

# Einführung in die Anwendungsorientierte Informatik (Köthe)

Robin Heinemann

December 9, 2016

## Contents

<b>1</b>	<b>Was ist Informatik?</b>	<b>3</b>
1.1	Teilgebiete . . . . .	3
1.1.1	theoretische Informatik ( <b>ITH</b> ) . . . . .	3
1.1.2	technische Informatik ( <b>ITE</b> ) . . . . .	3
1.1.3	praktische Informatik . . . . .	3
1.1.4	angewandte Informatik . . . . .	4
<b>2</b>	<b>Wie unterscheidet sich Informatik von anderen Disziplinen?</b>	<b>4</b>
2.1	Mathematik . . . . .	4
<b>3</b>	<b>Informatik</b>	<b>4</b>
3.1	Algorithmus . . . . .	4
3.2	Daten . . . . .	5
3.2.1	Beispiele für Symbole . . . . .	5
3.3	Einfachster Computer . . . . .	6
3.3.1	<b>TODO</b> Graphische Darstellung . . . . .	6
3.3.2	<b>TODO</b> Darstellung durch Übergangstabellen . . . . .	6
3.3.3	Beispiel 2: . . . . .	6
<b>4</b>	<b>Substitutionsmodell (funktionale Programmierung)</b>	<b>8</b>
4.1	Substitutionsmodell . . . . .	8
4.2	Bäume . . . . .	9
4.2.1	Beispiel . . . . .	10
4.3	Rekursion . . . . .	10
4.4	Präfixnotation aus dem Baum rekonstruieren . . . . .	10
4.5	Präfixnotation aus dem Baum rekonstruieren . . . . .	10
4.6	Berechnen des Werts mit Substitutionsmodell . . . . .	11

<b>5</b>	<b>Maschinensprachen</b>	<b>11</b>
5.1	Umwandlung in Maschinensprache . . . . .	11
<b>6</b>	<b>Funktionale Programmierung</b>	<b>12</b>
6.1	Beispiel . . . . .	12
6.2	Vorteile von Zwischenergebnissen . . . . .	13
6.3	Funktionale Programmierung in c++ . . . . .	13
<b>7</b>	<b>Prozedurale Programmierung</b>	<b>15</b>
7.1	Von der Funktionalen zur prozeduralen Programmierung . . . . .	15
7.2	Kennzeichen . . . . .	16
7.2.1	Prozeduren . . . . .	16
7.2.2	Steuerung des Programmablaufs . . . . .	16
7.2.3	Veränderung von Speicherzellen . . . . .	17
7.2.4	Schleifen . . . . .	18
7.2.5	prozedurale Wurzelberechnung . . . . .	18
7.2.6	for-Schleife . . . . .	19
<b>8</b>	<b>Datentypen</b>	<b>21</b>
8.1	Basistypen . . . . .	21
8.2	zusammengesetzte Typen . . . . .	21
8.3	Zeichenketten-Strings: . . . . .	21
<b>9</b>	<b>Umgebungsmodell</b>	<b>23</b>
<b>10</b>	<b>Referenzen</b>	<b>25</b>
<b>11</b>	<b>Container-Datentypen</b>	<b>26</b>
11.1	std::vector . . . . .	28
11.1.1	Effizienz von push\_back . . . . .	29
<b>12</b>	<b>Iteratoren</b>	<b>31</b>
<b>13</b>	<b>Insertion Sort</b>	<b>34</b>
<b>14</b>	<b>generische Programmierung</b>	<b>35</b>
<b>15</b>	<b>Effizienz von Algorithmen und Datenstrukturen</b>	<b>38</b>
15.1	Bestimmung der Effizienz . . . . .	38
15.1.1	wall clock . . . . .	38
15.1.2	algorithmische Komplexität . . . . .	39
15.1.3	Anwendung . . . . .	41

# 1 Was ist Informatik?

”Kunst” Aufgaben mit Computerprogrammen zu lösen.

## 1.1 Teilgebiete

### 1.1.1 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?

### 1.1.2 technische Informatik (ITE)

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

### 1.1.3 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

#### 1.1.4 angewandte Informatik

- Wie löst man Probleme aus einem anderem Gebiet mit Programmen?
- Informationstechnik
  - Buchhandlung, e-Kommerz, Logistik
- Web Programmierung
- scientific computing für Physik, Biologie
- Medizininformatik
  - bildgebende Verfahren
  - digitale Patientenakte
- Computer Linguistik
  - Sprachverstehen, automatische Übersetzung
- Unterhaltung: Spiele, special effects im Film

## 2 Wie unterscheidet sich Informatik von anderen Disziplinen?

### 2.1 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:  $y = \sqrt{x} \iff y \geq 0 \wedge y^2 = x (\implies x \geq 0)$

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

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

## 3 Informatik

Lösungswege, genauer Algorithmen

### 3.1 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 ( $\implies$  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

## 3.2 Daten

Daten sind Symbole,

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

⇒ 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"

