

Einführung in die Anwendungsorientierte Informatik (Köthe)

Robin Heinemann

14. Januar 2017

Inhaltsverzeichnis

1	Was ist Informatik?	3
1.1	Teilgebiete	3
1.1.1	theoretische Informatik (ITH)	3
1.1.2	technische Informatik (ITE)	3
1.1.3	praktische Informatik	4
1.1.4	angewandte Informatik	4
2	Wie unterscheidet sich Informatik von anderen Disziplinen?	4
2.1	Mathematik	4
3	Informatik	5
3.1	Algorithmus	5
3.2	Daten	5
3.2.1	Beispiele für Symbole	5
3.3	Einfachster Computer	6
3.3.1	TODO Graphische Darstellung	6
3.3.2	TODO Darstellung durch Übergangstabellen	6
3.3.3	Beispiel 2:	7
4	Substitutionsmodell (funktionale Programmierung)	8
4.1	Substitutionsmodell	9
4.2	Bäume	10
4.2.1	Beispiel	10
4.3	Rekursion	11
4.4	Präfixnotation aus dem Baum rekonstruieren	11
4.5	Präfixnotation aus dem Baum rekonstruieren	11
4.6	Berechnen des Werts mit Substitutionsmodell	12

5	Maschinensprachen	12
5.1	Umwandlung in Maschinensprache	12
6	Funktionale Programmierung	13
6.1	Beispiel	13
6.2	Vorteile von Zwischenergebnissen	13
6.3	Funktionale Programmierung in c++	14
7	Prozedurale Programmierung	16
7.1	Von der Funktionalen zur prozeduralen Programmierung	16
7.2	Kennzeichen	17
7.2.1	Prozeduren	17
7.2.2	Steuerung des Programmablaufs	18
7.2.3	Veränderung von Speicherzellen	18
7.2.4	Schleifen	19
7.2.5	prozedurale Wurzelberechnung	20
7.2.6	for-Schleife	21
8	Datentypen	22
8.1	Basistypen	22
8.2	zusammengesetzte Typen	22
8.3	Zeichenketten-Strings:	23
9	Umgebungsmodell	25
10	Referenzen	27
11	Container-Datentypen	29
11.1	std::vector	30
11.1.1	Effizienz von push_back	32
12	Iteratoren	34
13	Insertion Sort	37
14	generische Programmierung	38
15	Effizienz von Algorithmen und Datenstrukturen	42
15.1	Bestimmung der Effizienz	42
15.1.1	wall clock	42
15.1.2	algorithmische Komplexität	43
15.1.3	Anwendung	44
16	Zahlendarstellung im Computer	45
16.1	Natürliche Zahlen	45
16.1.1	Pitfalls	46

16.1.2 arithmetische Operationen	46
16.2 Ganze Zahlen	49
16.3 reelle Zahlen	50
17 Buchstabenzeichen	52
17.1 Geschichte	52
18 Eigene Datentypen	53
19 Objektorientierte Programmierung	54
19.1 Destruktoren	60
19.2 Kopier-Konstruktor	60
19.3 Standard-Konstruktor	61
19.4 rule of three	61
19.5 Vorteile der Kapselung:	61
19.6 Arithmetische Infix-Operationen	62
19.7 Objekte nachträglich verändern	63
19.8 Klasse Image	65

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

\implies Algorithmen transformieren nach bestimmten Regeln die Eingangsdaten (gegebene Symbole) in Ausgangsdaten (Symbole für das Ergebnis). Die Bedeutung / Interpretation der Symbole ist dem Algorithmus egal \triangleq „schematisch“

3.2.1 Beispiele für Symbole

- Zahlen
- Buchstaben
- Icons
- Verkehrszeichen

aber: heutige Computer verstehen nur Binärzahlen \implies 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	Timeout(Variante)
aus	(\implies {halb} \ {4 LEDs an})	%	(\implies {aus},{nichts})
halb	(\implies {voll},{8 LEDs an})	%	(\implies {aus},{nichts})
voll	(\implies {blinken an},{Timer starten})	%	(\implies {aus},{nichts})

Fortsetzung nächste Seite

Fortsetzung von vorheriger Seite

Zustände \ Ereignisse	Knopf drücken	Timeout	Timeout(Variante)
blinken an	(\Rightarrow {aus},{Alle LEDs aus, Timer stoppen})	(\Rightarrow {blinken aus},{alle LEDs aus, Timer starten})	(\Rightarrow {blinken aus},{8 LEDs aus})
blinken aus	(\Rightarrow {aus},{Alle LEDs aus, Timer stoppen})	(\Rightarrow {blinken an},{alle LEDs an, Timer starten})	(\Rightarrow {blinken an},{8 LEDs an})

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

3.3.3 Beispiel 2:

$$\begin{array}{rcl}
 1011010 & = 2 + 8 + 16 + 74 = 90_{\text{dez}} & (1) \\
 +0111001 & = 1 + 8 + 16 + 32 = 57_{\text{dez}} & (2) \\
 \hline
 10010011 & = 1 + 2 + 16 + 128 = 147_{\text{dez}} \checkmark & (3)
 \end{array}$$

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

- 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 der 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 Klammern 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 Klammern wieder einfügen

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

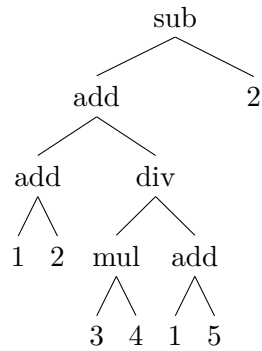
$$\begin{aligned}
 &1 + 2 + 3 * 4 / (1 + 5) - 2 \\
 &(((1 + 2) + ((3 * 4) / (1 + 5))) - 2) \\
 &sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2) \\
 &sub(add(3, div(12, 6)), 2) \\
 &sub(add(3, 2), 2) \\
 &sub(5, 2) \\
 &2
 \end{aligned}$$

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

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



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: $SpZ \leftarrow \text{Wert}$
3. Rechenoperationen in Reihenfolge des Substitutionsmodell ausführen und in der jeweiligen Speicherzelle speichern
 - Notation: $SpZ\text{-Ergebnis} \leftarrow \text{fname } SpZ\text{Arg1 } SpZ\text{Arg2}$
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
 \implies Definition eigener Funktionen

```

1 function sq(x) {
2     return x * x
3 }

```

$\Rightarrow d \leftarrow \text{sqrt}(\text{sq}(p) - q)$ 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;“

```

1 double a = ...;
2 double b = ...;
3 double c = ...;
4 double p = -0.5 b / a;
5 double q = c / a;
6 double d = std::sqrt(p*p - q);
7 double x1 = p + d;
8 double x2 = p - d;
9 std::cout << "x1: " << x1 << ", x2: " << x2 << std::endl;

```

- eigene Funktionen in C++

```

1 // Kommentar (auch /* */)
2 type_ergebnis fname(type_arg1 name1, ...) {
3     // Signatur / Funktionskopf / Deklaration
4     return ergebnis;
5     /* Funktionskörper / Definition / Implementation */
6 }

```

- ganze Zahl quadrieren:

```

1 int sq(int x) {
2     return x*x;
3 }

```

- reelle Zahl quadrieren:

```

1 double sq(double x) {
2     return x*x;
3 }

```

- beide Varianten dürfen in c++ gleichzeitig definiert sein \implies „function overloading“ \implies c++ wählt automatisch die richtige Variable anhand des Argumenttyps („overload resolution“)

```

1 int x = 2;
2 double y = 1.1
3 int x2 = sq(x) // int Variante
4 double y2 = sq(y) // double Variante

```

- jedes c++-Programm muss genau eine Funktion namens „main“ haben. Dort beginnt die Programmausführung.

```

1 int main() {
2     Code;
3     return 0;
4 }

```

- 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` \Rightarrow komplexe Zahlen
- `string` \Rightarrow Zeichenkettenverarbeitung
- um ein Modul zu benutzen muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren (Header inkludieren) \rightarrow `include <name>`

```

1  #include <iostream>
2  #include <string>
3  int main() {
4      std::cout << "Hello, world!" << std::endl;
5      std::string out = "mein erstes Programm\n";
6      std::cout << out;
7      return 0;
8  }

