Vertiefte Programmierpraxis

FK04

SoSem 2024

Klaus J. Ressel

© Klaus. J. Ressel, 18. März 2024

Inhaltsverzeichnis

1 Einführung **7**
1.1 Softwareentwicklungs-Prozes . . . 7
1.1.1 Entwicklungsziel . . . 7
1.1.2 Phasen und Aktivitäten des Entwicklungsprozesses . . . 8
1.1.3 Vorgehensmodelle . . . 12
1.2 In der Praxis . . . 14
1.3 Modellierung mit UML . . . 15
1.3.1 Hintergrund und Geschichte . . . 15
1.3.2 Srukturdiagramme der UML . . . 17
1.3.3 Verhaltensdiagramme der UML . . . 19
1.4 Technische Hilfsmittel . . . 24
1.4.1 Allgemeines . . . 24
1.4.2 Werkzeuge für die Projektorganisation/-Verwaltung . . . 26
1.4.3 Werkzeuge für die Modellierung . . . 27
1.4.4 Werkzeuge für die Versionsverwaltung . . . 28
1.4.5 Werkzeuge das Build-Management . . . 29
1.4.6 Werkzeuge für die Qualitätssicherung . . . 31
1.4.7 Werkzeuge für das Testen . . . 33
1.5 Ziele der Vorlesung . . . 36
1.6 Literaturhinweise . . . 37
1.6.1 Literaturhinweise zur Programmiersprache C++ . . . 37
1.6.2 Literaturhinweise zu Softwaredesign . . . 39
1.6.3 Literaturhinweise zu Entwurfsmuster . . . 40
1.6.4 Literaturhinweise zu UML . . . 41
1.6.5 Literaturhinweise zu Qt . . . 41
1.6.6 Literaturhinweise zu Testverfahren . . . 42
1.7 Die Programmiersprache C++ . . . 43
1.7.1 Überblick . . . 43
1.7.2 Entwicklung der Objektorientierten Programmierung . . . 43
1.7.3 Entwicklung von C++ . . . 44
1.7.4 Eigenschaften von C++ . . . 46
1.7.5 Popularität von C++ . . . 48
1.7.6 Laufzeit-Performance von C++ . . . 51
1.8 Aufgabe TDD am Bsp. von Soundex . . . 53
1.8.1 Soundex-Algorithmus . . . 54
1.8.2 TDD Lösung . . . 55

2 Unterschiede zwischen C++ und C **56**
2.1 Hello World Programm . . . 56
2.2 Namensbereiche . . . 58
2.3 Ein/Ausgabe mit Streams . . . 61
2.3.1 Stream-Klassen in C++ . . . 62
2.3.2 Werte Ausgeben . . . 63
2.3.3 Werte Einlesen . . . 64

2.3.4 Ein/Ausgabe im C-Stil . . . 66
2.3.5 Formatierte Ausgabe mit Flags . . . 67
2.3.6 Formatierte Ausgabe mit Manipulatoren . . . 73
2.4 Referenzen . . . 79
2.4.1 Referenzen als Parameter . . . 81
2.4.2 Vergleich Zeiger und Referenzen . . . 83
2.5 L-Wert und R-Wert . . . 84
2.6 Typ-Qualifizierer const . . . 85
2.6.1 Schreibschutz für Zeiger . . . 86
2.6.2 Schreibschutz für Referenzen . . . 89
2.6.3 Schlüsselwort constexpr . . . 92
2.7 Vereinheitlichte Initialisierung von Objekten . . . 94
2.8 Dynamische Speicherverwaltung mit new und delete . . . 95
2.8.1 Aufteilung des Hauptspeichers . . . 95
2.8.2 Dynamischen Speicher reservieren mit new . . . 96
2.8.3 Speicherfreigabe mit delete . . . 98
2.8.4 Dynamische Speicherverwaltung für Felder . . . 100
2.8.5 Dynamische Speicherverwaltung in C . . . 103
2.9 Komplexe Deklarationen lesen . . . 106
2.9.1 Erinnerung Funktionszeiger . . . 107
2.10 Argumentübergabe und Ergebnisrückgabe von Funktionen . . . 112
2.11 Deklaration und Definition in C++ . . . 119
2.11.1 Deklaration . . . 119
2.11.2 Definition . . . 120
2.11.3 Header-Dateien und Implementierungs-Dateien . . . 121
2.12 Default Parameter von Funktionen . . . 122
2.13 Inline-Funktionen . . . 124
2.14 Überladen von Funktionen . . . 125
2.15 Datentyp bool in C++ . . . 127
2.16 Der Typplatzhalter auto . . . 128
2.17 Bereichsbasierte for-Schleifen für Arrays und Container . . . 130
2.17.1 Bereichsbasierte for-Schleifen für Arrays . . . 130
2.17.2 Bereichsbasierte for-Schleifen für Container . . . 131
2.18 Verwendung von nullptr anstelle 0 oder NULL für den Nullpointer . . . 135
2.19 Zusammenfassende Aufgabe Gray-Code . . . 136

3 Grundkonzepte der Objektorientierten Programmierung (OOP) 141
3.1 Objektorientierte Vorgehensweise . . . 141
3.2 Klassifizierung/Abstraktion . . . 146
3.3 Objekte in Computer-Welt . . . 147
3.4 Klasse als Datentyp, Instanz, Objekt . . . 148
3.5 Datenkapselung (Information Hiding) . . . 149
3.6 Vererbung (Inheritance) . . . 151
3.7 Polymorphie . . . 152
3.8 Haus der Objekt-Orientierung . . . 154
3.9 UML Klassendiagramm . . . 155
3.9.1 Vererbung . . . 157
3.9.2 Assoziation . . . 158
3.9.3 Aggregation, Komposition . . . 159
3.10 Aufgabe OOA und OOD . . . 160

4 Klassen **162**
4.1 Einführung . . . 162
4.2 Definition von Klassen . . . 163
4.2.1 Zugriffskontrolle durch Modifikatoren . . . 163
4.2.2 Strukturen struct . . . 166
4.2.3 Daten-Member (Attribute) . . . 167
4.2.4 Member-Funktionen (Methoden) . . . 168
4.2.5 inline Funktionen/Methoden . . . 170
4.2.6 Konstante Member-Funktionen . . . 172
4.3 Statische Klassen-Member . . . 173
4.4 Instanzen von Klassen (Objekte) . . . 177
4.4.1 Operationen mit Objekten . . . 179
4.5 Konstruktoren . . . 181
4.5.1 Resssourcenbelegung ist Initialisierung (RAII) . . . 182
4.5.2 Default-Konstruktor . . . 184
4.5.3 Konstruktoren mit Member-Initialisierungsliste . . . 186
4.5.4 Notwendige Existenz eines Default-Konstruktors . . . 188
4.5.5 Standard-Methoden aktivieren, Methoden deaktivieren . . . 189
4.5.6 Delegierende Konstruktoren . . . 191
4.5.7 Konvertierende Konstruktoren . . . 192
4.5.8 Konstruktoren mit Default-Parametern . . . 194
4.5.9 Initialisierungslisten-Konstruktoren . . . 195
4.6 Destruktoren . . . 197

5 Überladung von Operatoren **201**
5.1 Grundlegende Konzepte . . . 201
5.2 Operatorfunktionen . . . 203
5.2.1 Freie Operatorfunktionen . . . 204
5.2.2 Das friend Konzept in C++ . . . 206
5.2.3 Operatorfunktionen als Member-Funktionen . . . 207
5.3 Ausgabe-Operator . . . 209
5.4 Indizierungs-Operator . . . 210
5.5 Arithmetik mit gemischten Typen . . . 211
5.6 Inkrement-Operator . . . 214
5.7 Kopieren und Zuweisen . . . 216
5.7.1 Bsp. Verletzung der Dreier-Regel . . . 217
5.7.2 Bsp. Erfüllung der Dreier-Regel . . . 219
5.8 Verschieben anstelle von Kopieren . . . 222
5.8.1 Bsp. R-werte-Referenz . . . 223

6 Vererbung **228**
6.1 Grundlegende Konzepte der Vererbung . . . 228
6.1.1 Vorteile der Vererbung . . . 229
6.1.2 Vererbung oder Templates . . . 230
6.1.3 Kardinalität der Vererbung . . . 231
6.2 Definition abgeleiteter Klassen in C++ . . . 232
6.2.1 Zugriff auf Basisklassen-Member . . . 233
6.2.2 Ableitung unterbinden mit final . . . 234
6.3 Member der abgeleiteten Klasse . . . 235
6.4 Konstruktoren und Vererbung . . . 237
6.5 Aufbau und Abbau von Objekten . . . 238

7 Polymorphie **241**
7.1 Liskovsches Substitutionsprinzip . . . 242
7.2 Cast-Operatoren . . . 243
7.2.1 C-Stil Cast-Operatoren . . . 243
7.2.2 Static Cast . . . 244
7.2.3 Dynamic Cast . . . 245
7.2.4 Const Cast . . . 247
7.2.5 Reinterpret Cast . . . 248
7.3 Laufzeit-Typ-Informationen . . . 249
7.3.1 Typinformationen über Dynamic Cast . . . 249
7.3.2 Typinformationen über typeid-Operator . . . 252
7.4 Virtuelle Methoden . . . 256
7.5 Abstrakte Klassen . . . 259
7.6 Schnittstellen-Programmierung . . . 261
7.7 Virtuelle Destruktoren . . . 264
7.8 Aufgabe KFZ-Steuer OOP . . . 268

8 Entwurfsmuster (Design Patterns) **271**
8.1 Singleton . . . 273
8.1.1 Bsp. Singleton (klassisch) . . . 274
8.1.2 Singleton (Scott Meyers) . . . 276
8.2 Composite . . . 278
8.2.1 Bsp. Impedanzberechnung . . . 279
8.3 Observer . . . 282
8.3.1 Pull und Push Modell . . . 283
8.3.2 Bsp. Observer . . . 284

9 Ausnahmebehandlung **288**
9.1 Laufzeitfehler/Logische Fehler . . . 288
9.1.1 Historische Programmfehler . . . 289
9.1.2 Fehlerhäufigkeit und Programmgröße . . . 291
9.2 Klassische Fehlerbehandlung mit Fehler-Nummern . . . 292
9.3 Konzept der Exceptions . . . 293
9.4 Exception Auswerfen mit throw . . . 295
9.5 try-catch Block . . . 296
9.6 Exception erneut auswerfen . . . 298
9.7 Exception Spezifikationen für Funktionen . . . 299
9.8 Bsp. Stack Unwinding . . . 300
9.9 RAII anstelle von finally . . . 302
9.10 Fehlerklassen der Std.-Bibliothek . . . 305
9.10.1 Basisklasse exception . . . 306
9.10.2 Unterklasse `ios_base::failure` . . . 307
9.11 Zusammenfassendes Beispiel . . . 309
9.12 Exception Missbrauch zur Ablaufsteuerung . . . 312
9.13 Aufgabe ArrayBool . . . 314

10 Templates **316**
10.1 Funktions-Templates . . . 317
10.1.1 Aufruf von Funktions-Templates . . . 318
10.1.2 Spezialisierung von Funktions-Templates . . . 319
10.1.3 Nicht-Typ-Parameter . . . 320
10.1.4 Aufgabe Bubble-Sort . . . 322
10.2 Klassen-Templates . . . 323

10.2.1 Klassen-Name . . . 324
10.2.2 Aufteilung des Quellcodes . . . 325
10.2.3 Methoden-Templates . . . 332
10.2.4 Aufgabe Template Stack . . . 336

11 Verwendung der Standardbibliothek 338
11.1 Allgemeines zur ISO-C++ Standard-Bibliothek . . . 338
11.1.1 Design-Aspekte . . . 339
11.1.2 Header-Dateien . . . 340
11.2 Komponenten der Standard-Bibliothek . . . 341
11.3 Dateiverarbeitung mit Streams . . . 345
11.3.1 Funktionalität der Stream-Klassen . . . 346
11.3.2 Vorgehensweise . . . 347
11.3.3 Klasse ofstream . . . 348
11.3.4 Klasse ifstream . . . 351
11.4 Algorithmen der STL . . . 354
11.4.1 Funktionsübergabe . . . 356
11.4.2 Funktionsübergabe als Funktionspointer . . . 357
11.4.3 Funktionsübergabe als Funktionsobjekt . . . 358
11.4.4 Funktionsübergabe mittels Lambda-Ausdruck . . . 361
11.5 Funktionen höherer Ordnung als Template . . . 365
11.6 Aufgaben Funktionale Programmierung . . . 368

12 Einführung in Qt 370
12.1 Graphische Benutzeroberflächen . . . 370
12.1.1 Ablauf Konsolen-Anwendung . . . 371
12.1.2 Ablauf GUI-Anwendung . . . 372
12.2 Toolkits . . . 373
12.3 Bestandteile von Qt . . . 374
12.3.1 Wichtige Qt Module (Qt Essentials) . . . 376
12.3.2 Klassenhierarchie von Qt . . . 378
12.4 Objektbäume und Besitz . . . 379
12.5 Erstellung eines GUIS mit Qt Creator . . . 380
12.5.1 Eigene GUI Klasse . . . 382
12.5.2 Hauptprogramm mit Eventloop-Start . . . 383
12.6 Signals und Slots . . . 384
12.6.1 Signal mit Slot verbinden . . . 385
12.7 Layouts . . . 390
12.8 Model-View-Control (MVC) Muster . . . 393
12.9 Zeichnen mit Qt . . . 395
12.9.1 Vorgehensweise beim Zeichnen mit QPainter . . . 396
12.9.2 Stift, Pinsel, Schrift . . . 397
12.9.3 Koordinatensysteme . . . 398
12.10 Qt Hauptfenster . . . 404

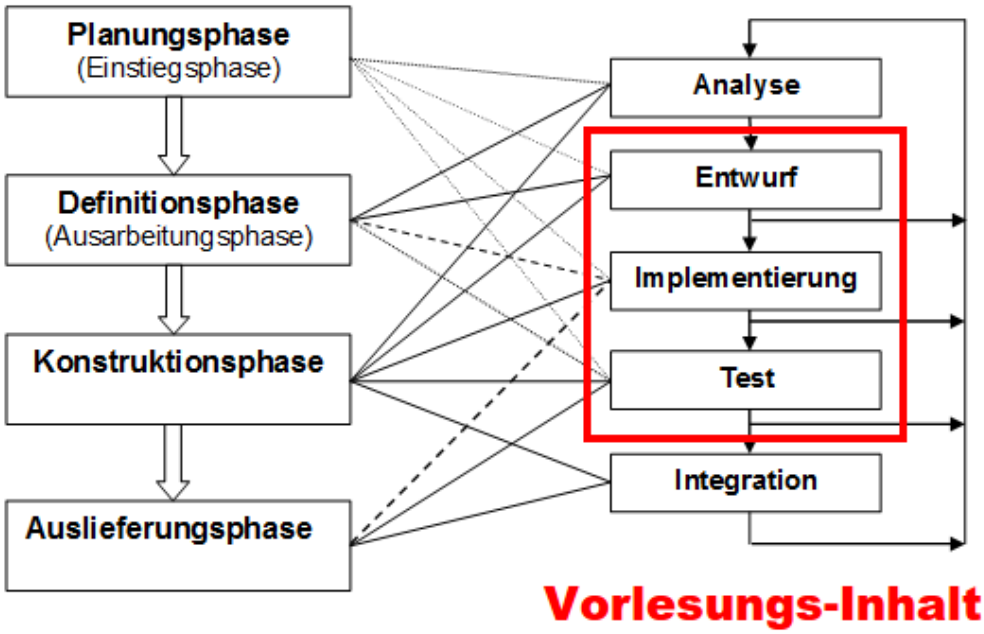
1. Einführung

1.1. Softwareentwicklungs-Prozess:

1.1.1. Entwicklungsziel:

- Das Ergebnis eines erfolgreichen Softwareentwicklungsprozesses soll ein Software-System sein, dass:
  - die vom Benutzer gestellten **funktionalen** und **nichtfunktionalen Anforderungen und Erwartungen** erfüllt sowie
  - **zeitgerecht** und **wirtschaftlich** entwickelt wurde
  - und **änderungs- und anpassungsfähig** ist
- **Funktionale Anforderungen** beschreiben das eigentliche Systemverhalten und die jeweiligen Funktionen des zu erstellenden Produktes.
- **Nichtfunktionale Anforderungen** zielen auf Qualitätsmerkmale ab, z. B. Zuverlässigkeit, Laufzeiteffizienz und kurze Antwortzeiten, einfache Benutzbarkeit, gute Wartbarkeit.
- Ein derartiger Prozess muss einerseits ein ausreichendes Maß an **Kreativität und Innovation** unterstützen, andererseits muss er die notwendige **Überprüfung und Steuerung** des Entwicklungsfortschritts sicherstellen.
- Diese Forderungen werden von einem iterativen und inkrementellen Entwicklungsprozess erfüllt.
- Ein derartiger Prozess besteht aus zeitlich nacheinander ablaufenden Phasen, innerhalb denen bestimmte Aktivitäts-Zyklen (Arbeitsschrittzyklen) wiederholt - mit teilweise unterschiedlicher Gewichtung - ausgeführt werden.

