

# Einführung in die Anwendungsorientierte Informatik (Köthe)

Robin Heinemann

November 25, 2016

## Contents

<b>1</b>	<b>Klausur 10.02.2016</b>	<b>2</b>
<b>2</b>	<b>Was ist Informatik?</b>	<b>2</b>
2.1	Teilgebiete . . . . .	2
2.1.1	theoretische Informatik ( <b>ITH</b> ) . . . . .	2
2.1.2	technische Informatik ( <b>ITE</b> ) . . . . .	3
2.1.3	praktische Informatik . . . . .	3
2.1.4	angewante Informatik . . . . .	3
<b>3</b>	<b>Wie unterscheidet sich Informatik von anderen Disziplinen?</b>	<b>4</b>
3.1	Mathematik . . . . .	4
<b>4</b>	<b>Informatik</b>	<b>4</b>
4.1	Algorithmus . . . . .	4
4.2	Daten . . . . .	4
4.2.1	Beispiele für Symbole . . . . .	5
4.3	Einfachster Computer . . . . .	5
4.3.1	<b>TODO</b> Graphische Darstellung . . . . .	5
4.3.2	<b>TODO</b> Darstellung durch Übergangstabellen . . . . .	6
4.3.3	Beispiel 2: . . . . .	6
<b>5</b>	<b>Substitutionsmodell (funktionale Programmierung)</b>	<b>7</b>
5.1	Substitutionsmodell . . . . .	8
5.2	Bäume . . . . .	9
5.2.1	Beispiel . . . . .	9
5.3	Rekursion . . . . .	10
5.4	Prefixnotation aus dem Baum rekonstruieren . . . . .	10
5.5	Prefixnotation aus dem Baum rekonstruieren . . . . .	10
5.6	Berechnen des Werts mit Substitutionsmethode . . . . .	11

<b>6</b>	<b>Maschinensprachen</b>	<b>11</b>
6.1	Umwandlung in Maschinensprache . . . . .	11
<b>7</b>	<b>Funktionale Programmierung</b>	<b>12</b>
7.1	Beispiel . . . . .	12
7.2	Vorteile von Zwischenergebnissen . . . . .	12
7.3	Funktionale Programmierung in c++ . . . . .	13
<b>8</b>	<b>Prozedurale Programmierung</b>	<b>15</b>
8.1	Von der Funktionalen zur prozeduralen Programmierung . . . . .	15
8.2	Kennzeichen . . . . .	16
8.2.1	Prozeduren . . . . .	16
8.2.2	Steuerung des Programmablaufs . . . . .	16
8.2.3	Veränderung von Speicherzellen . . . . .	17
8.2.4	Schleifen . . . . .	18
8.2.5	prozedurale Wurzelberechnung . . . . .	18
8.2.6	for-Schleife . . . . .	19
<b>9</b>	<b>Datentypen</b>	<b>20</b>
9.1	Basistypen . . . . .	20
9.2	zusammengesetzte Typen . . . . .	21
9.3	Zeichenketten-Strings: . . . . .	21
<b>10</b>	<b>Umgebungsmodell</b>	<b>23</b>
<b>11</b>	<b>Referenzen</b>	<b>25</b>
<b>12</b>	<b>Container-Datentypen</b>	<b>26</b>
12.1	std::vector . . . . .	28

## 1 Klausur 10.02.2016

## 2 Was ist Informatik?

”Kunst” Aufgaben mit Computerprogrammen zu lösen.

### 2.1 Teilgebiete

#### 2.1.1 theoretische Informatik (ITH)

- Berechenbarkeit: Welche Probleme kann man mit Informatik lösen und welche prinzipiell nicht?
- Komplexität: Welche Probleme kann man effizient lösen?

- Korrektheit: Wie beweist man, dass das Ergebnis richtig ist?  
Echtzeit: Dass das richtige Ergebnis rechtzeitig vorliegt.
- verteilte Systeme: Wie sichert man, dass verteilte Systeme korrekt kommunizieren?

### 2.1.2 technische Informatik (ITE)

- Auf welcher Hardware kann man Programme ausführen, wie baut man dies Hardware?
- CPU, GPU, RAM, HD, Display, Printer, Networks

### 2.1.3 praktische Informatik

- Wie entwickelt man Software?
- Programmiersprachen und Compiler: Wie kommuniziert der Programmierer mit der Hardware? **IPI, IPK**
- Algorithmen und Datenstrukturen: Wie baut man komplexe Programme aus einfachen Grundbausteinen? **IAL**
- Softwaretechnik: Wie organisiert man sehr große Projekte? **ISW**
- Kernanwendung der Informatik: Betriebssysteme, Netzwerke, Parallelisierung **IBN**
  - Datenbanksysteme **IDB1**
  - Graphik, Graphische Benutzerschnittstellen **ICG1**
  - Bild- und Datenanalyse
  - maschinelles Lernen
  - künstliche Intelligenz

### 2.1.4 angewante Informatik

- Wie löst man Probleme aus einem anderem Gebiet mit Programmen?
- Informationstechnik
  - Buchhandlung, e-commerce, Logistik
- Web programming
- scientific computing für Physik, Biologie
- Medizininformatik
  - bildgebende Verfahren
  - digitale Patientenakte

- computer linguistik
  - Sprachverstehen, automatische Übersetzung
- Unterhaltung: Spiele, special effect im Film

### 3 Wie unterscheidet sich Informatik von anderen Disziplinen?

#### 3.1 Mathematik

Am Beispiel der Definition  $a \leq b : \exists c \geq 0 : a + c = b$

Informatik:

Lösungsverfahren:  $a - b \leq 0$ , das kann man leicht ausrechnen, wenn man subtrahieren und mit 0 vergleichen kann.