### 3.2.1 Beispiele für Symbole

- Zahlen
- Buchstaben
- Icons
- Verkehrszeichen

aber: heutige Computer verstehen nur Binärzahlen ⇒ 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

### 3.3 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

#### 3.3.1 TODO Graphische Darstellung

graphische Darstellung: Zustände = Kreise, Zustandsübergänge: Pfeile

#### 3.3.2 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	( $\Rightarrow \{\text{aus}\}, \{\text{Alle LEDs aus, Timer stoppen}\}$ )	( $\Rightarrow \{\text{blinken aus}\}, \{\text{alle LEDs}\}$ )
blinken aus	( $\Rightarrow \{\text{aus}\}, \{\text{Alle LEDs aus, Timer stoppen}\}$ )	( $\Rightarrow \{\text{blinken an}\}, \{\text{alle LEDs}\}$ )

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

#### 3.3.3 Beispiel 2:

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)
1 0 0 1 0 0 1 1	$= 1 + 2 + 16 + 128 = 147_{\text{dez}} \checkmark$	(3)

**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

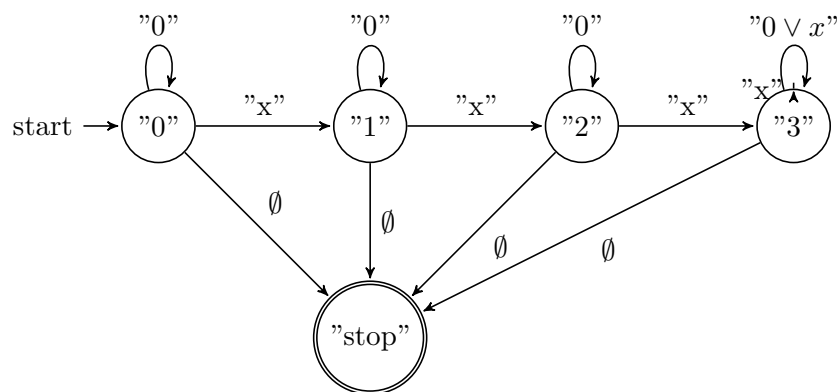
## TODO Skizze der Automaten

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) Spendiert 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 ausführen, 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.

## 4 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
  - $\text{mul}(\text{add}(1,2), \text{sub}(5,3)) \rightarrow 6$

### 4.1 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 des Standardbibliothek: "cmath"
- Für Arithmetik gebräuchlicher ist "Infix-Notation" mit Operator-Symbolen "+", "-", "\*", "/", "%"



- $\text{mul}(\text{add}(1,2), \text{sub}(5,3)) \iff ((1+2)*(5-3))$ 
  - oft besser, unter anderem weil man Klammer weglassen darf
    1. "Punkt vor Strichrechnung"  $3+4*5 \iff 3+(4*5)$ , mul, div, mod binden stärker als add, sub
    2. Operatoren gleicher Präzedenz werden von links nach rechts ausgeführt (links-assoziativ)  
 $1+2+3-4+5 \iff (((1+2)+3)-4)+5$
    3. äußere Klammer kann man weglassen  $(1+2) \iff 1+2$
- Computer wandeln Infix zuerst in Präfix Notation um
  1. weggelassene Klammer wieder einfügen
  2. Operatorensymbol durch Funktionsnamen ersetzen und an Präfix-Position verschieben

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

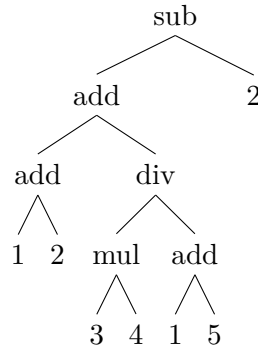
## 4.2 Bäume

- 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

#### 4.2.1 Beispiel

$$1 + 2 + 3 * 4 / (1 + 5) - 2$$

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



#### 4.3 Rekursion

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

#### 4.4 Präfixnotation 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 ")"

$\Rightarrow$

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

#### 4.5 Präfixnotation 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**

#### 4.6 Berechnen des Werts mit Substitutionsmodell

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

⇒ **post-order**

### 5 Maschinensprachen

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

#### 5.1 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-Ergebnis  $\leftarrow$  fname SpZArg1 SpZArg2

4. alles in Zahlencode umwandeln

- Funktionsnamen:

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

## 6 Funktionale Programmierung

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

### 6.1 Beispiel

Lösen einer quadratischen Gleichung:

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

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

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

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 Zwischenergebnis und Infix Notation