```

- overloading der arithmetischen Operationen
 - 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 Division 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 $\Rightarrow -12 / 4 = -2$

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

```

1 std::cout << "Hello\n"; // Infix
2 operator<<(std::cout, "Hello\n"; // Präfix

```

- Prozeduren in c++

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

```

1 void foo(int x) {
2     return;
3 }

```

2. Returnwert ignorieren

7.2.2 Steuerung des Programmablaufs

- Anweisungen zur Steuerung des Programmablaufs

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

- Funktional

```
1 int abs(int x) {  
2     return (x >= 0) ? x : -x;  
3 }
```

- Prozedural

```
1 int abs(int x) {  
2     if(x >= 0) {  
3         return x;  
4     } else {  
5         return -x;  
6     }  
7  
8     // oder  
9     if(x >= 0) return x;  
10    return -x;  
11 }
```

7.2.3 Veränderung von Speicherzellen

- Zuweisung: Speicherzellen können nachträglich verändert werden („read-write“)
 - prozedural:

```
1 int foo(int x) {    // x = 3  
2     int y = 2;  
3     int z1 = x * y; // z1 = 6  
4     y = 5;  
5     int z2 = z * y; // z2 = 15  
6     return z1 + z2; // 21  
7 }
```

- funktional:

```
1 int foo(int x) {    // x = 3  
2     int y1 = 2;  
3     int z1 = x * y1; // z1 = 6
```

```

4     int y2 = 5;
5     int z2 = z1 * y2; // z2 = 15
6     return z1 + z2;   // 21
7 }

```

- Syntax

```

1 name = neuer_wert;           // Zuweisung
2 typ name = neuer_wert;      // Initialisierung
3 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

```

1 while(Bedingung) {
2     // Code, wird ausgeführt solange Bedingung "true"
3 }

```

```

1 int counter = 0;
2 while(counter < 3) {
3     std::cout << counter << std::endl;
4     counter++; // Kurzform für counter = counter + 1
5 }

```

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 Inkrementieren \Rightarrow Bedingung immer „true“ \Rightarrow Endlosschleife \Rightarrow 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

```
1 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:

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

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:

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

```
1 double sqrt(double y) {
2     if(y < 0.0) {
3         std::cout << "Wurzel aus negativer Zahl\n";
4         return -1.0;
5     }
```

```

6     if(y == 0.0) return 0.0;
7
8     double x = y; // initial guess
9     double epsilon = 1e-15 * y;
10
11     while(abs(x * x - y) > epsilon) {
12         x = 0.5*(x + y / x);
13     }
14 }

```

7.2.6 for-Schleife

```

1 int c = 0;
2 while(c < 3) {
3     // unser Code
4     c++; // vergisst man leicht
5 }

```

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

Äquivalent zu der while Schleife oben ist:

```

1 for(int c = 0; c < 3; c++) {
2     // unser Code
3 }

```

Allgemeine Form:

```

1 for(init; Bedingung; Inkrement) {
2     // unser Code
3 }

```

- 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

```

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

```

Variante mit continue:

```
1 for(int i = 0; i < 10; i++) {  
2     if(c % 2 != 0) continue;  
3     std::cout << c << std::endl;  
4 }
```

```
1 for(int i = 0; i < 10; i += 2) {  
2     std::cout << c << std::endl;  
3 }
```

```
1 double sqrt(double y) {  
2     while(true) {  
3         x = (x + y / 2) / 2.0;  
4         if(abs(x * x - y) < epsilon) {  
5             return x;  
6         }  
7     }  
8 }
```

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:

```
1  std::string s = "abcde";
2  std::string s2 = s1;
3  std::string leer = "";
4  std::string leer(); // Abkürzung, default Konstruktor
```

- Länge

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

- Zuweisung

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

- Addition Aneinanderkettung von String („concatenate“)

```
1  std::string s3 = s + "ijh"; // "xyijh"
2  s3 = "ghi" + s; // "ghixy"
```

```

3 s3 = s + s; // "xyxy"
4 // aber nicht!!
5 s3 = "abc" + "def"; // Bug Literale unterstützen + mit ganz
  ↳ anderer Bedeutung
6 s3 = std::string("abc") + "def"; // Ok

```

- Add-Assignment: Abkürzung für Addition gefolgt von Zuweisung

```

1 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:

```

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

```

Anwendung: jedes Zeichen einzeln ausgeben

```

1 std::string s = "abcde";
2
3 for(int i = 0; i < s.size(); i++) {
4     std::cout << s[i] << std::endl;
5 }

```

String umkehren

```

1 int i = 0; // Anfang des Strings
2 int k = s.size() - 1; // Ende des String
3 while(i < k) {
4     char tmp = s[i];
5     s[i] = s[k];
6     s[k] = tmp;
7     i++; k--;
8 }

```

Variante 2: neuen String erzeugen

```

1 std::string s = "abcde";
2 std::string r = "";
3 for(int i = s.size() - 1; i >= 0; i--) {

```



```
4      r += s[i];  
5  }
```

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

```
1 - int global = 1;  
2   int main() {  
3       int l = 2;  
4       {  
5           int m = 3  
6       } // <- m wird freigegeben  
7   } // <- l wird freigegeben  
8   // <- 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:

```

1 namespace std {
2 double sqrt(double x);
3 namespace chrono {
4 class system_clock;
5 }
6 }
7
8 // Benutzung mit Namespace-Präfix:
9 std::sqrt(80);
10 std::chrono::system_clock clock;

```

Besonderheit: mehrere Blöcke mit selbem Namensraum werden verschmolzen Beispiel

```

1 int p = 2;
2 int q = 3;
3
4 int foo(int p) {
5     return p * q;
6 }
7
8 int main() {
9     int k = p * q; // beides global => 6 = 2 * 3
10    int p = 4; // lokales p verdeckt globales p
11    int r = p * q; // p lokal, q global => 12 = 4 * 3
12    int s = foo(p); // lokale p von main() wird zum lokalen p von
    ↪ foo() 12 = 4 * 3
13    int t = foo(q); // globales q wird zum lokalen p von foo() 9 = 3
    ↪ * 3
14    int q = 5;
15    int n = foo(q); // lokales q wird zum lokalen p von foo() 15 = 5
    ↪ * 3
16 }

```

10 Referenzen

sind neue (zusätzliche) Namen für vorhandene Speicherzellen

```

1 int x = 3; // neue Variable x mit neuer Speicherzelle
2 int & y = x; // Referenz: y ist neuer Name für x, beide haben die selbe
    ↪ Speicherzelle
3 y = 4; // Zuweisung an y, aber x ändert sich auch, das heißt x == 4
4 x = 5; // jetzt y == 5

```

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

```
1 int foo(int x) { // pass-by-value
2     x += 3;
3     return x;
4 }
5
6 int bar(int & y) { // pass-by-reference
7     y += 3; // Seiteneffekt der Funktion
8     return y;
9 }
10
11 void baz(int & z) { // pass-by-reference
12     z += 3;
13 }
14
15 int main() {
16     int a = 3;
17     std::cout << foo(a) << std::endl; // 5
18     std::cout << a << std::endl; // 3
19     std::cout << bar(a) << std::endl; // 5
20     std::cout << a << std::endl; // 5
21     baz(a);
22     std::cout << a << std::endl; // 8
23 }
```

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)

```
1  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:

```
1  #include <vector>
2
3  //          Elementtyp    Größe  Initialwert der Elemente
4  std::vector<double    > v(20    ,          0.0          );
5  // analog
6  std::vector<int>;
7  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

```

1  #include <map>
2  #include <unordered_map>
3
4  // Binärer Suchbaum
5  std::map;
6
7  // Hashtabelle, siehe Algorithmen und Datenstrukturen
8  std::unordered_map;
9
10 //      Schlüsseltyp  Elementtyp
11 std::map<int      , double> noten; noten[3121101] = 10;
12 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:

```

1  std::vector<double> v(20, 1.0);
2  std::vector<double> v; // leeres Array
3  std::vector<double> v = {1.0, -3.0, 2.2}; // "initializer list":
    ↪ Element für Anfangszustand

```

- Größe:

```

1  v.size();
2  v.empty(); // => v.size() == 0

```

- Größe ändern

```

1  v.resize(neue_groesse, initialwert);
2  // Dann:
3  // Fall 1: neue_groesse < size(): Element ab Index "neue_groesse"
    ↪ gelöscht die andern bleiben
4  // Fall 2: neue_groesse > size(): neue Elemente mit Initialwert am
    ↪ Ende anhängen, die anderen bleiben
5  // Fall 3: neue_groesse == size(): nichts passiert
6

```

```

7  v.push_back(neues_element); // ein neues Element am Ende anhängen
   ↪ (ähnlich string += "mehr")
8  v.insert(v.begin() + index, neues_element); // neues element an
   ↪ Position "index" einfügen 0 <= index <= size()
9  // Falls index == size(): am Ende anhängen, sonst: alte Elemente ab
   ↪ Index werden eine Position nach hinten verschoben (teuer)
10
11 v.pop_back(); // letztes Element löschen (effizient)
12 v.erase(v.begin() + index); // Element an Position index löschen,
   ↪ alles dahinter eine Position verschieben (teuer)
13 v.clear(); // alles löschen

```

- Zugriff

```

1  v[k]; // Element bei Index k
2  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

```

1  // erstes Element
2  v.begin()
3
4  // hinter letztem Element
5  v.end()

```

- im Header <algorithm>

- alle Elemente kopieren

```

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

```

- Elemente sortieren

```

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

```

- Elemente mischen:

```
1  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
 - a) alloziere neues statisches Array mit doppelt Kapazität
 - b) kopiere die Daten aus dem alten in das neue Array
 - c) gebe das alte Array frei
 - d) gehe zum Anfang des Algorithmus, jetzt ist wieder Reserve vorhanden
- das Umkopieren ist nicht zu teuer, weil es nur selten notwendig ist
- Beispiel:

```
1  std::vector<int> v;  
2  
3  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 allem 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`

```

1 v.size() // aktuelle Zahl der Elemente
2 v.capacity() // aktuelle Zahl Speicherzellen
3 assert(v.capacity() - v.size() >= 0) // Reserve
4 v.resize(new_size) // ändert immer v.size(), aber v.capacity() nur
  ↪ wenn < new_size
5 v.reserve(new_capacity) // ändert v.size() nicht, aber v.capacity()
  ↪ falls new_capacity >= size
6 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 (FI-FO)
- `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

```

1  for(int i = 0; i != v.size(); i++) {
2      int current = v[i]; // lesen
3      v[i] = new_value; // schreiben
4  }
```

- wir wollen eine so einfache Schleife für beliebige Container
 - der Index-Zugriff `v[]` 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:

```

1  for(auto iter = v.begin(); iter != v.end(); ++iter) {
2      int current = *iter; // lesen
3      *iter = new_value; // schreiben
4  }
5  // Abkürzung: range-based for loop
```

```

6  for(auto & element : v) {
7  int current = element; // lesen
8  element = new_value; // schreiben
9  }

```

- 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

```

1  (*iter).first; // Schlüssel
2  (*iter).second; // Wert
3  // Abkürzung
4  iter->first;
5  iter->second;

```

- bei std::map liefern die Iteratoren die Elemente in aufsteigender Reihenfolge der Schlüssel
- Die Funktion std::transform()
 - wir hatten: std::copy()

```

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

```

- std::transform:

```

1  // nach Kleinbuchstaben konvertieren
2  std::string source = "aAbCdE";
3  std::string target = source;
4  std::transform(source.begin(), source.end(), target.begin(),
   ↪ std::tolower); // Name einer Funktion, die ein einzelnes
   ↪ Element transformiert, t="aabcde"
5  // die Daten quadrieren
6  double sq(double x) { return x * x; }
7  std::transform(source.begin(), source.end(), target.begin(),
   ↪ sq); // target == {1, 4, 9, 16}

```

```

8 // das ist eine Abkürzung für eine Schleife
9 auto src_begin = source.begin();
10 auto src_end = source.end();
11 auto tgt_begin = target.begin();
12
13 for(; src_begin != src_end; src_begin++, tgt_begin++) {
14     *tgt_begin = sq(*src_begin);
15 }

```

- 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 \Rightarrow für Fortgeschrittene („functionpointer cast“)
 2. Funktionsobjekt \Rightarrow objekt-orientierte Programmierung
 3. Definiere eine namenlose Funktion \Rightarrow „Lambda-Funktion λ “
 - statt λ verwenden wir den Universalnamen `[]`

```

1 std::transform(source.begin(), source.end(),
  ↪ target.begin(), [](double x) { return x*x; }); //
  ↪ statt Funktionsname sq wie bei 1 steht hier die
  ↪ ganz Funktionsimplementation
2 // Returntyp setzt Computer automatisch ein, wenn es
  ↪ nur einen return-Befehl gibt.

```

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

```

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

```

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

```

1 std::vector<double> v = {4, 2, 3, 5, 1};
2 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

```

1 void insertion_sort(std::vector<double> & v) {
2     for(int i = 0; i < v.size(); i++) {
3         double current = v[i];
4         int j = i; // Anfangsposition der Lücke
5         while(j > 0) {
6             if(v[j - 1] < current) { // -> if(cmp(a, b))
7                 break; // j ist richtige Position der Lücke
8             }
9             v[j] = v[j - 1];

```

```

10         j--;
11     }
12     v[j] = current;
13 }
14 }

```

- andere Sortierung: definiere Funktor 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

```

1  std::sort(v.begin(), v.end()); // Standardsort mit "<"
2  std::sort(v.begin(), v.end(), [](double a, double b) { return a
   ↪ < b; }); // Standardsortierung aufsteigen
3  std::sort(v.begin(), v.end(), [](double a, double b) { return b
   ↪ < a; }); // absteigende Sortierung
4  std::sort(v.begin(), v.end(), [](double a, double b) { return
   ↪ std::abs(a) < std::abs(b); }); // Normal nach Betrag; //
   ↪ Normal nach Betrag
5  std::sort(v.begin(), v.end(), [](std::string a, std::string b) {
6      std::transform(a.begin(), a.end(), a.begin(), std::tolower);
7      std::transform(b.begin(), b.end(), b.begin(), std::tolower);
8      return a < b;
9  });

```

14 generische Programmierung

insertion_sort soll für beliebige Elementtypen funktionieren

```

1  template<typename T>
2  void insertion_sort(std::vector<T> & v) {
3      for(int i = 0; i < v.size(); i++) {
4          T current = v[i];
5          int j = i; // Anfangsposition der Lücke
6          while(j > 0) {
7              if(v[j - 1] < current) { // -> if(cmp(a, b))
8                  break; // j ist richtige Position der Lücke
9              }
10             v[j] = v[j - 1];
11             j--;
12         }
13         v[j] = current;

```

```
14     }
15 }
```

- 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 \Rightarrow 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

```
1 int sq(int x) {
2     return x * x;
3 }
4
5 double sq(double x) {
6     return x * x;
7 }
8
9 // und so weiter für komplexe und rationale Zahlen...
```

- Nachteil
 - wenn die Implementationen gleich sind \rightarrow nutzlose Arbeit
 - Redundanz ist gefährlich: korrigiert man einen Bug wird leicht eine Variante vergessen
- mit templates reicht eine Implementation

```
1 template<typename T> // T: Platzhalter für beliebigen Typ,
   ↪ wird später durch einen tatsächlichen Typ ersetzt
2 T sq(T x) {
3     return x * x; // implizierte Anforderung an den Typ T,
   ↪ er muss Multiplikation unterstützen, sonst:
   ↪ Fehlermeldung
4 }
```

- wie bei Substituieren von Variablen mit Werten, aber jetzt mit Typen

- Benutzung:
 - Typen für die Platzhalter hinter dem Funktionsnamen in spitzen klammern

```

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

```

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

```

1 sq(2); // == sq<int>(2) == 4
2 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) \implies für Fortgeschrittene
- Beispiel 2: Funktion, die ein Array auf Konsole ausgibt, für beliebige Elementtypen