Quadratwurzel:  $y = \sqrt{x} \Leftrightarrow y \geq 0 \wedge y^2 = x (\Rightarrow x > 0)$

Informatik: Algorithmus aus der Antike:  $y = \frac{x}{y}$  iteratives Verfahren:

Initial Guess  $y^{(0)} = 1$  schrittweise Verbesserung  $y^{(t+1)} = \frac{y^{(t)} + \frac{x}{y^{(t)}}}{2}$

### 4 Informatik

Lösungswege, genauer Algorithmen

#### 4.1 Algorithmus

**schematische** Vorgehensweise mit der jedes Problem einer bestimmten **Klasse** mit **endliche** vielen **elementaren** Schritten / Operationen gelöst werden kann

- schematisch: man kann den Algorithmus ausführen, ohne ihn zu verstehen ( $\Rightarrow$  Computer)
- alle Probleme einer Klasse: zum Beispiel: die Wurzel aus jeder beliebigen nicht-negativen Zahl, und nicht nur  $\sqrt{11}$
- endliche viele Schritte: man kommt nach endlicher Zeit zur Lösung
- elementare Schritte / Operationen: führen die Lösung auf Operationen oder Teilprobleme zurück, die wir schon gelöst haben

#### 4.2 Daten

Daten sind Symbole,

- die Entitäten und Eigenschaften der realen Welt im Computer representieren.
- die interne Zwischenergebnisse eines Algorithmus aufbewahren

$\Rightarrow$  Algorithmen transformieren nach bestimmten Regeln die Eingangsdaten (gegebene Symbole) in Ausgangsdaten (Symbole für das Ergebnis). Die Bedeutung / Interpretation der Symbole ist dem Algorithmus egal  $\hat{=}$  "schematisch"

#### 4.2.1 Beispiele für Symbole

- Zahlen
- Buchstaben
- Icons
- Verkehrszeichen

aber: heutige Computer verstehen nur Binärzahlen  $\Rightarrow$  alles andere muss man übersetzen  
Eingangsdaten: "Ereignisse":

- Symbol von Festplatte lesen oder per Netzwerk empfangen
- Benutzerinteraktion (Taste, Maus, ...)
- Sensor übermittelt Meßergebnis, Stoppuhr läuft ab

Ausgangsdaten: "Aktionen":

- Symbole auf Festplatte schreiben, per Netzwerk senden
- Benutzeranzeige (Display, Drucker, Ton)
- Stoppuhr starten
- Roboteraktion ausführen (zum Beispiel Bremsassistent)

Interne Daten:

- Symbole im Hauptspeicher oder auf Festplatte
- Stoppuhr starten / Timeout

### 4.3 Einfachster Computer

endliche Automaten (endliche Zustandsautomaten)

- befinden sich zu jedem Zeitpunkt in einem bestimmten Zustand aus einer vordefinierten endlichen Zustandsmenge
- äußere Ereignisse können Zustandsänderungen bewirken und Aktionen auslösen

#### 4.3.1 TODO Graphische Darstellung

graphische Darstellung: Zustände = Kreise, Zustandsübergänge: Pfeile

### 4.3.2 TODO Darstellung durch Übergangstabellen

Zeilen: Zustände, Spalten: Ereignisse, Felder: Aktion und Folgezustände

Zustände \ Ereignisse	Knopf drücken	Timeout
aus	$\Rightarrow \{\text{halb}\}$	
{4 LEDs an}	%	$(\Rightarrow \{\text{aus}\}, \{\text{nichts}\})$
halb	$(\Rightarrow \{\text{voll}\}, \{8 \text{ LEDs an}\})$	%
voll	$(\Rightarrow \{\text{blinken an}\}, \{\text{Timer starten}\})$	%
blinken an	$(\Rightarrow \{\text{aus}\}, \{\text{Alle LEDs aus, Timer stoppen}\})$	$(\Rightarrow \{\text{blinken aus}\}, \{\text{alle LEDs aus, Timer stoppen}\})$
blinken aus	$(\Rightarrow \{\text{aus}\}, \{\text{Alle LEDs aus, Timer stoppen}\})$	$(\Rightarrow \{\text{blinken an}\}, \{\text{alle LEDs an, Timer starten}\})$

Variante: Timer läuft immer (Signal alle 0.3s)  $\Rightarrow$  Timeout ignorieren im Zustand "aus", "halb", "voll"

### 4.3.3 Beispiel 2:

$$\begin{array}{rcl}
 1\ 0\ 1\ 1\ 0\ 1\ 0 & = 2 + 8 + 16 + 74 = 90_{\text{dez}} & (1) \\
 + 0\ 1\ 1\ 1\ 0\ 0\ 1 & = 1 + 8 + 16 + 32 = 57_{\text{dez}} & (2) \\
 \hline
 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 & = 1 + 2 + 16 + 128 = 147_{\text{dez}} \checkmark & (3)
 \end{array}$$

### Implementation mit Endlichen Automaten Prinzipien:

- wir lesen die Eingangsdaten von rechts nach links
- Beide Zahlen gleich lang (sonst mit 0en auffüllen)
- Ergebnis wird von rechts nach links ausgegeben

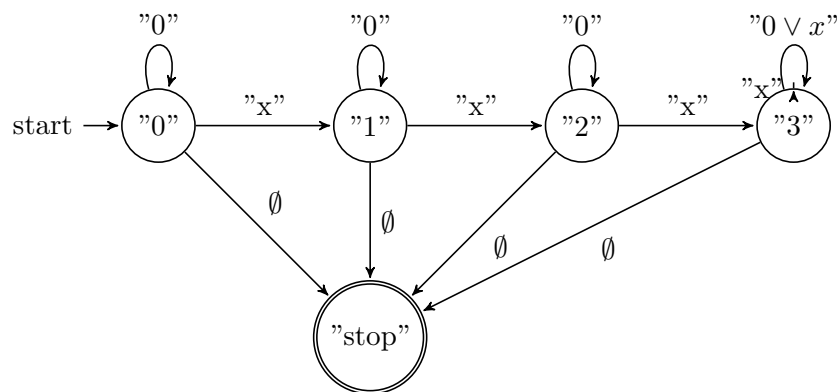
### TODO Skizze der Automaten

Zustand	Ereignis	Ausgeben
start	(0,1)	"1"
start	(1,0)	"1"
start	(0,0)	"0"
start	(1,1)	"0"
carry = 1	(1,1)	"1"
carry = 1	(0,1)	"0"
carry = 1	(1,0)	"0"
carry = 1	$\emptyset$	"1"

**Wichtig:** In jedem Zustand muss für **alle möglichen** Ereignisse eine Aktion und Folgezustand definiert werden. Vergisst man ein Ereignis zeigt der Automat undefiniertes Verhalten, also einen "Bug". Falls keine sinnvolle Reaktion möglich ist: neuer Zustand: "Fehler"  $\Rightarrow$  Übergang nach "Fehler", Aktion: Ausgeben einer Fehlermeldung

