virtuell definierten Funktionen mitführt.

Bemerkung:

- Wird eine als virtuell markierte Methode in einer abgeleiteten Klasse neu implementiert, so wird die Methode der *abgeleiteten Klasse* verwendet, wenn das Objekt für ein Basisklassenobjekt eingesetzt wird.
- Die Definition der Methode in der abgeleiteten Klasse muss genau mit der Definition in der Basisklasse übereinstimmen, ansonsten wird *überladen*!
- Das Schlüsselwort **virtual** muss in der abgeleiteten Klasse nicht wiederholt werden, es ist aber guter Stil dies zu tun.
- Die Eigenschaften virtueller Funktionen lassen sich nur nutzen, wenn auf das Objekt über Referenzen oder Zeiger zugegriffen wird! Bei einem Aufruf (*call-by-value*) von

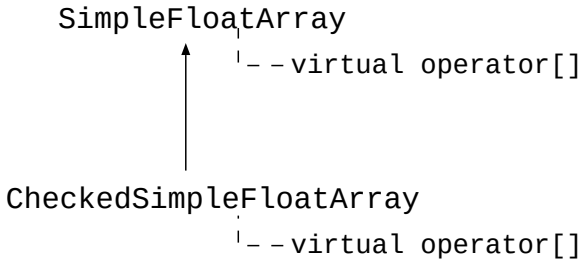
```
void g( SimpleFloatArray a )
{
  cout << a[1] << "  " << a[11] << endl;
}
```

  erzeugt der Copy-Konstruktor ein Objekt `a` vom Typ `SimpleFloatArray` (Slicing!) und innerhalb von `g()` wird entsprechend dessen `operator[]` verwendet.
- Virtuelle Funktionen stellen wieder eine Form des Polymorphismus dar („eine Schnittstelle — viele Methoden“).
- Der Zugriff auf eine Methode über die Tabelle virtueller Funktionen ist deutlich ineffizienter, was für Objektorientierung auf niedriger Ebene eine Rolle spielen kann.
- In vielen objektorientierten Sprachen (z. B. Smalltalk, Objective C, Common Lisp/CLOS) verhalten sich alle Methoden „virtuell“.
- In der Programmiersprache Java ist das virtuelle Verhalten der Normalfall, das Default-Verhalten von C++-Methoden kann man aber durch Hinzufügen des Schlüsselworts `static` erreichen.

10 Abstrakte Klassen

10.1 Motivation

Hatten:



Beobachtung: Beide Klassen besitzen dieselben Methoden und unterscheiden sich nur in der Implementierung von `operator[]`. Wir könnten ebenso `SimpleFloatArray` von `CheckedSimpleFloatArray` ableiten. Das Klassendiagramm drückt diese Symmetrie aber nicht aus.

Grund: `SimpleFloatArray` stellt sowohl die Definition der Schnittstelle eines ADT *Feld* dar, als auch eine Implementierung dieser Schnittstelle. Es ist aber sinnvoll, diese beiden Aspekte zu trennen.

10.2 Schnittstellenbasisklassen

Idee: Definiere eine möglichst allgemeine Klasse `FloatArray`, von der sowohl `SimpleFloatArray` als auch `CheckedSimpleFloatArray` abgeleitet werden.

Bemerkung: Oft will und kann man für (virtuelle) Methoden in einer solchen Basisklasse keine Implementierung angeben. In C++ kennzeichnet man sie dann mit dem Zusatz `= 0` am Ende. Solche Funktionen bezeichnet man als rein virtuelle (*engl.: pure virtual*) Funktionen.

Beispiel: **(FloatArray.hh)**

```
class FloatArray {
public:
  virtual ~FloatArray() {};
  virtual float& operator[](int i) = 0;
  virtual int numIndices () = 0;
  virtual int minIndex () = 0;
  virtual int maxIndex () = 0;
  virtual bool isMember (int i) = 0;
} ; // Dokumentation fehlt leider!
```

Bezeichnung: Klassen, die mindestens eine rein virtuelle Funktion enthalten, nennt man abstrakt. Das Gegenteil ist eine konkrete Klasse.

Bemerkung:

- Man kann keine Objekte von abstrakten Klassen instanzieren. Aus diesem Grund haben abstrakte Klassen auch *keine Konstruktoren*.
- Sehr wohl kann man aber Zeiger und Referenzen dieses Typs haben, die dann aber auf Objekte abgeleiteter Klassen zeigen.
- Eine abstrakte Klasse, die der Definition einer Schnittstelle dient, bezeichnen wir nach Barton/Nackman als Schnittstellenbasisklasse (*interface base class*).
- Schnittstellenbasisklassen enthalten üblicherweise keine Datenmitglieder und alle Methoden sind rein virtuell.
- Die Implementierung dieser Schnittstelle erfolgt in abgeleiteten Klassen.

Bemerkung: (Virtueller Destruktor) Eine Schnittstellenbasisklasse sollte einen *virtuellen* Destruktor

```
virtual ~FloatArray();
```

mit einer Dummy-Implementierung

```
FloatArray::~FloatArray() {}
```

besitzen, damit man dynamisch erzeugte Objekte abgeleiter Klassen durch die Schnittstelle der Basisklasse löschen kann. Beispiel:

```
void g( FloatArray* p )
{
  delete p;
}
```

Bemerkung: Der Destruktor darf nicht rein virtuell sein, da der Destruktor abgeleiteter Klassen einen Destruktor der Basisklasse aufrufen will.

10.3 Beispiel: geometrische Formen

Aufgabe: Wir wollen mit zweidimensionalen geometrischen Formen arbeiten. Dies sind von einer Kurve umschlossene Flächen wie Kreis, Rechteck, Dreieck, ...

Programm: Eine mögliche C++-Implementierung wäre folgende **(shape.cc)**:

```
#include<iostream>
#include<cmath>

const double pi = 3.1415926536;

class Shape
{
public:
  virtual ~Shape () {};
  virtual double area () = 0;
  virtual double diameter () = 0;
  virtual double circumference () = 0;
};

// works on every shape
double circumference_to_area (Shape &shape) {
  return shape.circumference()/shape.area();
}

class Circle : public Shape {
public:
  Circle (double r) {radius = r;}
```

```
    virtual double area () {
24    return pi*radius*radius;
    }
    virtual double diameter () {
27    return 2*radius;
    }
    virtual double circumference () {
30    return 2*pi*radius;
    }
  private:
33  double radius;
  };

36 class Rectangle : public Shape {
  public:
    Rectangle (double aa, double bb) {
39    a = aa; b = bb;
    }
    virtual double area () {return a*b;}
42  virtual double diameter () {
      return sqrt(a*a+b*b);
    }
45  virtual double circumference () {
      return 2*(a+b);
    }
48 private:
    double a, b;
  };
51
  int main ()
  {
54  Rectangle unit_square(1.0, 1.0);
    Circle unit_circle(1.0);
    Circle unit_area_circle(1.0/sqrt(pi));
57
    std::cout << "Das äVerhltnis von Umfang zu äFlche äbetrgt\n";
    std::cout << "Einheitsquadrat: "
60          << circumference_to_area(unit_square)
            << std::endl;
    std::cout << "Kreis mit äFlche 1: "
63          << circumference_to_area(unit_area_circle)
            << std::endl;
    std::cout << "Einheitskreis: "
66          << circumference_to_area(unit_circle)
            << std::endl;
    return 0;
69 }
```

Ergebnis: Wir erhalten als Ausgabe des Programms:

```
Das Verhaeltnis von Umfang zu Flaeche betraegt
Einheitsquadrat:     4
Kreis mit Flaeche 1: 3.54491
Einheitskreis:       2
```

10.4 Beispiel: Funktoren

→ Klasse der Objekte, die sich wie Funktionen verhalten.

Hatten: Definition einer Inkrementierer-Klasse in `Inkrementierer.cc`. Nachteile waren:

- Möchten beliebige Funktion auf die Listenelemente anwenden können.
- Syntax `ink.eval(...)` nicht optimal.

Dies wollen wir nun mit Hilfe einer Schnittstellenbasisklasse und der Verwendung von `operator()` verbessern.

Programm: **(Funktor.cc)**

```
#include <iostream>

class Function {
public:
  virtual ~Function () {};
  virtual int operator() (int) = 0;
};

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

void schleife (Function& func) {
  for (int i=1; i<10; i++)
    std::cout << func(i) << " ";
  std::cout << std::endl;
}

class Quadrat : public Function {
public:
  int operator() (int n) {return n*n;}
} ;

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

```
30    Quadrat quadrat;
      schleife (ink);
      schleife (quadrat);
33  }
```

Bemerkung: Unangenehm ist jetzt eigentlich nur noch, dass der Argument- und Rückgabetyp der Methode auf **int** festgelegt ist. Dies wird bald durch Schablonen (*Templates*) behoben werden.

10.5 Beispiel: Exotische Felder

Programm: Wir definieren folgende (schon gezeigte) Schnittstellenbasisklasse **(FloatArray.hh)**:

```
  class FloatArray {
  public:
3   virtual ~FloatArray() {};
    virtual float& operator[](int i) = 0;
    virtual int numIndices () = 0;
6   virtual int minIndex () = 0;
    virtual int maxIndex () = 0;
    virtual bool isMember (int i) = 0;
9 } ; // Dokumentation fehlt leider!
```

Von dieser kann man leicht `SimpleFloatArray` ableiten. Außerdem passt die Schnittstelle auf weitere Feldtypen, was wir im Folgenden zeigen wollen.

Dynamisches Feld

Wir wollen jetzt ein Feld mit variabel großer, aber zusammenhängender Indexmenge $I = \{o, o+1, \ldots, o+n-1\}$ mit $o, n \in \mathbb{Z}$ und $n \geq 0$ definieren. Wir gehen dazu folgendermaßen vor:

- Der Konstruktor fängt mit einem Feld der Länge 0 an ($o = n = 0$).
- `operator[]` prüft, ob $i \in I$ gilt; wenn nein, so wird der Indexbereich erweitert, ein entsprechendes Feld allokiert, die Werte aus dem alten Feld werden in das neue kopiert und das alte danach freigegeben.

Programm: **(DFA.cc)**

```
  class DynamicFloatArray : public FloatArray {
  public:
3   DynamicFloatArray () {n=0; o=0; p=new float[1];}

    virtual ~DynamicFloatArray() {delete[] p;}
6   virtual float& operator[](int i);
    virtual int numIndices () {return n;}
    int minIndex ()           {return o;}
```

```
9     int maxIndex ()            {return o+n-1;}
      bool isMember (int i) {return (i>=o)&&(i<o+n);}
    private:
12    int n;       // Anzahl Elemente
      int o;       // Ursprung der Indexmenge
      float *p;    // Zeiger auf built-in array
15  };

    float& DynamicFloatArray::operator[] (int i)
18  {
      if (i<o || i>=o+n)
        { // determine new size
21        int new_o, new_n;
          if (i<o) {
            new_o = i;
24          new_n = n+o-i;
          }
          else {
27          new_o = o;
            new_n = i-o+1;
          }
30        // allocate new array
          float *q = new float[new_n];
          // initialize and copy over
33        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];
36        // delete old array
          delete[] p;
          // remember new data
39        p = q;
          n = new_n;
          o = new_o;
42      }
      return p[i-o];
    }
```

Bemerkung:

- Im Konstruktor wird der Einfachheit halber bereits ein **float** allokiert.
- Der Copy-Konstruktor und Zuweisungsoperator müssten auch implementiert werden (um zu vermeiden, dass der Zeiger kopiert wird).

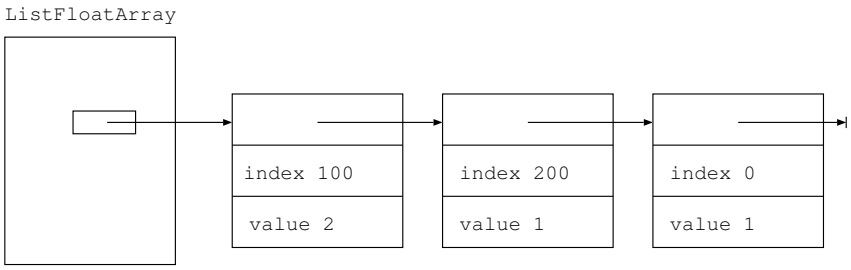
Listenbasiertes Feld

Problem: Wenn man `DynamicFloatArray` zur Darstellung von Polynomen verwendet, so werden Polynome mit vielen Nullkoeffizienten, z. B.

$$p(x) = x^{100} + 1 \text{ oder } q(x) = p^2(x) = x^{200} + 2x^{100} + 1$$

sehr ineffizient verwaltet.

Abhilfe: Speichere die Elemente des Feldes als einfach verkettete *Liste* von Index–Wert–Paaren:



Programm: **(LFA.cc)**