```

1 template<typename ElementType>
2 void print_vector(std::vector<ElementType> const & v) {
3     std::cout << "{";
4     if(v.size() > 0) {
5         std::cout << " " << v[0];
6         for(int i = 1; i < v.size(); i++) {
7             std::cout << ", " << v[i];
8         }
9     }
10    std::cout << " }";
11 }

```

- Verallgemeinerung für beliebige Container mittels Iteratoren:

```

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

```

- es genügen forward_iterators

```

1 Iterator iter2 = iter1; // Kopie erzeugen
2 iter1++; // zum nächsten Element
3 iter1 == iter2; // Zeigen sie auf das selbe Element?
4 iter1 != end;

```

```

5  *iter1; // Zugriff auf aktuelles Element
6
7  template<typename Iterator>
8  void print_container(Iterator begin, Iterator end) {
9      std::cout << "{}";
10     if(begin != end) { // Container nicht leer?
11         std::cout << " " << *begin++;
12         for(;begin != end; begin++) {
13             std::cout << ", " << *begin;
14         }
15         std::cout << "}";
16     }

```

· Beispiel 3: überprüfen, ob Container sortiert ist

```

1  template<typename E, typename CMP>
2  bool check_sorted(std::vector<E> const & v, CMP
   ↪ less_than) {
3      for(int i = 1; i < v.size(); i++) {
4          if(less_than(v[i], v[i - 1])) { // statt v[i] <
   ↪ v[i - 1], ausnutzen der Transitivität
5              return false;
6          }
7      }
8      return true;
9  }
10
11 // Aufruf:
12 std::vector<double> v = {1.0, 2.0, 3.0};
13 check_sorted(v, [](double a, double b) { return a < b; }
   ↪ ); // == true
14
15 check_sorted(v, [](double a, double b) { return a > b; }
   ↪ ); // == false
16
17 // Implementation für Iteratoren
18 template<typename Iterator, typename CMP>
19 bool check_sorted(Iterator begin, Iterator end, CMP
   ↪ less_than) {
20     if(begin == end) {
21         return true;
22     }
23     Iterator next = begin;
24     ++next;

```

```

25     for(; next != end; ++begin, ++next) {
26         if(less_than(*next, *begin)) {
27             return false;
28         }
29     }
30     return true;
31 }
32 // == 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>

```

1  #include <chrono>
2  #include <iostream>
3
4  int main() {
5      // alles zur Zeitmessung vorbereiten
6
7      auto start = std::chrono::high_resolution_clock::now(); // Startzeit
8      // code der gemessen werden soll
9      auto stop = std::chrono::high_resolution_clock::now();
10     std::chrono::duration<double> diff = stop - start; // Zeitdifferenz
11     std::cout << "Zeitdauer: " << diff.count() << " Sekunden\n" <<
        ↪ std::endl; // ausgeben
12 }

```

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 0.02 s und 3 s

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 **\$\$\$-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

```

1  int fib1(int k) {
2      if(k < 2) { // O(1)
3          return k; // O(1)
4      }
5      // O(1)
6      int f1 = 0; // letzten beiden Fibonacci Zahlen, anfangs die
           ↪ ersten beiden
7      int f2 = 1;
8      for(int i = 2; i <= k; i++) { // f(k) = k - 1 e O(k)
9          int f = f1 + f2; // O(1)
10         f1 = f2; // O(1)
11         f2 = f; // O(1)
12     } // gesamte Schleife: O(1)*O(k) = O(k)
13     return f2;
14 } // gesamte Funktion: teuerstes gewinnt: O(k)
15
16 // rekursive Variante:
17 int fib2(int k) {
18     if(k < 2) { // O(1)
19         return k; // O(1)
20     }
21     return fib2(k - 2) + fib2(k - 1);
22 }

```

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

Sei $f(k)$ die Anzahl der Schritte, $f'(k)$ die Schritte oberhalb, Annahme: jeder Knoten ist $O(1) \implies f(k) \in O(\text{Anzahl Knoten})$. Oberhalb ist der Baum vollständig (jeder innere Knoten hat zwei Kinder), Anzahl der Knoten im vollständigen Baum der Tiefe l :

$$1 + 2 + 4 + \dots + 2^l = 2^{l+1} - 1$$

16 Zahlendarstellung im Computer

Problem: es gibt ∞ viele Zahlen, aber der Computer ist endlich.

16.1 Natürliche Zahlen

Natürliche Zahlen $\implies x \geq 0$. c++ bietet Typen verschiedener Größe.

klassisch	mit Größe	Anzahl Bits	Bereich	Literale
unsigned char	uint8_t	(\geq) 8	0 - 255	
unsigned short	uint16_t	(\geq) 16	0 - 65535	
unsigned int	uint32_t	(\geq) 32	0 - 4×10^9	
unsigned long		32 oder 64		
unsigned long long	uint64_t	64	0 - 2×10^{19}	L

was passiert bei zu großen Zahlen?

- alle Operationen werden Modulo 2^m ausgeführt, wenn der Typ m Bits hat

```

1  uint8_t x = 250, y = 100;
2  uint8_t s = x + y; // 350 % 256 = 94
3  uint8_t p = x * y; // 25000 % 256 = 168

```

„integer overflow“: einfach Bits oberhalb von m wegwerfen

16.1.1 Pitfalls

```

1  std::vector<uint8_t> v = { ... };
2  uint8_t sum = 0; // FALSCH, da es zu overflow kommen kann
3  // verwende uint32_t, uint64_t, verhindern overflow mit hoher
   ↪ Wahrscheinlichkeit
4  for(int k = 0; k < v.size(); k++) {
5      sum += v[k];
6  }
7
8  // Endlosschleife, da i nie < 0, da unsigned
9  // Abhilfe: int verwenden
10 for(uint8_t i = v.size(); i >= 0; i++) {
11     // auf v[k] zugreifen
12 }

```

16.1.2 arithmetische Operationen

- Addition in Kapitel Automaten

Subtraktion Subtraktion kann auf Addition zurückgeführt werden Erinnerung: Restklassenarithmetik: (Modulo)

alle Zahlen mit dem gleichen Rest modulo k bilden „Äquivalenzklasse“, zum Beispiel

$$k = 4$$

$$\begin{array}{llll} 0 & \bmod 4 = 0 \equiv 4 & \bmod 4 \equiv 8 & \bmod 4 \equiv 12 \bmod 4 \dots \\ 1 & \bmod 4 = 1 \equiv 5 & \bmod 4 \equiv 9 & \bmod 4 \equiv 13 \bmod 4 \dots \\ 2 & \bmod 4 = 2 \equiv 6 & \bmod 4 \equiv 10 & \bmod 4 \equiv 14 \bmod 4 \dots \\ 3 & \bmod 4 = 3 \equiv 7 & \bmod 4 \equiv 11 & \bmod 4 \equiv 15 \bmod 4 \dots \end{array}$$

Ein Mitglied jeder Äquivalenzklasse wird Repräsentant.

Hier: kleinste Repräsentanten $0, \dots, (k-1)$, mit $k = 2^m$ sind das gerade die uint-Werte
Eigenschaft: man kann Vielfache nk addieren, ohne Äquivalenzklasse zu ändern:

$$(a-b) \bmod 2^m = \left(a + \underbrace{2^m - b}_{\text{x: Zweierkomplement}} \right) \bmod 2^m = (a+z) \bmod 2^m$$

$z = (2^m - b) \bmod 2^m$ lässt sich billig berechnen als $(\sim b + 1) \bmod 2^m$ Dabei ist \sim bitweise Negation (dreht alle Bits um)

$$\cdot m = 4, \sim(1001) = 0110$$

Satz 1.

$$(2^m - b) \bmod 2^m = (\sim b + 1) \bmod 2^m$$

Beweis.

$$\begin{aligned} b + \sim b &= 1111 \dots 1 = 2^m - 1 \\ \sim b + 1 &= 2^m - b \end{aligned}$$

Fall 1: $b > 0$

$$\begin{aligned} &\implies \sim b < 2^m - 1 \implies \sim b + 1 < 2^m \implies (\sim b + 1) \bmod 2^m = \sim b + 1 \\ &\implies (\sim b + 1) \bmod 2^m = (2^m - b) \bmod 2^m \end{aligned}$$