**TODO Skizze Fehlermeldung** Ein endlicher Automat hat nur ein Speicherelement, das den aktuellen Zustand angibt. Folge:

- Automat kann sich nicht merken, wie er in den aktuellen Zustand gekommen ist ("kein Gedächtnis")
- Automat kann nicht beliebig weit zählen, sondern nur bis zu einer vorgegebenen Grenze



Insgesamt: Man kann mit endlichen Automaten nur relativ einfache Algorithmen implementieren. (nur reguläre Sprachen) Spondiert man zusätzlichen Speicher, geht mehr:

- Automat mit Stack-Speicher (Stapel oder Keller)  $\Rightarrow$  Kellerautomat (Kontextfreie Sprachen)
- Automat mit zwei Stacks oder äquivalent Turing-Maschine kann alles auführen, was man intuitiv für berechenbar hält

Markov Modelle: endliche Automaten mit probabilistischen Übergängen. Bisher: Algorithmen für einen bestimmten Zweck (Problemklasse) Frage: Gibt es einen universellen Algorithmus für alle berechenbare Probleme? Betrachte formale Algorithmusbeschreibung als Teil der Eingabe des universellen Algorithmus.

## 5 Substitutionsmodell (funktionale Programmierung)

- einfaches Modell für arithmetische Berechnung "Taschenrechner"
- Eingaben und Ausgaben sind Zahlen (ganze oder reelle Zahlen). Zahlenkonstanten heißen "Literele"
- elementare Funktionen: haben eine oder mehrere Zahlen als Argumente (Parameter) und liefern eine Zahl als Ergebnis (wie Mathematik):
  - $\text{add}(1,2) \rightarrow 3$ ,  $\text{mul}(2,3) \rightarrow 6$ , analog  $\text{sub}()$ ,  $\text{div}()$ ,  $\text{mod}()$

- Funktionsaufrufe können verschachtelt werden, das heißt Argumente kann Ergebnis einer anderen Funktion sein
  - $\text{mul}(\text{add}(1,2), \text{sub}(5,3)) \rightarrow 6$

## 5.1 Substitutionsmodell

Man kann einen Funktionsaufruf, dessen Argument bekannt ist (das heißt Zahlen sind) durch den Wert des Ergebnisses ersetzen ("substituieren"). Geschachtelte Ausdrücke lassen sich so von innen nach außen auswerten.

$$\text{mul}(\text{add}(1, 2), \text{sub}(5, 3))$$

$$\text{mul}(3, \text{sub}(5, 3))$$

$$\text{mul}(3, 2)$$

$$6$$

- Die arithmetischen Operationen  $\text{add}()$ ,  $\text{sub}()$ ,  $\text{mul}()$ ,  $\text{div}()$ ,  $\text{mod}()$  werden normalerweise von der Hardware implementiert.
- Die meisten Programmiersprachen bieten außerdem algebraische Funktionen wie:  $\text{sqrt}()$ ,  $\text{sin}()$ ,  $\text{cos}()$ ,  $\text{log}()$ 
  - sind meist nicht in Hardware, aber vorgefertigte Algorithmen, werden mit Programmiersprachen geliefert, "Standardbibliothek"
- in C++: mathematisches Modul der Standardbibliothek: "cmath"
- Für Arithmetik gebräuchlicher ist "Infix-Notation" mit Operator-Symbolen "+", "-", "\*", "/", "%"
- $\text{mul}(\text{add}(1,2), \text{sub}(5,3)) \Leftrightarrow ((1+2)*(5-3))$ 
  - oft besser, unter anderem weil man Klammern weglassen darf
    1. "Punkt vor Strichrechnung"  $3+4*5 \Leftrightarrow 3+(4*5)$ ,  $\text{mul}$ ,  $\text{div}$ ,  $\text{mod}$  binden stärker als  $\text{add}$ ,  $\text{sub}$
    2. Operatoren gleicher Präzedenz werden von links nach rechts ausgeführt (links-assoziativ)
 
$$1+2+3-4+5 \Leftrightarrow (((1+2)+3)-4)+5$$
    3. äußere Klammer kann man weglassen  $(1+2) \Leftrightarrow 1+2$
- Computer wandeln Infix zuerst in Prefix Notation um
  1. weggelassene Klammern wieder einfügen



2. Operatorensymbol durch Funktionsnamen ersetzen und an Prefix-Position verschieben

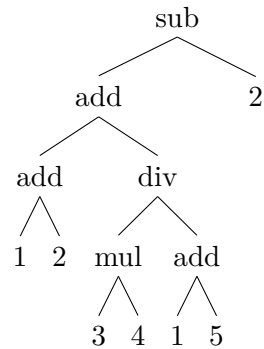
$$\begin{aligned}
 &1 + 2 + 3 * 4 / (1 + 5) - 2 \\
 &(((1 + 2) + ((3 * 4) / (1 + 5))) - 2) \\
 &sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2) \\
 &sub(add(3, div(12, 6)), 2) \\
 &sub(add(3, 2), 2) \\
 &sub(5, 2) \\
 &2
 \end{aligned}$$

## 5.2 Bäume

- bestehen aus Knoten und Kanten (Kreise und Pfeile)
- Kanten verbinden Knoten mit ihren Kind-knoten
- jeder Knoten (außer der Wurzel) hat genau ein Elternteil ("parent node")
- Knoten ohne Kinder heißen Blätter ("leaves / leaf node")
- Teilbaum
  - wähle beliebigen Knoten
  - entferne temporär dessen Elternkante, dadurch wird der Knoten temporär zu einer Wurzel, dieser Knoten mit allen Nachkommen bildet wieder einen Baum (Teilbaum des Originalbaumes)
- trivialer Teilbaum hat nur einen Knoten
- Tiefe: Abstand eines Knotens von der Wurzel (Anzahl der Kanten zwischen Knoten und Wurzel)
  - Tiefe des Baums: maximale Tiefe eines Knoten

### 5.2.1 Beispiel

$$\begin{aligned}
 &1 + 2 + 3 * 4 / (1 + 5) - 2 \\
 &sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)
 \end{aligned}$$



### 5.3 Rekursion

Rekursiv  $\hat{=}$  Algorithmus für Teilproblem von vorn.