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

$$a \leftarrow c/a$$

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

$$x_1 \leftarrow p + d$$

$$x_2 \leftarrow p - d$$

## 6.2 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  
}
```

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

## 6.3 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  $\implies$  "function overloading"  $\implies$  c++ wählt automatisch die richtig Variable anhand des Argumenttyps ("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 namens "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 +, -, \*, /, %
    - Präfix-Operatoren *operator*+, *operator*-, ...
  - Funktion der Standardbibliothek  $\hat{=}$  müssen "angefordert" werden
    - Namen beginnen mit "std::", "std::sin,..."
    - sind in Module geordnet, zum Beispiel
      - cmath  $\implies$  algebraische Funktion
      - complex  $\implies$  komplexe Zahlen
      - string  $\implies$  Zeichenkettenverarbeitung

- um ein Modul zu benutzen muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren (Header inkludieren) → `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 Operationen
    - overloading genau wie bei *sq*
    - $3 * 4 \implies$  int Variante
    - $3.0 * 4.0 \implies$  double Variante
    - $3 * 4.0 \implies$  automatische Umwandlung in höheren Typ, hier "double"  $\implies$  wird als  $3.0 * 4.0$  ausgeführt
  - $\implies$  Division unterscheidet sich
    - Integer-Division:  $12 / 5 = 2$  (wird abgerundet)
    - Double-Division:  $12.0 / 5.0 = 2.4$
    - $-12 / 5 = 2$  ( $\implies$  truncated Division)
    - $12.0 / 5.0 = 2.4$
    - Gegensatz (zum Beispiel in Python)
      - floor division  $\implies$  wird immer abgerundet  $\implies -12 / 4 = -2$

## 7 Prozedurale Programmierung

### 7.1 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 Funktionalität ("Taschenrechner")
- einfache Auswertung durch Substitutionsmodell → Auswertungsreihenfolge nach Post-Order
- mathematisch gut formalisierbar  $\implies$  Korrektheitsbeweise, besonders bei Parallelverarbeitung
- Rekursion ist mächtig und natürliche für bestimmte Probleme (Fakultät, Baum-Traversierung)
- Nachteile
  - viele Probleme lassen sich anders natürlicher ausdrücken (z.B. Rekursion vs. Iteration)
  - setzt unendlich viel Speicher voraus (  $\implies$  Memory Management notwendig  $\implies$  später)
  - Entitäten, die sich zeitlich verändern sind schwer zu modellieren
- Korollar: kann keine externen Ressourcen (z.B. Konsole, Drucker, ..., Bildschirm) ansprechen "keine Seiteneffekte"
  - $\implies$  Multi-Paradigmen-Sprachen, zum Beispiel Kombination von Funktionaler Programmierung und prozeduraler Programmierung

## 7.2 Kennzeichen

### 7.2.1 Prozeduren

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

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

- Prozeduren in c++

1. Funktion die "void" zurück gibt (Pseudotyp für "nichts")

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

2. Returnwert ignorieren

### 7.2.2 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;
}
```

### 7.2.3 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 * y1;   // z1 = 6
    int y2 = 5;
    int z2 = z1 * y2;   // z2 = 15
    return z1 + z2;    // 21
}
```

- Syntax

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

- $\Rightarrow$  Folgen: mächtiger, aber ermöglicht völlig neue Bugs  $\Rightarrow$  erhöhte Aufmerksamkeit beim Programmieren

- die Reihenfolge der Ausführung ist viel kritischer als beim Substitutionsmodell
- Programmierer muss immer ein mentales Bild des aktuellen Systemzustands haben

#### 7.2.4 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     | ∅       |

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

#### 7.2.5 prozedurale Wurzelberechnung

**Ziel**

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

**Methode** iterative Verbesserung mittels Newtonverfahren initial\_guess  $x^{(0)}$  ("geraten"),  $t = 0$   
while not\_good\_enough( $x^{(t)}$ ):  
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  $x^* = x + \Delta$

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

Iterationsvorschrift:

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

Anwendung auf Wurzel: setze  $f(x) = x^2 - y \implies$  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);
    }
}
```

### 7.2.6 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 Schleifenkörper
  - break: bricht die ganze 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;
        }
    }
}

```

## 8 Datentypen

### 8.1 Basistypen

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

- int, double, bool (  $\implies$  später mehr)

### 8.2 zusammengesetzte Typen

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

- wie das geht  $\implies$  später
- Standardtypen: in der C++ Standardbibliothek definiert, aktivieren 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  $\implies$  später

Durch "objekt-orientierte Programmierung" (  $\implies$  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

$\implies$  Punktsyntax: `type\_name t = init; t.foo(a1, a2);  $\hat{=}$  foo(t, a1, a2);`

### 8.3 Zeichenketten-Strings:

zwei Datentypen in c++

- klassischer c-string: `char[]` ("Charakter Array")  $\implies$  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 Konstruktor

```

- 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 unterstützen + 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];
}

```

## 9 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,  $\implies$  bessere Approximation von "unendlich viel Speicher"
- Umgebung beginnt normalerweise bei "{" und endet bei "}"
 

Ausnahmen:

  - *for*: Umgebung beginnt schon bei "for"  $\implies$  Laufvariable ist Teil der Umgebung
  - Funktionsdefinitionen: Umgebung beginnt beim Funktionskopf  $\implies$  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
  }