Fall 2: $b = 0$

$$\begin{aligned} &\implies \sim b = 2^m - 1 \\ &\sim b + 1 = 2^m \\ (\sim b + 1) \bmod 2^m &= 0 \\ 2^m - b &= 2^m \\ z = (2^m - b) \bmod 2^m &= (\sim b + 1) \bmod 2^m = 0 \end{aligned}$$

□

Multiplikation Neue Operationen: \ll und \gg (left und right shift). Verschiebt die Bits um k Positionen nach links oder rechts. Die herausgeschobenen Bits werden vergessen und auf der anderen Seite durch 0-Bits ersetzt.

```

1 // m = 8
2 assert(11011101b << 3 == 11101000b)
3 assert(11011101b >> 3 == 00011011b)

```

Satz 2.

$$x \ll k \equiv (x \cdot 2^k) \mod 2^m$$

$$x \gg k \equiv \left(\frac{x}{2^k}\right)$$

Operation $\&$ und $|$: bitweise und beziehungsweise oder-Verknüpfung (nicht verwechseln mit $\&\&$ und $||$ für logische Operationen) $m = 8$: $10110011 \& 1 =$

1	0	1	1	0	0	1	1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1

(testet, ob in linker Zahl Bit 0 gesetzt ist) $10110011 | 1 =$

1	0	1	1	0	0	1	1
0	0	0	0	0	0	0	1
1	0	1	1	0	0	1	1

kombiniere $\&$ mit \ll :

$$x \& (1 \ll k)$$

testet, ob in x Bit k gesetzt ist.

```

1 uint8_t mul(uint8_t x, uint8_t y) {
2     uint8_t res = 0;
3     for(int i = 0; i < 8; i++) {
4         if(y & (1 << i)) {
5             res += x;
6         }
7         x = x << 1; // x * 2
8     }
9 }

```

16.2 Ganze Zahlen

klassisch	mit Größe	Anzahl Bits	Bereich
signed char	int8_t	8	-128 ... 127
signed short	int16_t	16	$-2^{15} \dots 2^{15} - 1$
signed int	int32_t	32	$-2^{31} \dots 2^{31} - 2$
signed long		32 oder 64	
signed long long	int64_t	64	$-2^{63} \dots 2^{63} - 1$

Wird der erlaubte Bereich überschritten, ist Verhalten Compiler abhängig. In der Praxis: auch Modulo 2^m , aber mit anderen Repräsentanten.

für Restklassen:

statt $0 \dots 2^m$ bei unsigned jetzt $-2^{m-1} \dots 2^{m-2} - 1$

das heißt:

- $x < 2^{m-1}$: Repräsentant bleibt
- $x \geq 2^{m-1}$: neuer Repräsentant $x - 2^m$ ist gleiche Restklasse

Vorteil: $x, -, *$ kann von unsigned übernommen werden

a, b signed: $a \text{ OP } b \rightarrow c$ signed (interpretiere Bitmuster von a und b als unsigned und Interpretiere das Ergebnis dann als signed) Konsequenzen:

- bei negativer Zahl ist höchstes Bit 1, weil $x \rightarrow x - 2^m$ falls $x \geq 2^{m-1}$
- unäre Negation $-x$ durch Zweierkomplement

$$\begin{aligned}
 -x &= (\sim x + 1) \mod 2^m \\
 -0 &= (\sim 00000000 + 1) \mod 2^8 \\
 &= (11111111 + 1) \mod 2^8 \\
 &= \underbrace{(100000000)}_{2^8} \mod 2^8 = 0 - 1 = (\sim 00000001 + 1) \mod 2^8 \\
 &= (11111110 + 1) \mod 2^8 \\
 &= \underbrace{(11111111)}_{2^8 - 1 < 2^8} \mod 2^8 \\
 &= 11111111
 \end{aligned}$$

Ausnahmeregel für \gg bei negativen Zahlen: Compilerabhängig, meist wird links ein 1-Bit reingeschoben, damit Zahl negativ bleibt \implies es gilt immer noch $x \gg k = (x/2^k)$
 Reichen 64 Bit nicht aus (zum Beispiel bei moderner Verschlüsselung) verwende BigInt: Datentyp variabler Größe. Zum Beispiel GNU Multi-Precision Library

16.3 reelle Zahlen

c++ bietet Typen

Name	Größe	Bereich	kleinste Zahl	Literale
float	32bit	$-1 \times 10^{38} - 1 \times 10^{38}$	$10e-38$	4.0f
double	64bit	$-1 \times 10^{308} - 1 \times 10^{308}$	1×10^{-308}	4.0, 1e-2
long double	plattformabhängig, ≥ 64 bit			

Der c++ Standard legt die Größe nicht fest, aber praktisch alle gängigen CPUs benutzen Standard IEEE 754, c++ übernimmt HW-Implementation. Ziele der Definition von reellwertigen Typen:

- hohe Genauigkeit (viele gültige Nachkommastellen)
- Zahlen sehr unterschiedlicher Größenskalen (zum Beispiel Durchmesser eines Protons = 2×10^{-15} m vs. Durchmesser des sichtbaren Universum 1×10^{27} m) mit natürlichen Zahlen bräuchte man > 150 Stellen

elegante Lösung: halb-logarithmische Darstellung („floating point“). Datentyp ist aus 3 natürlichen Zahlen zusammengesetzt (aber alles von der CPU gekapselt)

- S (1-bit): Vorzeichen, 0 = „+“, 1 = „-“
- M (m-bit): Mantisse: Nachkommastellen
- E : (e-bit, Bias b): Exponent: Größenordnung

die eigentliche Zahl wird durch

$$x = (-1)^s \cdot (1 + M \cdot 2^{-m}) \cdot 2^{E-b}$$

- $M \cdot 2^{-m} \in [0, \frac{2^m-1}{2^m}) \in [0, 1)$
- $M \in [0, 2^m - 1]$
- $1 + M \cdot 2^{-m} \in [1, 2)$

Beispiel: natürliche Zahlen

x	$M \cdot 2^{-m}$	$E - b$	effektive Darstellung
1	0	0	$1 \cdot 2^0$
2	0	1	$1 \cdot 2^1$
3	0.5	1	$1.5 \cdot 2^1$
4	0	2	$1 \cdot 2^2$
5	0.25	2	$1.25 \cdot 2^2$

Konsequenz: alle ganzen Zahlen zwischen $-2^m, \dots, 2^m$ können exakt dargestellt werden und haben exakte Arithmetik. (IEEE 754) Werte für m, e, b

- float
 - $m = 23$
 - $e = 8$
 - $b = 127$
 - $2^{E-b} \in [2^{-126}, 2^{127}] \approx [10^{-38}, 10^{38}]$
- double
 - $m = 52$
 - $e = 11$
 - $b = 1024$
 - $2^{E-b} \in [2^{-1022}, 2^{1023}] \approx [10^{-308}, 10^{308}]$

Anzahl der Nachkommastellen: $\varepsilon = 2^{-m}$ (machine epsilon, unit last place (ULP))

- float $2^{-23} \approx 1 \times 10^{-7}$
- double $2^{-52} \approx 1 \times 10^{-16}$

ε ist die kleinste Zahl, sodass

$$(1.0 + \varepsilon) \neq 1.0$$

weil Nachkommastellen außerhalb der Mantisse (rechts von 2^{-m}) ignoriert werden. \implies Problem der Auslöschung von signifikanten Stellen. Wenn man zwei fast gleich große Zahlen subtrahiert, löschen sich fast alle Bits der Mantisse \implies nur wenige gültige Nachkommastellen überleben. Zum Beispiel:

$$0.1234567 - 0.1234566 = 0.000001 \quad (\text{nur eine gültige Nachkommastelle!})$$

$$1.0 - \cos x, x \rightarrow 0, x \approx 0 \implies \cos x \approx 1 \implies \text{Auslöschung}$$

x	Anzahl der gültigen Stellen	Additionstheorem $1 - \cos(x) = 2(\sin(x/2))^2$
0.001	9	15
1×10^{-8}	0 ($\cos(1 \times 10^{-8})$)	15