1.1.2. Phasen und Aktivitäten des Entwicklungsprozesses



1) Planungsphase:
(Einstiegsphase, Anfangsphase, Vorphase, inception phase)

- Ziel: Prüfung, ob ein Projekt realisiert werden soll.
- Festlegung von Zweck und Ziel eines Projektes,
- Durchführung von Trendstudien und Marktanalysen,
- Untersuchung der Durchführbarkeit (Aufwands-, Bedarfs- und Termin-Abschätzung, Risikoanalyse, Wirtschaftlichkeitsrechnung)
- Ergebnis: **Durchführbarkeitsstudie (feasability study)** bestehend aus: **Lastenheft (grobes Pflichtenheft)**, Projektkalkulation und Projektplan.

Phasen und Aktivitäten des Entwicklungsprozesses 2

2) Definitionsphase:
(**Ausarbeitungsphase, Elaborationsphase, elaboration phase**)

- Ziel: Festlegung der vollständigen, konsistenten, eindeutigen und durchführbaren Projektanforderungen
- Ermittlung und Analyse der Anforderungen (Systemanalyse):
  - Problembereichanalyse (**erste Exploration von Klassen**),
  - Abgrenzung des Software-Systems zur Systemumgebung (**Problembereichsmodell**)
  - Identifikation der **Nutzungsfälle (Use Cases)** (Nutzungsfallmodell)
  - Anforderungsspezifikation (**Pflichtenheft**)
  - Entwurf der grundlegenden Systemarchitektur (weitere Exploration von Klassen)
  - Konzipierung der externen Schnittstellen (z.B. Benutzeroberfläche)
  - Risikoanalyse und Priorisierung der Nutzungsfälle
  - Festlegung der Entwurfsstrategie
  - Planung der Systementwicklung in inkrementellen Entwicklungsschritten (Evolutionsplan, Entwurfsmodell)
  - Ergebnis: Entwurfsspezifikation.

Phasen und Aktivitäten des Entwicklungsprozesses 3

3) Konstruktionsphase:
(**construction phase**)

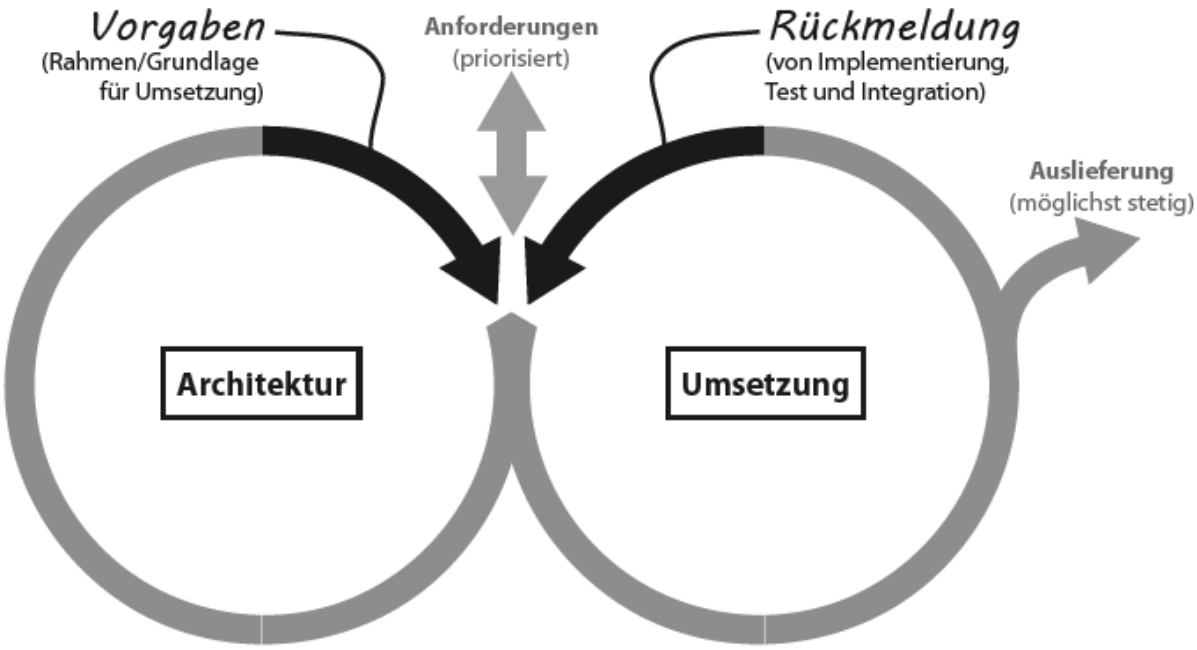
- Ziel: Erstellung eines Softwaresystems (Programm(e)), das die festgelegten Anforderungen erfüllt.
  - Entwicklung einer detaillierten Systemstruktur.
  - Identifikation und Definition weiterer Klassen/Objekte.
  - Konzipierung und Analyse der statischen Klassenbeziehungen (**Klassendiagramme**).
  - Konzipierung und Analyse der dynamischen Objektinteraktion (**Interaktionsdiagramme, Kollaborationsdiagramme**).
  - Erstellung statischer und dynamischer Modelle.
- Erstellung des Systems in einer Folge von Iterationen (zyklische Evolution):
  - Jeweils Analyse, Entwurf, Implementierung, Test und Integration.
  - Inkrementelles Hinzufügen weiterer Funktionalitäten.
  - Iterative Änderung des Codes (z.B. Umstrukturierung (Refactoring)).
  - Validierung jeder Iteration durch einen geeigneten Test.
  - Ergebnis: ausgetesteter Code mit Dokumentation, Benutzer- und Installationshandbücher, Prüf- und Testprotokolle.

Phasen und Aktivitäten des Entwicklungsprozesses 4

4) Auslieferungsphase:

(**Abnahme- u. Einführungsphase, Überleitungsphase, transition phase**)

- Ziel: Auslieferung eines fertiggestellten Produkts an den Kunden/Anwender.
- Abnahme des Produkts durch den Kunden (Abnahmetest mit Abnahmeprotokoll).
- Gegebenenfalls Korrektur von festgestellten Fehlern.
- Installation und Inbetriebnahme des Produkts.
- Schulung der Benutzer.



Architektur-Brezel: Iterative Architekturarbeit ist mit der Umsetzung verzahn, aus [Toth2015]

1.1.3. Vorgehensmodelle

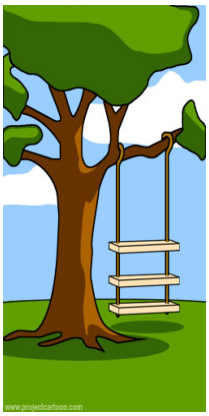
Es gibt unterschiedliche Vorgehensmodelle, wie diese einzelnen Phasen während eines Projektes durchlaufen werden:

- **Wasserfallmodell**: Erst nach dem kompletten Abschluss einer Phase wird mit der nachfolgenden Phase begonnen.
- **V-Modell**: Ähnlich dem Wasserfallmodell, aber in jeder Phase wird auch eine Qualitäts-sichernde Maßnahme gegenübergestellt.
- **Spiralenmodell**: Es werden iterativ alle Phasen mehrfach durchlaufen.
- **Rational Unified Prozess (RUP)** Vorgehensweise die auf ein gleichnamiges kommerzielles Programm der Firma Rational Software (IBM) zugeschnitten ist. Der RUP benutzt die Unified Modeling Language (UML) als Notationssprache.
- **Extreme Programming (XP)**: Iterativer Prozess der sich auf 12 Prinzipien stützt: (Pair-Programming, Kollektives Eigentum, Permanente Integration, Testgetriebene Entwicklung bzw. Permanentes Testen, Kundeneinbeziehung, Refactoring, Keine Überstunden, kurze Iterationen, Metapher (Verwendung von Kunden-Vokabular), Coding-Standards, Einfaches Design, Planning-Game).
- **Scrum**: Gegenentwurf zur Befehls-und-Kontroll-Organisation, hochqualifizierte, interdisziplinär besetzte Entwicklungsteams bekommen eine klare Zielvorgabe und sind in einem Sprint-Cyclus für die Umsetzung allein zuständig.

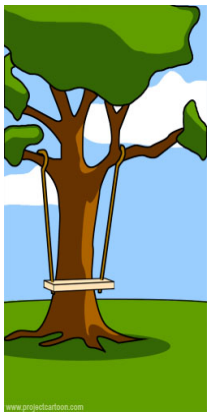
Vorgehensmodelle 2

- Die Historie hat gezeigt, dass es praktisch unmöglich ist in der frühen Phase eines Projektes die Anforderungen vollständig zu erfassen.
- Deshalb verlieren die erstgenannten strikt phasenorientierten Vorgehensmodelle immer mehr an Bedeutung.
- Seit Anfang der 1990er Jahre werden verstärkt die zuletzt genannten **agilen Vorgehensmodelle** eingesetzt.
- Sie zeichnen sich durch eine stark iterative und inkrementelle Vorgehensweise aus, die früh den Kunden miteinbezieht und in der sich das Team selbst organisiert.
- Insbesondere die Konstruktionsphase besteht aus zahlreichen Iterationen.
- In jeder Iteration wird ein vollständiger Aktivitätszyklus - z. Teil in sich auch wieder zyklisch - durchlaufen, wobei jeweils Software erstellt, getestet und integriert wird, die eine Teilmenge der Gesamtanforderungen des Projekts abdeckt.
- In jeder Entwicklungsphase können sich Rückwirkungen auf Ergebnisse und Festlegungen früherer Phasen ergeben.
- In allen Phasen und Tätigkeiten sollten die jeweiligen Ergebnisse und Festlegungen geeignet dokumentiert werden, z. B.:
  nach **IEEE 1417** oder **arc42**.

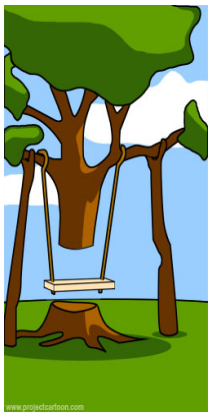
1.2. In der Praxis



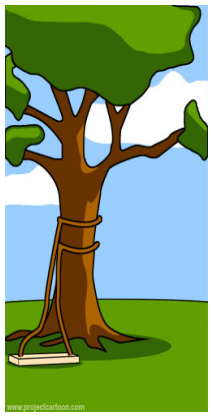
Auftrag des Kunden



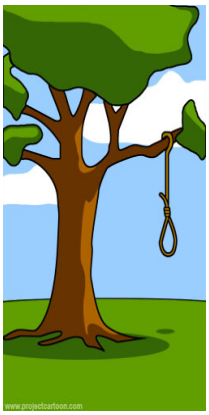
wie es der Projektleiter verstanden hat



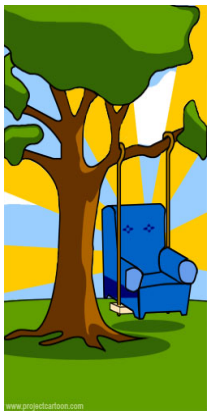
wie es der Designer entwirft



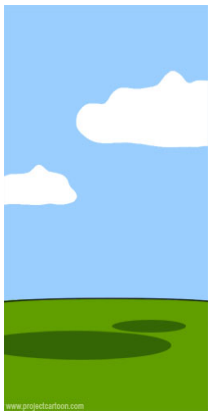
wie es programmiert wurde



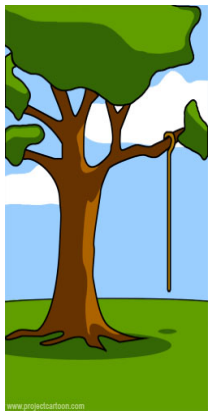
wie es ausgeliefert wird



wie es das Marketing verkauft

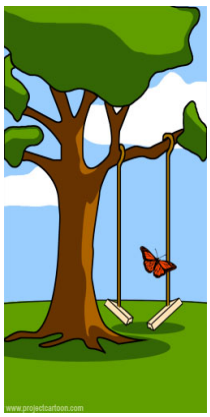


wie es dokumentiert ist

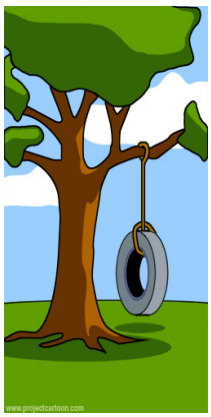


nach dem neuem Servicepack

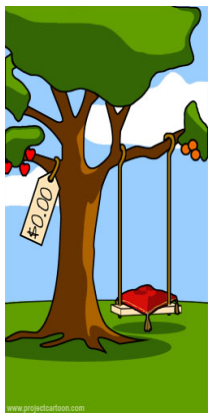
Servicepack 8



nach längerem Gebrauch



was der Kunde eigentlich benötigt hätte



die Open Source Lösung

1.3. Modellierung mit UML

1.3.1. Hintergrund und Geschichte

- Die **Unified Modeling Language (UML)** (deutsch: vereinheitlichte Modellierungssprache) ist eine grafische Modellierungssprache zur Spezifikation, Konstruktion, Dokumentation und Visualisierung von Software-Systemen.
- Mit ihrer Hilfe lassen sich Analyse und Design im Softwareentwicklungsprozess beschreiben.
- Im Sinne einer Sprache definiert UML dabei Bezeichner für die meisten bei einer Modellierung wichtigen Begriffe und legt mögliche Beziehungen zwischen diesen Begriffen fest.
- UML definiert weiterhin grafische Notationen für diese Begriffe und für Modelle statischer Strukturen und dynamischer Abläufe, die man mit diesen Begriffen formulieren kann.
- Mittlerweile hat sich die UML jedoch zu einer allgemeinen Notation für andere Beschreibungen entwickelt, zum Beispiel für Datenbanken oder Workflow-Anwendungen.
- Die UML wurde in den 90er-Jahren von den **drei Amigos** Grady Booch, Ivar Jacobson und James Rumbaugh durch Zusammenführung einiger früherer Ansätze entwickelt.
- Zunächst wurde sie vor allem von dem Entwicklungstool-Hersteller Rational gefördert und weiterentwickelt.
- Später wurde sie von der **Object Management Group (OMG)** weiter entwickelt, standardisiert und fand allgemeine Verbreitung.
- UML ist ISO genormt (ISO/IEC 19505 für Version 2.4.1).

Hintergrund und Geschichte 2

- Die Spezifikation erweitert sich ständig mit dem Aufkommen neuer Softwaretechniken.
- Ende 2003 wurde die neue Version 2.0 veröffentlicht, die einige Erweiterungen und Verbesserungen bietet.
- Die UML 2.0 beinhaltet Konzepte wie **Model-Driven Architecture (MDA)** und **Geschäftsprozessmodellierung (BPM)** und unterstützt **Echtzeitmodellierung (RT)**.
- Eine aktuelle Version des Standards lässt sich unter

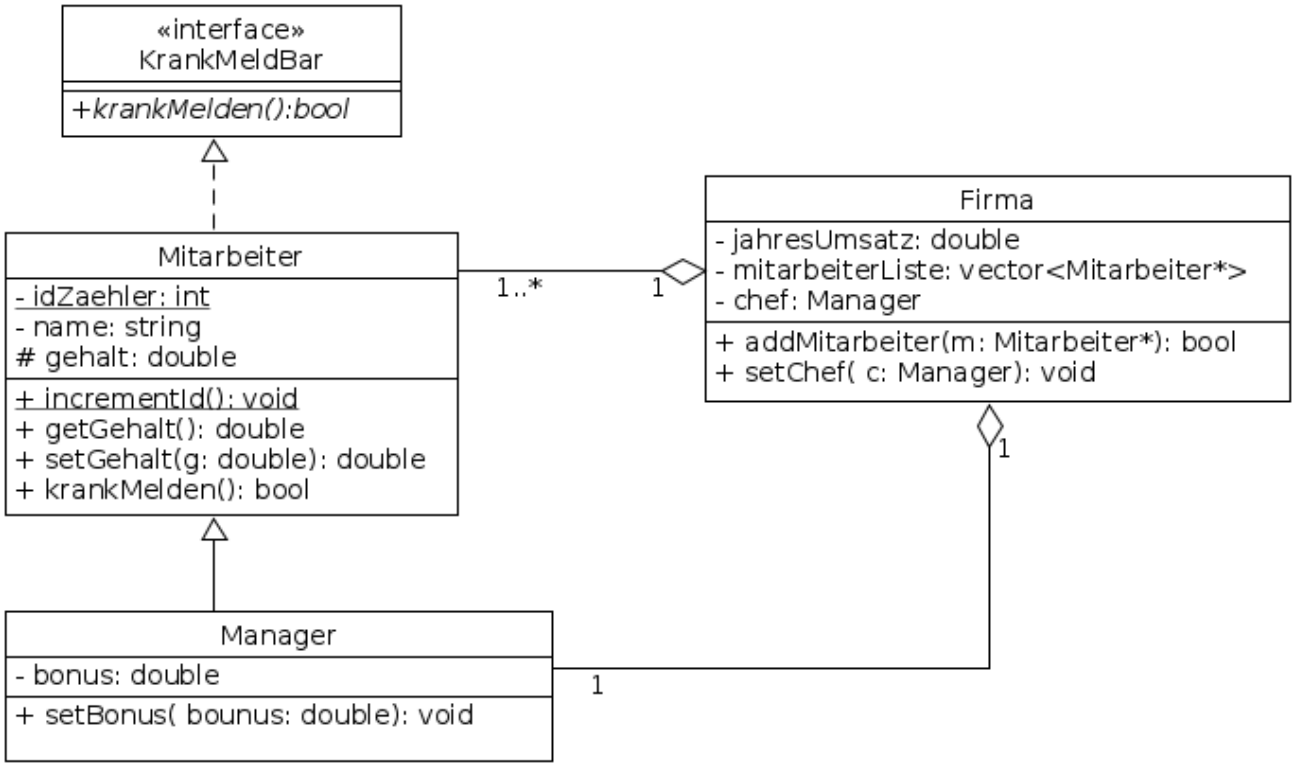
  `https://www.omg.org/spec/UML/2.3/Infrastructure/PDF/`

  einsehen.