### 5.4 Prefixnotation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist: Drucke die Zahl
2. sonst:
  - Drucke Funktionsnamen
  - Drucke "("
  - Wiederhole den Algorithmus ab 1 für das linke Kind (Teilbaum mit Wurzel = linkes Kind)
  - Drucke ","
  - Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind)
  - Drucke ")"

$\Rightarrow$

$sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$

### 5.5 Prefixnotation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist: Drucke die Zahl
2. sonst:
  - Drucke Funktionsnamen
  - Drucke "("
  - Wiederhole den Algorithmus ab 1 für das linke Kind (Teilbaum mit Wurzel = linkes Kind)
  - Drucke Operatorsymbol

- Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind)
- Drucke ")"

⇒

*sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)*

⇒ **inorder**

## 5.6 Berechnen des Werts mit Substitutionsmethode

1. Wenn Wurzel dein Blatt gib Zahl zurück
2. sonst:
  - Wiederhole den Algorithmus ab 1 für das linke Kind (Teilbaum mit Wurzel = rechtes Kind), speichere Ergebnis als "lhs"
  - Wiederhole den Algorithmus ab 1 für das rechte Kind (Teilbaum mit Wurzel = rechtes Kind), speichere Ergebnis als "rhs"
  - berechne funktionsname(lhs,rhs) und gebe das Ergebnis zurück

⇒ **postorder**

## 6 Maschinensprachen

- optimiert für die Hardware
- Gegensatz: höhere Programmiersprachen (c++)
  - optimiert für Programmierer
- Compiler oder Interpreter übersetzen Hoch- in Maschinensprache

### 6.1 Umwandlung in Maschinensprache

1. Eingaben und (Zwischen)-Ergebnisse werden in Speicherzellen abgespeichert ⇒ jeder Knoten im Baum bekommt eine Speicherzelle
2. Speicherzellen für Eingaben initialisieren
  - Notation: SpZ ← Wert
3. Rechenoperationen in Reihenfolge des Substitutionsmodell ausführen und in der jeweiligen Speicherzelle speichern
  - Notation: SpZ-Ergebniss ← fname SpZArg1 SpZArg2
4. alles in Zahlencode umwandeln
  - Funktionsnamen:

Opcode	Wert
init	1
add	2
sub	3
mul	4
div	5

## 7 Funktionale Programmierung

- bei Maschinensprache werden Zwischenergebnisse in Speicherzellen abgelegt
- das ist auch in der funktionalen Programmierung eine gute Idee
- Speicherzellen werden durch Namen (vom Programmierer vergeben) unterschieden

### 7.1 Beispiel

Lösen einer quadratischen Gleichung:

$$ax^2 + bx + c = 0$$

$$x^2 - 2px + q = 0, p = -\frac{b}{2a}, q = \frac{c}{a}$$

$$x_1 = p + \sqrt{p^2 - q}, x_2 = p - \sqrt{p^2 - q}$$

ohne Zwischenergebnisse:

$$x_1 \leftarrow \text{add}(\text{div}(\text{div}(b, a), -2), \text{sqrt}(\text{sub}(\text{mul}(\text{div}(b, a), -2), \text{div}(\text{div}(b, a) - 1)), \text{div}(c, a)))$$

mit Zwischenergebniss und Infix Notation

$$p \leftarrow b/c / -2 \text{ oder } p \leftarrow -0.5 * b/a$$

$$a \leftarrow c/a$$

$$d \leftarrow \text{sqrt}(p * p - q)$$

$$x_1 \leftarrow p + d$$

$$x_2 \leftarrow p - d$$

### 7.2 Vorteile von Zwischenergebnissen

1. lesbarer
2. redundante Berechnung vermieden. Beachte: In der funktionalen Programmierung können die Speicherzellen nach der Initialisierung nicht mehr verändert werden
3. Speicherzellen und Namen sind nützlich um Argumente an Funktionen zu übergeben  
⇒ Definition eigener Funktionen

```
function sq(x) {
    return x * x
}
```

$\Rightarrow d \leftarrow \text{sqrt}(sq(p) - q)$  Speicherzelle mit Namen "x" für das Argument von *sq*

### 7.3 Funktionale Programmierung in c++

- in c++ hat jede Speicherzelle einen Typ (legt Größe und Bedeutung der Speicherzelle fest)

- wichtige Typen

int	ganze Zahlen
double	reelle Zahlen
std::string	Text

int: 12, -3

double:  $-1.02, 1.2e - 4 = 1.2 * 10^{-4}$

std::string: "text"

- Initialisierung wird geschrieben als "typename spzname = Wert;"

```
double a = ...;
double b = ...;
double c = ...;
double p = -0.5 b / a;
double q = c / a;
double d = std::sqrt(p*p - q);
double x1 = p + d;
double x2 = p - d;
std::cout << "x1: " << x1 << ", x2: " << x2 << std::endl;
```

- eigene Funktionen in C++

```
// Kommentar (auch /* */)
type_ergebnis fname(type_arg1 name1, ...) {
    // Signatur / Funktionskopf / Deklaration
    return ergebnis;
    /* Funktionskörper / Definition / Implementation */
}
```

- ganze Zahl quadrieren:

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

- reelle Zahl quadrieren:

```
double sq(double x) {
    return x*x;
}
```

- beide Varianten dürfen in c++ gleichzeitig definiert sein  $\Rightarrow$  "function overloading"  $\Rightarrow$  c++ wählt automatisch die richtig Variable anhand des Argumenttypes ("overload resolution")

```
int x = 2;
double y = 1.1
int x2 = sq(x) // int Variante
double y2 = sq(y) // double Variante
```

- jedes c++-Programm muss genau eine Funktion names "main" haben. Dort beginnt die Programmausführung.

```
int main() {
    Code;
    return 0;
}
```