Quadratische Gleichung:

$$ax^2 + bx + c = 0, b > 0$$

$$x_1 = \frac{1}{2a}(-b + \sqrt{b^2 - 4ac})$$

falls $a \cdot c \wedge b^2 \gg 4ac \implies \sqrt{b^2 - 4ac} \approx b \implies x_1 \approx -b + b + (\varepsilon') \approx 0 \implies$ Auslöschung, wenig gültige Stellen. Also umstellen:

$$\begin{aligned}
 x_1 &= \frac{1}{2a} \frac{b^2 - (b^2 - 4ac)}{-b - \sqrt{b^2 - 4ac}} \\
 &= \frac{2c}{\underbrace{-b - \underbrace{\sqrt{b^2 - 4ac}}_{\approx b}}_{\approx -2b}}
 \end{aligned}$$

\Rightarrow keine Auslösung

Dies tritt auch bei Aufgabe 8.3 der Übungszettel auf

Ausnahmeregeln (spezielle Werte)

· normal: $E \in [1, 2^e - 2]$

· $E = 2^e - 1$ (größtmöglicher Wert):
$$\begin{cases} x = -\infty & M = 0 \wedge S = 1 \\ x = \infty & M = 0 \wedge S = 0 \\ x = \text{NaN} & M = 0 \end{cases}$$

· $\pm\infty$: $\frac{1.0}{0.0}, \frac{-1.0}{0.0}, \dots$

· NaN: $\frac{0.0}{0.0}, \sqrt{-1.0}, \infty \cdot 0$

· $E = 0$ (kleinstmöglicher Wert):
$$\begin{cases} -0 & M = 0 \wedge S = 1 \\ 0 & M = 0 \wedge S = 0 \\ \text{denormalisierte Zahlen (für sehr kleine Werte)} & M > 0 \end{cases}$$

17 Buchstabenzeichen

Buchstabenzeichen: „glyphs“ müssen durch Zahlen repräsentiert werden „Zeichencode“

17.1 Geschichte

1963	ASCII	7-bit	Zeichen der englischen Schreibmaschine (keine Umlaute)
1964 ... 2000		8-bit Codes	mit Umlauten, Akzenten, kyrillische Zeichen, aber 8-bit sind zu wenig, um alles abzudecken \Rightarrow viele verschiedene 8-bit Codes: DOS Codepage 437, Codepage 850 (Konsole im deutschen Windows), Windows-1252 („ANSI“) \approx ISO 8859-1 \approx ISO 8859.15 (Westeuropäische Sprachen)
1991 ... heute	Unicode	anfangs 16-bit, jetzt \approx 21-bit	alles (chinesisch, Hyroglyphen, Emojis, ...)

3 Codierungen für Unicode:

- UTF-8: variable length code: pro glyph 1 ... 4 uint8
- UTF-16: variable length code: pro glyph 1 ... 2 uint16

- UTF-32: fixed length code: pro glyph 1 uint32 pro glyph

In c++:

- char: 8-bit Codes
- wchar_t: 16-bit (Windows), 32-bit (Linux)
- u16char_t, u32char_t:

leider sehr Plattformabhängig

Symbol	DOS	ANSI	UTF-8
ö	148	246	195 182
€	221	128	226 130 172

⇒ ICU Library.

hat man alle Zeichen korrekt, ist Problem noch nicht gelöst: alphabetische Sortierung:

- kontext abhängig
- sprach abhängig
- :
- deutschen Wörterbuch: wie a
- deutsches Telefonbuch: wie ae
- schwedisch: hinter Zeichen $\overset{\circ}{a}$

```

1 #include <locale>
2 #include <codecvt>
3 std::sort(v.begin(), v.end(), std::locale("se_SE.UTF-8")); // für
  ↪ schwedisch (falls se_SE.UTF-8 installiert)

```

18 Eigene Datentypen

3 Möglichkeiten

- enum: Aufzählungstypen ⇒ Selbststudium
- struct: strukturierte Daten, zusammengesetzte Typen
- class: wie struct, auf objekt-orientiert

struct und enum schon in C, struct und class fast gleich

```

1 struct TypeName { // Neuer Typ
2     type_name1 var_name1; // existierende Typen
3     type_name2 var_name2;
4     // ...
5 }; // semikolon WICHTIG, falls vergessen: Fehlermeldung
6
7 // Beispiel:
8 struct Date {
9     // Datenmember, Membervariables
10    int day;
11    int month;
12    int year;
13 };
14
15 Date easter(int year) {
16     // Datum ausrechnen
17
18     // Datum zurückgeben
19     Date d;
20     d.day = day; // Punktsyntax kennen wir schon
21     d.month = month;
22     d.year = year;
23     return d;
24 }
25
26 struct Character {
27     wchar_t clear;
28     wchar_t encrypted;
29     int count;
30 };
31
32 Character c;
33 c.count = 0;
34 c.count += 1;

```

19 Objektorientierte Programmierung

Eigene Datentypen mit Kapselung. Eigene Datentypen sind zusammengesetzt aus einfachen existierenden Datentypen (Ausnahme: enum), zwei Arten:

- offene Typen: Benutzer kann auf interne Daten zugreifen, „C-style types“ wenige Beispiele in der Standardbibliothek:

- `std::pair`
- `std::tm` (von C übernommen)
- `std::div_t` (Resultat von `div()`, zwei Returnwerte: Quotient und Rest)
- gekapselte Typen: Benutzer kann nicht auf interne Daten zugreifen „private“. Alle Benutzerzugriffe über Öffentliches Interface „public“. Vorteile:
 - komplexe Details zur Verwaltung bleiben verborgen
 - öffentliches Interface (hoffentlich) einfach, zum Beispiel `std::vector`
 - interne Details können bei Bedarf geändert werden, ohne dass sich das öffentliche Interface ändert \implies Benutzer muss Code nicht ändern, aber plötzlich geht Programm schneller, „Rückwärts-Kompatibilität“

Wie erreicht man Kapselung?

Zwei Schlüsselwörter für eigene Typen „class“ (Konvention OOP), „struct“ (von C übernommen)

Zwei Schlüsselwörter für Kapselung: „public“ (öffentlicher Teil), „private“ (gekapselter Teil). `class` ist standardmäßig „private“, `struct` standardmäßig „public“

```

1  class MyType {
2      // private by default
3      public:
4      // jetzt öffentlich
5      private:
6      // wieder private
7      public:
8      // etc.
9  }; // <--- Semikolon wichtig
10 struct MyType {
11     // öffentlich by default
12     private:
13     // jetzt privat
14     public:
15     // etc
16 };

```

Benutzer können nur auf Funktionalität in public-Teil zugreifen.

Die im zusammengesetzten typ enthaltenen daten heißen „Daten-Members“, „member variables“ sind normalerweise „private“ \implies

- kann nachträglich geändert werden, zum Beispiel `complex` real/imaginär vs. Betrag/Phase
- Benutzer kann nicht unbeabsichtigt die Konsistenz verletzen (zum Beispiel versentlicht negativer Betrag bei `complex`)

Running example: Punkt-Klasse für 2-dimensionalen Punkt

```
1 class Point {
2     double x_; // Koordinaten als private
3     double y_; // Datenmenber (Konvention "_" am Ende)
4 };
```

dieser Typ ist unbenutzbar, weil alle privat.

unverzichtbar: öffentliche Funktion zum Initialisieren des Speichers = „Konstruktoren“

- Prozeduren innerhalb der Klasse, Name gleich Klassennamen, Returntyp muss weggelassen werden (fügt Compiler automatisch ein) \implies nur Seiteneffekt: neues Objekt initialisieren, also die Konstruktoren der Datenmember aufrufen.
- zur Erinnerung: zwei Möglichkeiten für normale Variableninitialisierung

```
1 double z = 1.0;
2 double z(1.0); // nur diese Syntax im Konstruktor erlaubt.
3 double z{1,0}; // seit c++11
4
5 class Point {
6     double x_;
7     double y_;
8 public:
9     Point(double x, double y) :x_(x), // Member x_ auf Initialwert x
        ↪ vom Prozedurargument setzen
10    y_(y) // Member y_ auf Initialwert y vom Prozedurargument setzen
11    {
12        // normaler Rumpf der Prozedur, hier leer, weil nichts zu tun
13        // Datenmenber hier schon fertig initialisiert und können
        ↪ benutzt werden
14    }
15 };
16 Point p(1.0, 2.0);
17 Point q = {3.0, 4.0}
```