1.3.2. Strukturdiagramme der UML

UML2.3 kennt die folgenden sieben Strukturdiagramme:

- **Klassendiagramm (Class Diagram)**:
  - Zur grafischen Darstellung (Modellierung) der statische Ansicht eines Programmentwurfs ist das Klassendiagramm einer der wichtigsten Diagrammtypen.
  - Ein Klassendiagramm stellt die Elemente von Klassen dar, also ihre Attribute (Objekt-Eigenschaften) und ihre Methoden (Verhalten des Objektes).
  - Es stellt auch die Beziehungen (Assoziation, Vererbung, Aggregation, Komposition, Schnittstellen) der Klassen untereinander dar.



Strukturdiagramme 2

- **Kompositionsstrukturdiagramm (Montagediagramm)**:

  zeigt einheitlich das Innere eines Klassifizierers (Metaklasse) und dessen Wechselwirkung mit seiner Umgebung.

- **Komponentendiagramm (component diagram)**:

  zeigt eine bestimmte Sicht auf die Struktur des modellierten Systems. Die Darstellung umfasst dabei typischerweise Komponenten mit deren Schnittstellen bzw. Ports. Es zeigt auch, wie Komponenten über Abhängigkeitsbeziehungen und Konnektoren miteinander verbunden sind

- **Verteilungsdiagramm (deployment diagram)**:

  dient der Darstellung der Verteilung von Komponenten auf Rechenknoten.

- **Objektdiagramm (object diagram)**:

  zeigt die Struktur des modellierten Systems für einen limitierten Zeitabschnitt bzw. konkreten Zeitpunkt (Momentaufnahme). Die Darstellung umfasst dabei typischerweise Objekte, die Belegung ihrer Attribute und Assoziationen.

- **Paketdiagramm (package diagram)**

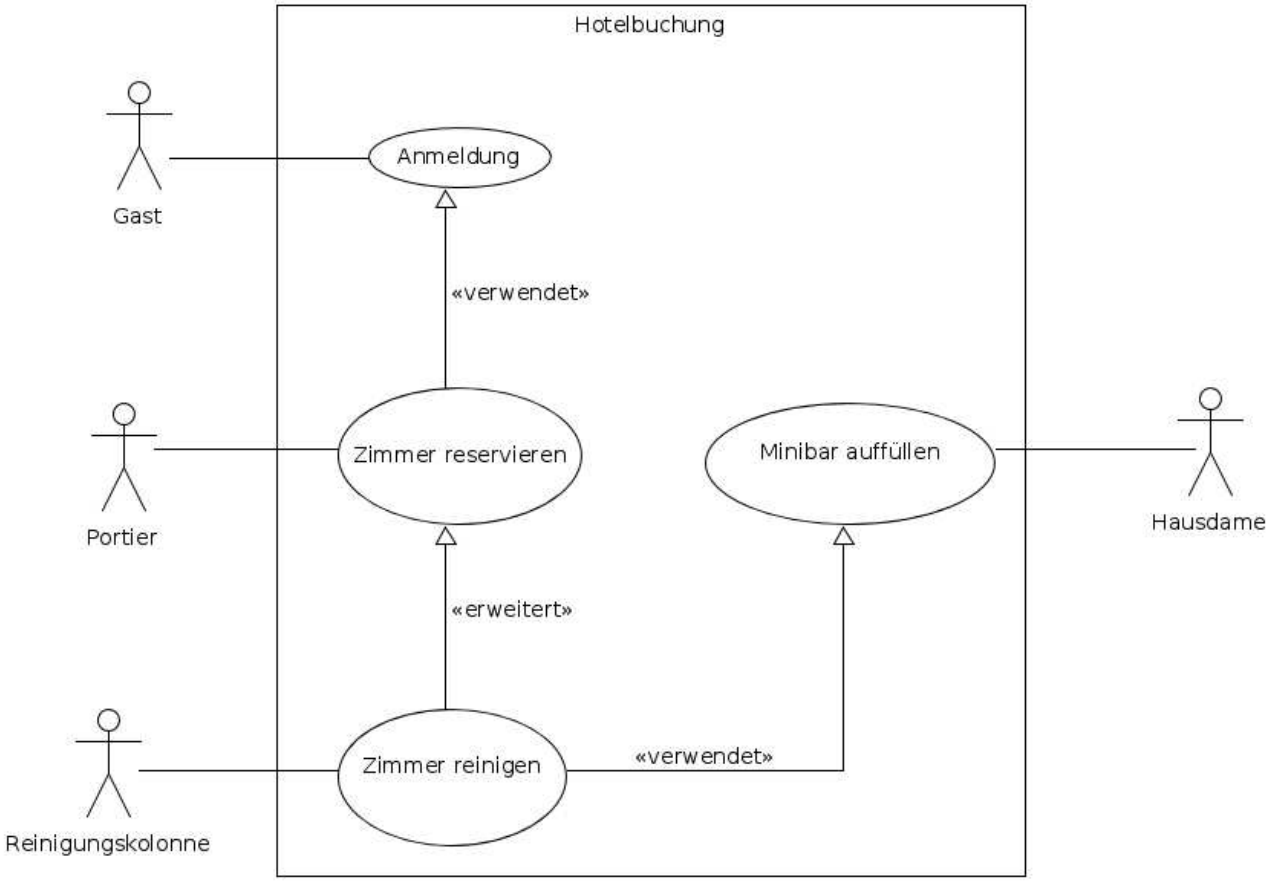
  zeigt eine bestimmte Sicht auf die Struktur des modellierten Systems. Die Darstellung umfasst dabei typischerweise Pakete, Paketverschmelzungen, Paketimports und Abhängigkeitsbeziehungen.

- **Profildiagramm**:

  wird in der Metamodellebene verwendet, um sogenannte Stereotypen mit Klassen anzuzeigen.

1.3.3. Verhaltensdiagramme der UML

- **Anwendungsfalldiagramm (use case diagram)**: stellt die potenziellen Fähigkeiten eines neuen Systems dar und beantworten die Frage: **Was soll mein geplantes System eigentlich leisten?** Jeder Anwendungsfall zeigt wie das System mit Nutzern (**Akteuren**) (dargestellt als **Strich-Männchen**) oder anderen Systemen interagiert, um ein bestimmtes Ziel zu erreichen. Ein Anwendungsfall ist ein zeitlich in sich abgeschlossener Vorgang. Er wird stets durch einen Akteur initiiert und liefert ein festgelegtes Ergebnis.



Der Anwendungsfall *Zimmer reservieren* verwendet die Daten von *Anmeldung*. Falls das Zimmer noch nicht gereinigt ist muss der Anwendungsfall *Zimmer reinigen* ausgeführt werden, er erweitert deshalb *Zimmer reservieren*. *Zimmer reinigen* beinhaltet auch immer *Minibar auffüllen*.

Verhaltensdiagramme 2

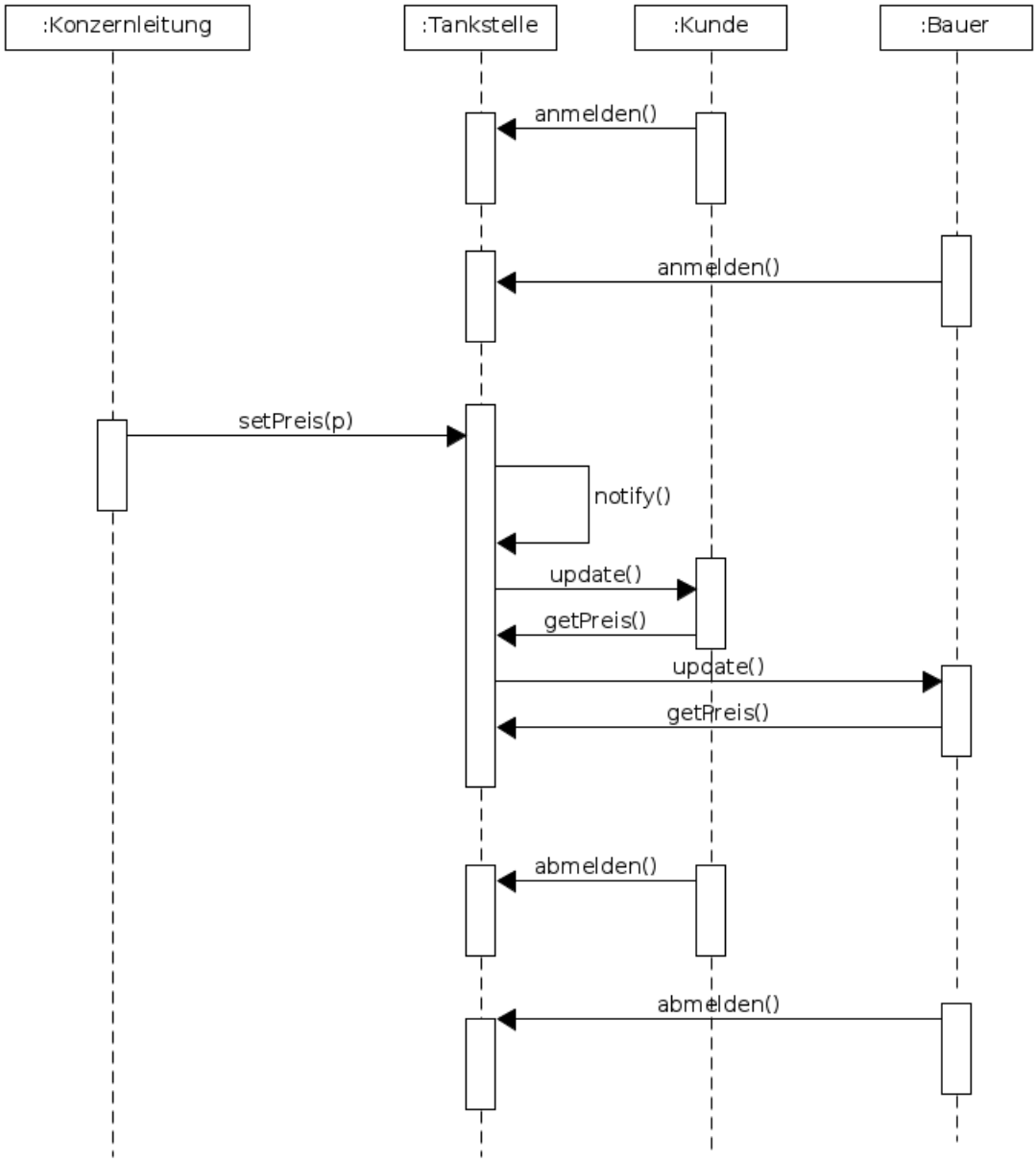
- **Sequenzdiagramm (Sequence Diagram)**

  zeigt die Beziehungen zwischen Objekten sowie deren Austausch von **Nachrichten (Methoden-Aufruf)** im Zeitablauf.

  In der Horizontalen werden die an der Interaktion beteiligten Objekte angegeben.

  In der vertikalen werden die **Lebenslinien (swim-lane)** (Zeit-Verlauf) der Objekte dargestellt.

  Bsp.: Beobachtung einer Tankstelle durch Kunden-/Bauern-Objekt.

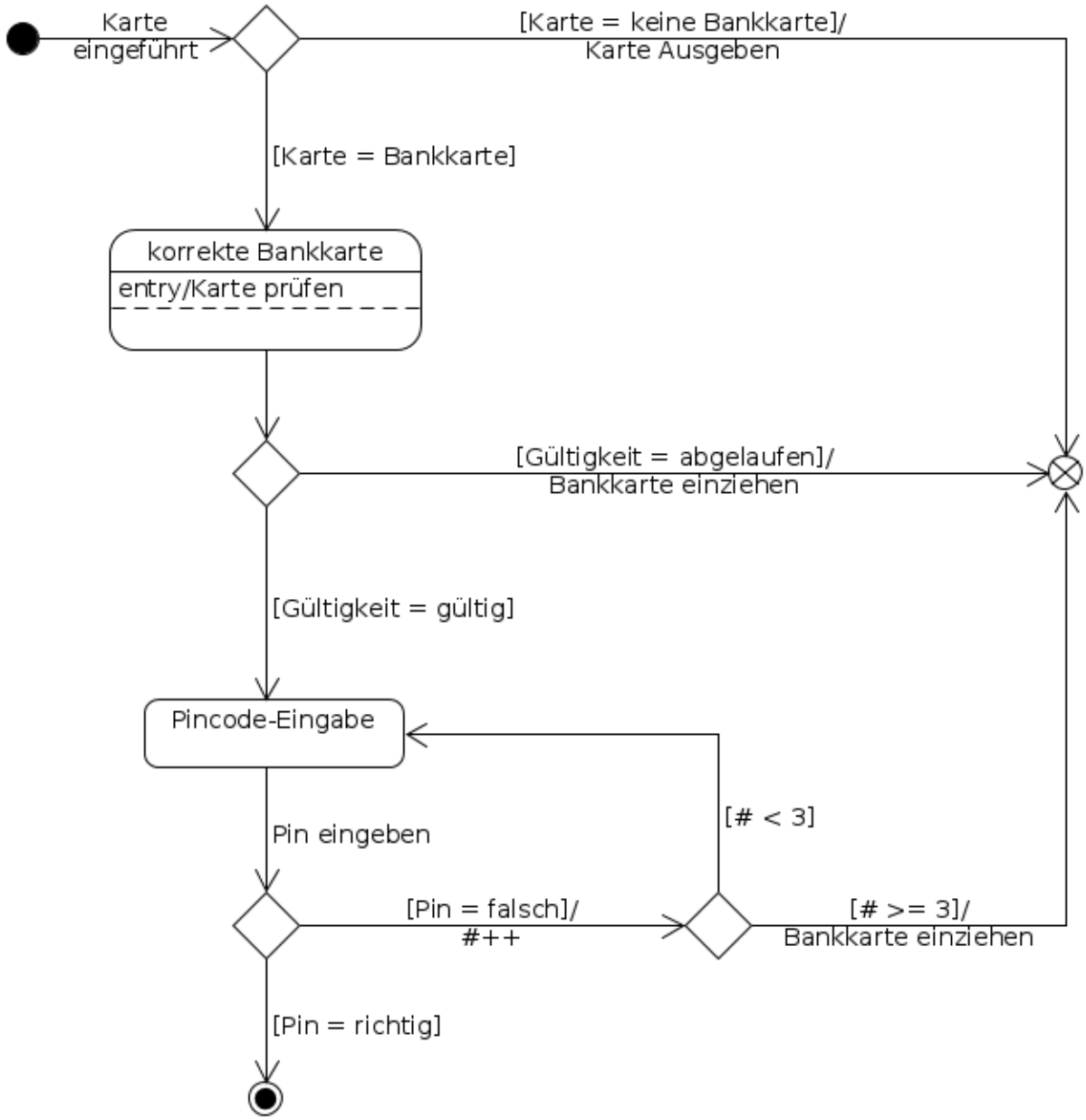


Verhaltensdiagramme 3

- **Zustandsdiagramm (state diagram)**

  stellt einen endlichen Automaten in einer UML-Sonderform grafisch dar und zeigt die verschiedenen Zustände eines Objekts und die Übergänge zwischen diesen Zuständen.