- return aus der "main" Funktion ist optional
- Regel von c++ für erlaubte Name
  - erstes Zeichen: Klein- oder Großbuchstaben des englischen Alphabets, oder "\_"
  - optional: weitere Zeichen oder, "\_" oder Ziffer 0-9
- vordefinierte Funktionen:
  - eingebaute  $\hat{=}$  immer vorhanden
    - Infix-Operatoren +, -, \*, /, %
    - Prefix-Operatoren *operator*+, *operator*-, ...
  - Funktion der Standardbibliothek  $\hat{=}$  müssen "angefordert" werden
    - Namen beginnen mit "std::", "std::sin,..."
    - sind in Module geordnet, zum Beispiel
      - cmath  $\Rightarrow$  algebraische Funktion
      - complex  $\Rightarrow$  komplexe Zahlen
      - string  $\Rightarrow$  Zeichenkettenverarbeitung
  - um ein Modul zu benutzen muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren (Header includieren)  $\rightarrow$  include <name>

```
#include <iostream>
#include <string>
int main() {
```

```

        std::cout << "Hello, world!" << std::endl;
        std::string out = "mein erstes Programm\n";
        std::cout << out;
        return 0;
    }

```

- overloading der arithmetischen Operationene
  - overloading genau wie bei *sq*
    - $3 * 4 \Rightarrow$  int Variante
    - $3.0 * 4.0 \Rightarrow$  double Variante
    - $3 * 4.0 \Rightarrow$  automatische Umwandlung in höheren Typ, hier "double"  $\Rightarrow$  wird als  $3.0 * 4.0$  ausgeführt
- $\Rightarrow$  Devision unterscheidet sich
  - Integer-Division:  $12 / 5 = 2$  (wird abgerundet)
  - Double-Division:  $12.0 / 5.0 = 2.4$
  - $-12 / 5 = 2$  ( $\Rightarrow$  truncated Division)
  - $12.0 / 5.0 = 2.4$
  - Gegensatz (zum Beispiel in Python)
    - floor division  $\Rightarrow$  wird immer abgerundet  $\Rightarrow -12 / 4 = -2$

## 8 Prozedurale Programmierung

### 8.1 Von der Funktionalen zur prozeduralen Programmierung

- Eigenschaften der Funktionalen Programmierung:
  - alle Berechnungen durch Funktionsaufruf, Ergebnis ist Rückgabe
  - Ergebnis hängt nur von den Werten der Funktionsargumente ab, nicht von externen Faktoren "referentielle Integrität"
  - Speicherzellen für Zwischenergebnisse und Argumente können nach Initialisierung nicht geändert werden "write once"
  - Möglichkeit rekursiver Funktionsaufrufe (jeder Aufruf bekommt eigene Speicherzellen)
    - Vorteile
      - natürliche Ausdrucksweise für arithmetische und algebraische Funktionaliät ("Taschenrechner")
      - einfache Auswertung durch Substitutionsmodell  $\rightarrow$  Auswertungsreihenfolge nach Post-Order
      - mathematisch gut formalisierbar  $\Rightarrow$  Korrektheitsbeweise, besonders bei Parallelverarbeitung

- Rekursion ist mächtig und natürliche für bestimmte Probleme (Fakultät, Baumtraversierung)
- Nachteile
  - viele Probleme lassen sich anders natürlicher ausdrücken (z.B. Rekursion vs. Iteration)
  - setzt unendlich viel Speicher voraus ( $\Rightarrow$  Memory management notwendig  $\Rightarrow$  später)
  - Entitäten, die sich zeitlich verändern schwer zu modellieren
- Korollar: kann keine externen Ressourcen (z.B. Konsole, Drucker, ..., Bildschirm) ansprechen "keine Seiteneffekte"
- $\Rightarrow$  Multi-Paradigmen-Sprachen, zum Beispiel Kombination von Funktionaler Programmierung und prozeduraler Programmierung

## 8.2 Kennzeichen

### 8.2.1 Prozeduren

- Prozeduren: Funktionen, die nichts zurückgeben, haben nur Seiteneffekte

- Beispiel

```
std::cout << "Hello\n"; // Infix
operator<<(std::cout, "Hello\n"); // Prefix
```

- Prozeduren in c++

1. Funktion die "void" zurückgibt (Pseudotyp für "nichts")

```
void foo(int x) {
    return;
}
```

2. Returnwert ignorieren

### 8.2.2 Steuerung des Programmablaufs

- Anweisungen zur Steuerung des Programmablaufs

```
if(), else, while(), for()
```

- Funktional

```
int abs(int x) {
    return (x >= 0) ? x : -x;
}
```

- Prozedural



```

int abs(int x) {
    if(x >= 0) {
        return x;
    } else {
        return -x;
    }

    // oder
    if(x >= 0) return x;
    return -x;
}

```

### 8.2.3 Veränderung von Speicherzellen

- Zuweisung: Speicherzellen können nachträglich verändert werden ("read-write")

- prozedural:

```

int foo(int x) {      // x = 3
    int y = 2;
    int z1 = x * y;   // z1 = 6
    y = 5;
    int z2 = z * y;   // z2 = 15
    return z1 + z2;   // 21
}

```

- funktional:

```

int foo(int x) {      // x = 3
    int y1 = 2;
    int z1 = x * y;   // z1 = 6
    int y2 = 5;
    int z2 = z * y;   // z2 = 15
    return z1 + z2;   // 21
}

```

- Syntax

```

name = neuer_wert;      // Zuweisung
typ name = neuer_wert;  // Initialisierung
typ const name = neuer_wert; // write once

```

- ⇒ Folgen: mächtiger, aber ermöglicht völlig neue Bugs ⇒ erhöhte Aufmerksamkeit beim Programmieren
  - die Reihenfolge der Ausführung ist viel kritischer als beim Substitutionsmodell
  - Programmierer muss immer ein mentales Bild des aktuellen Systemzustands haben

### 8.2.4 Schleifen

Der gleiche Code soll oft wiederholt werden

```
while(bedingung) {  
    // code, wird ausgeführt solange Bedingung "true"  
}  
  
int counter = 0;  
while(counter < 3) {  
    std::cout << counter << std::endl;  
    counter++; // Kurzform für counter = counter + 1  
}
```

counter	Bedingung	Ausgabe
0	true	0
1	true	1
2	true	2
3	false	$\emptyset$

- in c++ beginnt Zählung meist mit 0 ("zero based")
- vergisst man Inkrementierung  $\Rightarrow$  Bedingung immer "true"  $\Rightarrow$  Endlosschleife  $\Rightarrow$  Bug
- drei äquivalente Schreibweisen für Inkrementierung:
  - counter = counter + 1; // assignment  $\hat{=}$  Zuweisung
  - counter += 1; // add-assignment  $\hat{=}$  Abkürzung
  - ++counter; // pre-increment