Spezieller Konstruktor: ohne Argument = Standardkonstruktor, „default constructor“, initialisiert Objekt in Standardzustand. Zum Beispiel bei Zahlen auf 0 setzen, hier auf Koordinatenursprung Point origin; \iff Point origin(0.0, 0.0);

```
1 class Point {
2     // ... wie zuvor
3     Point() : x_(0.0), y_(0.0) {} // Standardkonstruktor
4 };
```

um mit Punkt-Objekten zu arbeiten, brauchen wir weitere Funktionen: zwei Möglichkeiten

- Member-Funktionen: innerhalb der Klasse definiert (wie Konstruktor), dürfen auf alles private zugreifen, können als private oder public definiert sein
- freie Funktionen: normale Funktionen außerhalb der Klasse, die ein Argument des neuen Typs haben, können nur auf Öffentliches Interface zugreifen

wichtigste Vertreter der Member-Funktionen: Zugriffsfunktionen „Getter“: erlauben Benutzer, aktuellen Zustand abzufragen:

```
1 std::vector<int> v = {1, 2, 3};
2 v.size(); // getter für Arraygröße
3 Point p(1,0, 2.0);
4 p.x() // -> 1.0 x-Koordinate abfragen
5 p.y() // -> 2.0 y-Koordinate abfragen
```

Member-Funktionen werden mit Punkt-Syntax aufgerufen: `p.x()`, das Objekt für dem Punkt ist das „nullte“ Argument der Funktion, Compiler macht daraus `x(p)`. Bei Implementation der Member Funktion schreibt man das nullte Argument nicht hin. Der Compiler stellt es automatisch unter dem Namen „(*this)“ zur Verfügung.

```
1 class Point {
2     // ... wie zuvor
3     double x() const {
4         return (*this).x_;
5     }
6
7     double y() const {
8         return (*this).y_;
9     }
10 };
```

Meist kann man `(*this)` weglassen, wenn eindeutig ist, welchen Member man meint, fügt es der Compiler es automatisch ein. Getter-Funktionen sind read-only (ändern die Member-Variablen nicht), man sollte sie deshalb mittels „const“ explizit als read-only markieren. Vorteile:

- Programmierer kann Member-Variablen nicht irrtümlich ändern
- Funktion kann in Kontexten benutzt werden, wo das Objekt (nulltes Argument) explizit als read-only markiert ist.

```
1 Point const cp(1.0, 2.0); // write-once
2 cp.x();
```

Point auf Konsole ausgeben: nach String wandeln und String ausgeben

· zwei Möglichkeiten:

```
1 // Member Funktion:
2 std::cout << p.to_string() << endl;
3 class Point {
4     // ... wie zuvor
5     std::string to_string() const {
6         std::string res;
7         res += "[" + std::to_string((*this).x()) + "," +
8             ↪ std::to_string((*this).y()) + "]"
9         return res;
10    }
11 };
12 // freie Funktion:
13 std::cout << to_string(p) << endl;
14 std::string to_string(Point p) const {
15     std::string res;
16     res += "[" + std::to_string((*this).x()) + "," +
17         ↪ std::to_string((*this).y()) + "]"
18     return res;
19 }
```

Was man wählt ist Geschmackssache (freie Funktion ist kompatibel zu std::to_string, kleiner Vorteil).

Punkte vergleichen:

```
1 class Point {
2     // ... wie zuvor
3     bool equals(Point other) const {
4         return (*this).x() == other.x() && (*this).y() == other.y();
5     }
6     bool operator==(Point other) const {
7         return (*this).x() == other.x() && (*this).y() == other.y();
8     }
9     bool operator!=(Point other) const {
10        return (*this).x() != other.x() || (*this).y() != other.y();
11    }
```

```

11     }
12 }
13
14 Point p(1.0, 2.0);
15 Point origin;
16 assert(p.equals(p));
17 assert(!p.equals(origin));
18 // üblicher: Infix-Notation
19 assert(p == p);
20 assert(!(p == origin));
21 assert(p != origin);

```

Für Infix-Notation muss Prefix-Varante operator== und operator!= implementiert werden.

```

1 p == origin; // -> p.operator==(origin) -> operator==(point, origin)

```

zwei weitere Funktionen:

- neuen Punkt erzeugen, transponiert, das heißt x- und y-Koordinate vertauscht.
- verschoben um Punkt

```

1 class Point {
2     // ... wie zuvor
3     Point transpose() const {
4         Point res((*this).y(), (*this).x());
5         return res;
6     }
7     Point translate(Point v) const {
8         Point res((*this).x() + v.x(), (*this).y() + v.y());
9         return res;
10    }
11 }

```

Jede Klasse hat bestimmte spezielle Member-Funktionen:

- Konstruktoren bringen Objekt in wohldefinierten Anfangszustand
- Destruktoren: Entsorgt ein nicht mehr benötigtes Objekt (meist am Ende der Umgebung)
- Zuweisungsoperatoren: um Objekt per Zuweisung („=“) zu überschreiben \implies später

19.1 Destruktoren

jede Klasse muss genau einen Destruktor haben, wenn der Programmierer das nicht explizit implementiert, fügt Compiler ihn automatisch ein. Syntax:

```
1 class klassenName {
2 public:
3     ~klassenName(){
4         // Implementation
5     }
6 };
```

Der automatische Destruktor ruft einfach die Destruktoren aller Member-Variablen auf. Meist ist das ausreichend, aber in bestimmten Situationen muss der Programmierer zusätzliche Aktionen implementieren. Beispiele dafür:

1. manuelle Speicherverwaltung: Destruktor muss nicht mehr benötigten Speicher an Betriebssystem zurückgeben (\Rightarrow später), zum Beispiel Destruktor von `std::vector`, `std::string` (Kapselung: Benutzer merken davon nichts)
2. manuelles Dateimanagement: Destruktor muss Datei schließen, also Dateien aus dem Festplattencache auf Platte übertragen und Metadaten für Datei schreiben, zum Beispiel `std::ofstream`, `std::ifstream`
3. Abmelden von einem Service (ausloggen, Verbindung beenden)

19.2 Kopier-Konstruktor

Spezieller Kontruktor: Kopier-Konstruktor zum Erzeugen einer Kopie eines vorhandenen Objekts, das heißt neue Speicherzelle mit gleichem Inhalt.

```
1 Point p(1.0, 2.0); // Konstruktor mit Initialwert
2 Point q = p;       // Kopierkonstruktor
3 Point r(p);        // Kopierkonstruktor
4
5 int foo(Point q) {
6     // ...
7 }
8 int bar(Point const & g) {
9     // ...
10 }
11 int main() {
12     Pont p(1.0, 2.0);
13     foo(p) // Kopierkonstruktor, um lokales q in foo aus p zu erzeugen,
           ↪ "pass-by-value"
```

```

14     bar(p) // g ist neuer Name für p, keine neue Speicherzelle, kein
        ↪ Kopierkonstruktor, "pass-by-reference"
15 }
16
17 // Syntax:
18 class KlassenName {
19 public:
20     klassenName(klassenName const & existing) {
21         // ...
22     }
23 };

```

Der Compiler erzeugt Kopier-Konstruktor automatisch, falls nicht explizit programmiert = ruft Kopier-Konstruktor aller Member-Variablen auf, meistens richtig, Ausnahmen wie oben.

19.3 Standard-Konstruktor

Spezieller Konstruktor: Standard-Konstruktor („default constructor“): ohne Argumente, bringt Object in Standard-Zustand, zum Beispiel „0“ bei Zahlen, „“ bei (Leerstring) bei std::string, leeres Array std::vector, (0, 0) bei Point. Syntax:

```

1 class klassenName {
2 public:
3     klassenName() {}
4     klassenName() : Initialisierungserte der Member-Variablen {}
5 };

```

Compiler erzeugt Standard-Konstruktor automatisch, falls es **keinen** eenutzerdefinierten Konstruktor gibt.

19.4 rule of three

„rule of three“: Faustregel: wenn es notwendig ist, eine der drei Funktionen Destruktor, Kopierkonstruktor, Kopier-Zuweisung (\implies später) explizit zu implementieren, müssen alle drei explizit implementiert werden

19.5 Vorteile der Kapselung:

1. Benutzung der Klasse ist viel einfacher, weil unwichtige Details verborgen sind
2. interne Implementatoin kann geändert werden, ohne den Benutzer zu Folgeänderungen zu zwingen, weil externe Schnittstelle erhalten bleibt.