```
class ListFloatArray :
  public FloatArray {      // Ableitung
public:
  ListFloatArray ();       // leeres Feld

  ~ListFloatArray();       // ersetzt ~FloatArray

  virtual float& operator[](int i);
  virtual int numIndices ();
  virtual int minIndex ();
  virtual int maxIndex ();
  virtual bool isMember (int i);
private:
  struct FloatListElem { // lokale Struktur
    FloatListElem *next;
    int index;
    float value;
  };
  FloatListElem* insert (int i, float v);
  FloatListElem* find (int i);

  int n;             // Anzahl Elemente
  FloatListElem *p;// Listenanfang
} ;

// 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;
36  n = n+1;
    return q;
}

ListFloatArray::FloatListElem*
ListFloatArray::find (int i)
42  {
    for (FloatListElem* q=p; q!=0; q = q->next)
      if (q->index==i)
45      return q;
    return 0;
}

48  // Konstruktor
ListFloatArray::ListFloatArray ()
51  {
    n = 0; // alles leer
    p = 0;
54  }

    // Destruktor
57  ListFloatArray::~ListFloatArray ()
    {
    FloatListElem* q;

60
    while (p!=0)
      {
63      q = p;        // q ist erstes
        p = q->next; // entferne q aus Liste
        delete q;
66    }
}

69  float& ListFloatArray::operator[] (int i)
    {
    FloatListElem* r=find(i);
72  if (r==0)
      r=insert(i,0.0); // index einfuegen
    return r->value;
75  }

    int ListFloatArray::numIndices ()
78  {
    return n;
}

81
    int ListFloatArray::minIndex ()
    {
84  if (p==0) return 0;
```

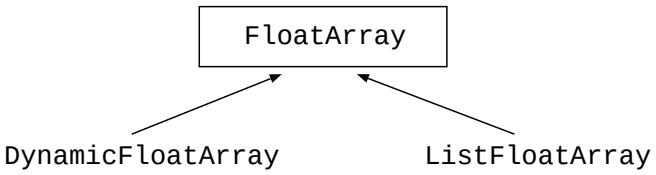
```
    int min=p->index;
    for (FloatListElem* q=p->next;
87        q!=0; q = q->next)
      if (q->index<min) min=q->index;
    return min;
90  }

    int ListFloatArray::maxIndex ()
93  {
    if (p==0) return 0;
    int max=p->index;
96  for (FloatListElem* q=p->next;
        q!=0; q = q->next)
      if (q->index>max) max=q->index;
99  return max;
    }

102 bool ListFloatArray::isMember (int i) {
    return (find(i)!=0);
    }
```

Bemerkung:

- Das Programm verwendet eine unsortierte Liste. Man hätte auch eine sortierte Liste verwenden können (Vorteile?).
- Für die Index–Wert–Paare wird innerhalb der Klassendefinition der zusammengesetzte Datentyp `FloatListElem` definiert.
- Die privaten Methoden dienen der Manipulation der Liste und werden in der Implementierung der öffentlichen Methoden verwendet. Merke: Innerhalb einer Klasse können wiederum Klassen definiert werden!

Anwendung

Wir haben somit folgendes Klassendiagramm:



Da sowohl `DynamicFloatArray` als auch `ListFloatArray` die durch `FloatArray` definierte Schnittstelle erfüllen, kann man nun Methoden für `FloatArray` schreiben, die auf beiden abgeleiteten Klassen funktionieren.

Als Beispiel betrachten wir folgendes Programm, welches `FloatArray` wieder zur Polynom-Multiplikation verwendet (der Einfachheit halber ohne es in eine Klasse `Polynomial` zu packen).

Programm: **(UseFloatArray.cc)**

```
#include <iostream>

#include "FloatArray.hh"
#include "DFA.cc"
#include "LFA.cc"

void polyshow (FloatArray& f) {
  for (int i=f.minIndex(); i<=f.maxIndex(); i=i+1)
    if (f.isMember(i)  && f[i]!=0.0)
      std::cout << "+" << f[i] << "*x^" << i;
  std::cout << std::endl;
}

void polymul (FloatArray& a, FloatArray& b, FloatArray& c) {
  // Loesche a
  for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1)
    if (a.isMember(i))
      a[i] = 0.0;

  // a = b*c
  for (int i=b.minIndex(); i<=b.maxIndex(); i=i+1)
    if (b.isMember(i))
      for (int j=c.minIndex(); j<=c.maxIndex(); j=j+1)
        if (c.isMember(j))
          a[i+j] = a[i+j]+b[i]*c[j];
}

int main ()
{
  // funktioniert mit einer der folgenden Zeilen:
  //DynamicFloatArray f,g;
  ListFloatArray f,g;

  f[0] = 1.0; f[100] = 1.0;

  polymul(g,f,f);
  polymul(f,g,g);
  polymul(g,f,f);
  polymul(f,g,g); // f=(1+x^100)^16

  polyshow(f);
}
```

Ausgabe:

```
+1*x^0+16*x^1000+120*x^2000+560*x^3000+1820*x^4000+4368*x^5000+8008*x^6000
+11440*x^7000+12870*x^8000+11440*x^9000+8008*x^10000+4368*x^11000
+1820*x^12000+560*x^13000+120*x^14000+16*x^15000+1*x^16000
```

Bemerkung:

- Man kann nun sehr „spät“, nämlich erst in der `main`-Funktion entscheiden, mit welcher Art Felder man tatsächlich arbeiten will.
- Je nachdem, wie vollbesetzt der Koeffizientenvektor ist, ist entweder `DynamicFloatArray` oder `ListFloatArray` günstiger.
- Schlecht ist noch die Weise, in der allgemeine Schleifen über das Feld implementiert werden. Die Anwendung auf `ListFloatArray` ist sehr ineffektiv! Eine Abhilfe werden wir bald kennenlernen (Iteratoren).

10.6 Zusammenfassung

In diesem Abschnitt haben wir gezeigt, wie man mit Hilfe von Schnittstellenbasisklassen eine Trennung von

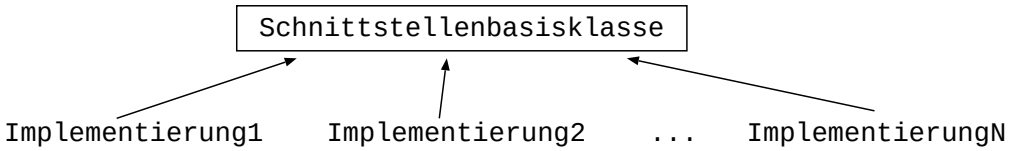
- Schnittstellendefinition und
- Implementierung

erreicht.

Dies gelingt durch

- rein virtuelle Funktionen in Verbindung mit
- Vererbung.

Typischerweise erhält man Klassendiagramme der Form:



Man *erzeugt* Objekte konkreter (abgeleiteter) Klassen und *benutzt* diese Objekte durch die Schnittstellenbasisklasse:

Create objects, use interfaces!

11 Generische Programmierung

11.1 Funktionsschablonen

Motivation: Auswechseln von Datentypen in streng typgebundenen Sprachen.

Definition: Eine Funktionsschablone (*Function Template*) entsteht, indem man die Präambel

```
template <class T>
```

bzw.

```
template <typename T>
```

einer Funktionsdefinition voranstellt. In der Schablonendefinition kann man `T` dann wie einen vorhandenen Datentyp verwenden. Dieser Typ muss keine Klasse sein, was die Einführung der äquivalenten Definition mittels **typename** motivierte.

Programm: Vertauschen des Inhalts zweier gleichartiger Referenzen:

```
template <class T>
void swap( T& a, T& b )
{
  T t = a;
  a = b; b = t;
}

int main()
{
  int a = 10, b = 20;
  swap( a, b );
  double x = 1.0, y = 2.0;
  swap( x, y );
}
```

- Bei der *Übersetzung* von `swap(a,b)` generiert der Übersetzer die Version `swap(int& a, int& b)` und übersetzt sie (es sei denn, es gibt schon genau so eine Funktion).
- Bei der *Übersetzung* von `swap(x,y)` generiert der Übersetzer die Version `swap(double& a, double& 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).
- Dies nennt man automatische Template-Instanzierung.

Programm: Beispiel: Maximum

```
template <class T> T max( T a, T b )
{
  if ( a < b ) return b; else return a;
}
```

Bemerkung: Hier muss für den Typ `T` ein **operator**< definiert sein.

Beispiel: wieder funktionales Programmieren

Problem: Der Aufruf virtueller Funktionen erfordert Entscheidungen zur Laufzeit, was in einigen (wenigen) Fällen zu langsam sein kann.

Abhilfe: Verwendung von Funktionsschablonen.

Programm: Funktionales Programmieren mit Schablonen **(Funktional-statisch.cc)**:

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

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

class Quadrat {
public:
  int operator() (int n) {return n*n;}
} ;

template<class T>
void schleife (T& func) {
  for (int i=1; i<10; i++)
    cout << func(i) << " ";
  cout << endl;
}

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

Bemerkung:

- Es werden die passenden Varianten der Funktion `schleife` erzeugt.
- Unterschied zur Variante mit der gemeinsamen Basisklasse Function:
  - Statt genau einer gibt es nun mehrere Varianten der Funktion `schleife`.
  - Methodenaufrufe am Argument `func` erfolgen **nicht** über virtuelle Funktionen.
- Nachteil: Leider haben wir aber keine Schnittstellendefinition mehr.

Bezeichnung: Man nennt diese Technik auch statischen Polymorphismus, da die Methodenauswahl zur Übersetzungszeit erfolgt. Im Gegensatz dazu bezeichnet man die Verwendung virtueller Funktionen als dynamischen Polymorphismus.

Empfehlung: Wenden Sie diese oder ähnliche Techniken (wie etwa die sogenannten *Expression Templates*) nur an, wenn es unbedingt notwendig ist. Untersuchen Sie auch vorher das Laufzeitverhalten (Profiling), denn laut Donald E. Knuth (ursprünglich wohl von C. A. R. Hoare) gilt:

> *Premature optimization is the root of all evil!*

11.2 Klassenschablonen

Problem: Unsere selbstdefinierten Felder und Listen sind noch zu unflexibel. So hätten wir beispielsweise auch gerne Felder von **int**-Zahlen.

Bemerkung: Dieses Problem rührt von der statischen Typbindung von C/C++ her und tritt bei Sprachen mit dynamischer Typbindung (Scheme, Python, ...) nicht auf. Allerdings ist es für solche Sprachen viel schwieriger hocheffizienten Code zu generieren.

Abhilfe: Die C++-Lösung für dieses Problem sind parametrisierte Klassen, die auch Klassenschablonen (*Class Templates*) genannt werden.

Definition: Eine Klassenschablone entsteht, indem man der Klassendefinition die Präambel **template** <**class** T> voranstellt. In der Klassendefinition kann dann der Parameter `T` wie ein Datentyp verwendet werden.

Beispiel:

```
// Schablonendefinition
template <class T> class SimpleArray
{
public:
  SimpleArray( int s, T f );
...
};
// Verwendung
SimpleArray<int> a( 10, 0 );
SimpleArray<float> b( 10, 0.0 );
```

Bemerkung:

- `SimpleArray` alleine ist kein Datentyp!
- `SimpleArray<int>` ist ein neuer Datentyp, d. h. Sie können Objekte dieses Typs erzeugen, oder ihn als Parameter/Rückgabewert einer Funktion verwenden.
- Der Mechanismus arbeitet wieder zur *Übersetzungszeit* des Programmes. Bei *Übersetzung* der Zeile

```
SimpleArray<int> a( 10, 0 );
```

generiert der Übersetzer den Programmtext für `SimpleArray<int>`, der aus dem Text der Klassenschablone `SimpleArray` entsteht, indem alle Vorkommen von `T` durch **int** ersetzt werden. Anschließend wird diese Klassendefinition übersetzt.

- Da der Übersetzer selbst C++-Programmcode generiert, spricht man auch von generischer Programmierung.
- Den Vorgang der Erzeugung einer konkreten Variante einer Klasse zur Übersetzungszeit nennt man auch Template-Instanzierung. Allerdings gibt es bei Klassen, im Gegensatz zu Funktionen, keine automatische Instanzierung.
- Der Name Schablone (*Template*) kommt daher, dass man sich die parametrisierte Klasse als Schablone vorstellt, die zur Anfertigung konkreter Varianten benutzt wird.

Programm: **(SimpleArray.hh)**

```
template <class T>
class SimpleArray {
public:
  SimpleArray (int s, T f);
  SimpleArray (const SimpleArray<T>&);
  SimpleArray<T>& operator=
    (const SimpleArray<T>&);
  ~SimpleArray();