```

```

} // <- 1 wird freigegeben
// <- global wird freigegeben

```

- Umgebungen können beliebig geschachtelt werden  $\implies$  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  $\implies$  siehe unten)
  - $\implies$  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  $\implies$  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  $\implies$  Fehlermeldung
    - $\implies$  ist der Name in mehreren Umgebungen vorhanden gilt der zuerst gefundene (Typ, Wert)
- $\implies$  Programmierer muss selbst darauf achten, dass
  1. bei der Suche die gewünschte Speicherzelle gefunden wird  $\implies$  benutze "sprechende Namen"
  2. der aktuelle Wert der richtig ist  $\implies$  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-Präfix:

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

## 10 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, in der 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 beiden Umgebungen teilen  $\implies$  verwende Referenzen
- häufig will man vermeiden, dass eine Variable kopiert wird (pass-by-value)
  - Durch pass-by-reference braucht man keine Kopie  $\implies$  typisch "const &", also read-only, keine Seiteneffekte

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

int bar(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; // 3
    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

## 11 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) → 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 dasselbe 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,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

## 11.1 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 Position index löschen, alles dahinter ein
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
```

### 11.1.1 Effizienz von push\\_back

Warum ist push\\_back() effizient? (bei std::vector)

- veraltete Lehrmeinung: Arrays sind nur effizient wenn statisch (das heißt Größe zur Compilezeit, oder spätestens bei Initialisierung, bekannt)
  - sonst: andere Datenstruktur verwenden, zum Beispiel verkettete Liste (std::list)
- modern: bei vielen Anwendungen genügt, wenn Array (meist) nur am Ende vergrößert wird (zum Beispiel push\\_back())
  - dies kann sehr effizient unterstützt werden  $\implies$  dynamisches Array
- std::vector verwaltet intern ein statisches Array der Größe "capacity", v.capacity()  $\geq$  c.size()
  - wird das interne Array zu klein  $\implies$  wird automatisch auf ein doppelt so großes umgeschaltet
  - ist das interne Array zu groß, bleiben unbenutzte Speicherzellen als Reserve
- Verhalten bei push\\_back():
  1. noch Reserve vorhanden: lege neues Element in eine unbenutzte Speicherzelle  $\implies$  billig

## 2. keine Reserve

- alloziere neues statisches Array mit doppelt Kapazität
- kopiere die Daten aus dem alten in das neue Array
- gebe das alte Array frei
- gehe zum Anfang des Algorithmus, jetzt ist wieder Reserve vorhanden

· das Umkopieren ist nicht zu teuer, weil es nur selten notwendig ist

· Beispiel:

```
std::vector<int> v;
```

```
for(int i = 0; i < 32; i++) v.push_back(k);
```

| k     | capacity vor push_back() | capacity nach push_back() | size() | Reserve | #Umkopieren |
|-------|--------------------------|---------------------------|--------|---------|-------------|
| 0     | 0                        | 1                         | 1      | 0       | 0           |
| 1     | 1                        | 2                         | 2      | 0       | 1           |
| 2     | 2                        | 4                         | 3      | 1       | 2           |
| 3     | 4                        | 4                         | 4      | 0       | 2           |
| 4     | 4                        | 8                         | 5      | 3       | 4           |
| 5-7   | 8                        | 8                         | 8      | 0       | 0           |
| 8     | 8                        | 16                        | 9      | 7       | 8           |
| 9-15  | 16                       | 16                        | 16     | 0       | 0           |
| 16    | 16                       | 32                        | 17     | 15      | 16          |
| 17-31 | 32                       | 32                        | 32     | 0       | 0           |

· was kostet das:

- 32 Elemente einfügen = 32 Kopien extern  $\Rightarrow$  intern
- aus altem Array ins neu kopieren  $(1 + 2 + 4 + 8 + 16) = 31$  kopieren intern  $\Rightarrow$  intern
- $\Rightarrow$  im Durchschnitt sind pro Einfügung 2 Kopien nötig
- $\Rightarrow$  dynamisches Array ist doppelt so teuer wie das statische  $\Rightarrow$  immer noch sehr effizient

· relevante Funktionen von std::vector

```
v.size() // aktuelle Zahl der Elemente
v.capacity() // aktuelle Zahl Speicherzellen
assert(v.capacity() - v.size() >= 0) // Reserve
v.resize(new_size) // ändert immer v.size(), aber v.capacity() nur wenn < new_size
v.reserve(new_capacity) // ändert v.size() nicht, aber v.capacity() falls new_capacity > v.capacity()
v.shrink_to_fit() // == v.reserve(v.size()) Reserve ist danach 0, wenn Endgröße erreicht
```

· wenn Reserve > size: capacity kann auch halbiert werden

- wichtige Container der c++ Standardbibliothek
- wir hatten dynamische Arrays `std::string`, `std::vector`, assoziative Arrays `std::map`, `std::unordered_map`
- `std::set`, `std::unordered_set`: Menge, jedes Element ist höchstens einmal enthalten zum Beispiel Duplikate
- `std::stack` (Stapel, Keller): unterstützt `push` und `pop()` mit Last in- First out Semantik (LIFO) äquivalent zu `push_back()` und `pop_back()` bei `std::vector`
- `std::queue` (Warteschlange) `push()` und `pop()` mit First in-first out Semantik (FIFO)
- `std::deque` ("double-ended queue") gleichzeitig `stack` und `queue`, `push`, `pop_front()`, `pop_back()`
- `std::priority_queue`, `push()` und `pop()` - Element mit höchster niedrigster Priorität (user defined)

## 12 Iteratoren

- für Arrays lautet die kanonische Schleife