### 8.2.5 prozedurale Wurzelberechnung

#### Ziel

```
double sqrt(double y);
```

**Methode** iterative Verbesserung mittel Newtonverfahren initial<sub>guess</sub>  $x^{(0)}$  ("geraten"), t = 0

while not<sub>goodenough</sub>( $x^{(t)}$ ):

update  $x^{(t+1)}$  from  $x^{(t)}$  (zum Beispiel  $x^{(t+1)} = x^{(t)} + \Delta^{(t)}$  additives update,  $x^{(t+1)} = x^{(t)}\Delta^{(t)}$  multiplikatives update)

t = t + 1

**Newtonverfahren** Finde Nullstellen einer gegebenen Funktion  $f(x)$ , das heißt suche  $x^*$  sodass  $f(x^*) = 0$  oder  $|f(x^*)| < \varepsilon$  Taylorreihe von  $f(x)$ :  $f(x + \Delta) \approx f(x) + f'(x)\Delta +$  setze  $x^* = x + \Delta$

$$0 \stackrel{!}{=} f(x^*) \approx f(x) + f'(x)\Delta = 0 \Rightarrow \Delta = -\frac{f(x)}{f'(x)}$$

Iterationsvorschrift:

$$x^{(t+1)} = x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})}$$

Anwendung auf Wurzel: setze  $f(x) = x^2 - y \Rightarrow$  mit  $f(x^*) = 0$  gilt

$$(x^*)^2 - y = 0 \quad (x^*)^2 = y \quad x^* = \sqrt{y} \quad f'(x) = 2x$$

Iterationsvorschrift:

$$x^{(t+1)} = x^{(t)} - \frac{(x^{(t)})^2 - y}{2x^{(t)}} = \frac{x^{(t)^2} + y}{2x^{(t)}}$$

```
double sqrt(double y) {
    if(y < 0.0) {
        std::cout << "Wurzel aus negativer Zahl\n";
        return -1.0;
    }
    if(y == 0.0) return 0.0;

    double x = y; // initial guess
    double epsilon = 1e-15 * y;

    while(abs(x * x - y) > epsilon) {
        x = 0.5*(x + y / x);
    }
}
```

### 8.2.6 for-Schleife

```
int c = 0;
while(c < 3) {
    // unser code
    c++; // vergisst man leicht
}
```

Bei der while Schleife kann man leicht vergessen  $c$  zu inkrementieren, die for Schleife ist idiotensicher

Äquivalent zu der while Schleife oben ist:

```
for(int c = 0; c < 3; c++) {
    // unser code
}
```

Allgemeine Form:

```
for(init; Bedingung; Inkrement) {
    // unser code
}
```

- Befehle, um Schleifen vorzeitig abubrechen
  - continue: Bricht aktuelle Iteration ab und springt zum Schleifenkopf
  - break: bbricht ganz Schleife ab und springt hinter das Schleifenende
  - return: beendet Funktion und auch die Schleife

Beispiel: nur gerade Zahlen ausgeben

```
for(int i = 0; i < 10; i++) if(c % 2 == 0) std::cout << c << std::endl;
```

Variante mit continue:

```
for(int i = 0; i < 10; i++) {
    if(c % 2 != 0) continue;
    std::cout << c << std::endl;
}
```

```
for(int i = 0; i < 10; i += 2) {
    std::cout << c << std::endl;
}
```

```
double sqrt(double y) {
    while(true) {
        x = (x + y / 2) / 2.0;
        if(abs(x * x - y) < epsilon) {
            return x;
        }
    }
}
```

## 9 Datentypen

### 9.1 Basistypen

Bestandteil der Sprachsyntax und normalerweise direkt von der Hardware unterstützt (CPU)

- int, double, bool (⇒ später mehr)

## 9.2 zusammengesetzte Typen

mit Hilfe von "struct" oder "class" aus einfachen Typen zusammengesetzt

- wie das geht  $\Rightarrow$  später
- Standardtypen: in der C++ Standardbibliothek definiert, aktiviere durch `#include <module_name>`
  - `std::string`, `std::complex`, etc.
- externe Typen: aus anderer Bibliothek, die man zuvor herunterladen und installieren muss
- eigene Typen: vom Programmierer selbst implementiert  $\Rightarrow$  später

Durch "objekt-orientierte Programmierung" ( $\Rightarrow$  später) erreicht man, dass zusammengesetzte Typen genauso einfach und bequem und effizient sind wie Basistypen (nur c++, nicht c)

- "Kapselung": die interne Struktur und Implementation ist für Benutzer unsichtbar
- Benutzer manipuliert Speicher über Funktionen ("member functions")  $\hat{=}$  Schnittstelle des Typs, "Interface", API

$\Rightarrow$  Punktsyntax: `type_name t = init; t.foo(a1, a2);  $\hat{=}$  foo(t, a1, a2);`

## 9.3 Zeichenketten-Strings:

zwei Datentypen in c++

- klassischer c-string: `char[]` ("charakter array")  $\Rightarrow$  nicht gekapselt, umständlich
- c++ string: `std::string` gekapselt und bequem (nur dieser in der Vorlesung)
- string literale: "Zeichenkette", einzelnes Zeichen: `'z'` ("z" = Kette der Länge 1)  
Vorsicht: die String-Literale sind c-strings (gibt keine c++ string-Literale), müssen erst in c++ strings umgewandelt werden, das passiert meist automatisch

- `#include <string>`
- Initialisierung:

```
std::string s = "abcde";
std::string s2 = s1;
std::string leer = "";
std::string leer(); // Abkürzung, default constructor
```
- Länge

```
s.size();
assert(s.size() == 5);
assert(leer.size() == 0);
s.empty() // Abkürzung für s.size() == 0
```

- Zuweisung

```
s = "xy";
s2 = leer;
```

- Addition Aneinanderkettung von String ("concatenate")

```
std::string s3 = s + "ijh"; // "xyijh"
s3 = "ghi" + s; // "ghixy"
s3 = s + s; // "xyxy"
// aber nicht!!
s3 = "abc" + "def"; // Bug literale unterstütze + mit ganz anderer Bedeutung
s3 = std::string("abc") + "def"; // Ok
```