  T& operator[](int i);
  int numIndices ();
  int minIndex ();
  int maxIndex ();
  bool isMember (int i);

private:
  int n;  // Anzahl Elemente
  T *p;   // Zeiger auf built-in array
};
```

Bemerkung: Syntaktische Besonderheiten:

- Wird die Klasse selbst als Argument oder Rückgabewert im Rumpf der Definition benötigt, schreibt man `SimpleArray<T>`.
- Im Namen des Konstruktors bzw. Destruktors taucht *kein* `T` auf. Der Klassenparameter parametrisiert den Klassennamen, nicht aber die Methodennamen.
- Die Definition des Destruktors (als Beispiel) lautet dann:

```
template <class T>
SimpleArray<T>::~SimpleArray() { delete[] p; }
```

Programm: Methodenimplementierung **(SimpleArrayImp.cc)**:

```
// Destruktor
template <class T>
SimpleArray<T>::~SimpleArray () {
  delete[] p;
}

// Konstruktor
template <class T>
SimpleArray<T>::SimpleArray (int s, T v)
{
  n = s;
  p = new T[n];
  for (int i=0; i<n; i=i+1) p[i]=v;
}

// Copy-Konstruktor
template <class T>
SimpleArray<T>::SimpleArray (const
                  SimpleArray<T>& a) {
  n = a.n;
  p = new T[n];
  for (int i=0; i<n; i=i+1)
    p[i]=a.p[i];
}

// Zuweisungsoperator
template <class T>
SimpleArray<T>& SimpleArray<T>::operator=
  (const SimpleArray<T>& a)
{
  if (&a!=this) {
    if (n!=a.n) {
      delete[] p;
      n = a.n;
      p = new T[n];
    }
    for (int i=0; i<n; i=i+1) p[i]=a.p[i];
  }
  return *this;
}

template <class T>
inline T& SimpleArray<T>::operator[] (int i)
{
  return p[i];
}

template <class T>
inline int SimpleArray<T>::numIndices ()
{
```

```
51    return n;
    }

54  template <class T>
    inline int SimpleArray<T>::minIndex ()
    {
57    return 0;
    }

60  template <class T>
    inline int SimpleArray<T>::maxIndex ()
    {
63    return n-1;
    }

66  template <class T>
    inline bool SimpleArray<T>::isMember (int i)
    {
69    return (i>=0 && i<n);
    }

72  template <class T>
    std::ostream& operator<< (std::ostream& s,
                  SimpleArray<T>& a)
75  {
      s << "#(␣";
      for (int i=a.minIndex(); i<=a.maxIndex(); i=i+1)
78      s << a[i] << "␣";
      s << ")" << std::endl;
      return s;
81  }
```

Programm: Verwendung **(UseSimpleArray.cc)**:

```
    #include<iostream>

3   #include "SimpleArray.hh"
    #include "SimpleArrayImp.cc"

6   int main ()
    {
      SimpleArray<float> a(10,0.0); // erzeuge
9     SimpleArray<int> b(25,5);     // Felder

      for (int i=a.minIndex(); i<=a.maxIndex(); i++)
12      a[i] = i;
      for (int i=b.minIndex(); i<=b.maxIndex(); i++)
        b[i] = i;
15
      std::cout << a << std::endl << b << std::endl;
```

```
18      // hier wird der Destruktor gerufen
    }
```

Beispiel: Feld fester Größe

Bemerkung:

- Eine Schablone kann auch mehr als einen Parameter haben.
- Als Schablonenparameter sind nicht nur Klassennamen, sondern z. B. auch Konstanten von eingebauten Typen erlaubt.

Anwendung: Ein Feld fester Größe könnte folgendermaßen definiert und verwendet werden:

```
template <class T, int m>
class SimpleArrayCS
{
public:
  SimpleArrayCS( T f );
  ...
private:
  T p[m]; // built-in array fester Groesse
};

...

SimpleArrayCS<int, 5> a( 0 );
SimpleArrayCS<float, 3> a( 0.0 );
...
```

Bemerkung:

- Die Größe ist hier auch zur Übersetzungszeit festgelegt und muss nicht mehr gespeichert werden.
- Da nun keine Zeiger auf dynamisch allokierte Objekte verwendet werden, sind für Copy-Konstruktor, Zuweisung und Destruktor die Defaultmethoden ausreichend.
- Der Compiler kann bei bekannter Feldgröße unter Umständen effizienteren Code generieren, was vor allem für kleine Felder interessant ist (z. B. Vektoren im $\mathbb{R}^2$ oder $\mathbb{R}^3$).
- *Es ist ein wichtiges Kennzeichen von C++, dass Objektorientierung bei richtigem Gebrauch auch für sehr kleine Datenstrukturen ohne Effizienzverlust angewendet werden kann.*

Beispiel: Smart Pointer

Problem: Dynamisch erzeugte Objekte können ausschließlich über Zeiger verwaltet werden. Wie bereits diskutiert, ist die konsistente Verwaltung des Zeigers (bzw. der Zeiger) und des Objekts nicht einfach:

- Objekte müssen gelöscht werden bevor der letzte Zeiger darauf gelöscht wird.

Abhilfe: Entwerfe einen neuen Datentyp, der anstatt eines Zeigers verwendet wird. Mittels Definition von **operator*** und **operator->** kann man erreichen, dass sich der neue Datentyp wie ein eingebauter Zeiger benutzen lässt. In Copy-Konstruktor, Destruktor und Zuweisungsoperator wird dann die Speicherverwaltung eingebaut.

Bezeichnung: Ein Datentyp mit dieser Eigenschaft wird intelligenter Zeiger (*smart pointer*) genannt.

Die C++-Standardbibliothek bietet solche Zeigertypen an:

- std::unique_ptr<T> behandelt den Fall, dass es genau einen Zeiger auf ein dynamisch erzeugtes Objekt gibt. Der intelligente Zeiger existiert nur innerhalb einer Umgebung.
- std::shared_ptr<T> behandelt den allgemeineren Fall dass es mehrere Zeiger auf ein dynamisch erzeugtes Objekt gibt. Mittels *reference counting* wird (automatisch) festgestellt wann der letzte Zeiger gelöscht wird.

Wir demonstrieren hier anhand der Klasse Ptr<T> wie man so etwas implementieren kann.

Programm: **(Zeigerklasse zum *reference counting*, Ptr.hh)**

```
template<class T>
class Ptr {
  struct RefCntObj {
    int count;
    T* obj;
    RefCntObj (T* q) { count = 1; obj = q; }
  };
  RefCntObj* p;

  void report () {
    std::cout << "refcnt = " << p->count << std::endl;
  }
  void increment () {
    p->count = p->count + 1;
    report();
  }
  void decrement () {
    p->count = p->count - 1;
    report();
    if (p->count==0) {
      delete p->obj; // Geht nicht fuer Felder!
      delete p;
    }
  }
```

```
public:
  Ptr () { p=0; }

  Ptr (T* q) {
    p = new RefCntObj(q);
    report();
  }

  Ptr (const Ptr<T>& y) {
    p = y.p;
    if (p!=0) increment();
  }

  ~Ptr () {
    if (p!=0) decrement();
  }

  Ptr<T>& operator= (const Ptr<T>& y) {
    if (p!=y.p) {
      if (p!=0) decrement();
      p = y.p;
      if (p!=0) increment();
    }
    return *this;
  }

  T& operator* () { return *(p->obj); }

  T* operator-> () { return p->obj; }
};
```

Programm: (Anwendungsbeispiel, PtrTest.cc)

```
#include<iostream>

#include"Ptr.hh"

int g (Ptr<int> p) {
  return *p;
}

int main ()
{
  Ptr<int> q = new int(17);
  std::cout << *q << std::endl;
  int x = g(q);
  std::cout << x << std::endl;
  Ptr<int> z = new int(22);
  q = z;
```

```
    std::cout << *q << std::endl;
18  // nun wird alles automatisch geloescht !
  }
```

Bemerkung:

- Man beachte die sehr einfache Verwendung durch Ersetzen der eingebauten Zeiger (die natürlich nicht weiterverwendet werden sollten!).
- Nachteil: mehr Speicher wird benötigt (das `RefCntObj`)
- Es gibt verschiedene Möglichkeiten, *reference counting* zu implementieren, die sich bezüglich Speicher- und Rechenaufwand unterscheiden.
- Die hier vorgestellte Zeigerklasse funktioniert (wegen **delete**[]) nicht für Felder!
- *Reference counting* funktioniert *nicht* für Datenstrukturen mit Zykeln $\rightsquigarrow$ andere Techniken zur automatischen Speicherverwaltung notwendig.

11.3 Effizienz generischer Programmierung

Beispiel: Bubblesort

Aufgabe: Ein Feld von Zahlen $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ ist zu sortieren. Die Sortierfunktion liefert als Ergebnis eine Permutation $a' = (a'_0, a'_1, a'_2, \ldots, a'_{n-1})$ der Feldelemente zurück, so dass

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$$

Idee: Der Algorithmus Bubblesort ist folgendermaßen definiert:

- Gegeben sei ein Feld $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ der Länge n.
- Durchlaufe die Indizes $i = 0, 1, \ldots, n-2$ und vergleiche jeweils a_i und a_{i+1}. Ist $a_i > a_{i+1}$ so vertausche die beiden. Beispiel:

	17	10	8	16
$i = 0$	10	17	8	16
$i = 1$	10	8	17	16
$i = 2$	10	8	16	17

  Am Ende eines solchen Durchlaufes steht die größte der Zahlen sicher ganz rechts und ist damit an der richtigen Position.
- Damit bleibt noch ein Feld der Länge $n-1$ zu sortieren.

Satz: t_{cs} sei eine obere Schranke der Kosten für einen Vergleich und einen swap (Tausch), und n bezeichne die Länge des Felds. Falls t_{cs} nicht von n abhängt, so hat Bubblesort eine asymptotische Laufzeit von $O(n^2)$.

Beweis:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} t_{cs} = t_{cs}\sum_{i=0}^{n-1} i = t_{cs}\frac{(n-1)n}{2} = O(n^2)$$

Programm: **(bubblesort_.cc)**

```
/* ist in namespace std schon enthalten:
   template <class T> void swap (T& a, T&b) {
   T t = a;
   a = b;
   b = t;
   }
*/

template <class C> void bubblesort (C& a) {
  for (int i=a.maxIndex(); i>=a.minIndex(); i=i-1)
    for (int j=a.minIndex(); j<i; j=j+1)
      if (a[j+1]<a[j])
        std::swap(a[j+1], a[j]);
}
```

Bemerkung:

- Die Funktion `bubblesort` benötigt, dass auf *Elementen* des Feldes der Vergleichsoperator **operator<** definiert ist.
- Die Funktion benutzt die *öffentliche Schnittstelle* der Feldklassen, die wir programmiert haben, d. h. für `C` können wir jede unserer Feldklassen einsetzen!

Mit folgender Routine kann man Laufzeiten verschiedener Programmteile messen:

Programm: **(timestamp.cc)**

```
#include <ctime>
// Setzt Marke und gibt Zeitdifferenz zur letzten Marke zurueck
clock_t last_time;
double time_stamp () {
    clock_t current_time = clock();
    double duration =
        ((double)(current_time-last_time)) / CLOCKS_PER_SEC;
    last_time = current_time;
    return duration;
}
```

Dies wenden wir auf Bubblesort an:

Programm: **Bubblesort für verschiedene Feldtypen (UseBubblesort.cc)**

```
#include<iostream>

// SimpleFloatArray mit virtuellem operator[]
```

```
#include "SimpleFloatArrayV.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

// templatisierte Variante mit variabler Groesse
#include "SimpleArray.hh"
#include "SimpleArrayImp.cc"

// templatisierte Variante mit Compile-Zeit Groesse
#include "SimpleArrayCS.hh"
#include "SimpleArrayCSImp.cc"

// dynamisches listenbasiertes Feld
#include "FloatArray.hh"
#include "ListFloatArrayDerived.hh"
#include "ListFloatArrayImp.cc"

// Zeitmessung
#include "timestamp.cc"

// generischer bubblesort
#include "bubblesort_.cc"