```
for(int i = 0; i != v.size(); i++) {
    int current = v[i]; // lesen
    v[i] = new_value; // schreiben
}
```
  - wir wollen eine so einfache Schleife für beliebige Container
    - der Index-Zugriff `v[i]` ist bei den meisten Container nicht effizient
    - Iteratoren sind immer effizient  $\implies$  es gibt sie in allen modernen Programmiersprachen, aber Details sehr unterschiedlich
    - Analogie: Zeiger einer Uhr, Cursor in Textverarbeitung
      - $\implies$  ein Iterator zeigt immer auf ein Element des Containers, oder auf Spezialwert "ungültiges Element"
    - in c++ unterstützt jeder Iterator 5 Grundoperationen
      1. Iterator auf erstes Element erzeugen: `auto iter = v.begin();`
      2. Iterator auf "ungültiges Element" erzeugen: `auto end = v.end();`
      3. Vergleich `iter1 == iter2` (Zeigen auf gleiches Element), `iter != end`:
- iter zeigt **nicht** auf ungültiges Element
1. zum nächsten weitergehen: `++iter`. Ergebnis ist `v.end()`, wenn man vorher beim letzten Element war

2. auf Daten zugreifen: `*iter` ("Dereferenzierung") analog `v[k]`

kanonische Schleife:

```
for(auto iter = v.begin(); iter != v.end(); ++iter) {
    int current = *iter; // lesen
    *iter = new_value; // schreiben
}
// Abkürzung: range-based for loop
for(auto & element : v) {
    int current = element; // lesen
    element = new_value; // schreiben
}
```

- Iteratoren mit den 5 Grundoperationen heißen "forward iterator" (wegen `++iter`)
- "bidirectional iterators": unterstützen auch `--iter`, zum vorigen Element ((fast) alle Iteratoren in `std`)
- "random access iterators": beliebige Sprünge `*iter += 5; iter -= 3;`
- Besonderheit für assoziative Arrays (`std::map`, `std::unordered_map`) Schlüssel und Werte können beliebig gewählt werden

- $\Rightarrow$  das aktuelle Element ist ein Schlüssel / Wert -Paar, das heißt Iterator gibt Schlüssel und Wert zurück

```
(*iter).first; // Schlüssel
(*iter).second; // Wert
// Abkürzung
iter->first;
iter->second;
```

- bei `std::map` liefern die Iteratoren die Elemente in aufsteigender Reihenfolge der Schlüssel

- Die Funktion `std::transform()`

- wir hatten: `std::copy()`

```
std::vector<double> source = {1, 2, 3, 4};
std::vector<double> target(source.size());
std::copy(source.begin(), source.end(), target.begin());
```

- `std::transform`:

```
// nach Kleinbuchstaben konvertieren
std::string source = "aAbCdE";
std::string target = source;
std::transform(source.begin(), source.end(), target.begin(), std::tolower); // 1
// die Daten quadrieren
double sq(double x) { return x * x; }
```



```

std::transform(source.begin(), source.end(), target.begin(), sq); // target == +
// das ist eine Abkürzung für eine Schleife
auto src_begin = source.begin();
auto src_end = source.end();
auto tgt_begin = target.begin();

for(; src_begin != src_end; src_begin++, tgt_begin++) {
    *tgt_begin = sq(*src_begin);
}

```

- Der Argumenttyp der Funktion muss mit dem source Elementtyp kompatibel sein. Der Returntyp der Funktion muss mit dem Target-Elementtyp kompatibel sein.
- Das letzte Argument von std::transform() muss ein Funktor sein (verhält sich wie eine Funktion), drei Varianten:

1. normale Funktion, z.B. sq. Aber: wenn Funktion für mehrere Argumenttypen überladen ist (overloading) (zum Beispiel, wenn es sq(double) und sq(int) gibt), muss der Programmierer dem Compiler sagen, welche Version gemeint ist  $\implies$  für Fortgeschrittene ("functionpointer cast")
2. Funktionsobjekt  $\implies$  objekt-orientierte Programmierung
3. Definiere eine namenlose Funktion  $\implies$  "Lambda-Funktion  $\lambda$ "

- statt  $\lambda$  verwenden wir den Universalnamen  $\square$

```

std::transform(source.begin(), source.end(), target.begin(), [] (double)
// Returntyp setzt Computer automatisch ein, wenn es nur einen return-B

```

- Lambda-Funktionen können noch viel mehr  $\implies$  für Fortgeschrittene
- std::transform() kann in-place arbeiten (das heißt source-Container überschreiben), wenn source und target gleich

```

std::transform(source.begin(), source.end(), source.begin(), sq);

```

- Die Funktion std::sort() zum in-place sortieren eines Arrays