  Bsp.: Zustände eines Bankautomatens beim Authentifizieren:



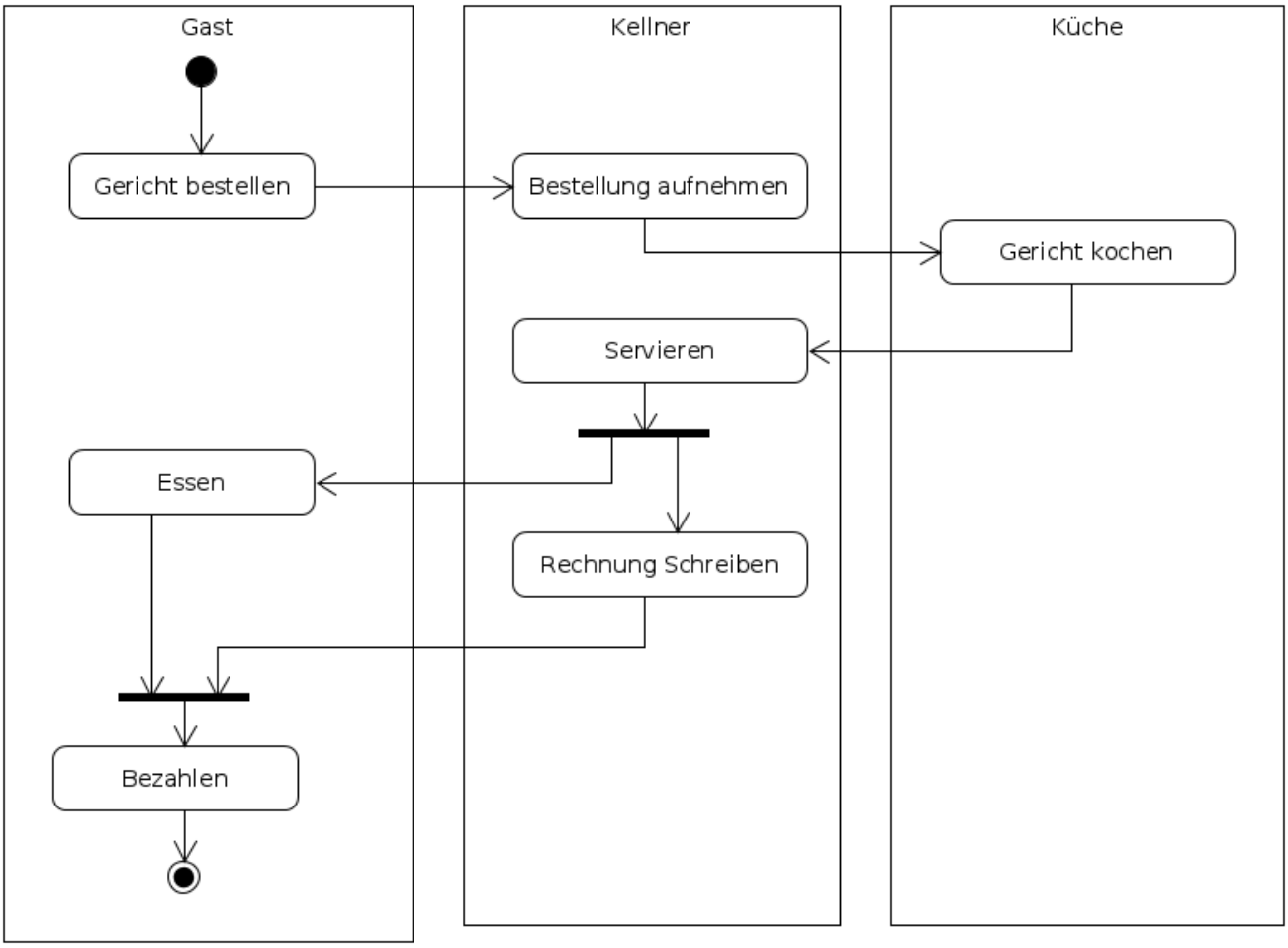
Verhaltensdiagramme 4

- **Aktivitätsdiagramm (Activity Diagram)**

  beschreibt meist den Ablauf eines Anwendungsfalls, es eignet sich aber auch zur Modellierung einzelner Aktivitäten innerhalb eines Systems.

  Dabei wird der logische und zeitliche Ablauf von verschiedenen elementaren Aktionen dargestellt.

  Bsp.: Gerichtbestellung mit der Aufteilung in Verantwortungsbereiche mit **Parallelisierung (Splitting)** und **Synchronisation**.



Verhaltensdiagramme 5

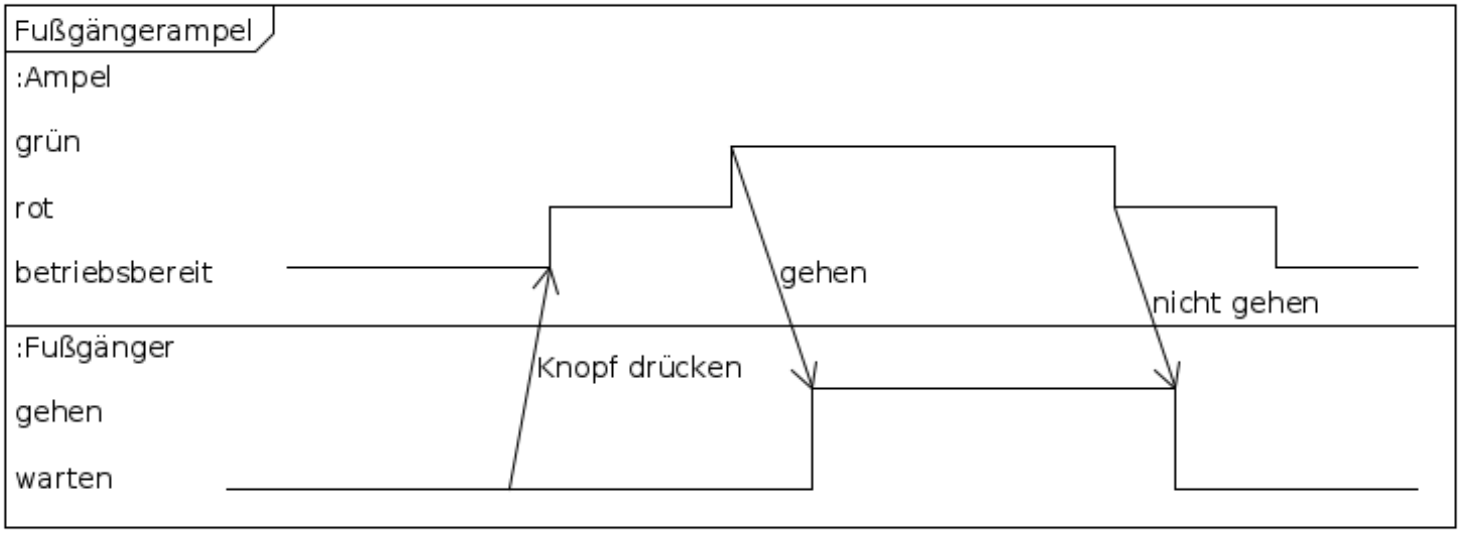
- **Kommunikationsdiagramm (communication diagram)**

  zeigt das Wechselspiel und den Nachrichtenaustausch von Teilen einer komplexen Struktur.
  Auf einer mittleren Abstraktionsebene bildet es das Zusammenspiel der Kommunikationspartner zur Lösung einer gemeinsamen Aufgabe ab.
  Es beantwortet die Frage: **Welche Teile einer komplexen Struktur arbeiten wie zusammen, um eine bestimmte Funktion zu erfüllen?**

- **Interaktionsübersichtsdiagramm (interaction overview diagram)**

  zeigt das Zusammenspiel verschiedener Aktionen auf einer höheren Ebene. Es beantwortet die Frage: **In welcher Reihenfolge und unter welchen Bedingungen finden Interaktionen statt?**

- **Zeitverlaufsdiagramm (timing-diagramm)** zeigt das zeitliche Verhalten von **Classifiern** (z.B. Klasse, Aktivität, Assoziation) in einem System. Es beantwortet die Frage: **Wann befinden sich verschiedene Interaktionspartner in welchem Zustand?**



1.4. Technische Hilfsmittel

1.4.1. Allgemeines

- An jeder der Entwicklungsphasen sind personelle Ressourcen (human resources) nötig und beteiligt.
- Zentrales Thema des Softwareentwicklungsprozesses ist deshalb auch die Kommunikation, der am Entwicklungsprozess beteiligten Personen.
- Diese erstreckt sich über alle Entwicklungsphasen und Aktivitäten hinweg.
- Es sollten deshalb Konventionen der Kommunikation vorab festgelegt werden, die eine gemeinsame Art der Kommunikation für bestimmte Aktivitäten festlegen.
- Dies geschieht auch durch die Auswahl und Festlegung technischer Hilfsmittel.
- Technische Hilfsmittel zur Unterstützung der personellen Ressourcen sollen deshalb:
  - Die Beteiligten mit aktuellen und relevanten Informationen des Softwareentwicklungsprozesses versorgen.
  - Einfach zu handhaben sein und am Besten in gewissen Punkten auch automatisierte Mechanismen zur Verfügung stellen.
  - Die nötigen Informationen möglichst ortsunabhängig (und über Zugriffsrechte geregelt) zur Verfügung zu stellen.
- Ein gewähltes technisches Hilfsmittel begleitet den Entwicklungsprozess phasenübergreifend vom Zeitpunkt des Einsatzes bis zum Ende des Entwicklungsprozesses.

Allgemeines 2

- Änderung eines technischen Hilfsmittels während eines Entwicklungsprozesses ist möglich und manchmal auch nötig (z. B. Wechsel der Entwicklungsumgebung), oft aber nicht gewünscht (z.B. Mehraufwand durch Migration der Daten).
- Ziel der Entwicklung solcher Hilfsmittel ist es, so früh wie möglich im Softwareentwicklungsprozess zum Einsatz zu kommen.
- Solche Produkte werden wie folgt bezeichnet:
  - Projektmanagement-Tools
  - Softwareentwicklungstools
  - CASE-Tools (Computer-Aided Software Engineering)
- Meist umfassen diese aber nicht alle Entwicklungsphasen, deshalb trifft man (besonders im Open Source-Entwicklungsbereich) auf eine Sammlung mehrerer technischer Hilfsmittel.

1.4.2. Werkzeuge für die Projektorganisation/-verwaltung

- **MS Project**

  Dieses Werkzeug kommt bereits ab der Planungsphase zum Einsatz. Diese Werkzeuge besitzen oft auch automatisierte Mechanismen zur Erstellung von Berichten, Analysen und Präsentationen und zur Kommunikation.

- **Office-Produkte (z. B. MS Office, LibreOffice, ...)**

  Kommen ebenfalls bereits ab der Planungsphase zum Einsatz. Die wichtigsten Komponenten sind Textverarbeitung, Präsentationssoftware, Tabellenkalkulation. Sie dienen zur Kommunikation (Präsentation, Dokumentation) in jeder Entwicklungsphase. Sie werden inzwischen teilweise durch andere Werkzeuge (Projektorganisationsssoftware, Webpräsentation, u.ä.) abgelöst oder über diese aufgerufen.

- **Internetdienste (z. B. Newsgruppen, Mailinglisten, Wiki, elektronische Foren, Bugtracking-Systeme)** Gewinnen besonders ab der Konstruktionsphase an Bedeutung. Werden meist zur Kommunikation nach außen eingesetzt als interaktive Informationsplattform für den Anwender (Kunden). So z. B. zur Produktpräsentation (Anleitungen, HOWTOs, ...) und als Feedback-Möglichkeit (Bugreports, Verbesserungsvorschläge, ...). Auch unmittelbare Beteiligte des Softwareentwicklungsprozesses nutzen diese Formen der Kommunikation untereinander (z. B. geschlossene Newsgruppen). In diesem Fall findet man diese Werkzeuge bereits in früheren Phasen der Softwareentwicklung. Erreichbarkeit der Beteiligten wird durch die Verwendung dieser Werkzeuge erhöht und Ortsunabhängigkeit hergestellt.

1.4.3. Werkzeuge für die Modellierung

- Kommerziell: Visual Paradigm, Rational Rose,
- Open Source: ArgoUML, Dia, eclipse mit Papyrus bzw. Uml2Tools, Umbrello, Violet, UMLet.
- Dieses Werkzeug kommt meist ab der Definitionsphase (u. U. auch schon in der Planungsphase) zum Einsatz, verliert aber leider manchmal in den späteren Entwicklungsphasen (Konstruktion, Auslieferung) an Bedeutung.
- Die gemeinsame Kommunikation findet hier über UML statt.
- Sie unterstützen die Diagrammerstellung, Erstellung eines Grundgerüstes für den Quellcode, und auch des **Reverse Engineering** (Erstellung von Diagrammen aus dem Quellcode).

1.4.4. Werkzeuge für die Versionsverwaltung

- Git, Subversion (SVN), Concurrent Version System (CVS) RCS, LinCVS, WinCVS.
- Sie dienen zur zentralen Ablage (**Repository**) von Quellcode und Dokumentation.
- Dieses Repository kann auch auf einem entfernten Server liegen, somit wird der Softwareentwicklungsprozess ortsunabhängig.
- Sie regeln die Zusammenarbeit von verschiedenen Entwicklern und stellen sicher, dass diese nicht gleichzeitig dieselbe Stelle des Quellcodes bearbeiten, also die Änderungen von einem Entwickler durch einen anderen Entwickler irrtümlich gelöscht werden.
- Sie speichern auch unterschiedliche Arbeitszustände der Entwicklung, die ggf.in unterschiedliche **Branches** aufgespalten werden können.
- Einzeländerungen können mit einem Kommentar in das Repository eingecheckt werden.
- Änderungen können auch leicht wieder rückgängig gemacht werden, in dem eine ältere Version ausgecheckt wird.
- Historisch bedingt kommt die Versionsverwaltung meist erst in der Konstruktionsphase zum Einsatz, jedoch wäre auch ein früherer Einsatz denkbar (Ablegen von Spezifikationen, Lastenheft, u. ä.) .
- Frontends zum Zugriff auf die Versionsverwaltung von Dateien werden bereits in Integrierte Entwicklungsumgebungen und Internetdienste eingebettet.

1.4.5. Werkzeuge für das Build-Management

- Diese Erzeugen eine neue ausführbare Version der gesamten Software und vergeben eine **Versions-Nummer** in der Form MAJOR.MINOR.PATH (z.B.: 5.4.1), die wie folgt inkrementiert wird:
  - Die MAJOR Nummer wird erhöht, wenn inkompatible API-Änderungen vorgenommen werden.
  - Die MINOR Nummer wird erhöht, wenn Funktionalität hinzukommt, das Verhalten (Schnittstellen) der Software sich aber nicht ändert.
  - Die PATH Nummer wird erhöht, wenn abwärts kompatible Bugfixes vorgenommen wurden.
- Bei Programmiersprachen, die kompiliert werden, müssen die ausführbare Version zusätzlich in einem **Artefakt-Repository** gespeichert werden, in dem auch eine Aufzeichnung (Logging) des Build-Prozesses festgehalten wird.
- Bei kleineren Projekten reicht es den Build-Prozess mit der Hilfe von Makefiles (make, qmake, cmake) oder Ant für Java oder Maven zu automatisieren.
- Bei größeren Projekten wird eine Integrierte Entwicklungsumgebung (IDE) , z. B. Eclipse, Code::Blocks, Netbeans, QtCreator, Visual Studio verwendet.
- Dieses Werkzeug kommt meist ab der Konstruktionsphase zum Einsatz.
- Die Entwicklung der IDEs geht aber bereits in Richtung Integration der Modellierung (z. B. über UML), um sie somit bereits in der Definitionsphase einsetzen zu können.

Werkzeuge für das Build-Management 2

- Die Kommunikation findet hier meist über kommentierten Code und in das Projekt integrierte Dokumentation statt (z. B. Doxygen, API-Dokumentation, CHANGELOG, ...).
- In neuerer Zeit werden auch **Web-basierte Continous Integration Systems, z.B. Jenkins**, verwendet.
  - Statt die Software nur in großen Zeitabständen (üblicherweise kurz vor der Auslieferung) zu erstellen,
  - wird sie immer wieder in kleinen Zyklen direkt in der geplanten Systemumgebung erstellt und getestet.
  - So können Integrationsprobleme früh erkannt werden.
  - Weiterhin steht zum Testen jederzeit eine lauffähige Version der Software permanent zur Verfügung.