// Zufallsgenerator
#include "Zufall.cc"

const int n = 8000;

static Zufall z(93576);

template <class T>
void initialisiere (T & a) {
    for (int i=0; i<n; i=i+1)
        a[i] = z.ziehe_zahl();
}

int main ()
{
    SimpleArrayCS<float,n> a(0.0);
    SimpleArray<float> b(n,0.0);
    SimpleFloatArray c(n,0.0);
    ListFloatArray d;

    initialisiere (a); initialisiere (b);
    initialisiere (c); initialisiere (d);

    time_stamp();
    std::cout << "SimpleArrayCS ...";
```

```
54      bubblesort(a);
        std::cout << a[1] << " " << time_stamp() << " sec" << std::
            endl;
        std::cout << "SimpleArray ...";
57      bubblesort(b);
        std::cout << b[1] << " " << time_stamp() << " sec" << std::
            endl;
        std::cout << "SimpleFloatArrayV ...";
60      bubblesort(c);
        std::cout << c[1] << " " << time_stamp() << " sec" << std::
            endl;
        // std::cout << "ListFloatArray ...";
63      // bubblesort(d);
        // std::cout << time_stamp() << " sec" << std::endl;
    }
```

Ergebnis:

Ergebnisse vom WS 2002/2003:

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	—	—	—	—

WS 2011/2012, gcc 4.5.0, 2.26 GHz Intel Core 2 Duo:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.0029	0.0089	0.034	0.138	0.557	2.205
`SimpleArray`	0.0031	0.0098	0.039	0.156	0.622	2.499
`SimpleFloatArrayV`	0.0110	0.0204	0.083	0.330	1.322	5.288

WS 2014/2015, gcc 4.9.2 -O3, 2.6 GHz Intel Core i7:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.00096	0.0032	0.013	0.070	0.328	1.4048
`SimpleArray`	0.00096	0.0034	0.013	0.0727	0.329	1.4095
`SimpleFloatArrayV`	0.0008	0.0027	0.011	0.0579	0.297	1.4353
`ListFloatArray`	1.056	8.999	—	—	—	—

WS 2015/2016, gcc 5.2.1 -O3, 4.0 GHz Intel Core i7-4790K:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.000238	0.000936	0.00372	0.0148	0.0594	0.240
`SimpleArray`	0.000237	0.000938	0.00371	0.0148	0.0593	0.240
`SimpleFloatArrayV`	0.000185	0.000716	0.00282	0.0112	0.0447	0.182
`ListFloatArray`	0.646912	6.68187	—	—	—	—

WS 2015/2016, gcc 5.2.1, no opt, 4.0 GHz Intel Core i7-4790K:

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.00208	0.00837	0.0330	0.132	0.545	2.192
`SimpleArray`	0.00208	0.00833	0.0329	0.132	0.538	2.200
`SimpleFloatArrayV`	0.00241	0.00948	0.0379	0.152	0.613	2.511
`ListFloatArray`	1.846	14.2928	—	—	—	—

WS 2019/2020, gcc 8.3.0, -O3, 2,7 GHz Intel Core i7

n	1000	2000	4000	8000	16000	32000
`SimpleArrayCS`	0.0012	0.004	0.015	0.070	0.327	1.417
`SimpleArray`	0.0009	0.0035	0.013	0.064	0.308	1.431
`SimpleFloatArrayV`	0.001	0.0038	0.014	0.068	0.330	1.490
`ListFloatArray`	0.87.	7.0	—	—	—	—

Bemerkung:

- Die ersten fünf Zeilen zeigen deutlich den $O(n^2)$-Aufwand: Verdopplung von n bedeutet vierfache Laufzeit.
- Die Zeilen fünf und vier zeigen die Laufzeit für die Variante mit einem virtuellem `operator[]` bzw. eine Version der Klassenschablone, bei der das Schlüsselwort `inline` vor der Methodendefinition des `operator[]` weggelassen wurde. Diese beiden Varianten sind etwa viermal langsamer als die vorherigen.
- Eine Variante mit eingebautem Feld (nicht vorgestellt, ohne Klassen) ist am schnellsten, gefolgt von den zwei Varianten mit Klassenschablonen, die unwesentlich langsamer sind.
- Beachte auch den Einfluss des Optimizers (gcc-Option `-O3`).
- `ListFloatArray` ist die listenbasierte Darstellung des Feldes mit Index-Wert-Paaren. Diese hat Komplexität $O(n^3)$, da nun die Zugriffe auf die Feldelemente Komplexität $O(n)$ haben.

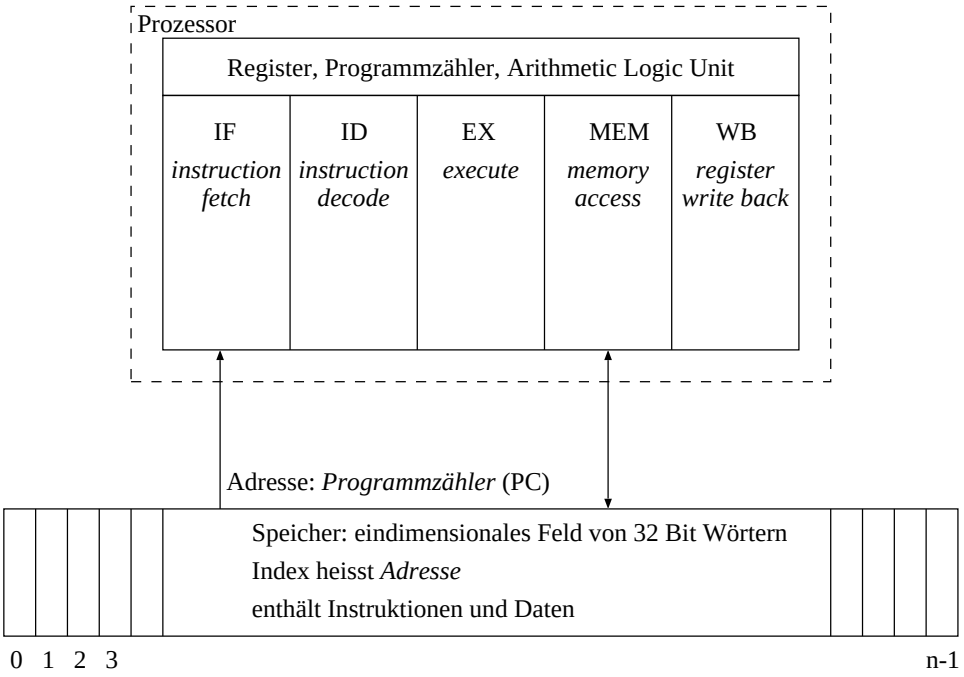
Frage: Warum sind die Varianten auf Schablonenbasis (mit inlining) schneller als die Variante mit virtueller Methode?

RISC

Bezeichnung: RISC steht für *Reduced Instruction Set Computer* und steht für eine Kategorie von Prozessorarchitekturen mit verhältnismäßig einfachem Befehlssatz. Gegenpol: CISC=*Complex Instruction Set Computer*.

Geschichte: RISC stellt heutzutage den Großteil aller Prozessoren dar (vor allem bei eingebetteten Systemen (Handy, PDA, Spielekonsole, etc.), wo das Verhältnis Leistung/Verbrauch wichtig ist). Für PCs ist allerdings noch mit den Intel-Chips die CISC-Technologie dominant (mittlerweile wurden aber auch dort viele RISC-Techniken übernommen).

Aufbau eines RISC-Chips



Befehlszyklus Bezeichnung: Ein typischer RISC-Befehl lässt sich in Teilschritte unterteilen, die von verschiedener Hardware (in der CPU) ausgeführt werden:

1. IF: Holen des nächsten Befehls aus dem Speicher. Ort: *Programmzähler*.
2. ID: Dekodieren des Befehls, Auslesen der beteiligten Register.
3. EX: Eigentliche Berechnung (z. B. Addieren zweier Zahlen).
4. MEM: Speicherzugriff (entweder Lesen oder Schreiben).
5. WB: Rückschreiben der Ergebnisse in Register.

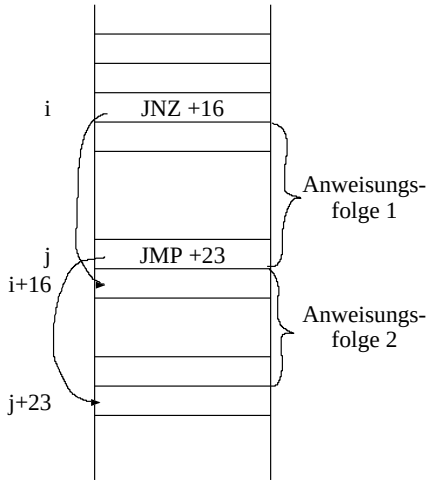
Dies nennt man Befehlszyklus (*instruction cycle*).

Pipelining Diese Stadien werden nun für aufeinanderfolgende Befehle überlappend ausgeführt (Pipelining).

IF	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6	Instr7
ID	—	Instr1	Instr2	Instr3	Instr4	Instr5	Instr6
EX	—	—	Instr1	Instr2	Instr3	Instr4	Instr5
MEM	—	—	—	Instr1	Instr2	Instr3	Instr4
WB	—	—	—	—	Instr1	Instr2	Instr3

Probleme mit Pipelining Sehen wir uns an, wie eine `if`-Anweisung realisiert wird:

```
if ( a == 0 )
{
  <Anweisungsfolge 1>
}
else
{
  <Anweisungsfolge 2>
}
```



Problem: Das Sprungziel des Befehls `JNZ +16` steht erst am Ende der dritten Stufe der Pipeline (EX) zur Verfügung, da ein Register auf 0 getestet und 16 auf den PC addiert werden muss.

Frage: Welche Befehle sollen bis zu diesem Punkt weiter angefangen werden?

Antwort:

- Gar keine, dann bleiben einfach drei Stufen der Pipeline leer (pipeline stall).
- Man *rät* das Sprungziel (branch prediction unit) und führt die nachfolgenden Befehle spekulativ aus (ohne Auswirkung nach aussen). Notfalls muss man die Ergebnisse dieser Befehle wieder verwerfen.

Bemerkung: Selbst das Ziel eines unbedingten Sprungbefehls stünde wegen der Addition des Offset auf den PC erst nach der Stufe EX zur Verfügung (es sei denn, man hat extra Hardware dafür).

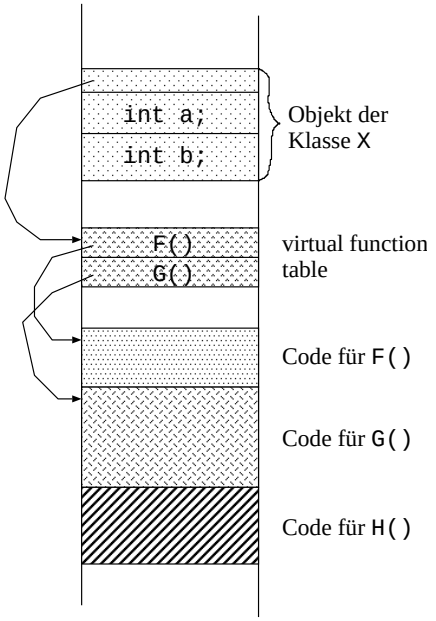
Funktionsaufrufe Ein Funktionsaufruf (Methodenaufruf) besteht aus folgenden Operationen:

- Sicherung der Rücksprungadresse auf dem Stack
- ein unbedingter Sprungbefehl
- der Rücksprung an die gespeicherte Adresse
- + eventuelle Sicherung von Registern auf dem Stack

Diese Liste gilt genauso für CISC-Architekturen. Ein Funktionsaufruf ist also normalerweise mit erheblichem Aufwand verbunden. Darüberhinaus optimiert der Compiler nicht über Funktionsaufrufe hinweg, was zu weiteren Geschwindigkeitseinbussen führt.

Realisierung virtueller Funktionen Betrachte folgende Klassendefinition und ein konkretes Objekt im Speicher:

```
class X
{
public:
  int a;
  int b;
  virtual void F();
  virtual void G();
  void H();
};