19.6 Arithmetische Infix-Operationen

Ziel der OO-Programmierung: Arbeiten mit Nutzer-definierten Datenstrukturen möglichst einfach, wie mit eingebauten. Zum Beispiel arithmetische Infix-Operationen

```
1 Point p(2.0, 3.0), q(4.0, 5.0);
2 Point r = 2.5 * p + q;
3 assert(r == Point(9.0, 12.5));
```

dazu muss man die entsprechenden Prefix-Funktionen implementieren

- Addition:

```
1 Point operator+(Point p1, Point p2) {
2     Point res(p1.x() + p2.x(), p1.y() + p2.y());
3     return res;
4 }
5
6 Point operator+(Point & p1, Point & p2) {
7     Point res(p1.x() + p2.x(), p1.y() + p2.y());
8     return res;
9 }
```

- subtraktion, elementweise Multiplikation und Division genauso („+“ überall durch „-“, „*“, „/“ ersetzen)
- Skalierungsoperation: Multiplikation von Punkt mit Zahl, das heißt zwei verschiedene Argumenttypen \Rightarrow zwei Versionen: $2.5 * p$ und $p * 2.5$

```
1 Point operator*(double s, Point p) {
2     Point res(s * p.x(), s * p.y());
3     return res;
4 }
5
6 Point operator*(Point p, double s) {
7     Point res(p.x() * s, p.y() * s);
8     return res;
9 }
```

alle diese Versionen können dank „function overloading“ gleichzeitig implementiert sein

Bisher: freie Funktionen. Falls das erste Argument vom Typ Point oder Point const & ist, kann man die Funktionen alternativ als Member-Funktion implementieren:

```

1 class Point {
2     // ... wie bisher
3     Point operator+(Point const & other) const {
4         Point res(x_ * other.x_, y_ * other.y_);
5         return res;
6     }
7 };

```

Vorteil von Member Funktionen: Zugriff auf private Member der Klasse (hier nicht notwendig).

Nachteil:

- nur möglich, wenn das linke Argument vom Klassentyp ist $p * 2.5$ kann Member Funktion sein, $2.5 * p$ nicht
- nur möglich, wenn man die Klassendefinition ändern darf

19.7 Objekte nachträglich verändern

Bisher: Alle Objekte waren write-once, das heißt Speicher wurde im Konstruktor initialisiert und war dann unveränderlich. \Rightarrow Paradigma der funktionalen Programmierung - „referentielle Integrität“ Prozedurale Programmierung erfordert Möglichkeit, Objekte zu ändern, zum Beispiel um entsprechende Änderungen in der realen Welt wiederzuspiegeln (zum Beispiel Student besteht Prüfung). Möglichkeit 1: Setter-Funktionen

```

1 class Point {
2     // ... wie bisher
3
4     void setX(double new_x) {
5         x_ = new_x;
6     }
7
8     void setY(double new_y) {
9         y_ = new_y;
10    }
11 };

```

Möglichkeit 2: Index-Zugriff, wie bei `std::vector`

```

1 Point p(2, 3);
2 assert(p[0] == 2 && p[1] == 3);
3 p[0] = 4;
4 p[1] = 5;

```

```

5  assert(p[0] == 4 && p[1] == 5);
6
7  // dazu muss die Member-Funktion operator[] implementiert werden
8  class Point {
9      // ... wie bisher
10
11      double operator[](int index) const {
12          if(index == 0) { return x_; }
13          if(index == 1) { return y_; }
14          // andernfalls Fehlermeldung (Exception -> später)
15      }
16
17      double operator[](int index) {
18          // implementation identisch / gleicher Quellcode, aber
19          // unterschiedlicher Maschinencode
20      }
21  }
22
23  // Verwendung (Langform):
24  Point p(2.0, 3.0);
25  double & x = p[0]; // neue Namen x, y für Variablen p.x_ und p.y_
26  double & y = p[1];
27  x = 4.0; // ändert indirekt auch p.x_
28  y = 5.0; // ändert indirekt auch p.y_

```

Möglichkeit 3 : Zuweisungsoperatoren

```

1  Point p(2, 3), g(4, 5);
2  p = 1.0; // beide Koordinaten mit 1.0 überschreiben
3  assert(p == Point(1.0, 1.0)) p = q; // Speicherzelle p werden die gleichen
4                                     // Werte zugewiesen wie Speicherzelle q
5  assert(p == Point(4.0, 5.0)) Point & r =
6      q; // Gegesatz r und q sind Namen für gleiche Speicherzelle
7
8  class Point {
9      // ... wie bisher
10
11      void operator=(double v) {
12          x_ = v;
13          y_ = v;
14      }
15
16      // copy assignment operator (analog zu copy constructor)

```



```

17     operator=(Point const & other) {
18         x_ = other.x_;
19         y_ = other.y_;
20     }
21 }

```

Bemerkung:

1. implementiert der Programmierer keinen copy assignment Operator, implementiert Compiler ihn automatisch (wie Kopierkonstruktor): ruft copy assignment für alle Member-Variablen auf
2. man implementiert meist

```

1 Point & operator=(...) {
2     // ... wie bisher
3     return *this;
4 }

```

Vorteil: man kann Zuweisung verketteten:

```

1 r = p = q; // -> r = (p = q)

```

arithmetische Zuweisung:

```

1 Point p(2, 3),
2 q(4, 5);
3 p += q; // add -assign: Abkürzung für p = p + q
4 assert(p == Point(6, 8));
5
6 class Point {
7     // ... wie bisher
8     Point & operator+=(Point const & other) {
9         x_ += other.x_;
10        y_ += other.y_;
11        return *this;
12    }
13 };

```

19.8 Klasse Image

Speichert 2D Bild (analog: Matrix) zunächst nur Graubilder, später: Farbbilder. Beispiel für dynamische Datenstruktur, Größe erst zu Laufzeit bekannt und änderbar. Besteht aus

Pixeln („picture elements“), die mit 2 Indizes x und y angesprochen werden Problem: Speicher ist 1-dimensional, Lösung: Lege Zeilen hintereinander ab. \implies Übergangsformeln:

$$i = x + y \cdot \text{width}$$

$$x = 1y \qquad \qquad \qquad = \frac{1}{\text{width}}$$

```

1  class Image {
2      int width_, height_;
3      std::vector<uint16_t> data_;
4
5  public:
6      Image() : width_(0), height_(0) {}
7      Image(unsigned int w, unsigned int h)
8          : width_(w), height_(h), data_(w * h, 0) {}
9      int width() const { return width_; }
10     int height() const { return height_; }
11     int size() const { return width_ * height_; }
12     void resize(unsigned int w, unsigned int h) {
13         width_ = w;
14         height_ = h;
15         data_.resize(w * h);
16     }
17     uint16_t get(int x, int y) const { return data_[x + y * width_]; }
18     void set(int x, int y, uint16_t v) { data_[x + y * width_] = v; }
19
20     // Zugriff bequemer: wünschenswert:
21     // uint16_t v = image[1, 2];
22     // image[1, 2] = 255;
23     // geht nicht, weil im Indexoperator nur 1 Argument sein darf
24     // -> verwende Stattdessen runde Klammern
25     uint16_t operator()(int x, int y) const {
26         return get(x, y); // return data_[x + y * width_];
27     }
28     uint16_t operator()(int x, int y) {
29         return return data_[x + y * width_];
30     }
31     // damit: uint16_t = image(1, 2); image(1, 2) = 255;
32 };
33
34
35 std::string to_string(Image const & im) {
36     std::string res;

```

```

37     for(int x = 0; x < im.width(); x++) {
38     for(int y = 0; y < im.height(); y++) {
39         res += std::to_string(im(x, y)) + " ";
40     }
41         res.pop_back();
42         res += "\n";
43     }
44
45     return res;
46 }

```

PGM-Format („Portable GrayMap“): reines Textformat: my_image.pgm:

```

P2
# Kommentar
<width> <height>
255
Ausgabe von to_string

```