- Add-Assignment: Abkürzung für Addition gefolgt von Zuweisung

```
s += "nmk"; // s = s + "nmk" => "xynmk"
```

- die Zeichen werden intern in einem C-Array gespeichert (Array = "Feld")  
Array: zusammenhängende Folge von Speicherzellen des gleichen Typs, hier 'char' (für einzelne Zeichen), Die Länge wird (bei std::string) automatisch angepasst, die einzelnen Speicherzellen sind durchnummeriert in c++: von 0 beginnend  $\hat{=}$  Index

- Indexoperator:

```
s[index]; // gibt das Zeichen an Position "index" zurück
```

Anwendung: jedes Zeichen einzeln ausgeben

```
std::string s = "abcde";

for(int i = 0; i < s.size(); i++) {
    std::cout << s[i] << std::endl;
}
```

String umkehren

```
int i = 0; // Anfang des Strings
int k = s.size() - 1; // Ende des String
while(i < k) {
    char tmp = s[i];
    s[i] = s[k];
    s[k] = tmp;
    i++; k--;
}
```

Variante 2: neuen String erzeugen

```
std::string s = "abcde";
std::string r = "";
for(int i = s.size() - 1; i >= 0; i--) {
    r += s[i];
}
```

## 10 Umgebungsmodell

Gegenstück zum Substitutionsmodell (in der funktionalen Programmierung) für die prozedurale Programmierung

- Regeln für Auswertung von Ausdrücken
  - Regeln für automatische Speicherverwaltung
    - Freigeben nicht mehr benötigter Speicherzellen,  $\Rightarrow$  bessere Approximation von "unendlich viel Speicher"
  - Umgebung beginnt normalerweise bei "{" und endet bei "}"  
Ausnahmen:
    - *for*: Umgebung beginnt schon bei "for"  $\Rightarrow$  Laufvariable ist Teil der Umgebung
    - Funktionsdefinitionen: Umgebung beginnt beim Funktionskopf  $\Rightarrow$  Speicherzellen für Argumente und Ergebnis gehören zur Umgebung
    - globale Umgebung außerhalb aller "{" }" klammern
  - automatische Speicherverwaltung
    - Speicherzellen, die in einer Umgebung angelegt werden (initialisiert, deklariert) werden, am Ende der Umgebung in umgekehrter Reihenfolge freigegeben
    - Compiler fügt vor "}" automatisch die Notwendigen Befehle ein
    - Speicherzellen in der globalen Umgebung werden am Programmende freigegeben
- ```
- int global = 1;
  int main() {
    int l = 2;
    {
      int m = 3
    } // <- m wird freigegeben
  } // <- l wird freigegeben
  // <- global wird freigegeben
```
- Umgebungen können beliebig geschachtelt werden  $\Rightarrow$  alle Umgebungen bilden einen Baum, mit der globalen Umgebung als Wurzel
  - Funktionen sind in der globalen Umgebung definiert
    - Umgebung jeder Funktion sind Kindknoten der globalen Umgebung (Ausnahme: Namensräume  $\Rightarrow$  siehe unten)  
 $\Rightarrow$  Funktions Umgebung ist **nicht** in der Umgebung, wo die Funktion aufgerufen wird
  - Jede Umgebung besitzt eine **Zuordnungstabelle** für alle Speicherzellen, die in der Umgebung definiert wurden

| Name | Typ | aktueller Wert |
|------|-----|----------------|
| 1    | int | 2              |

- jeder Name kann pro Umgebung nur einmal vorkommen
- Ausnahme Funktionsnamen können mehrmals vorkommen bei function overloading (nur c++)
- Alle Befehle werden relativ zur aktuellen Umgebung ausgeführt
  - aktuell: Zuordnungstabelle der gleichen Umgebung und aktueller Wert zum Zeitpunkt des Aufrufs  
Beispiel:  $c = a * b$ ;  
Regeln:
    - wird der Name (nur  $a, b, c$ ) in der aktuellen Zuordnungstabelle gefunden
      1. Typprüfung  $\Rightarrow$  Fehlermeldung, wenn Typ und Operation nicht zusammenpassen
      2. andernfalls, setze aktuellen Wert aus Tabelle in Ausdruck ein (ähnlich Substitutionsmodell)
    - wird Name nicht gefunden: suche in der Elternumgebung weiter
    - wird der Name bis zur Wurzel (globale Umgebung) nicht gefunden  $\Rightarrow$  Fehlermeldung
    - $\Rightarrow$  ist der Name in mehreren Umgebungen vorhanden gilt als zuerst gefundene (Typ, Wert)
- $\Rightarrow$  Programmierer muss selbst darauf achten, dass
  1. bei der Suche die gewünschte Speicherzelle gefunden wird  $\Rightarrow$  beutze "sprechende Namen"
  2. der aktuelle Wert der richtig ist  $\Rightarrow$  beachte Reihenfolge der Befehle!
- Namensraum: spezielle Umgebungen in der globalen Umgebung (auch geschachtelt) mit einem Namen  
Ziele:
  - Gruppieren von Funktionalität in Module (zusätzlich zu Headern)
  - Verhinderung von Namenskollisionen

Beispiel: c++ Standardbibliothek:

```
namespace std {
double sqrt(double x);
namespace chrono {
class system_clock;
}
}
```



// Benutzung mit Namespace-Prefix:

```
std::sqrt(80);
```

```
std::chrono::system_clock clock;
```

Besonderheit: mehrere Blöcke mit selbem Namensraum werden verschmolzen

Beispiel

```
int p = 2;
```

```
int q = 3;
```

```
int foo(int p) {
```

```
    return p * q;
```

```
}
```

```
int main() {
```

```
    int k = p * q; // beides global => 6 = 2 * 3
```

```
    int p = 4; // lokales p verdeckt globales p
```

```
    int r = p * q; // p lokal, q global => 12 = 4 * 3
```

```
    int s = foo(p); // lokale p von main() wird zum lokalen p von foo() 12 = 4 * 3
```

```
    int t = foo(q); // globales q wird zum lokalen p von foo() 9 = 3 * 3
```

```
    int q = 5;
```

```
    int n = foo(g); // lokales q wird zum lokalen p von foo() 15 = 5 * 3
```

```
}
```