X x;
x.F();
x.H();
```



Bemerkung:

- Für jede Klasse gibt es eine Tabelle mit Zeigern auf den Programmcode für die virtuellen Funktionen dieser Klasse. Diese Tabelle heißt *virtual function table* (VFT).
- Jedes Objekt einer Klasse, die virtuelle Funktionen enthält, besitzt einen Zeiger auf die VFT der zugehörigen Klasse. Dies entspricht im wesentlichen der Typinformation, die bei Sprachen mit dynamischer Typbindung den Daten hinzugefügt ist.
- Beim Aufruf einer virtuellen Methode generiert der Übersetzer Code, welcher der VFT des Objekts die Adresse der aufzurufenden Methode entnimmt und dann den Funktionsaufruf durchführt. Welcher Eintrag der VFT zu entnehmen ist, ist zur *Übersetzungszeit* bekannt.
- Der Aufruf nichtvirtueller Funktionen geschieht ohne VFT. Klassen (und ihre zugehörigen Objekte) ohne virtuelle Funktionen brauchen keinen Zeiger auf eine VFT.
- Für den Aufruf virtueller Funktionen ist immer ein Funktionsaufruf notwendig, da erst zur Laufzeit bekannt ist, welche Methode auszuführen ist.

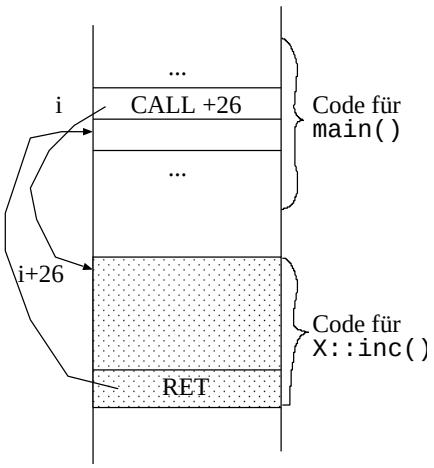
Inlining Problem: Der Funktionsaufruf sehr kurzer Funktionen ist relativ langsam.

Beispiel:

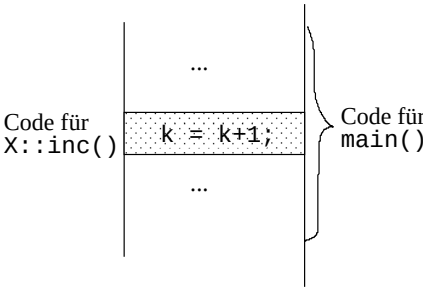
```
class X {
public:
  void inc();
private:
  int k;
};

inline void X::inc()
{
  k = k + 1;
}
```

```
void main()
{
  X x;
  x.inc();
}
```

Ohne das Schlüsselwort `inline` in der Methodendefinition generiert der C++-Übersetzer einen Funktionsaufruf für `inc()`:



Mit dem Schlüsselwort `inline` in der Methodendefinition setzt der Übersetzer den Code der Methode am Ort des Aufrufes direkt ein falls dies möglich ist:



Bemerkung:

- Inlining ändert nichts an der Semantik des Programmes.
- Das Schlüsselwort `inline` ist nur ein *Vorschlag* an den Compiler. Z. B. wird es für rekursive Funktionen ignoriert.
- Virtuelle Funktionen können nicht inline ausgeführt werden, da die auszuführende Methode zur Übersetzungszeit nicht bekannt ist.
- *Aber:* Änderungen der Implementation einer Inline-Funktion in einer Bibliothek machen normalerweise die erneute Übersetzung von anderen Programmteilen notwendig!

Bemerkung: Es sei auch nochmal eindringlich an Knuth's Wort *"Premature optimization is the root of all evil"* erinnert. Bevor Sie daran gehen, Ihr Programm durch Elimination virtueller Funktionen und Inlining unflexibler zu machen, sollten Sie folgendes tun:

1. Überdenken Sie den Algorithmus!
2. Messen Sie, wo der „Flaschenhals“ wirklich liegt (Profiling notwendig).

3. Überlegen Sie, ob die erreichbare Effizienzsteigerung den Aufwand wert ist.

Beispielsweise ist die *einzig sinnvolle Verbesserung* für das Sortierbeispiel am Anfang dieses Abschnitts das Verwenden eines besseren Algorithmus!

11.4 Zusammenfassung

- Klassenschablonen definieren parametrisierte Datentypen und sind daher besonders geeignet, um allgemein verwendbare Konzepte (ADT) zu implementieren.
- Funktionsschablonen definieren parametrisierte Funktionen, die auf verschiedenen Datentypen (mit gleicher Schnittstelle) operieren.
- In beiden Fällen werden konkrete Varianten der Klassen/Funktionen zur Übersetzungszeit erzeugt und übersetzt (generische Programmierung).
- Diese Techniken sind für Sprachen mit *dynamischer Typbindung* meist unnötig. Solche Sprachen brauchen aber in vielen Fällen Typabfragen zur Laufzeit, was dazu führt, dass der erzeugte Code nicht mehr hocheffizient ist.

Nachteile der generischen Programmierung

- Es wird viel Code erzeugt. Die Übersetzungszeiten template-intensiver Programme können unerträglich lang sein.
- Es ist keine *getrennte Übersetzung* möglich. Der Übersetzer muss die Definition aller vorkommenden Schablonen kennen. Dasselbe gilt für Inline-Funktionen. Dies erfordert dann z. B. auch spezielle Softwarelizenzen.
- Das Finden von Fehlern in Klassen/Funktionenschablonen ist erschwert, da der Code für eine konkrete Variante nirgends existiert. Empfehlung: testen Sie zuerst mit einem konkreten Datentyp und machen Sie dann eine Schablone daraus.

12 Containerklassen

12.1 Motivation

Bezeichnung: Klassen, die eine Menge anderer Objekte verwalten (man sagt aggregieren) nennt man Containerklassen.

Beispiele: Wir hatten: Liste, Stack, Feld. Weitere sind: *binärer Baum* (*binary tree*), Warteschlange (*queue*), Menge (*set*), Abbildung (*map*), ...

Bemerkung: Diese Strukturen treten sehr häufig als Komponenten in größeren Programmen auf. Ziel von Containerklassen ist es, diese Bausteine in wiederverwendbarer Form zur Verfügung zu stellen (*code reuse*).

Vorteile:

- Weniger Zeitaufwand in Entwicklung und Fehlersuche.
- Klarere Programmstruktur, da man auf einer höheren Abstraktionsebene arbeitet.

Werkzeug: Das Werkzeug zur Realisierung effizienter und flexibler Container in C++ sind Klassenschablonen.

Bemerkung: In der C++-Standardbibliothek sind Containerklassen in hoher Qualität im Rahmen der Standard Template Library (STL) implementiert. Diese sollten in der Praxis verwendet werden.

In diesem Abschnitt möchten wir trotzdem demonstrieren wie solche Containerklassen implementiert werden können und implementieren daher unsere eigenen Varianten. Allerdings wird nicht behauptet, dass die STL-Container genau so implementiert sind, es wird nur das Prinzip demonstriert.

Ziel: Sie sind am Ende dieses Kapitels motiviert, die STL zu verwenden und können die Konzepte verstehen.

12.2 Kurzer Ausflug in die STL

Der Container std::vector<T> entspricht unserem dynamischen Feld von Elementen vom Typ T mit konsekutivem Indexbereich ab 0:

```
std::vector<int> x(10,77); // 10 Elemente mit Wert 77
x[0] = x[5];
```

Alternativ kann man den Container folgendermaßen erzeugen

```
std::vector<int> y(0); // leerer Container
for (int i=0; i<10; ++i) y.push_back(i);
```

Durchlaufen der Elemente gelingt hier mittels

```
for (int i=0; i<x.size(); ++i) x[i] = i;
```

Interessant ist auch die Methode x.resize(100) zur Veränderung der Größe

Der Container std::array<T,n> realisiert ein Feld mit fester Größe, welche zur Übersetzungszeit bekannt sein muss:

```
std::array<int,10> a;      // 10 int Elemente
std::array<double,100> b; // 100 double Elemente
```

Damit ist std::array<T,n> ein Ersatz für eingebaute Felder ohne dessen Nachteile:

- a.size() liefert die Größe des Feldes
- a ist kein Zeiger T*

Allerdings bietet std::array<T,n> nicht die Methode push_back, da ja seine Größe unveränderbar ist

Nun gibt es auch andere Container, z.B. die (doppelt verkettete) Liste std :: list <T>. Die kann man mit der push_back Methode analog erzeugen

```
std::list<int> l;
for (int i=0; i<10; ++i) l.push_back(i);
```

Das Durchlaufen einer Liste vom Anfang zum Ende gelingt mittels Iteratoren:

```
for (std::list<int>::iterator it=l.begin();
        it!=l.end(); ++it)
  sum += *it;
```

Ein Iterator verhält sich wie ein Zeiger auf ein Element der Liste. Auch das Einfügen und löschen geht damit:

```
std::list<int>::iterator it1 = l.begin();
++it1;
l.insert(it1,17); // üfge neues Element *vor* ein
```

Mittels Iteratoren gelingt es auch verschiedene Container in der selben Art und Weise zu traversieren. Z.B. klappt auch mit dem vector:

```
int sum = 0;
for (std::vector<int>::iterator it=x.begin();
       it!=x.end(); ++it)
  sum += *it;
```

Die STL erlaubt die Programmierung generischer Algorithmen die mittels duck typing auf verschiedenen Containern arbeiten können:

```
template<typename C>
void sort (C& c)
{ ... }
```

- Container werden in Kategorien eingeteilt (nur vorwärts iterierbar, rückwärts iterierbar, wahlfreier Zugriff)
- Beispiel: maximales Element finden

Übersicht über die Container:

- std :: vector dynamisches Feld, std :: array Feld fester Größe, std :: list doppelt verkettete Liste, std :: slist einfach verkettete Liste
- std :: queue first in first out, std :: stack
- std :: set Menge, enthalten sein, Vereinigung und Schnitt, std :: priority_queue größtes Element zuerst, std :: map Abbildung eines Schlüssels zu Werten

12.3 Listenschablone

Bei diesem Entwurf ist die Idee, das Listenelement und damit auch die Liste als Klassenschablone zu realisieren. In jedem Listenelement wird ein Objekt der Klasse `T`, dem Schablonenparameter, gespeichert.

Programm: Definition und Implementation (Liste.hh)

```
template<class T>
class List {
public:
  // Infrastruktur
  List() { _first=0; }
  ~List();

  // Listenelement als nested class
  class Link {
    Link* _next;
  public:
    T item;
    Link (T& t) {item=t;}
    Link* next () {return _next;}
    friend class List<T>;
  };

  Link* first() {return _first;}
  void insert (Link* where, T t);
  void remove (Link* where);

private:
  Link* _first;
  // privater Copy-Konstruktor und Zuweisungs-
  // operator da Defaultvarianten zu fehlerhaftem
  // Verhalten fuehren
  List (const List<T>& l) {};
  List<T>& operator= (const List<T>& l) {};
};

template<class T> List<T>::~List()
{
  Link* p = _first;
  while (p!=0)
    {
      Link* q = p;
      p = p->next();
      delete q;
    }
}

template<class T>
void List<T>::insert (List<T>::Link* where, T t)
{
  Link* ins = new Link(t);
  if (where==0)
    {
      ins->_next = _first;
```

```
      _first = ins;
    }
  else
    {
      ins->_next = where->_next;
      where->_next = ins;
    }
}

template<class T>
void List<T>::remove (List<T>::Link* where)
{
  Link* p;
  if (where==0)
    {
      p = _first;
      if (p!=0) _first = p->_next;
    }
  else
    {
      p = where->_next;
      if (p!=0) where->_next = p->_next;
    }
  delete p;
}
```

Programm: **Verwendung (UseListe.cc)**

```
#include<iostream>
#include"Liste.hh"

int main () {
  List<int> list;

  list.insert(0,17);
  list.insert(0,34);
  list.insert(0,26);

  for (List<int>::Link* l=list.first();
       l!=0; l=l->next())
    std::cout << l->item << std::endl;
  for (List<int>::Link* l=list.first();
       l!=0; l=l->next())
    l->item = 23;
}
```

Bemerkung:

- Diese Liste ist homogen, d. h. alle Objekte im Container haben den gleichen Typ. Eine heterogene Liste könnte man als Liste von Zeigern auf eine gemeinsame Basisklasse realisieren.

- Speicherverwaltung wird von der Liste gemacht. Listen können kopiert und als Parameter übergeben werden (sofern Copy-Konstruktor und Zuweisungsoperator noch mittels deep copy implementiert werden).
- Zugriff auf die Listenelemente erfolgt über eine offengelegte *nested class*. Die Liste wird als friend deklariert, damit die Liste den `next`-Zeiger manipulieren kann, nicht jedoch der Benutzer der Liste.

12.4 Iteratoren

Problem: Grundoperation aller Container sind

- Durchlaufen aller Elemente,
- Zugriff auf Elemente.

Um Container austauschbar verwenden zu können, sollten diese Operationen mit der gleichen Schnittstelle möglich sein. Die Schleife für eine Liste sah aber ganz anders aus als bei einem Feld.

Abhilfe: Diese Abstraktion realisiert man mit Iteratoren. Iteratoren sind zeigerähnliche Objekte, die auf ein Objekt im Container zeigen (obwohl der Iterator nicht als Zeiger realisiert sein muss).

Prinzip:

```
template <class T> class Container
{
public:
  class Iterator
  {  // nested class definition
     ...
  public:
    Iterator();
    bool operator!=( Iterator x );
    bool operator==( Iterator x );
    Iterator operator++();       // prefix
    Iterator operator++( int );  // postfix
    T& operator*() const;
    T* operator->() const;
    friend class Container<T>;
  };
  Iterator begin() const;
  Iterator end() const;
  ... // äSpezialitten des Containers
};