1.4.6. Werkzeuge für die Qualitätssicherung

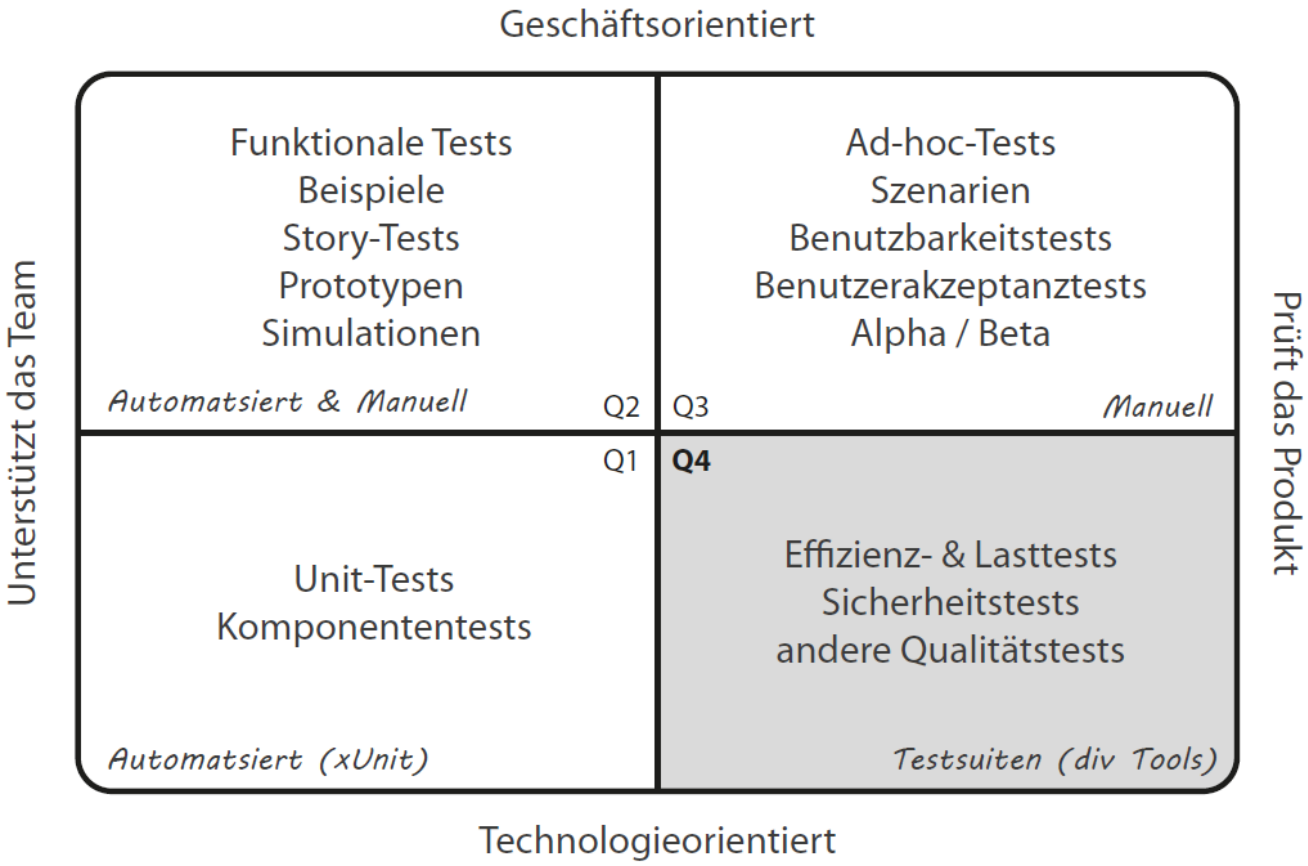
- **Kodierrichtlienien**:
  - Geben den Stil und die Formatierung des Quellcodes strikt vor, damit der Quellcode unabhängig vom Entwickler immer die gleiche Form hat und dadurch leicht zu Lesen ist.
  - Vor dem Einchecken einer Quellcode-Datei wird überprüft, ob die Kodoerrichtlinien eingehalten wurden.
  - Hierfür können Werkzeuge wie z.B. cpplint.py für C++ Programme, welches die Kodierrichtlinnien gemäß dem Google Style überprüft, verwendet werden.
- **Statische Quellcode Analyse (Linting)**:
  - Z. B. durch SonarQube, FindBugs, Scan-Build, PMD, Clang-StaticAnalyzer. Flint, cppcheck.
  - Dient zur Bestimmung der Qualität des Quellcodes nach gewissen Metriken, z. B.: Die Anzahl der Zeilen in einer Funktion/Modul.
  - Die Länge von Variablennamen und Methoden-Namen geben eine gewisse Information über die Lesbarkeit.
  - Die Anzahl Parameter von Funktionen und Methoden gibt Aufschluss über ihre Komplexität.
  - Die Anzahl der Attribute pro Klasse gibt ein Indiz für die Informationsvielfalt pro Klasse.
  - Die Tiefe von Vererbungshierarchien gibt Auskunft darüber, ob Vererbung eingesetzt wurde. Hohe Werte können für einen zu starken Vererbungseinsatz sprechen, der die Wiederverwendbarkeit erschweren kann.

Werkzeuge für die Qualitätssicherung 2

- **Dynamische Quellcode Analyse**:
  - Z. B.: durch valgrind, heob, Purify, Insure++, Dr. Memory.
  - Hat das Ziel, illegale Lese- und Speicherzugriffe auf den Speicher, z. B. durch Feldgrenzenüberschreitung oder Zugriff auf bereits freigegebene Bereiche, **Memory-Leaks**, zu entdecken.
  - Hierfür wird zur Laufzeit des Programms jeder Speicherzugriff genauestens überwacht und protokolliert.

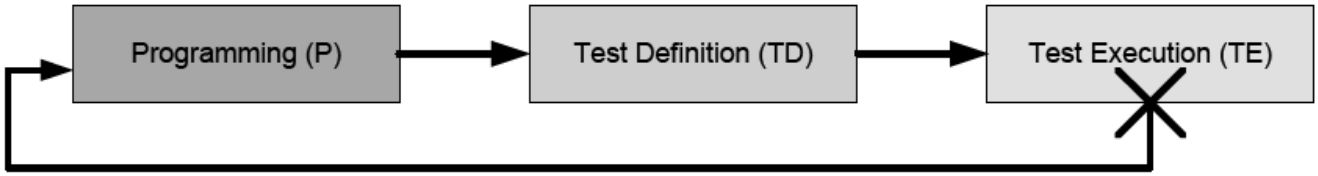
1.4.7. Werkzeuge für das Testen

- Eine naheliegende Idee ist es, für das Testen einer Software wiederum Software einzusetzen.
- Die theoretische Informatik liefert allerdings schon mit dem Beispiel des **Halteproblem** die Erkenntnis, dass es unmöglich ist ein Programm zu erstellen, welches als Eingabe ein anderes Programm (z.B. in der Form von Quellcode) erhält und feststellt, ob das eingegebene Programm nach endlich vielen Schritten terminiert.
- Deshalb kann eine Software nie total automatisiert vollständig getestet werden. Die vier Testquadranten von Crispin und Gregory geben Aufschluss darüber welche Test in der Praxis erforderlich sind (aus [Toth2015]):

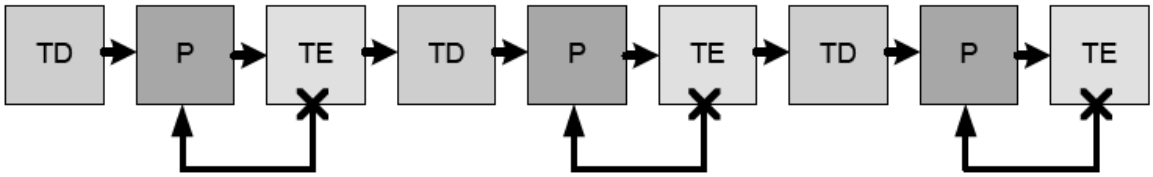


1.4.8. Werkzeuge für das Testen 2

- Die notwendigen Tests in den Quadranten Q1, Q2, und Q4 können aber durch den Einsatz von Werkzeugen z. T. automatisiert werden:
- Für die im Quadranten Q1 auszuführenden Komponententests (Unit-Tests, Modul-Tests) gibt es für viele Programmiersprachen Frameworks, mit denen sich Testfälle erstellen lassen, z.B.: JUnit, CppUnit, Boost Test Library, Qt QTestLib, Google Test und Google Mock.
- Auf diesen Frameworks basieren auch die Vorgehensmethoden Test-first-Design, Test-first-Development oder **Test-Driven-Development (TDD)**.
- Dabei werden vor der Erstellung des eigentlichen Quellcodes die Tests definiert und Testroutienen erstellt und erst danach der eigentliche Quellcode in kleinen Schritten so entwickelt, dass immer mehr der Testroutinen fehlerfrei ausgeführt werden können.



(a) Traditionelle Einbettung von Software-Testen.



(b) Test-First Development / Test-Driven Development.

1.4.8. Werkzeuge für das Testen 3

- Weiterhin existieren Werkzeuge, mit denen der **Überdeckungsgrad**, d. h. der Anteil des Quellcodes, der durch die Tests überprüft wird, bestimmt werden kann, u.B.: lcov, gcov, Testwell CTC++.
- Die funktionalen Test im Quadranten Q2 überprüfen ob die richtigen Komponenten erstellt wurden.
- In Verbindung mit den Vorgehensweisen **Behavior Driven Development (BDD)** bzw. **Specification Driven Development (SDD)** kommen Werkzeuge wie z.B. Cucumber, JBehave, Framework for Integrated Test (FIT), FitNesse, Concordion, Catch zum Einsatz, mit denen die Anforderungen in einem speziellen textuellen Format formuliert werden, um daraus automatisiert Test zu generieren.
- Für die Tests im Quadranten Q4 (Stresstests, Lasttests, Volumentests, Performance-Tests, statische und dynamische Sicherheitstest) können ebenfalls diverse Werkzeuge wie z.B. RadView, SilkPerformer, LoadRunner, Fortify verwendet werden.

1.5. Ziele der Vorlesung

- Die Vorlesung gibt einen Überblick über die wichtigsten **Techniken und Werkzeuge**, die in **großen Softwareprojekten** zum Einsatz kommen.
- Für die Problemanalyse und den Softwareentwurf wird die Verwendung von **UML** vermittelt.
- Die richtige Auswahl und Anwendung der wichtigsten **Entwurfsmuster** wird trainiert.
- Für die Implementierung wird neben den schon bekannten **prozeduralen und objektorientierten Paradigmen** auch das **generische und das funktionale Programmierparadigma** angewendet,
- Ziel ist die Erstellung von **Clean Code**.
- Die **Verwendung von Bibliotheken und Frameworks**, insbesondere das richtige **Lesen der zugehörigen Dokumentation (API)**, wird anhand von Beispielen (Serialisierung mit JSON) geübt.
- Die verwendete Programmiersprache C++ ist nebensächlich, der Schwerpunkt liegt auf den Konzepten.

1.6. Literaturhinweise

Literaturhinweise zur Programmiersprache C++

1. Wolf, Jürgen: *Grundkurs C++ : C++-Programmierung verständlich erklärt*. 3. Aufl.. Bonn: Rheinwerk Verlag GmbH, 2016. (12,90 Euro)
2. Louis, Dirk: *C++, Das komplette Starterkit für den einfachen Einstieg in die Programmierung*. M: Carl Hanser Verlag GmbH Co KG, 2018.
3. Stadler, Andreas; Tholen, Marco: *Das C++ Tutorial : Crash-Kurs und Repetitorium für Ingenieure und Naturwissenschaftler*. 1. Aufl.. Berlin Heidelberg New York: Springer-Verlag, 2018.
4. Breymann, Ulrich: *Der C++-Programmierer* , Carl Hanser Verlag GmbH, 2018.
5. Stroustrup,Bjarne: *Eine Tour durch C++*, Carl Hanser Verlag GmbH, 2015.
6. Will, Torsten: *C++ Das umfassende Handbuch* , Rheinwerk Verlag GmbH, 2018.
7. Schneeweiss, Ralf: *Moderne C++ Programmierung : Klassen, Templates, Design Patterns*. Berlin Heidelberg: Springer Science Business Media, 2006.
8. Kirch, Ulla und Prinz, Peter: *C++ - Lernen und professionell anwenden*. Heidelberg: MITP-Verlags GmbH Co. KG, 2018.
9. Kirch, Ulla: *C++ - Das Übungsbuch* , mitp Verlags GmbH, 2013.

Literaturhinweise zur Programmiersprache C++ 2

10. Meyers, Scott: *Effektives modernes C++*. Köln: O'Reilly Germany, 2015.

11. Stroustrup, Bjarne: *Die C++-Programmiersprache : aktuell zum C++11-Standard*. Carl Hanser Verlag GmbH Co KG, 2015.

12. Kuhlins, Stefan und Schrader, Martin: *Die C++-Standardbibliothek : Einführung und Nachschlagewerk* . Berlin Heidelberg New York: Springer-Verlag, 2013.

13. Alexandrescu, Andrei: *Modern C++ Design : Generic Programming and Design Patterns Applied*. Boston: Addison-Wesley Professional, 2001.

14. Cukic, Ivan: *Functional Programming in C++ : How to Improve Your C++ Programs Using Functional Techniques*. Birmingham: Manning Publications Company, 2018.

1.6.2. Literaturhinweise zu Softwaredesign

1. Starke, Gernot: *Effektive Softwarearchitekturen : Ein praktischer Leitfaden*. 7. überarbeitete. München: Hanser, Carl, 2015.
2. Toth, Stefan: *Vorgehensmuster für Softwarearchitektur : Kombinierbare Praktiken in Zeiten von Agile und Lean*. Carl Hanser Verlag GmbH Co KG, 2015.
3. Schatten, Alexander ; Biffl, Stefan ; Demolsky, Markus ; Gostischa-Franta, Erik ; Östreicher, Thomas ; Winkler, Dietmar: *Best Practice Software-Engineering : Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. 2010. Aufl.. Berlin Heidelberg New York: Springer-Verlag, 2010.
4. Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl.. Berlin Heidelberg New York: Springer-Verlag, 2011.

1.6.3. Literaturhinweise zu Entwurfsmuster

1. Eilebrecht, Karl ; Starke, Gernot ; Starke, Gernot: *Patterns kompakt : Entwurfsmuster für effektive Software-Entwicklung*. 4. Aufl.. Berlin Heidelberg New York: Springer-Verlag, 2014.
2. Freeman, Eric ; Freeman, Elisabeth ; Sierra, Kathy ; Bates, Bert: *Entwurfsmuster von Kopf bis Fuß*. 1. Aufl.. Köln: O'Reilly Germany, 2006.
3. Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. München: Pearson Deutschland GmbH, 2004.

Literaturhinweise zu UML und Qt

1.6.4. Literaturhinweise zu UML

1. Rupp, Christine, Queins Stefan, die SOPHISTen: *UML 2 glasklar : Praxiswissen für die UML-Modellierung*. Carl Hanser Verlag GmbH Co KG, 2012.
2. Kleuker, Stephan: *Grundkurs Software-Engineering mit UML: Der pragmatische Weg zu erfolgreichen Softwareprojekten*, Springer Vieweg, 2018.
3. Oestereich, Bernd: *Analyse und Design mit UML 2 : objektorientierte Softwareentwicklung* Oldenbourg, 2005.
4. Balzert, Heide ; Balzert, Helmut: *Lehrbuch der Objektmodellierung : Analyse und Entwurf mit der UML 2*. 2. Aufl. 2004. Heidelberg: Spektrum Akademischer Verlag, 2011.

1.6.5. Literaturhinweise zu Qt

1. Blanchette, Jasmin ; Summerfield, Mark: *C++-GUI-Programmierung mit Qt 4 : die offizielle Einführung*. 2. aktualisierte Auflage. München: Pearson Deutschland GmbH, 2009.
2. Wolf, Jürgen: *Qt 4 - GUI-Entwicklung mit C+ : das umfassende Handbuch ; Grundlagen, Praxis, Referenz; plattformunabhängige Anwendungen mit Qt 4*; Bonn: Galileo Press, 2007.
3. Ezust, Alan und Ezust, Paul: *Introduction to Design Patterns in C++ with Qt*, Prentice Hall, 2011.
4. Lazar, Guillaume und Penea, Robin: *Mastering Qt 5*. Birmingham: Packt Publishing Ltd, 2016.

1.6.6. Literaturhinweise zu Testverfahren

1. Langr, Jeff: *Testgetriebene Entwicklung mit C++ : Sauberer Code. Bessere Produkte*. 1. Auflage. Heidelberg: dpunkt.verlag, 2014.
2. Spillner, Andreas und Breymann, Ulrich: *Lean Testing für C++-Programmierer : Angemessen statt aufwendig testen*. Heidelberg: dpunkt.verlag, 2016.
3. Baumgartner, Manfred ; Klonk, Martin ; Pichler, Helmut ; Seidl, Richard ; Tanczos, Siegfried: *Agile Testing : Der agile Weg zur Qualität*. 2. Aufl.. M: Carl Hanser Verlag GmbH Co KG, 2018.

1.7. Die Programmiersprache C++

1.7.1. Überblick:

- C++ ist eine Weiterentwicklung der Programmiersprache C.
- C ist bis auf wenige unbedeutende Ausnahmen als Untermenge in C++ enthalten.
- Erweiterung von C um Elemente zur Realisierung der Objektorientierten Programmierung (OOP)
- und um zusätzliche nicht-objektorientierte Sprachelemente.
- C++ ist eine Kombination aus einer **prozeduralen** und einer **objektorientierten** Programmiersprache (hybride Programmiersprache).
- In den aktuellen Versionen (s.u) sind auch **funktionale** Bestandteile wie Lambdas enthalten.