```

std::vector<double> v = {4, 2, 3, 5, 1};
std::sort(v.begin(), v.end()); // v == {1, 2, 3, 4, 5}

```

- std::sort ruft intern den <-Operator des Elementtyps auf, um Reihenfolge zu bestimmen
- die <-Operation muss eine totale Ordnung der Elemente definieren:
  - $a < b$  muss für beliebige  $a, b$  ausführbar sein
  - transitiv:  $(a < b) \wedge (b < c) \implies (a < c)$
  - anti-symmetrisch:  $\neg(a < b) \wedge \neg(b < a) \implies a == b$

## 13 Insertion Sort

schnellster Sortieralgorithmus für kleine Arrays ( $n \leq 30$ ) hängt von Compiler und CPU ab

- Idee von Insertion Sort:
  - wie beim Aufnehmen und Ordnen eines Kartenblatts
  - gegeben: bereits sortierte Teilmenge bis Position  $k - 1$  Karten bereits in Fächer
  - Einfügen des  $k$ -ten Elements an richtiger Stelle  $\rightarrow$  Erzeuge Lücke an richtiger Position durch verschieben von Elementen nach rechts
  - Wiederholung für  $k = 1, \dots, N$
  - Beispiel:

|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 2 | 3 | 5 | 1 |
| 4 | — | 3 | 5 | 1 |
| — | 4 | 3 | 5 | 1 |
| 2 | 4 | 3 | 5 | 1 |
| 2 | 4 | — | 5 | 1 |
| 2 | — | 4 | 5 | 1 |
| 2 | 3 | 4 | 5 | 1 |
| 2 | 3 | 4 | — | 1 |
| 2 | 3 | 4 | 5 | 1 |
| 2 | 3 | 4 | 5 | — |
| — | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |

```
void insertion_sort(std::vector<double> & v) {
    for(int i = 0; i < v.size(); i++) {
        double current = v[i];
        int j = i; // Anfangsposition der Lücke
        while(j > 0) {
            if(v[j - 1] < current) { // -> if(cmp(a, b))
                break; // j ist richtige Position der Lücke
            }
            v[j] = v[j - 1];
            j--;
        }
        v[j] = current;
    }
}
```

- andere Sortierung: definiere Funktor  $\text{cmp}(a, b)$ , der das gewünschte kleiner realisiert (gibt genau dann "true" zurück, wenn  $a$  "kleiner"  $b$  nach neuer Sortierung)

- neue Sortierung am besten per Lambda-Funktion an `std::sort` übergeben

```
std::sort(v.begin(), v.end()); // Standard sort mit "<"
std::sort(v.begin(), v.end(), [](double a, double b) { return a < b; }); // Standard sort mit Lambda-Funktion
std::sort(v.begin(), v.end(), [](double a, double b) { return b < a; }); // absteigend sortieren
std::sort(v.begin(), v.end(), [](double a, double b) { return std::abs(a) < std::abs(b); }); // nach Absolutwert sortieren
std::sort(v.begin(), v.end(), [](std::string a, std::string b) {
    std::transform(a.begin(), a.end(), a.begin(), std::tolower);
    std::transform(b.begin(), b.end(), b.begin(), std::tolower);
    return a < b;
});
```

## 14 generische Programmierung

`insertion\_sort` soll für beliebige Elementtypen funktionieren

```
template<typename T>
void insertion_sort(std::vector<T> & v) {
    for(int i = 0; i < v.size(); i++) {
        T current = v[i];
        int j = i; // Anfangsposition der Lücke
        while(j > 0) {
            if(v[j - 1] < current) { // -> if(cmp(a, b))
                break; // j ist richtige Position der Lücke
            }
            v[j] = v[j - 1];
            j--;
        }
        v[j] = current;
    }
}
```

- Ziel: benutze template-Mechanismus, damit **eine** Implementation für viele verschiedene Typen verwendbar ist
  - erweitert funktionale und prozedurale und objekt-orientierte Programmierung
- zwei Arten von Templates ("Schablone"):
  1. Klassen-templates für Datenstrukturen, zum Beispiel Container sollen beliebige Elementtypen unterstützen
    - Implementation  $\implies$  später
    - Benutzung: Datenstrukturname gefolgt vom Elementtyp in spitzen Klammern (`std::vector<double>`), oder mehrere Typen, zum Beispiel Schlüssel und Wert bei `std::map<std::string, double>`

2. Funktionen-Templates: es gab schon function overloading

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

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

// und so weiter für komplexe und rationale Zahlen...

- Nachteil
  - wenn die Implementationen gleich sind → nutzlose Arbeit
  - Redundanz ist gefährlich: korrigiert man einen Bug wird leicht eine Variante vergessen

- mit templates reicht eine Implementation

```
template<typename T> // T: Platzhalter für beliebigen Typ, wird später durch  
T sq(T x) {  
    return x * x; // implizierte Anforderung an den Typ T, er muss Multiplik  
}
```

- wie bei Substituieren von Variablen mit Werten, aber jetzt mit Typen
- Benutzung:
  - Typen für die Platzhalter hinter dem Funktionsnamen in spitzen klammern