// Verwendung
Container<int> c;
for ( Container<int>::Iterator i=c.begin(); i!=c.end(); ++i )
  std::cout << *i << std::endl;
```

Bemerkung:

- Der `Iterator` ist als Klasse innerhalb der Containerklasse definiert. Dies nennt man eine geschachtelte Klasse (*nested class*).
- Damit drückt man aus, dass `Container` und `Iterator` zusammengehören. Jeder Container wird seine eigene Iteratorklasse haben.
- Innerhalb von `Container` kann man `Iterator` wie jede andere Klasse verwenden.
- `friend class Container<T>` bedeutet, dass die Klasse `Container<T>` auch Zugriff auf die *privaten* Datenmitglieder der Iteratorklasse hat.
- Die Methode `begin()` des Containers liefert einen Iterator, der auf das erste Element des Containers zeigt.
- `++i` bzw. `i++` stellt den Iterator auf das *nächste* Element im Container. Zeigte der Iterator auf das letzte Element, dann ist der Iterator gleich dem von `end()` gelieferten Iterator.
- `++i` bzw. `i++` manipulieren den Iterator für den sie aufgerufen werden. Als Rückgabewert liefert `++i` den neuen Wert des Iterators, `i++` jedoch den alten.
- Bei der Definition unterscheiden sie sich dadurch, dass der Postfix-Operator noch ein **int**-Argument erhält, das aber keine Bedeutung hat.
- `end()` liefert einen Iterator, der auf „das Element nach dem letzten Element“ des Containers zeigt (siehe oben).
- `*i` liefert eine Referenz auf das Objekt im Container, auf das der Iterator `i` zeigt. Damit kann man sowohl `x = *i` als auch `*i = x` schreiben.
- Ist das Objekt im Container von einem zusammengesetzten Datentyp (also `struct` oder `class`), so kann mittels `i-><Komponente>` eine Komponente selektiert werden. Der Iterator verhält sich also wie ein Zeiger.

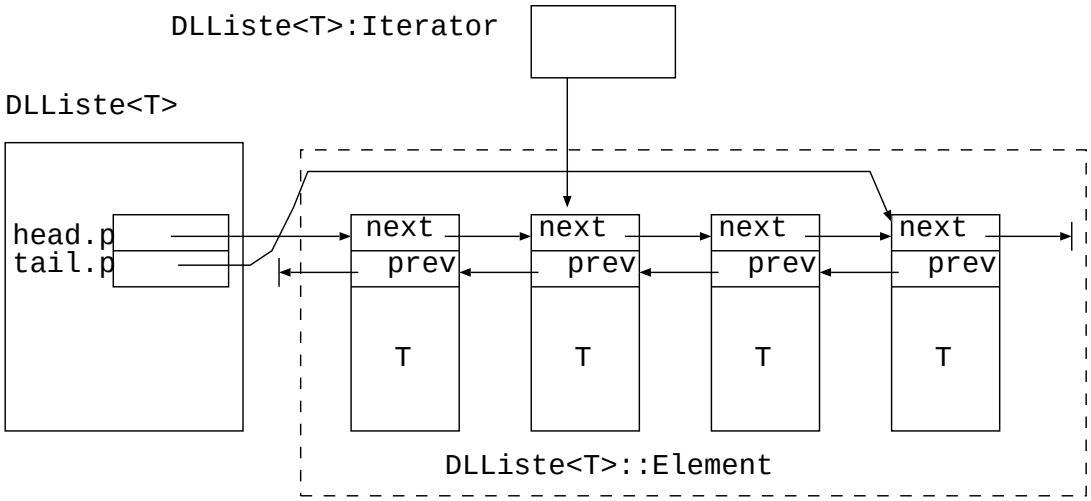
Machen wir ein Beispiel ...

12.5 Doppelt verkettete Liste

Anforderungen:

- Vorwärts- *und* Rückwärtsdurchlauf
- Das Einfügen vor oder nach einem Element soll eine $O(1)$-Operation sein. Die Position wird durch einen Iterator angegeben.
- Das Entfernen eines Elementes soll eine $O(1)$-Operation sein. Das zu entfernende Element wird wieder durch einen Iterator angegeben
- Für die Berechnung der Größe der Liste akzeptieren wir einen $O(N)$-Aufwand.

Struktur



Bemerkung:

- Intern werden die Listenelemente durch den Datentyp `Element` repräsentiert. Dieser private, geschachtelte, zusammengesetzte Datentyp ist außerhalb der Klasse nicht sichtbar.
- Die Einfügeoperationen erhalten Objekte vom Typ `T`, erzeugen dynamisch ein Listenelement und *kopieren* das Objekt in das Listenelement.
- Damit kann man Listen für beliebige Datentypen erzeugen. Die Manipulation gelingt mit Hilfe der Iteratorschnittstelle. Der Iterator kapselt insbesondere den Zugriff auf die außerhalb der Liste nicht bekannten `Element`-Objekte.

Implementation

Programm: **DLL.hh**

```
template<class T>
class DLList {

  // das interne Listenelement
  struct Element {
    Element* next;
    Element* prev;
    T item;
    Element (T &t) {
      item = t;
      next = prev = 0;
    }
  };

public:
  typedef T MemberType;   // Merke Grundtyp
```

```
18      // der iterator kapselt Zeiger auf Listenelement
        class Iterator {
        private:
21        Element* p;
        public:
          Iterator () { p=0; }
24        Iterator (Element* q) { p=q; }
          bool operator!= (Iterator x) {
            return p!=x.p;
27        }
          bool operator== (Iterator x) {
            return p==x.p;
30        }
          Iterator operator++ () { // prefix
            p=p->next;
33          return *this;
          }
          Iterator operator++ (int) { // postfix
36          Iterator tmp = *this;
            p=p->next;
            return tmp;
39        }
          Iterator operator-- () { // prefix
            p=p->prev;
42          return *this;
          }
          Iterator operator-- (int) { // postfix
45          Iterator tmp = *this;
            p=p->prev;
            return tmp;
48        }
          T& operator* () { return p->item; }
          T* operator-> () { return &(p->item); }
51        friend class DLList<T>; // Liste man. p
        } ;

54      // Iteratoren
        Iterator begin () const {return head;}
        Iterator end () const {return Iterator();}
57      Iterator rbegin () const {return tail;}
        Iterator rend () const {return Iterator();}

60      // Konstruktion, Destruktion, Zuweisung
        DLList ();
        DLList (const DLList<T>& list);
63      DLList<T>& operator= (const DLList<T>&);
        ~DLList();

66      // Listenmanipulation
```

```
    Iterator insert (Iterator i, T t);// einf. vor i
    void erase (Iterator i);
69  void append (const DLList<T>& l);
    void clear ();
    bool empty () const;
72  int size () const;
    Iterator find (T t) const;

75  private:
    Iterator head;    // erstes Element der Liste
    Iterator tail;    // letztes Element der Liste
78  };


81  // Insertion
    template<class T>
    typename DLList<T>::Iterator
84  DLList<T>::insert (Iterator i, T t)
    {
    Element* e = new Element(t);
87  if (empty())
      {
        assert(i.p==0);
90      head.p = tail.p = e;
      }
    else
93    {
        e->next = i.p;
        if (i.p!=0)
96        {   // insert before i
            e->prev = i.p->prev;
            i.p->prev = e;
99          if (head==i)
              head.p=e;
          }
102     else
          {   // insert at end
            e->prev = tail.p;
105         tail.p->next = e;
            tail.p = e;
          }
108   }
    return Iterator(e);
    }
111
    template<class T>
    void DLList<T>::erase (Iterator i)
114 {
    if (i.p==0) return;
```

```
117     if (i.p->next!=0)
          i.p->next->prev = i.p->prev;
        if (i.p->prev!=0)
120       i.p->prev->next = i.p->next;

        if (head==i) head.p=i.p->next;
123     if (tail==i) tail.p=i.p->prev;

        delete i.p;
126   }

      template<class T>
129   void DLList<T>::append (const DLList<T>& l) {
        for (Iterator i=l.begin(); i!=l.end(); i++)
          insert(end(),*i);
132   }

      template<class T>
135   bool DLList<T>::empty () const {
        return begin()==end();
      }
138
      template<class T>
      void DLList<T>::clear () {
141     while (!empty())
          erase(begin());
      }
144
      // Constructors
      template<class T> DLList<T>::DLList () {}
147
      template<class T>
      DLList<T>::DLList (const DLList<T>& list) {
150     append(list);
      }

153   // Assignment
      template<class T>
      DLList<T>&
156   DLList<T>::operator= (const DLList<T>& l) {
        if (this!=&l) {
          clear();
159       append(l);
        }
        return *this;
162   }

      // Destructor
165   template<class T> DLList<T>::~DLList() { clear(); }
```

```
// Size method
template<class T> int DLList<T>::size () const {
  int count = 0;
  for (Iterator i=begin(); i!=end(); i++)
    count++;
  return count;
}

template<class T>
typename DLList<T>::Iterator DLList<T>::find (T t) const {
  DLList<T>::Iterator i = begin();
  while (i!=end())
    {
      if (*i==t) break;
      i++;
    }
  return i;
}

template <class T>
std::ostream& operator<< (std::ostream& s, DLList<T>& a) {
  s << "(";
  for (typename DLList<T>::Iterator i=a.begin();
       i!=a.end(); i++)
    {
      if (i!=a.begin()) s << " ";
      s << *i;
    }
  s << ")" << std::endl;
  return s;
}
```

Verwendung

Programm: **UseDLL.cc**

```
#include<cassert>
#include<iostream>
#include"DLL.hh"
#include"Zufall.cc"

int main ()
{
    Zufall z(87124);
    DLList<int> l1,l2,l3; // std::list<int>

    // Erzeuge 3 Listen mit je 5 Zufallszahlen
    for (int i=0; i<5; ++i)
        l1.insert(l1.end(), i);
```

```
    for (int i=0; i<5; i=i+1)
        l2.insert(l2.end(), z.ziehe_zahl());
    for (int i=0; i<5; i=i+1)
        l3.insert(l3.end(), z.ziehe_zahl());

    // Loesche alle geraden in der ersten Liste
    DLList<int>::Iterator i,j;
    i=l1.begin();
    while (i!=l1.end())
    {
        j=i; // merke aktuelles Element
        ++i; // gehe zum naechsten
        if (*j%2==0) l1.erase(j);
    }

    // Liste von Listen ...
    DLList<DLList<int> > ll;
    ll.insert(ll.end(),l1);
    ll.insert(ll.end(),l2);
    ll.insert(ll.end(),l3);
    std::cout << ll << std::endl;
    std::cout << "Laenge: " << ll.size() << std::endl;
}
```

Diskussion

- Den Rückwärtsdurchlauf durch eine Liste `c` erreicht man durch:

```
for ( DLList<int>::Iterator i=c.rbegin();
      i!=c.rend(); --i )
  std::cout << *i << endl;
```

- Die Objekte (vom Typ `T`) werden beim Einfügen in die Liste kopiert. Abhilfe: Liste von Zeigern auf die Objekte, z. B. `DLList<int*>`.
- Die Schlüsselworte **const** in der Definition von `begin`, `end`, ... bedeuten, dass diese Methoden ihr Objekt nicht ändern.
- Innerhalb einer Template-Definition werden geschachtelte Klassen nicht als Typ erkannt. Daher muss man den Namen explizit mittels **typename** als Typ kennzeichnen.

Beziehung zur STL-Liste

Die entsprechende STL-Schablone heißt `list` und unterscheidet sich von unserer Liste unter anderem in folgenden Punkten:

- Man erhält die Funktionalität durch **#include** <list>.
- Die Iterator-Klasse heißt `iterator` statt `Iterator`.

- Es gibt zusätzlich einen `const_iterator`. Auch unterscheiden sich Vorwärts- und Rückwärtsiteratoren (`reverse_iterator`).
- Sie hat einige Methoden mehr, z. B. `push_front`, `push_back`, `front`, `back`, `pop_front`, `pop_back`, `sort`, `reverse`, ...
- Die Ausgabe über „`std::cout <<`“ ist nicht definiert.

12.6 Feld

Wir fügen nun die Iterator-Schnittstelle unserer `SimpleArray<T>`-Schablone hinzu.

Programm: **(Array.hh)**

```
template <class T> class Array {
public:
  typedef T MemberType;  // Merke Grundtyp

  // Iterator fuer die Feld-Klasse
  class Iterator {
  private:
    T* p; // Iterator ist ein Zeiger ...
    Iterator(T* q) {p=q;}
  public:
    Iterator() {p=0;}
    bool operator!= (Iterator x) {
      return (p!=x.p);
    }
    bool operator== (Iterator x) {
      return (p==x.p);
    }
    Iterator operator++ () {
      p++;
      return *this;
    }
    Iterator operator++ (int) {
      Iterator tmp = *this;
      ++*this;
      return tmp;
    }
    T& operator* () const {return *p;}
    T* operator-> () const {return p;}
    friend class Array<T>;
  } ;