1.7.2. Entwicklung der Objektorientierten Programmierung:

1967	**SIMULA**, erstmalige Definition des Begriffs *Klasse*
1974	**SMALLTALK**, entwickelt von Kay, Goldberg, Ingalls (Palo Alto Research Center): Programmierumgebung bestehend aus Betriebssystem, Sprachinterpreter und graphischer Benutzeroberfläche.
ab 1980	Erweiterung konventioneller Sprachen um **objektorientierte Elemente** z.B. **C++**, **CLOS (Common Lisp Object System)**, **Object-PASCAL** (Apple),
1995	**JAVA** (Sun Microsystems) und **PYTHON** (Open Source)
2000	**C sharp** (Microsoft)

1.7.3. Entwicklung von C++

1980 **C with Classes**, entwickelt von Bjarne Stroustrup (Bell Labs von AT&T).

1983 Änderung des Namens in **C++**.

1998 **ANSI/ISO-Standard für C++, (C++98)**

Entwicklung von C++ 2

2006 **Technischer Report 1**
Enthalten sind im TR1 u. a. **reguläre Ausdrücke**, verschiedene **intelligente Zeiger**,

ungeordnete assoziative Container, eine **Zufallszahlenbibliothek**, Hilfsmittel für die **C++-Metaprogrammierung**, **Tupel** sowie numerische und mathematische Bibliotheken.

Außerdem sind sämtliche Bibliothekserweiterungen der 1999 überarbeiteten **Programmiersprache C (C99)** in einer an C++ angepassten Form enthalten.

2011 **C++11**: Bibliotheken für Nebenläufigkeit (Threads), Mehrprozessorsysteme, Typinferenz zur Ableitung von Ergebnistypen aus Ausdrücken (`auto`) und die sogenannten **R-Value References**, die das **Verschieben** von Objekten (Move-Semantik) ermöglichen.

2014 **C++14**: generische Lambdas.

2017 **C++17**: Typ `std::byte` für byte-weisen Zugriff auf den Speicher.

1.7.4. Eigenschaften von C++

- **International Organization for Standardization (ISO) genormte** Programmiersprache.
- Weitgehend Plattform-unabhängig, da viele (Cross)Compiler existieren.
- C++ Programme sind in der Regel sehr **Laufzeit-effizient**.
- Unterstützt sowohl das **prozedurale**, als auch das **objektorientierte** und das **funktionale Programmierparadigma** sowie die **generische Programmierung**.
- C++ ist deshalb eine universell einsetzbare modulare Programmiersprache,
- die sowohl die **maschinennahe Programmierung** ermöglicht,
- als auch die Programmierung auf hohem Abstraktionsniveau.
- C++ wird deshalb verwendet für
  - **Systemprogrammierung**
    (Betriebssysteme, eingebettete Systeme, virtuelle Maschinen, Treiber, Signalprozessoren)
  - **Anwendungsprogrammierung**
    vor allem dort, wo hohe Anforderungen an die Effizienz gestellt werden
    z.B. viele CAD-Programme und Finite-Elemente-Programme und Simulations-Programme sind in C++ geschrieben.

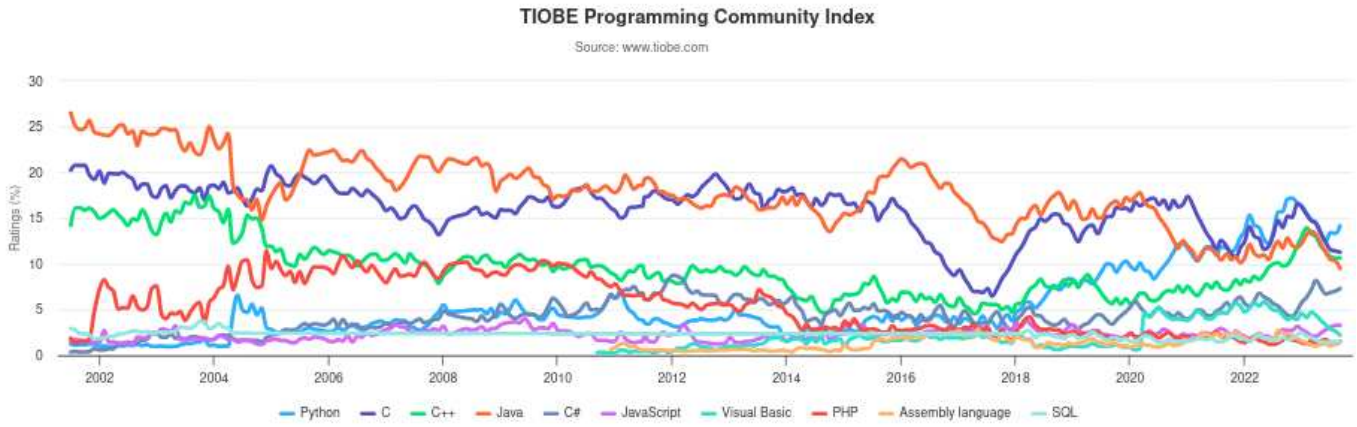
Eigenschaften von C++ 2

- Die umfangreiche **C++-Standardbibliothek** beinhaltet viele nützliche Datenstrukturen und Algorithmen.
- C++ ist sehr komplex, mächtig und manchmal fehleranfällig.
- Man muss viel lernen und üben, um es gut zu beherrschen.
- C++ hat z.B. keine Garbage Collection, der Programmierer ist selber für eine saubere Speicherverwaltung verantwortlich.
- Zitat von Bjarne Stroustrup:

  > *In C++ it is harder to shoot yourself in the foot, but when you do, you blow off your whole leg.*

- Wer C++ beherrscht kann sich ganz leicht in jede andere Programmiersprache einarbeiten.

1.7.5. Popularität von C++



Sep 2023	Sep 2022	Change	Programming Language	Ratings	Change
1	1		Python	14.16%	-1.58%
2	2		C	11.27%	-2.70%
3	4	^	C++	10.65%	+0.90%
4	3	v	Java	9.49%	-2.23%
5	5		C#	7.31%	+2.42%
6	7	^	JavaScript	3.30%	+0.48%
7	6	v	Visual Basic	2.22%	-2.18%
8	10	^	PHP	1.55%	-0.13%
9	8	v	Assembly language	1.53%	-0.96%
10	9	v	SQL	1.44%	-0.57%
11	15	^^	Fortran	1.28%	+0.26%
12	12		Go	1.19%	+0.03%
13	14	^	MATLAB	1.19%	+0.13%
14	22	^^	Scratch	1.08%	+0.51%
15	13	v	Delphi/Object Pascal	1.02%	-0.07%
16	16		Swift	1.00%	+0.02%
17	26	^^	Rust	0.97%	+0.47%
18	18		R	0.97%	+0.02%
19	20	^	Ruby	0.95%	+0.30%
20	34	^^	Kotlin	0.90%	+0.59%

Popularität von C++ 2

Vorsicht:

- Die Platzierung im Ranking ist stark davon anhängig, **wie der Index erhoben wurde**.
- Mit der Hilfe der Top 10 Auswahl kann man aber gut beurteilen, ob das eigene Wissen hinsichtlich Programmiersprachen noch aktuell ist, oder eine Fortbildung in Richtung neuerer Sprachen dringend erforderlich ist.
- Wie die Historie gezeigt hat: Programmiersprachen kommen und gehen, aber die Konzepte und Programmier-Paradigmen bleiben über die Jahre bestehen.
- Sind die Konzepte gut verstanden, ist es sehr leicht sich in die Syntax einer neuen Sprache einzuarbeiten.
- So fehlt z. B. aktuell in den Studiengängen die Ausbildung hinsichtlich Java.
- Wer aber die Konzepte basierend auf C++ verinnerlicht hat, kann sich innerhalb eines Tages in Java einarbeiten.
- Der unten angegebene **PYPL PopularitY of Programming Language Index** (wie oft wurde auf google nach Tutorials für eine bestimmte Programmiersprache gesucht)

  (`https://pypl.github.io/PYPL.html`)

  zeigt eine andere Platzierung, aber die Menge der Top 10 ist ähnlich zum TIOBE Index.

Die Popularität von C++ 3

PYPL PopularitY of Programming Language Index

(https://pypl.github.io/PYPL.html)

Worldwide, Oct 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	28.05 %	+0.1 %
2		Java	15.88 %	-1.0 %
3		JavaScript	9.27 %	-0.3 %
4		C#	6.79 %	-0.2 %
5		C/C++	6.59 %	+0.3 %
6		PHP	4.86 %	-0.4 %
7		R	4.45 %	+0.4 %
8		TypeScript	2.93 %	+0.1 %
9	↑↑	Swift	2.69 %	+0.7 %
10	↓	Objective-C	2.29 %	+0.2 %
11	↑↑	Rust	2.05 %	+0.4 %
12	↓↓	Go	1.95 %	-0.1 %
13	↓	Kotlin	1.76 %	-0.1 %
14		Matlab	1.53 %	+0.1 %
15		Ruby	1.04 %	-0.1 %
16	↑↑↑	Ada	1.04 %	+0.2 %
17	↓	VBA	0.94 %	-0.1 %
18		Dart	0.9 %	+0.1 %
19	↓↓	Powershell	0.89 %	-0.0 %
20		Scala	0.69 %	-0.0 %

1.7.6. Laufzeit-Performance von C++

Vergleich z.B. auf der Web-Seite:

`https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

N-Körper-Simulation:

n-body

source	secs	mem	gz	busy	cpu load
C++ g++	**7.70**	1,772	1879	8.02	4% 0% 0% 100%
Java	22.67	34,892	1429	23.01	1% 0% 0% 100%

n-body

source	secs	mem	gz	busy	cpu load
Python 3	865.18	8,176	1196	874.96	2% 20% 79% 0%
Java	22.67	34,892	1429	23.01	1% 0% 0% 100%

n-body

source	secs	mem	gz	busy	cpu load
Python 3	865.18	8,176	1196	874.96	2% 20% 79% 0%
C gcc	7.30	8	1391	7.33	0% 0% 1% 100%

Laufzeit-Performance von C++ 2

Vergleich z.B. auf der Web-Seite:

`https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

Mandelbrot-Mange:

mandelbrot

source	secs	mem	gz	busy	cpu load
C++ g++	**1.51**	25,708	1791	6.01	99% 99% 99% 100%
Java	5.93	76,272	796	23.28	97% 99% 98% 98%

mandelbrot

source	secs	mem	gz	busy	cpu load
Python 3	259.50	48,192	688	1,036.70	100% 100% 100% 100%
Java	5.93	76,272	796	23.28	97% 99% 98% 98%

mandelbrot

source	secs	mem	gz	busy	cpu load
Python 3	259.50	48,192	688	1,036.70	100% 100% 100% 100%
C gcc	1.64	27,016	1135	6.54	100% 99% 100% 100%

1.8. Aufgabe TDD am Bsp. von Soundex

- In dieser Aufgabe wird die testgetriebene Entwicklung, *Test Driven Development (TDD)* am Beispiel des Soundex-Algorithmus vorgestellt.
- Der **Soundex-Algorithmus** ordnet englischen Wörtern, die ähnlich klingen, das gleiche Soundex-Codewort zu.
- Er wird unter anderem in vielen Anwendungen für die Suche eingesetzt, um auch bei Schreibfehlern Ergebnisse anzuzeigen.
- Wir beschränken uns hier auf eine vereinfachte Version für englische Wörter.
- Er kann aber auch für andere Sprachen angepasst werden.
- Ziel der Aufgabe ist es, Ihnen das Vorgehen von TDD zu demonstrieren:
  - Die Anforderungen werden am Anfang nur sehr grob spezifiziert.
  - Die genaue Spezifizierung erfolgt durch die Vorgabe von vielen kleinen Tests.
  - Die Implementierung erfolgt iterativ in kleiner Schritten.
  - Es wird immer nur soviel Programm-Code erstellt, um den nächsten Test zu bestehen.
  - Nach dem Bestehen eines Testes erfolgen ggf. Aufräum-Arbeiten (Refactoring).

1.8.1. Soundex-Algorithmus

1. Der Soundex-Algorithmus ordnet jedem (englischen) Wort ein Soundex-Codewort zu.
2. Jedes Soundex-Codewort besteht aus einem Großbuchstaben gefolgt von drei Ziffern.
3. Der Anfangs-Großbuchstabe entspricht dem Anfangs-Buchstaben des Wortes, der ggf. in einen Großbuchstaben gewandelt werden muss.
4. Dem Anfangsbuchstaben folgende Vokale `A, E, I, O, U`, die Buchstaben `Y, H, W` und andere Zeichen werden verworfen.
5. Konsonanten, die dem Anfangsbuchstaben folgen, werden durch folgende Soundex-Ziffern ersetzt:

Konsonanten	Soundex-Ziffer
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

6. Wenn zwei angrenzende Buchstaben in die selbe Soundex-Ziffer umgewandelt werden, erscheint diese Soundex-Ziffer nur einmal.
7. Die Umwandlung endet, wenn das Soundex-Codewort genau die Länge 4 hat.
8. Hat das Wort zu wenige Zeichen, dann werden die fehlenden Stellen des Soundex-Codewortes mit Nullen aufgefüllt.

1.8.2. TDD Lösung

- Im Moodle-Kurs finden Sie im Verzeichnis für Kapitel 1 das Qt-Test-Projekt `TestSoundex` als zip-Datei gepackt.
- Nach der Installation des Qt-Creators und Google-Test entpacken Sie das Projekt und öffnen Sie es mit dem Qt-Creator (durch öffnen der Datei `TestSoundex.pro`).
- Fügen Sie nun dem Projekt eine `cpp`-Datei `soundex.cpp` und eine zugehörige Header-Datei `soundex.h` hinzu.
- Kompilieren Sie das Projekt und führen es danach aus. Wie man sieht werden alle Tests nicht bestanden.
- Fügen Sie nun jeweils nur soviel Codezeilen in ANSI-C (das Sie ja sehr gut beherrschen und das auch eine Teilmenge von C++ ist) hinzu, um jeweils den nächsten Test zu bestehen.
- Die Aufgabe ist erst gelöst, wenn alle Tests bestanden werden.

2. Unterschiede zwischen C++ und C

2.1. Hello World Programm

```
//-----------------------
// helloWorld.cpp
// Copyright 03.10.2019
// Author: KJR
//-----------------------

/* Hello World Programm
   in C++              */

#include <iostream>

using namespace std;

int main(void){

  cout << "Hello World\n";
  cout << "Ich mag C++ sehr" << endl;
  return 0;
}
```

Kommentare wie in ANSI C:

- Einzeiliger Kommentar wird eingeleitet durch `//` und gilt bis zum Zeilenende.
- Block-Kommentar wird eingeleitet durch `/*` und beendet durch `*/`, darf nicht verschachtelt werden.

Hello World Programm 2

```
#include <iostream>

using namespace std;

int main(void){

  cout << "Hello World\n";
  cout << "Ich mag C++ sehr" << endl;
  return 0;
}
```

Inkludieren von Header-Dateien:

- Header-Dateien der C++ Standard-Bibliothek werden in die spitzen Klammern < und > eingeschlossen aber **ohne die Extension** `.h` anzugeben.
- Der Präprozessor generiert daraus

  ```
  namespace std{
    #include <iostream.h>
  }
  ```

  und bewirkt, dass alle Elemente der C++ Standard-Bibliothek sich im **Namensraum (namespace)** `std` befinden.
- Aus Kompatibilitätsgründen können auch Header-Dateien der C++ Standard-Bibliothek mit Extension eingebunden werden **(sehr schlechter Programmierstil)**:

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

  Dann befinden sich aber die inkludierten Elemente im **globalen Namensraum** und nicht im üblichen Namensraum `std`.

2.2. Namensbereiche (namespace)

- In C++ gelten Gültigkeits und Sichtbarkeitsbereiche für Namen.
- Namen sind nur **nach der Deklaration und nur innerhalb des Blockes gültig, in dem sie deklariert wurden**.
- Durch die Verwendung von gleichen Namen z.B. in Bibliotheken kann es zu Namenskonflikten kommen.
- Deshalb ermöglicht C++ die Bildung von **Namensbereichen (Namensräumen)**:

```
namespace nameFuerNamensraum{

  // Deklarationen, z.B.
  double preis;
}
```

- Auf die Elemente eines Namensraumes kann dann nur noch über den **voll qualifizierten Namen** mittels **Scope-Operator** zugegriffen werden, z.B.:

```
nameFuerNamensraum::preis = 42.0;
```

- Falls man absolut sicher ist dass es zu keinen Namenskonflikten kommt, kann man mit der Anweisung

```
using namespace nameFuerNamensraum;
```

  alle Elemente eines Namensraumes bekannt machen.