```
sq<int>(2) == 4;  
sq<double>(3.0) == 9.0,
```

- meist kann man die Typenangabe <type> weglassen, weil der Computer sie anhand des Argumenttyps automatisch einsetzt:

```
sq(2); // == sq<int>(2) == 4  
sq(3.0); // == sq<double>(3.0) == 9
```

- kombiniert man templates mit Overloading, wird die ausprogrammierte Variante vom Compiler bevorzugt. Komplizierte Fälle (Argument teilweise Template, teilweise hard\_coded) ⇒ für Fortgeschrittene
- Beispiel 2: Funktion, die ein Array auf Konsole ausgibt, für beliebige Elementtypen

```
template<typename ElementType>  
void print_vector(std::vector<ElementType> const & v) {  
    std::cout << "{";  
    if(v.size() > 0) {
```

```

        std::cout << " " << v[0];
        for(int i = 1; i < v.size(); i++) {
            std::cout << ", " << v[i];
        }
    }
    std::cout << " }";
}

```

- Verallgemeinerung für beliebige Container mittels Iteratoren:

```

std::list<int> l = {1, 2, 3};
print_container(l.begin(), l.end()); // "{1,2,3}"

```

- es genügen forward\_\_iterators

```

Iterator iter2 = iter1; // Kopie erzeugen
iter1++; // zum nächsten Element
iter1 == iter2; // Zeigen sie auf das selbe Element?
iter1 != end;
*iter1; // Zugriff auf aktuelles Element

```

```

template<typename Iterator>
void print_container(Iterator begin, Iterator end) {
    std::cout << "{";
    if(begin != end) { // Container nicht leer?
        std::cout << " " << *begin++;
        for(;begin != end; begin++) {
            std::cout << ", " << *begin;
        }
    }
    std::cout << "}";
}

```

- Beispiel 3: überprüfen, ob Container sortiert ist

```

template<typename E, typename CMP>
bool check_sorted(std::vector<E> const & v, CMP less_than) {
    for(int i = 1; i < v.size(); i++) {
        if(less_than(v[i], v[i - 1])) { // statt v[i] < v[i - 1], ausnutzen
            return false;
        }
    }
    return true;
}

```

```

// Aufruf:
std::vector<double> v = {1.0, 2.0, 3.0};
check_sorted(v, [](double a, double b) { return a < b; }); // == true

```

```

check_sorted(v, [](double a, double b) { return a > b; }); // == false

// Implementation für Iteratoren
template<typename Iterator, typename CMP>
bool check_sorted(Iterator begin, Iterator end, CMP less_than) {
    if(begin == end) {
        return true;
    }
    Iterator next = begin;
    ++next;
    for(; next != end; ++begin, ++next) {
        if(less_than(*next, *begin)) {
            return false;
        }
    }
    return true;
}
// == std::is_sorted

```

- Bemerkung: Compiler-Fehlermeldungen bei Template-Code sind oft schwer zu interpretieren,  $\Rightarrow$  Erfahrung nötig aber: Compiler werden darin immer besser, besonders clang-Compiler
- mit Templates kann man noch viel raffinierter Dinge machen, zum Beispiel Traits-Klassen, intelligent libraries template meta programming  $\Rightarrow$  nur für Fortgeschrittene

## 15 Effizienz von Algorithmen und Datenstrukturen

### 15.1 Bestimmung der Effizienz

2 Möglichkeiten:

1. Messe die "wall clock time" - wie lange muss man auf das Ergebnis warten
2. unabhängig von Hardware benutzt man das Konzept der algorithmischen Komplexität

#### 15.1.1 wall clock

wall clock time misst man zum Beispiel mit dem Modul <chrono>