  // Iterator Methoden
  Iterator begin () const {
    return Iterator(p);
  }
  Iterator end () const {
    return Iterator(&(p[n])); // ja, das ist ok!
```

```
  }

  // Konstruktion; Destruktion und Zuweisung
  Array(int m) {
    n = m;
    p = new T[n];
  }
  Array (const Array<T>&);
  Array<T>& operator= (const Array<T>&);
  ~Array() {
    delete[] p;
  }

  // Array manipulation
  int size () const {
    return n;
  }
  T& operator[](int i) {
    return p[i];
  }

private:
  int n;  // Anzahl Elemente
  T *p;   // Zeiger auf built-in array
};

// Copy-Konstruktor
template <class T>
Array<T>::Array (const Array<T>& a) {
  n = a.n;
  p = new T[n];
  for (int i=0; i<n; i=i+1)
    p[i]=a.p[i];
}

// Zuweisung
template <class T>
Array<T>& Array<T>::operator= (const Array<T>& a) {
  if (&a!=this) {
    if (n!=a.n) {
      delete[] p;
      n = a.n;
      p = new T[n];
    }
    for (int i=0; i<n; i=i+1) p[i]=a.p[i];
  }
  return *this;
}

// Ausgabe
```

```
template <class T>
std::ostream& operator<< (std::ostream& s, Array<T>& a) {
  s << "array " << a.size() <<
    " elements = [" << std::endl;
  for (int i=0; i<a.size(); i++)
    s << "     " << i << "  " << a[i] << std::endl;
  s << "]" << std::endl;
  return s;
}
```

Bemerkung:

- Der Iterator ist als Zeiger auf ein Feldelement realisiert.
- Die Schleife

  `for (Array<int>::Iterator i=a.begin(); i!=a.end(); ++i) ...`

  entspricht nach Inlining der Methoden einfach

  `for (int* p=a.p; p!=&a[100]; p=p+1) ...`

  und ist somit nicht langsamer als handprogrammiert!
- Man beachte auch die Definition von `MemberType`. Dies ist praktisch innerhalb eines Template **template** <**class** C>, wo der Datentyp eines Containers `C` dann als `C::MemberType` erhalten werden kann.

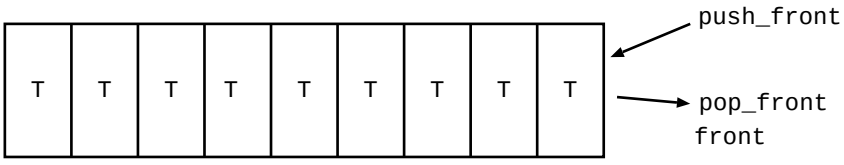
Programm: **Gleichzeitige Verwendung** `DLList/Array` **(UseBoth.cc)**:

```
#include<cassert>
#include<iostream>

#include"Array.hh"
#include"DLL.hh"
#include"Zufall.cc"

int main () {
  Zufall z(87124);
  Array<int> a(5);
  DLList<int> l;

  // Erzeuge Array und Liste mit 5 Zufallszahlen
  for (int i=0; i<5; i=i+1) a[i] = z.ziehe_zahl();
  for (int i=0; i<5; i=i+1)
    l.insert(l.end(), z.ziehe_zahl());

  // Benutzung
  for (Array<int>::Iterator i=a.begin(); i!=a.end(); i++)
    std::cout << *i << std::endl;

  std::cout << std::endl;

  for (DLList<int>::Iterator i=l.begin(); i!=l.end(); i++)
```

```
    std::cout << *i << std::endl;
}
```

Bemerkung: Die STL-Version von `Array` erhält man mit **#include** <vector>. Die Klassenschablone heißt `vector` anstatt `Array`.

12.7 Stack



Schnittstelle:

- Konstruktion eines Stack.
- Einfügen eines Elementes vom Typ `T` oben (`push`).
- Entfernen des obersten Elementes (`pop`).
- Inspektion des obersten Elementes (`top`).
- Test ob Stack voll oder leer (`empty`).

Programm: **Implementation über DLList (Stack.hh)**

```
template<class T>
class Stack : private DLList<T> {
public :
  // Default-Konstruktoren + Zuweisung OK

  bool empty () {return DLList<T>::empty();}
  void push (T t) {
    this->insert(DLList<T>::begin(),t);
  }
  T top () {return *DLList<T>::begin();}
  void pop () {this->erase(DLList<T>::begin());}
} ;
```

Bemerkung:

- Wir haben den Stack als Spezialisierung der doppelt verketteten Liste realisiert. Etwas effizienter wäre die Verwendung einer einfach verketteten Liste gewesen.
- Auffallend ist, dass die Befehle `top`/`pop` getrennt existieren (und `pop` keinen Wert zurückliefert). Verwendet werden diese Befehle nämlich meist gekoppelt, so dass auch eine Kombination `pop` ← `top`+`pop` nicht schlecht wäre.

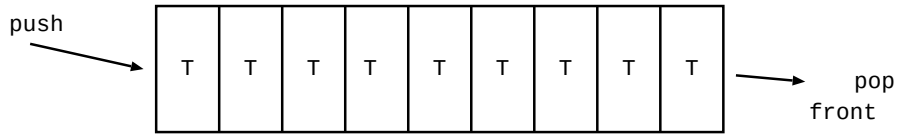
Programm: **Anwendung: (UseStack.cc)**

```
#include<cassert>
#include<iostream>

#include"DLL.hh"
#include"Stack.hh"

int main ()
{
  Stack<int> s1;
  for (int i=1; i<=5; i++)
    s1.push(i);

  Stack<int> s2(s1);
  s2 = s1;
  while (!s2.empty())
    {
      std::cout << s2.top() << std::endl;
      s2.pop();
    }
}
```

Bemerkung: Die STL-Version erhält man durch **#include** <stack>. Die Klassenschablone heißt dort `stack` und hat im Wesentlichen dieselbe Schnittstelle.

12.8 Queue

Eine Queue ist eine Struktur, die Einfügen an einem Ende und Entfernen nur am anderen Ende erlaubt:



Anwendung: Warteschlangen.

Schnittstelle:

· Konstruktion einer leeren Queue
· Einfügen eines Elementes vom Typ `T` am Ende
· Entfernen des Elementes am Anfang
· Inspektion des Elementes am Anfang
· Test ob Queue leer

Programm: **(Queue.hh)**

```
template<class T>
class Queue : public DLList<T> {
public :
  // Default-Konstruktoren + Zuweisung OK
  bool empty () {
    return DLList<T>::empty();
  }
  T front () {
    return *DLList<T>::begin();
  }
  T back () {
    return *DLList<T>::rbegin();
  }
  void push (T t) {
    this->insert(DLList<T>::end(),t);
  }
  void pop () {
    this->erase(DLList<T>::begin());
  }
};
```

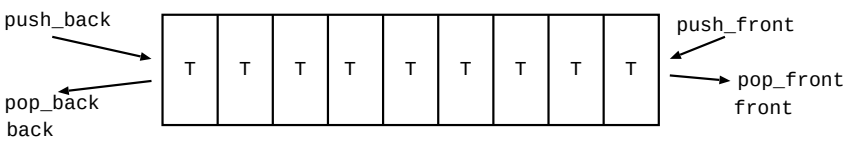
Bemerkung: Die STL-Version erhält man durch **#include** <queue>. Die Klassenschablone heißt dort `queue` und hat im Wesentlichen dieselbe Schnittstelle wie `Queue`.

Programm: Zur Abwechslung verwenden wir mal die STL-Version: **(UseQueueSTL.cc)**

```
#include<queue>
#include<iostream>

int main () {
  std::queue<int> q;
  for (int i=1; i<=5; i++)
    q.push(i);

  while (!q.empty()) {
    std::cout << q.front() << std::endl;
    q.pop();
  }
}
```

12.9 DeQueue

Eine DeQueue (*double-ended queue*) ist eine Struktur, die Einfügen und Entfernen an beiden Enden erlaubt:



Schnittstelle:

· Konstruktion einer leeren `DeQueue`
· Einfügen eines Elementes vom Typ `T` am Anfang oder Ende
· Entfernen des Elementes am Anfang oder am Ende
· Inspektion des Elementes am Anfang oder Ende
· Test ob `DeQueue` leer

Programm: **(DeQueue.hh)**

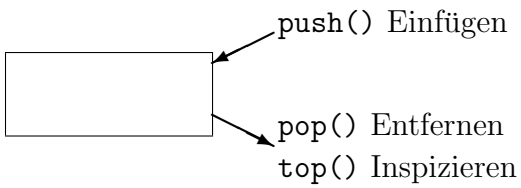
```
template<class T>
class DeQueue : private DLList<T> {
public :
  // Default-Konstruktoren + Zuweisung ok
  bool empty ();
  void push_front (T t);
  void push_back (T t);
  T pop_front ();
  T pop_back ();
  T front ();
  T back ();
} ;
```

Bemerkung: Die STL-Version erhält man auch hier mit **#include** <queue>. Die Klassenschablone heißt `deque`.

12.10 Prioritätswarteschlangen

Bezeichnung: Eine *Prioritätswarteschlange* ist eine Struktur, in die man Objekte des Grundtyps `T` einfüllen kann und von der jeweils das *kleinste* (MinPriorityQueue) bzw. das *größte* (MaxPriorityQueue) der eingegebenen Elemente als nächstes entfernt werden kann. Bei gleich großen Elementen verhält sie sich wie eine Queue.

Bemerkung: Auf dem Grundtyp `T` muß dazu die Relation < mittels dem **operator**< zur Verfügung stehen.



Schnittstelle:

- Konstruktion einer leeren MinPriorityQueue.

- Einfügen eines Elementes vom Typ `T` (`push`).
- Entfernen des kleinsten Elementes im Container (`pop`).
- Inspektion des kleinsten Elementes im Container (`top`).
- Test ob MinPriorityQueue leer (`empty`).

Programm: Hier die Klassendefinition **(MinPriorityQueue.hh)**:

```
template<class T>
class MinPriorityQueue : public DLList<T> {
private:
  typename DLList<T>::Iterator find_minimum();
public :
  // Default-Konstruktoren + Zuweisung OK
  bool empty ();
  void push (T t);  // Einfuegen
  void pop ();      // Entferne kleinstes
  T top ();         // Inspiziere kleinstes
} ;
```

Und die Implementation **(MinPriorityQueueImp.cc)**:

```
template<class T>
bool MinPriorityQueue<T>::empty () {
  return DLList<T>::empty();
}

template<class T>
void MinPriorityQueue<T>::push (T t) {
  this->insert(DLList<T>::begin(),t);
}

template<class T>
typename DLList<T>::Iterator
MinPriorityQueue<T>::find_minimum () {
  typename DLList<T>::Iterator min=DLList<T>::begin();
  for (typename DLList<T>::Iterator i=DLList<T>::begin();
       i!=DLList<T>::end(); i++)
    if (*i<*min) min=i;
  return min;
}

template<class T>
inline void MinPriorityQueue<T>::pop () {
  this->erase(find_minimum());
}

template<class T>
inline T MinPriorityQueue<T>::top () {
  return *find_minimum();
}
```

Bemerkung:

- Unsere Implementierung arbeitet mit einer einfach verketteten Liste. Das Einfügen hat Komplexität $O(1)$, das Entfernen/Inspizieren jedoch $O(n)$.
- Bessere Implementationen verwenden einen Heap, was zu einem Aufwand der Ordnung $O(\log n)$ führt.
- Analog ist die Implementation der `MaxPriorityQueue`.

Bemerkung: Die STL-Version erhält man auch durch **#include** <queue>. Die Klassenschablone heißt `priority_queue` und implementiert eine `MaxPriorityQueue`. Man kann allerdings den Vergleichsoperator auch als Template-Parameter übergeben (etwas lästig).

12.11 Set

Ein Set (Menge) ist ein Container mit folgenden Operationen:

- Konstruktion einer leeren Menge.
- Einfügen eines Elementes vom Typ `T`.
- Entfernen eines Elementes.
- Test auf Enthaltensein.
- Test ob Menge leer.

Programm: Klassendefinition **(Set.hh)**:

```
template<class T>
class Set : private DLList<T> {
public :
  // Default-Konstruktoren + Zuweisung OK

  typedef typename DLList<T>::Iterator Iterator;
  Iterator begin();
  Iterator end();

  bool empty ();
  bool member (T t);
  void insert (T t);
  void remove (T t);
  // union, intersection, ... ?
} ;
```

Implementation **(SetImp.cc)**:

```
template<class T>
typename Set<T>::Iterator Set<T>::begin() {
  return DLList<T>::begin();
}
```