## 11 Referenzen

sind neue (zusätzliche) Namen für vorhandene Speicherzellen

```
int x = 3; // neue Variable x mit neuer Speicherzelle
```

```
int &y = x; // Referenz: y ist neuer Name für x, beide haben die selbe Speicherzelle
```

```
y = 4; // Zuweisung an y, aber x ändert sich auch, das heißt x == 4
```

```
x = 5; // jetzt y == 5
```

```
int const &z = x; // read-only Referenz, das heißt z = 6 ist verboten
```

```
x = 6; // jetzt auch z == 6
```

Hauptanwendung:

- die Umgebung, wo eine Funktion aufgerufen wird und die Umgebung der Implementation sind unabhängig, das heißt Variablen der einen Umgebung sind in der anderen nicht sichtbar
- häufig möchte man Speicherzellen in beider Umgebungen teilen  $\Rightarrow$  verwende Referenzen
- häufig will man vermeiden, dass eine Variable kopiert wird (pass-by-value)

- Durch pass-by-reference brauch man keine Kopie  $\Rightarrow$  typisch "const &", also read-only, keine Seiteneffekte

```
int foo(int x) { // pass-by-value
    x += 3;
    return x;
}

int var(int & y) { // pass-by-reference
    y += 3; // Seiteneffekt der Funktion
    return y;
}

void baz(int & z) { // pass-by-reference
    z += 3;
}

int main() {
    int a = 3;
    std::cout << foo(a) << std::endl; // 5
    std::cout << a << std::endl; // 2
    std::cout << bar(a) << std::endl; // 5
    std::cout << a << std::endl; // 5
    baz(a);
    std::cout << a << std::endl; // 8
}
```

in der funktionalen Programmierung sind Seiteneffekte grundsätzlich verboten, mit Ausnahmen, zum Beispiel für Ein-/Ausgabe

## 12 Container-Datentypen

Dienen dazu, andere Daten aufzubewahren

- Art der Elemente:
  - homogene Container: alle Elemente haben gleichen Type (typisch für c++)
  - heterogene Container: Elemente könne verschiedene Typen haben (z.B. Python)
- nach Größen
  - statische Container: feste Größe, zur Compilezeit bekannt
  - dynamische Container: Größe zur Laufzeit veränderbar
- Arrays sind die wichtigsten Container, weil effizient auf Hardware abgebildet und einfach zu benutzen

- klassisch: Arrays sind statisch, zum Beispiel C-Arrays (hat c++ geerbt)
 

```
int a[20];
```
- modern: dynamische Arrays
  - Entdeckung einer effizienten Implementation
  - Kapselung durch objekt-orientierte Programmierung (sonst zu kompliziert)
- wir kennen bereits ein dynamisches Array: `std::string` ist Abbildung `int` (Index)  $\rightarrow$  `char` (Zeichen), mit  $0 \leq \text{index} < \text{s.size}()$ 
  - wichtigste Funktion: `s.size()` (weil Größe dynamisch), `s[4]` Indexzugriff, `s+="mehr"` Zeichen anhängen
- wir wollen das selbe Verhalten für beliebige Elementtypen:
 

```
#include <vector>

//          Elementtyp    Größe    Initialwert der Elemente
std::vector<double    > v(20    ,          0.0          );
// analog
std::vector<int>;
std::vector<std::string>;
```
- weitere Verallgemeinerung: Indextyp beliebig (man sagt dann "Schlüssel-Typ") "assoziatives Array"
  - typische Fälle:
    - Index ist nicht im Bereich  $(0, \text{size}]$ , zum Beispiel Matrikelnummern
    - Index ist string, zum Beispiel Name eines Studenten

```
#include <map>
#include <unordered_map>

// Binärer Suchbaum
std::map;

// Hashtabelle, siehe Algorithmen und Datenstrukturen
std::unordered_map;

//          Schlüsseltyp    Elementtyp
std::map<int          , double> noten; noten[3121101] = 10;
std::map<std::string, double> noten; noten["krause"] = 10;
```

  - Indexoperationen wie beim Array
  - Elemente werden beim 1. Zugriff automatisch erzeugt (dynamisch)

- alle dynamischen und assoziativen Arrays unterstützen `a.size()` zum Abfragen der Größe

## 12.1 `std::vector`

- Erzeugen:

```
std::vector<double> v(20, 1.0);
std::vector<double> v; // leeres Array
std::vector<double> v = {1.0, -3.0, 2.2}; // "initializer list": Element für Anfangs
```

- Größe:

```
v.size();
v.empty(); // => v.size() == 0
```

- Größe ändern

```
v.resize(neue_groesse, initialwert);
// Dann:
// Fall 1: neue_groesse < size(): Element ab Index "neue_groesse" gelöscht die anderen
// Fall 2: neue_groesse > size(): neue Elemente mit Initialwert am Ende anhängen, die
// Fall 3: neue_groesse == size(): nichts passiert
```

```
v.push_back(neues_element); // ein neues Element am Ende anhängen (ähnlich string +=)
v.insert(v.begin() + index, neues_element); // neues element an Position "index" einfügen
// Falls index == size(): am Ende anhängen, sonst: alte Elemente ab Index werden ein
```

```
v.pop_back(); // letztes Element löschen (effizient)
v.erase(v.begin() + index); // Element an Pos index löschen, alles dahinter eine Pos
v.clear(); // alles löschen
```

- Zugriff

```
v[k]; // Element bei Index k
v.at(k); // wie v[k], aber Fehlermeldung, wenn nicht 0 <= k < size() (zum Debuggen)
```

- Funktionen für Container benutzen in c++ immer Iteratoren, damit sie für verschiedene Container funktionieren

- Iterator-Range

```
// erstes Element
v.begin()

// hinter letztem Element
v.end()
```

- im Header `<algorithm>`
- alle Elemente kopieren

```
std::vector<double> source = {1.0, 2, 3, 4, 5};
std::vector<double> target(source.size(), 0.0);
std::copy(source.begin(), source.end(), target.begin());
std::copy(source.begin() + 2, source.end() - 1, target.begin()); // nur index 2
```

- Elemente sortieren

```
std::sort(v.begin(), v.end()); // "in-place" sortieren
```
- Elemente mischen:

```
std::random_shuffle(v.begin(), v.end()); "in-place" mischen
```