  Auf sie kann dann direkt über ihren Namen ohne Scope-Operator zugegriffen werden.

  Dies gilt aber als sehr schlechter Programmierstil!!

- Viel sicherer ist es nur einzelne Elemente mit einer `using` Anweisung bekannt zu machen:

```
using nameFuerNamensraum::preis;
```

Bsp. Namensbereiche

```
#include <iostream>

// Variable im globalen Namensraum
char message[] = "Guten Tag";

// Funktion im globalen Namensraum
void gruessen(){
  std::cout << "Endlich C++" << std::endl;
}

// Definition eines eigenen Namensraumes
// (Gueltigkeitsbereiches)
namespace gruss {
  char message[] = "Wirklich sehr!!";

  void gruessen(){
    std::cout << "Ich mag C++" << std::endl;
  }
}

// Hauptprogramm main
int main(void) {
  std::cout << ::message << std::endl;
  std::cout << "Hello World" << std::endl;
  gruessen();
  gruss::gruessen();
  std::cout << gruss::message << std::endl;
  return 0;
}
```

Aufgabe: Was gibt das Programm aus? Man überprüfe dies am Rechner!

Aufgabe Namensbereiche

Was wird ausgegeben? Man überprüfe dies am Rechner!!

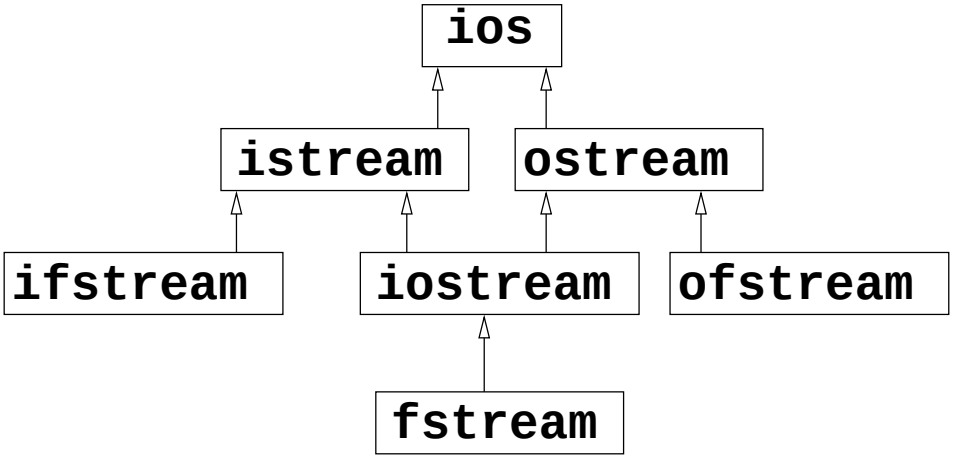
```
#include <iostream>
using std::cout;
using std::endl;
int i = 100;
namespace A {
  int i = 13;
  void print(){
    cout << "in A\n";
  }
  namespace B {
    int i = 42;
    void print() {
      cout << "in B\n";
    }
  }
}
int main(){
  cout << i << endl;
  int i = 1000;
  cout << i << endl << ::i << endl;
  {
    int i = 999;
    cout << i << endl;
    cout << A::i << endl;
    cout << A::B::i << endl;
  }
  A::print();
  A::B::print();
  return 0;
}
```

2.3. Ein/Ausgabe mit Streams

- In C++ erfolgt die Ein/Ausgabe **einheitlich und objektorientiert** mittels **Streams (Datenflüsse, Datenrutschen)**.
- In einem Stream fließen die Daten von einem Eingang zu einem Ausgang.
- Als Eingang bzw. Ausgang können ganz unterschiedliche Objekte verwendet werden, z.B. Tastatur (Standard-Eingabe), Bildschirm (Standard-Ausgabe), Dateien, Zeichenketten, Netzwerksockets, etc..
- Alle Elemente der C++-Standardbibliothek befinden sich im Namensraum `std`.
- Durch das Inkludieren der Header-Datei `<iostream>` sind die folgenden Ein-/Ausgabeströme vordefiniert:

`cin`	**Standard-Eingabestrom (Tastatur)**, gepuffert (in C `stdin`)
`cout`	**Standard-Ausgabestrom (Bildschirm)**, gepuffert (in C `stdout`)
`cerr`	Strom für **ungepufferte Fehlerausgabe** (in C `stderr`)
`clog`	Strom für **gepufferte Fehlerausgabe**

2.3.1. Stream-Klassen in C++



- Die Klasse `ios` ist die Basisklasse aller C++ Stream-Klassen.
- In ihr sind alle gemeinsamen Eigenschaften der C++ Streams enthalten:
  Verwaltung der Verbindung zum Datenstrom, Fehlerflags, Formatierung der Daten.
- Die Eingabeströme sind Objekte der vordefinierten Klasse `istream` (der Anfangsbuchstabe `i` steht für Input)
- Die Ausgabeströme sind Objekte der vordefinierten Klasse `ostream` (der Anfangsbuchstabe `o` steht für Output)
- Bidirektionale Ströme sind Objekte der Klasse `iostream`.
  In der Header-Datei `<iostream>` werden alle grundlegenden Stream-Klassen (auch die Standard-Streams `cin`, `cout`, `cerr`, `clog`) definiert bis auf die File-Stream-Klasse und die String-Stream-Klasse.
- Die File-Streams werden in der Klasse `fstream` definiert, die Header-Datei `<fstream>` bindet automatisch die Header-Datei `<iostream>` mit ein.
- Die String-Streams werden in der Klasse `strstream` definiert (nicht abgebildet).

2.3.2. Werte Ausgeben

Einzelne Ausgabe:

```
Ausgabestrom << Ausdruck
```

- *Bildlich:* Wert des Ausdruckes wird in den Ausgabestrom geschoben.
- Da obiger Ausdruck als Wert eine Referenz (s.u.) auf den Ausgabestrom (z.B. `cout`) wieder zurückliefert und der Operator << links-assoziativ ist (Auswertung erfolgt von links nach rechts), lassen sich auch mehrere Werte durch Verkettung ausgeben:

Verketteter Aufruf des Ausgabeoperators:

```
Ausgabestrom << Ausdruck1 << Ausdruck2 << Ausdruck3;
```

Zeilenwechsel innerhalb einer Ausgabe:

- mittels Escape-Sequenz `'\n'`, z.B.:

  ```
  cout << "Zeile1\nZeile2";
  ```

- oder durch sog. **Manipulator** `endl`, der auch die Ausgabe aller gepufferter Zeichen bewirkt, z.B.:

  ```
  cout << "Zeile1" << endl << "Zeile2" << endl;
  ```

2.3.3. Werte Einlesen

```
Eingabestrom >> variable;
```

- Liest Wert vom Eingabestrom (z.B. `cin`) ein und speichert ihn in der Variablen `variable`.
- *Bildlich:* Wert wird vom Eingabestrom in die Variable geschoben
- Typ von `variable` bestimmt erwartete Eingabe.
- Bei unpassender Eingabe werden ggf. Fehlerflags im Eingabestrom-Objekt gesetzt, Pufferinhalt bleibt bestehen
- Wiederaufsetzen nach unzulässigen Eingaben, z.B. für Eingabestrom `cin`:
  - Eingabepuffer löschen mit `cin.sync();`
  - Fehlerflags zurücksetzen mit `cin.clear();`
  - maximal `n` Eingabezeichen bis zum Zeichen `c` ignorieren mit `cin.ignore(n, c);`
- Da obiger Ausdruck als Wert eine Referenz auf den Eingabestrom (z.B. `cin`) zurück liefert und der Operator `>>` links-assoziativ ist, lassen sich durch Verkettung auch die Werte für mehrere Variable einlesen: `cin >> var1 >> var2 ... >> varn;`
- Wo erforderlich wird der von einem Ein- bzw Ausgabe-Ausdruck zurückgelieferte Wert implizit in einen `void*` bzw. `int`-Wert (als Status) umgewandelt.

  Dieser Wert ist `NULL` (bzw `0`), wenn das Dateiende erreicht ist bzw. ein Fehler aufgetreten ist.

  Dies erlaubt Überprüfungen, wie z.B.:

```
while (cin >> wert) { /* tu was */ }
```

Bsp. Ein/Ausgabe

```
#include <iostream>
#include <cmath>
using std::cout;
using std::cin;
using std::endl;
int main() {
  double radius, umfang;
  char anfangVorname, anfangNachname;

  // Umfang-Berechnung
  cout << "Eingabe Radius:";
  cin >> radius;
  umfang = 2 * M_PI * radius;
  cout << "Der Kreis hat den Umfang: "
       << umfang << endl;

  // Anfangsbuchstaben von Vorname
  // und Nachname
  cout << "Eingabe Vorname blank Nachname:";
  cin >> anfangVorname;
  cin.ignore(20, ' ');
  cin >> anfangNachname;
  cout << "Ihre Initialien lauten: "
       << anfangVorname
       << " und "
       << anfangNachname
       << endl;
  return 0;
}
```

2.3.4. Ein/Ausgabe im C-Stil

- Da C als Teilmenge in C++ enthalten ist, ist aus Kompatibilitätsgründen auch in C++ die Ein/Ausgabe im C-Stil möglich **(aber sehr schlechter Programmierstil)**.
- Die aus C stammenden Header-Dateien erhalten in C++ die Vorsilbe `c`, z.B. `<cstdio>`.

Beispiel:

```
#include <cstdio>

using std::printf;
using std::scanf;

int main() {
  int gluecksZahl;
  printf("Eingabe einer ganzen Zahl:");
  scanf("%d", &gluecksZahl);
  printf("Gleuckszahl lautet %d\n", gluecksZahl);
  return 0;
}
```

Aufgaben:

a) Man schreibe das oben angegebene *Hello World Programm* so um, dass **keine einzige** `using` Anweisung verwendet wird.

b) Man schreibe das obige Programm zur Eingabe einer Glückszahl so um, dass nur Streams verwendet werden.

c) Man erstelle ein Programm, das sowohl mit Streams als auch im C-Stil eine Gleitpunktzahl vom Typ `double` einliest und diese wieder (im C-Stil mit 5 Nachkommastellen) ausgibt.

2.3.5. Formatierte Ausgabe mit Flags

- In welcher Form Werte ausgegeben werden wird durch **Flags (Schalter)** in der Basisklasse `ios` gesteuert.
- Jedes Flag besitzt eine Standardeinstellung.
- Flags sind entweder unabhängige Bits oder einzelne Bits innerhalb einer ganzzahligen Variable (Bitfeld) vom Typ `ios::fmtflags`

Formatflag	Bedeutung	Bitfeld
`left`	linksbündige Ausgabe	`adjustfield`
`right`	rechtssbündige Ausgabe	`adjustfield`
`internal`	Vorz. links-, Wert rechtsbündig	`adjustfield`
`dec`	ganzzahlige Aus- bzw Eingabe dezimal	`basefield`
`oct`	ganzzahlige Aus- bzw Eingabe oktal	`basefield`
`hex`	ganzzahlige Aus- bzw Eingabe hexadezimal	`basefield`
`showbase`	Ausgabe mit Zahlensystem-Kennung (0 für oktal, 0x für hexadezimal)	
`showpoint`	immer Ausgabe des Dezimalpunktes und abschließenden Nullen (bei Gleitpunktzahlen)	
`showpos`	explizite Ausgabe eines pos. Vorzeichens	
`uppercase`	Ausgabe von Großbuchstaben bei hexadezimalen Zahlen und Gleitpunktzahlen in Exponentialdarstellung	
`fixed`	Ausgabe von Gleitpunktzahlen in Festpunktdarstellung	`floatfield`
`scientific`	Ausgabe von Gleitpunktzahlen in Exponentialdarstellung	`floatfield`
`skipws`	Überlesen von führenden Whitespaces	
`unitbuf`	Leeren des Puffers nach jeder Ausgabe	
`boolalpha`	Ausgabe von Bool-Werten als Text, ansonsten als 0/1	
`stdio`	Flush `cout`, `cerr` nach jeder Ausgabe	

Formatierte Ausgabe mit Flags 2

Memberfunktionen der Klasse ios zum Ermitteln bzw. Setzen der Formatflags

```
ios::fmtflags flags(void) const
```

liefert die Formatflags als Rückgabewert zurück.

```
ios::fmtflags flags(ios::fmtflags bits)
```

- setzt alle Formatflags auf die im Parameter `bits` enthaltenen Werte
- die alten Formatflags werden überschrieben
- gibt die alten Formatflags als Rückgabewert zurück

```
ios::fmtflags setf(ios::fmtflags onbits)
```

- setzt die im Parameter `onbits` gesetzten Formatflags
- die übrigen Formatflags werden nicht beeinflußt
- gibt die alten Formatflags als Funktionswert zurück

Formatierte Ausgabe mit Flags 3

Memberfunktionen der Klasse ios zum Ermitteln bzw. Setzen der Formatflags

```
ios::fmtflags setf(ios::fmtflags onbits,
                   ios::fmtflags mask)
```

- setzt die durch den 2. Parameter `mask` festgelegten Formatflags (üblicherweise ein Bitfeld) zurück und setzt anschließend die im 1. Parameter `onbits` gesetzten Formatflags.
- Typische Anwendung: Setzen von Formatflags innerhalb eines Bitfelds unter gleichzeitigem Rücksetzen der übrigen Flags dieses Bitfelds (innerhalb eines der definierten Bitfelder darf immer nur 1 Flag gesetzt sein)
- die übrigen Formatflags (außerhalb des durch `mask` festgelegten Bitfelds) werden nicht beeinflußt.
- gibt die alten Formatflags als Rückgabewert zurück

```
ios::fmtflags unsetf(ios::fmtflags offbits)
```

- setzt die im Parameter `offbits` gesetzten Formatflags zurück
- die übrigen Formatflags werden nicht beeinflußt
- gibt die alten Formatflags als Rückgabewert zurück

Mit der Hilfe des bitweisen ODER-Operators | können einzelne Flags kombiniert werden:

```
cout.setf(std::ios::showbase | std::ios::uppercase);
```

Bsp Formatierte Ausgabe mit Flags

```
#include <iostream>

using std::cout;
using std::clog;
using std::endl;
using std::ios;

int main(){

  // Fuer ganze Zahlen
  int i = 255;
  // Speichere aktuellen Status der Format-Flags
  ios::fmtflags myFlags = cout.flags();
  clog << "myFlags: " << myFlags << endl;
  // myFlags: 4098

  // Umschalten auf hexadezimale Ausgabe
  cout.setf(ios::hex, ios::basefield);

  // Umschalten auf Ausgabe der Zahlensystem-Kennung
  cout.setf(ios::showbase);
  cout << "i = " << i << endl;
  // i = 0xff


  // Ausgabe des aktuellen Status der Format-Flags
  clog << "myFlags: " << cout.flags() << endl;
  // myFlags: 4616

  // Umschalten auf Gorssbuchstaben
  cout.setf(ios::uppercase);
  cout << "i = " << i << endl;
  // i = 0XFF

  // Ausgabe des aktuellen Status der Format-Flags
  clog << "myFlags: " << cout.flags() << endl;
  // myFlags: 21000
```

Bsp Formatierte Ausgabe mit Flags 2

```
// Zurueckcksetzen der Ausgabe der Zahlensystem-Kennung
cout.unsetf(ios::showbase);

// Umschalten auf dezimale Ausgabe
cout.setf(ios::dec, ios::basefield);
cout << "i = " << i << endl;
// i = 255

// Fuer Gleitpunktzahlen
double x = 4712.123456;

cout << "x = " << x << endl;
// x = 4712.12
// Umschalten auf Wissenschaftliche Darstellung
cout.setf(ios::scientific, ios::floatfield);
cout << "x = " << x << endl;
// x = 4.712123E+03

// Ausgabe des aktuellen Status der Format-Flags
clog << "myFlags: " << cout.flags() << endl;
// myFlags: 20738

// Setzen der Feld-Breite
cout << "x = ";
cout.width(20); // gilt nur fuer die naechste Ausgabe
                // unschoen, die Ausgabe-Kette muss
                // unterbrochen werden
                // deshalb bessere Loesung:
                // Manipulatoren (s.u.)

clog << "Aktuele Feldbreite: " << cout.width() << endl;
// Aktuele Feldbreite: 20
cout << x << endl;
// x =         4.712123E+03
```

Bsp Formatierte Ausgabe mit Flags 3

```
  // Linksbuendige Ausgabe:
  cout.setf(ios::left, ios::adjustfield);
  cout << "x = ";
  cout.width(20);
  cout << x << endl;
  //x = 4.712123E+03

  // Setzen der Genauigkeit auf 3 Nachkommastellen
  cout.precision(3);
  cout << "x = ";
  cout.width(20);
  cout << x << endl;
  // x = 4.712E+03

  // Zureucksetzen auf Rechtsbuendige Ausgabe
  cout.setf(ios::right, ios::adjustfield);

  // Setze ? als Fuellzeichen
  cout.fill('?');
  cout << "x = ";
  cout.width(20);
  cout << x << endl;
  // x = ??????????4.712E+03
  return 0;
}
```

2.3.6. Formatierte Ausgabe mit Manipulatoren

- Die Formatflags können auch (einfacher) mittels Manipulatoren gesetzt werden.
- Manipulatoren sind spezielle I/O-Operanden, die aber wie normale I/O-Operanden verwendet werden.
- Verkettete I/O-Ausrücke müssen zur Formatierung deshalb nicht unterbrochen wreden.
- Für die Verwendung der **Manipulatoren mit Parameter** muss die Header-Datei `<iomanip>` inkludiert werden.