```

#include <chrono>
#include <iostream>

int main() {
    // alles zur Zeitmessung vorbereiten

```

```

    auto start = std::chrono::high_resolution_clock::now(); // Startzeit
    // code der gemessen werden soll
    auto stop = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> diff = stop - start; // Zeitdifferenz
    std::cout << "Zeitdauer: " << diff.count() << " Sekunden\n" << std::endl; // ausgeben
}

```

Pitfalls:

- moderne Compiler optimieren oft zu gut, das heißt komplexe Berechnungen werden zur Compilezeit ausgeführt und ersetzt  $\implies$  gemessene Zeit ist viel zu kurz.  
Abhilfen:
  - Daten nicht "hard-wired", sondern zum Beispiel von Platte lesen
  - "volatile" Schlüsselwort "volatile int k = 3;"
- der Algorithmus ist schneller als die clock  $\implies$  rufe den Algorithmus mehrmals in einer Schleife auf
- die Ausführung ihres Programms kann vom Betriebssystem jederzeit für etwas wichtigeres unterbrochen werden (zum Beispiel Mail checken)  $\implies$  gemessene Zeit zu lang  $\implies$  messe mehrmals und nimm die kürzeste Zeit (meist reicht 3 bis 10 fach)
- Faustregel: Messung zwischen 0.02s und 3s

Nachteil: Zeit hängt von der Qualität der Implementation, den Daten (insbesondere der Menge) und der Hardware ab

### 15.1.2 algorithmische Komplexität

Algorithmische Komplexität ist davon unabhängig, ist eine Art theoretisches Effizienzmaß. Sie beschreibt, wie sich die Laufzeit verlängert, wenn man mehr Daten hat.

*Beispiel 1.* Algorithmus braucht für  $n$  Elemente  $x$  Sekunden, wie lange dauert es für  $2n$ ,  $10n$  für große  $n$

Bei effiziente Algorithmen steigt der Aufwand mit  $n$  nur langsam (oder bestenfalls gar nicht)

Grundidee:

1. berechne, wie viele elementare Schritte der Algorithmus in Abhängigkeit von  $n$  benötigt  $\implies$  komplizierte Formel  $f(n)$
2. vereinfache  $f(n)$  in eine einfache Formel  $g(n)$ , die dasselbe wesentliche Verhalten hat. Die Vereinfachung erfolgt mittels **O-Notation** und ihren Verwandten  
Gegeben:  $f(n)$  und  $g(n)$

a)  $g(n)$  ist eine asymptotische (für große  $n$ ) obere Schranke für  $f(n)$  (" $f(n) \leq g(n)$ "),  $f(n) \in O(g(n))$  " $f(n)$  ist in der Komplexitätsklasse  $g(n)$ ", wenn es ein  $n_0$  (Mindestgröße) gibt und  $C$  (Konstante) gibt, sodass  $\forall n > n_0 : f(n) \leq Cg(n) \iff f(n) \in O(g(n))$

b)  $g(n)$  ist asymptotische untere Schranke für  $f(n)$  ( $f(n) \geq g(n)$ )

$$f(n) \in \Omega(g(n)) \iff \exists n_0, C : \forall n > n_0 f(n) \geq Cg(n)$$

c)  $g(n)$  ist asymptotisch scharfe Schranke für  $f(n)$  ( $f(n) = g(n)$ )

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

Regeln:

1.  $f(n) \in \Theta(f(n)) \implies f(n) \in O(f(n)), f(n) \in \Omega(f(n))$
2.  $c'f(n) \in \Theta(f(n))$
3.  $O(f(n)) \cdot O(g(n)) \in O(f(n)g(n))$  Multiplikationsregel
4.  $O(f(n)) + O(g(n)) \in O(\text{"max"}(f(n), g(n)))$  Additionsregel  
 formal: wenn  $f(n) \in O(g(n)) \implies O(f(n)) + O(g(n)) \in O(g(n))$   
 $g(n) \in O(f(n)) \implies O(f(n)) + O(g(n)) \in O(f(n))$
5.  $n^p \in O(n^q)$  wenn  $p \leq q$

Beliebte Wahl für  $g(n)$

- $O(1)$  "konstante Komplexität"  
elementare Operation "+, -, \*, /", Array-Zugriff  $v[k]$  ( $v$ : `std::vector`)
- $O(\log(n))$  "logarithmische Komplexität"  
zum Beispiel: auf Element von `std::map` zugreifen  $m[k]$  ( $m$ : `std::map`)
- $O(n)$  "lineare Komplexität"  
zum Beispiel `std::transform()` ( $n$  = Anzahl der transformierten Elemente)
- $O(n \log(n))$  "n log n", "log linear" "linearithmisch"  
Beispiel: `std::sort`
- $O(n^2)$  "quadratische Komplexität"
- $O(n^p)$  "polynomielle Komplexität"
- $O(2^n)$  "exponentielle Komplexität"

Beispiel 2.

$$f(n) = 1 + 15n + 4n^2 + 7n^3 \in O(n^3)$$



### 15.1.3 Anwendung

1. Fibonacci-Zahlen:  $f_k = f_{k-2} + f_{k-1}$

|       |   |   |   |   |   |   |   |    |    |
|-------|---|---|---|---|---|---|---|----|----|
| k     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  |
| $f_k$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |

```
int fib1(int k) {
    if(k < 2) { // O(1)
        return k; // O(1)
    }
    // O(1)
    int f1 = 0; // letzten beiden Fibonacci Zahlen, anfangs die ersten beiden
    int f2 = 1;
    for(int i = 2; i <= k; i++) { // f(k) = k - 1 e O(k)
        int f = f1 + f2; // O(1)
        f1 = f2; // O(1)
        f2 = f; // O(1)
    } // gesamte Schleife: O(1)*O(k) = O(k)
    return f2;
} // gesamte Funktion: teuerstes gewinnt: O(k)

// rekursive Variante:
int fib2(int k) {
    if(k < 2) { // O(1)
        return k; // O(1)
    }
    return fib2(k - 2) + fib2(k - 1);
}
```

- sehr ineffizient, weil alle Fibonacci-Zahlen  $< k$  mehrmals berechnet werden

Sei  $f(k)$  die Anzahl der Schritte, Annahme: jeder Knoten ist  $O(1) \implies f(k) \in O(\text{Anzahl Knoten})$