```
6   template<class T>
    typename Set<T>::Iterator Set<T>::end() {
      return DLList<T>::end();
9   }

    template<class T>
12  bool Set<T>::empty () {
      return DLList<T>::empty();
    }
15
    template<class T>
    inline bool Set<T>::member (T t) {
18    return find(t)!=DLList<T>::end();
    }

21  template<class T>
    inline void Set<T>::insert (T t)
    {
24    if (!member(t))
        DLList<T>::insert(DLList<T>::begin(),t);
    }
27
    template<class T>
    inline void Set<T>::remove (T t)
30  {
      typename DLList<T>::Iterator i = find(t);
      if (i!=DLList<T>::end())
33      erase(i);
    }
```

Bemerkung:

- Die Implementierung hier basiert auf der doppelt verketteten Liste von oben (private Ableitung!).
- Einfügen, Suchen und Entfernen hat die Komplexität $O(n)$.
- Auf dem Typ `T` muss der Vergleichsoperator **operator**< definiert sein. (Set gehört zu den sog. sortierten, assoziativen Containern).

12.12 Map

Bezeichnung: Eine Map ist ein assoziatives Feld, das Objekten eines Typs `Key` Objekte eines Typs `T` zuordnet.

Beispiel: Telefonbuch:

Meier	→	504423
Schmidt	→	162300
Müller	→	712364
Huber	→	8265498

Diese Zuordnung könnte man realisieren mittels:

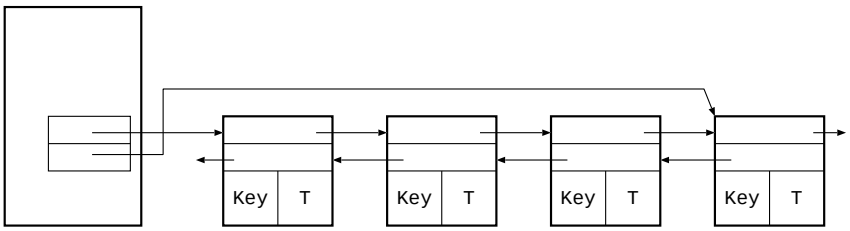
```
Map<string, int> telefonbuch;
telefonbuch["Meier"] = 504423;
...
```

Programm: **Definition der Klassenschablone (Map.hh)**

```
// Existiert schon als std::pair
// template<class Key, class T>
// struct pair {
//     Key first;
//     T second;
// };

template<class Key, class T>
class Map : private DLList<pair<Key,T> > {
public:

  T& operator[](const Key& k);

  typedef typename DLList<pair<Key,T> >::Iterator Iterator;
  Iterator begin() const;
  Iterator end() const;
  Iterator find(const Key& k);
};
```

Bemerkung:

- In dieser Implementation von Map werden je zwei Objekte der Typen `Key` (*Schlüssel*) und `T` (*Wert*) zu einem Paar vom Typ `pair<Key,T>` kombiniert und in eine doppelt verkettete Liste eingefügt:



- Ein Objekt vom Typ `Key` kann nur einmal in einer Map vorkommen. (Daher ist das Telefonbuch kein optimales Beispiel.)
- Wir haben einen Iterator zum Durchlaufen des Containers.
- Auf dem Schlüssel muss der Vergleichsoperator **operator**< definiert sein.
- `find(Key k)` liefert den Iterator für den Wert, ansonsten `end()`.

- Der Aufwand einer Suche ist wieder $O(n)$.

13 Verschiedenes

In diesem Abschnitt wollen wir noch einige Punkte erwähnen, die für das Programmieren wichtig sind.

13.1 Rechtliches

Lizenzen

Software ist normalerweise urheberrechtlich geschützt. Das geht so weit, dass es nicht erlaubt ist, Software ohne Erlaubnis des Besitzers in irgendeiner Weise zu nutzen. Diese Erlaubnis wird durch verschiedene Lizenzen erteilt, die dem Benutzer mehr oder weniger Einschränkungen auferlegen.

Wichtige Lizenzen

- Public Domain: der Code ist völlig frei, alles ist erlaubt. Vorsicht: nach deutschem Recht kann ein Programmierer seine Rechte in dieser Weise nicht aufgeben!
- BSD-Lizenz (nach dem Betriebssystem BSD Unix benannt): schließt Verantwortung des Urhebers für Fehler aus, erlaubt dem Benutzer alles außer Verändern der Lizenz selbst.
- GNU GPL (GNU General Public License): erlaubt dem Benutzer privat alles, bei Weitergabe an Dritte muss er aber auf deren Anfrage hin auch die Quelltexte unter der GPL nachliefern. Dasselbe muss er auch für angebundene Bibliotheken tun, die daher „kompatible“ Lizenzen haben müssen. Der Linux-Kernel steht unter der GPL.
- MPL (Mozilla Public License), QPL (Qt Public License): ähnlich GPL, aber dem Erstautor (Netscape, Trolltech, ...) werden besondere Rechte eingeräumt.
- Akademische Lizenzen: Gebrauch im akademischen Bereich erlaubt, sonst besondere Erlaubnis nötig.
- Kommerzielle Lizenzen: Oft erhebliche Einschränkungen, normalerweise ist kein Quellcode enthalten.

Wer hat die Rechte an Software?

Generell hat der Schöpfer eines Werks das Urheberrecht. Allerdings besteht oft ein Vertrag, der die *Nutzungsrechte* dem Arbeitgeber überträgt. Im akademischen Bereich ist das oft die entsprechende Hochschule.

Wenn der Arbeitgeber nicht an einer Nutzung der Software interessiert sein sollte (er muss aber gefragt werden!), so können die Rechte nach einer Wartezeit an den Angestellten rückübertragen werden.

Softwarepatente

Leider werden vor allem in den USA Patente auf softwaretechnische Ideen erteilt, die eigentlich nicht patentierbar sein sollten (z. B. Fortschrittsbalken, One-Click-Shopping), weil sie „offensichtlich“ sind (d. h. viele Programmierer hätten dieselbe Lösung für dasselbe Problem gefunden) oder aber erst nach einer wissenschaftlichen Veröffentlichung patentiert wurden (z. B. RSA-Kryptographie). Auch wenn solche Patente in Europa bisher noch nicht gültig sind, so kann es natürlich Schwierigkeiten bei in die USA exportierter Software geben.

13.2 Software-Technik (Software-Engineering)

Werkzeuge

- Editor (muss Sprachelemente kennen, kann in großen Programmsystemen Definitionen finden, Teile separat kompilieren und testen)
- Versionsverwaltung: Rückverfolgung von Änderungen, Vereinigung der Änderungen verschiedener Programmierer, Release-Management; Unix: CVS, Subversion, Git, ...
- Debugger: Beobachtung des laufenden Programms; Unix: gdb
- Profiling: Messen des Speicher- und Rechenzeitverbrauchs; Programme: valgrind (kommerziell: Purify), gprof;
- Testumgebung z. B. CppUnit.
- Entwicklungsumgebung (IDE): alles integriert

Beobachtungen aus der Praxis

- Große Programmpakete sind die Arbeit vieler Programmierer.
- Die Produktivität von Programmierern weist gigantische Unterschiede auf. Zwischen Durchschnitts- und Spitzenprogrammierer kann man mit etwa einer Größenordnung rechnen.
- Kleine Gruppen von Programmierern sind relativ gesehen am produktivsten. Insbesondere Zweiergruppen können sich gut ergänzen (pair programming).
- Es ist besonders effektiv, wenn ein Architekt ein Programmprojekt leitet. Dieser Architekt sollte
  - ein guter Programmierer sein,
  - sich mit vorhandenen Bibliotheken auskennen,
  - sich auch im Anwendungsbereich auskennen und
  - Autorität besitzen.
- Als sehr gut hat sich inkrementelle Entwicklung erwiesen: man fängt mit einem Prototyp an, der sukzessive verfeinert wird. Es wird darauf geachtet, dass möglichst immer eine lauffähige Version des Programms zur Verfügung steht.

- Ambitionierte Konzepte brauchen (zu viel) Zeit. Dies kann im Wettlauf mit anderen das Ende bedeuten.
- Ein Großteil der Kosten entsteht oft bei der Wartung der Software. Schlechte Arbeit am Anfang (z. B. verfehlter Entwurf) kann hier verhältnismäßig große Kosten verursachen.

Was kann schiefgehen?

- Manchmal ist die fachliche Qualifikation von Personen in Leitungsfunktionen (Architekt, Manager) schlecht, was zu teilweise schwerwiegenden Fehlentscheidungen führt.
- Insbesondere sind Manager (und Programmierer ebenso) viel zu optimistisch mit Abschätzungen der Schwierigkeit eines Programmprojekts. Ninety-ninety rule (Tom Cargill, Bell Labs):

  > The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.

- Einsparungen und Druck von oben können dazu führen, dass gerade die besten Programmierer gehen.
- Die Einstellung neuer Programmierer, um eine Deadline einzuhalten, kann die Arbeit noch weiter verzögern, weil diese eingearbeitet werden müssen.

Literatur: Frederick P. Brooks: *The Mythical Man-Month* (Klassiker)

Schlagworte

- Wasserfallmodell: Planung (*Systems Engineering*), Definition (*Analysis*), Entwurf (*Design*), Implementierung (*Coding*), Testen (*Testing*), Einsatz und Wartung (*Maintenance*)
- Agile Software-Entwicklung: XP (Extreme Programming, Pair programming), Scrum (Tägliche kurze Treffen, Sprints)
- Rational Unified Process: macht Wasserfallmodell iterativ, benutzt UML (Universal Modeling Language)
- XML: Datenaustauschformat, menschen- und maschinenlesbar, einfache Syntax der Form `<name>...</name>`. Wird leider beliebig kompliziert durch *Schemas* (DTD, XML Schema)
- Entwurfsmuster (Design Patterns): Die Idee ist es, wiederkehrende Muster in Programmen jenseits der Sprachelemente zu identifizieren und zu nutzen. Entwurfsmuster wurden bekannt durch das Buch: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995, ISBN 0-201-63361-2.

Bemerkung: Vorsicht vor Schlagworten, wenn sie als Allheilmittel verkauft werden. Viele Programmieraufgaben sind fundamental schwierig, und es wird auf absehbare Zeit dafür keine einfache Lösung geben geben.

13.3 Wie werde ich ein guter Programmierer?

Inhalt der Vorlesung

- Grundlagen des Programmierens in C/C++
- Kennenlernen verschiedener Programmiertechniken (funktional, imperativ, objektorientiert, generisch)
- Erkennen der Bedeutung effizienter Algorithmen

Bemerkung: Auch wenn C/C++ gerade für Anfänger nicht die einfachste Sprache ist, sind grundlegende Kenntnisse darin unbedingt notwendig. Das von der Syntax her ähnliche Java und C/C++ decken einen großen Teil des Software-Markts ab.

Nicht behandelt

Die Vorlesung hat aber viele interessante Themen nicht behandelt:

- Programmiertechniken: Logikprogrammierung, Programmierung von parallelen Prozessen, ...
- Programmierung von Web-Anwendungen, GUI-Programmierung, ...
- Programmverifikation und Qualitätssicherung
- Werkzeuge
- Testen (unit tests (Modultest), system testing, regression testing)
- Modellierung (Flussdiagramme, Struktogramme, Klassendiagramme, unified modeling language (UML))

Wie geht es nun weiter?

Lesen Sie einmal Peter Norvigs Artikel "Teach yourself programming in ten years!" (`http://www.norvig.com/21-days.html`)

Er empfiehlt unter anderem:

- Finden Sie einen Weg, um Spaß am Programmieren zu finden! (Wenn nicht mit C++, dann mit einfacheren Sprachen wie Python oder Scheme.)
- Sprechen Sie mit anderen Programmierern, und lesen Sie anderer Leute Programme.
- Programmieren Sie selber!
- Wenn es Ihnen Spaß macht, besuchen Sie weitere Informatik-Vorlesungen.

- Arbeiten Sie an Projekten mit. Seien Sie einmal schlechtester, einmal bester Programmierer bei einem Projekt.
- Lernen Sie mehrere unterschiedliche Programmiersprachen.
- Lernen Sie Ihren Computer näher kennen (wie lang braucht eine Instruktion, ein Speicherzugriff, etc.)

Ich möchte noch folgende recht bodenständige Tips hinzufügen:

- Englisch ist heutzutage die Sprache sowohl der Naturwissenschaften als auch der Programmierung. Üben Sie Ihr Englisch, wo immer Sie können!
- Wenn Sie noch nicht mit zehn Fingern tippen können, versuchen Sie es zu lernen, sobald Sie einmal einen langen Text zu schreiben haben (z. B. Bachelor- oder Masterarbeit).

Index

Literatur