Name	Anwendung für	Wirkung
`setw(int w)`	Ein-/Ausgabe	Setzen der Feldbreite auf `w` Stellen (entspricht `width()`)
`setprecision(int p)`	Ausgabe	Setzen der Genauigkeit auf `p` Stellen (entspricht `precision()`)
`setfill(int c)`	Ausgabe	Setzen des Füllzeichens auf das Zeichen `c` (entspricht `fill()`)
`setbase(int b)`	Ausgabe	Setzen der Basis des Zahlensystems auf `b` (nur für b = 8, 10, 16)
`setiosflags(fmtflags f)`	Ein-/Ausgabe	Setzen der in `f` spezifizierten Formatflags (entspricht `setf()`)
`resetiosflags(fmtflags f)`	Ein-/Ausgabe	Rücksetzen der in `f` spezifizierten Formatflags (entspricht `unsetf()`)

Formatierte Ausgabe mit Manipulatoren 2

Die **parameterlosen Manipulatoren** sind in der Header-Datei `<iostream>` deklariert.

Name	Anwendung für	Wirkung
`endl`	Ausgabe	Newline ausgeben und Puffer leeren
`ends`	Ausgabe	NULL-Char. ausgeben und Puffer leeren
`flush`	Ausgabe	Puffer leeren
`ws`	Eingabe	Whitespace-Character überlesen
`dec`	Ein-/ Ausgabe	dezimale Darstellung ganzer Zahlen
`hex`	Ein-/ Ausgabe	hexadezimale Darstellung ganzer Zahlen
`oct`	Ein-/ Ausgabe	oktale Darstellung ganzer Zahlen
`left`	Ausgabe	linksbündige Ausgabe einstellen
`right`	Ausgabe	rechtsbündige Ausgabe einstellen
`internal`	Ausgabe	VZ linksbündig, Wert rechtsbündig
`fixed`	Ausgabe	Dezimalbruchdarstellung einstellen
`scientific`	Ausgabe	Exponentialdarstellung einstellen
`boolalpha`	Ausgabe	`ios::boolalpha` setzen
`noboolalpha`	Ausgabe	`ios::boolalpha` rücksetzen
`showpos`	Ausgabe	`ios::showpos` setzen
`noshowpos`	Ausgabe	`ios::showpos rücksetzen`
`uppercase`	Ausgabe	`ios::uppercase` setzen
`nouppercase`	Ausgabe	`ios::uppercase` rücksetzen
`showbase`	Ausgabe	`ios::showbase` setzen
`noshowbase`	Ausgabe	`ios::showbase` rücksetzen
`showpoint`	Ausgabe	`ios::showpoint` setzen
`noshowpoint`	Ausgabe	`ios::showpoint` rücksetzen
`skipws`	Eingabe	`ios::skipws` setzen
`noskipws`	Eingabe	`ios::skipws` rücksetzen

Bsp. Formatierte Ausgabe mit Manipulatoren

```
#include <iostream>
#include <iomanip> // fuer Manipulatoren
                   // mit Parameter

using std::cout;
using std::endl;

int main(){

  // Fuer ganze Zahlen
  int i = 255;

  // Umschalten auf hexadezimale Ausgabe
  // Umschalten auf Ausgabe der Zahlensystem-Kennung
  cout << "i = " << std::hex << std::showbase << i << endl;
  // i = 0xff

  // Umschalten auf Gorssbuchstaben
  cout << "i = " << std::uppercase << i << endl;
  // i = 0XFF

  // Umschalten auf dezimale Ausgabe
  cout << "i = " << std::dec << i << endl;
  // i = 255

  // Fuer Gleitpunktzahlen
  double x = 4712.123456;
  cout << "x = " << x << endl;
  x = 4712.12

  // Umschalten auf Wissenschaftliche Darstellung
  cout << "x = " << std::scientific << x << endl;
  // x = 4.712123E+03
```

Bsp. Formatierte Ausgabe mit Manipulatoren 2

```
  // Setzen der Feld-Breite
  cout << "x = " << std::setw(20) << x << endl;
  // x =         4.712123E+03

  // Linksbuendige Ausgabe:
  cout << "x = " << std::setw(20)
       << std::left << x << endl;
  // x = 4.712123E+03

  // Setzen der Genauigkeit auf 3 Nachkommastellen
  cout << "x = " << std::setw(20)
       << std::setprecision(3) << x << endl;
  // x = 4.712E+03

  // Zureucksetzen auf Rechtsbuendige Ausgabe
  // Setze ? als Fuellzeichen
  cout << "x = " << std::setw(20) << std::right
       << std::setfill('?')
       << x << endl;
  // x = ??????????4.712E+03
  return 0;
}
```

Aufgabe zur Eingabe und formatierten Ausgabe

- Es sollen maximal `N = 5` Nettopreise eingelesen werden.
- Die eingelesenen Nettopreise und die zu berechnenden Bruttopreise sind in geeigneten Feldern zu speichern.
- Im C++11 Stil kann eine Konstante wie folgt deklariert werden:

```
constexpr int N = 5;
```

- In Tabellenform sollen mit Hilfe von Manipulatoren die Nummer des Preises (Feldbreite 4, linksbündig), der Nettopreis (Feldbreite 8, 2 Nachkommastellen, rechtsbündig) und der Bruttopreis (Feldbreite 10, 2 Nachkommastellen, rechtsbündig) ausgegeben werden.

Aufgabe zur Parameter-Übergabe

1) Was wird ausgegeben? Warum?

```
#include <iostream>
using std::cout;
using std::endl;
void swapF(int x, int y){
  int hilf = x;
  x = y;
  y = hilf;
}
int main(){
  int i = 5, j = 8;
  swapF(i, j);
  cout << "i = " << i << ", j = " << j << endl;
  return 0;
}
```

2) Was wird ausgegeben? Warum?

```
#include <iostream>
using std::cout;
using std::endl;
int main(void) {
  int var1 = 42;
  int* ptr;
  ptr = &var1;
  *ptr = 13;
  cout << "Wert von var1:" << var1 << endl;
  int var2 = 999;
  ptr = &var2;
  cout << "Wert von ptr:" << ptr << endl;
  cout << "Wert von *ptr:" << *ptr << endl;
  return 0;
}
```

3) Korrektur mittels Pointer

Man korrigiere die obige falsche `swapF` Funktion, indem man mit **Pointern** eine richtige Funktion `swapP` erstellt.

2.4. Referenzen

```
typ& referenzname = objektname;
```

Bsp.:

```
int i = 2;
int& ri = i;
ri = 42;
```

- Eine als **Referenz** deklarierte Variable ist ein zweiter Name (Alias), der für ein existierendes Objekt (Initialisierer) vergeben wird. **Sie muss bei der Deklaration initialisiert werden**.
- *Vorsicht:* Im Fall der Deklaration einer Referenz steht der Operator `&` **links** vom Zuweisungs-Operator. Wird der Operator `&` als **Adressoperator** verwendet steht er in der Regel **rechts** vom Zuweisungsoperator (z.B.: `int* pi = &i;`)
- Die Deklaration einer Referenz erzeugt kein neues Objekt!
- Es wird kein eigener Speicher für die Referenz reserviert.
- Von einer Referenz kann keine Adresse mit dem Adress-Operator ermittelt werden.
- Es gibt keine Zeiger (Pointer) die auf Referenz-Typen zeigen.
- Das Objekt (Initialisierer), für das die Referenz einen zweiten Namen definiert, muss bei der Deklaration angegeben werden und muss ein **L-Value** (Wert der einen eigenen Speicher hat und links vom Gleichheitszeichen stehen kann) sein. Es kann nicht gewechselt werden.
- Eine Zuweisung der Referenz entspricht danach einer Zuweisung des Objektes, auf das die Referenz verweist.

Referenzen 2

- Der Typ der Referenz muss zum Typ des Objektes passen, auf das die Referenz verweist.

```
int i = 2;
double& rd = i; // Fehler!!
```

- Bei einer Referenz auf eine Konstante darf der Initialisierer auch ein Ausdruck sein, der kein L-Value ist.

  Es wird in diesem Fall ein tempräres Objekt erzeugt und mit dem Wert des Initialisiers initialisiert.

```
const double& rd = 1.5;
void f(int p) {
  // ri ist Name fuer temporaeres Objekt
  const int& ri = 2*p;
  // .....
}
++rd;  // Fehler da const!!
```

- Bsp.: Referenzen auf Felder (Arrays):

```
char cfeld[] = "Hello World";
char (&rf)[sizeof(cfeld)] = cfeld; // rf ist
                                   // weiterer Name fuer cfeld
                                   // Klammerung ist notwendig

char& rc = cfeld[2];  // rc ist weiterer Name fuer
                      // die Speichestelle cfeld[2]

char* pc = rf;
char*& prc = pc;    // pc und prc sind Namen fuer den
                    // selben char Pointer, der auf
                    // den Anfang des char-Arrays
                    // cfeld zeigt
```

- Man kann keine Arrays von Referenzen erzeugen.

2.4.1. Referenzen als Parameter

call-by-reference-Mechanismus:

- Werden Funktions-Parameter als Referenzen deklariert, werden diese beim Funktionsaufruf mit den übergebenen Argumenten initialisiert.
- Dabei werden die Werte **nicht kopiert**, sondern die Parameter sind ein zweiter Name!
- Jede Änderung innerhalb der Funktion am Parameter wirkt sich auf den Aktual-Parameter aus!
- Vorteil: Zum schreibenden (verändernden) Zugriff auf die Aktual-Parameter müssen keine Zeiger mehr übergeben werden.

  Es wird trotzdem das **Kopieren vermieden**, was bei großen Objekten erheblich die **Laufzeit verbessert!**

Bsp. swap:

```
#include <iostream>

using std::cout;
using std::endl;

void swapR(int& x, int& y) {
  int hilf = x;
  x = y;
  y = hilf;
}

int main() {
  int i = 5, j = 8;
  swapR(i, j);
  cout << "i = " << i << ", j = " << j << endl;
  return 0;
}
```

Aufgabe Referenzen als Parameter

- Man erstelle eine Funktion `void minMaxMittel()`.
- Die Funktion soll das Minimum, das Maximum und den Mittelwert der `n` Elemente eines `double`-Feldes `vec` bestimmen.
- Das Feld `vec` und die Element-Anzahl `n` sind als Parameter zu übergeben.
- Weiterhin sollen alle Ergebnisse als Referenzen zurück geben werden.
- Man teste diese Funktion in einem Hauptprogramm.

2.4.2. Vergleich Zeiger und Referenzen

Zeiger	Referenz
Eigenständiges Objekt im Speicher	nur Aliasname für anderes programmiersprachliches Objekt
hat eigene Adresse	hat *keine* eigene Adresse
kann uninitialisiert sein	muß immer initialisiert sein
zeigt nicht immer auf ein Objekt (z.B. `NULL`-Zeiger `nullptr`)	verweisen immer auf ein Objekt
Zeiger auf Zeiger sind möglich	Referenz kann nicht auf andere Referenz verweisen.

- Zeiger können auch auf Zeigervariablen zeigen.
- Zulässige Zeigertypen sind z.B. auch `T**` und `T***` usw.

Bsp.: Was bewirkt folgendes Programm?

```
int i1 = 13;
int i2 = 42;
int * ptr = &i1;
int **ptrptr;
ptrptr = &ptr;
**ptrptr = 4711;
*ptrptr = &i2;
*ptr = 12345;
```

2.5. L-Wert und R-Wert

L-Wert (engl. L-value):

- Ausdruck, der eine Speicherstelle bezeichnet, in die etwas geschrieben werden darf und die eine Adresse besitzt.
- Kann auf der **linken Seite vom Zuweisungs-Operator stehen** (*L-value* = *left value*).
- Ist ein **Objekt** (zusammenhängender Speicherbereich), der eine **Identität besitzt** und **nicht verschiebbar ist**.
- Bsp.:
  - normale Variable
  - `ptr` und `*ptr` für normale Zeigervariable `ptr`
  - `++x`, für eine Variable `x`

R-Wert (engl. R-value):

- Ein Wert, der kein L-Wert ist und **verschiebbar ist**.
- Ausdruck für einen Wert, der gelesen werden kann.
- Kann auf **der rechten Seite vom Zuweisungs-Operator** stehen (*R-value* = *right value*).
- Kann aber nicht auf der linken Seite einer Zuweisung stehen.
- Bsp.:
  - Konstanten
  - (temporäre) arithmetische Ausdrücke mit elementaren Datentypen,
  - z.B. `x + 1` wenn `x` eine `int`-Variable ist.
  - `*ptr` für Read-Only-Zeiger `ptr` vom Typ `const T*`

2.6. Typ-Qualifizierer const

Grundregel in C++: `const` bedeutet Schreibschutz

Verwendung:

- Definition von Konstanten zur Vermeidung von *Magic Numbers*.
- Schreibschutz bei der Parameter-Übergabe von Zeigern und Referenzen.

```
// Schutz vor unbeabsichtigter Veraenderung
const int pufferGroesse = 512;

//  ...

// diese unbeabsichtigte Veraenderung
// liefert einen Syntax-Fehler
if ( pufferGroesse = 1 ) {
  // ....
}
```

Konvertierungsregel für Typen mit const

- **Schreibschutz kann dazu-, aber nicht weggenommen werden!**
- Wenn z.B. `T` ein Pointer- oder Referent-Typ ist gilt:
  - erlaubt: Zuweisung von `T` an `const T`
  - verboten: Zuweisung von `const T` an `T`

2.6.1. Schreibschutz für Zeiger

a) **Zeiger auf Konstante (Read-Only-Zeiger):**

```
const Typ* ptr
```

oder

```
Typ const* ptr
```

- Zeiger `ptr` zeigt auf einen konstanten Wert (Read-Only-Zeiger)
- Häufige Verwendung in Parameterlisten
  Es wird dadurch garantiert, dass Argumente nicht verändert werden.
- Wenn als Argument auch ein konstantes Objekt erlaubt sein soll, muss als Formal-Parameter ein Read-Only-Zeiger deklariert werden.
- Zuweisung an `ptr` ist erlaubt, d. h. der Zeiger selber kann umgesetzt werden
- Zuweisung an `*ptr` ist nicht erlaubt.

```
const char* gruss1 = "Hello";
const char* gruss2 = "Hallo";
gruss1[1] = 'a';       // Syntax-Fehler
gruss1 = gruss2;       // OK
gruss1 = "Guten Tag"; // OK

int i = 42;
int* ptr1 = &i;
const int* cptr = &i; //Read-Only-Zeiger
*cptr = 15; //Syntaxfehler, Schreiben verboten
cptr = ptr1; //O.K.
ptr1 = cptr; //Syntaxfehler: int* -> const int*
```

Schreibschutz für Zeiger 2

b) **Konstanter Zeiger**

```
Typ* const ptr = initialwert;
```

- Zuweisung an `ptr` ist nicht erlaubt, d. h. Zeiger kann im Verlauf auf keine andere Speicherposition zeigen.
- Zuweisung an `*ptr` ist erlaubt, d. h. der Inhalt der Speicherposition, auf die `ptr` zeigt, kann verändert werden.

```
int i = 42;
int* const ptrc = &i; //Konstanter Zeiger
int *ptr = &i;
ptrc = ptr;   //Syntaxfehler
*ptrc = 4711; //O.K.
```

c) **Konstanter Read-Only Zeiger:**

```
const Typ* const ptr = initialwert;
```

- Zeigervariable und Variable, worauf Zeiger zeigt, sind konstant
- Zuweisung an `ptr` ist nicht erlaubt
- Zuweisung an `*ptr` ist nicht erlaubt.

```
int i = 2, j = 3;
const int* const ptr = &i;
*ptr = 4; // Syntax-Fehler
ptr = &j; // Syntax-Fehler
```

Aufgabe Schreibschutz für Zeiger

Welche der folgenden Anweisungen sind zulässig/unzulässig? Man begründe die Antwort.

```
int i = -1;                    // a)

const int ic = i;              // b)

const int* pic = &ic;          // c)

int* const cpi = &ic;          // d)

const int* const cpic = &ic; // e)

i = ic;                        // f)

pic = &ic;                     // g)

cpi = pic;                     // h)

pic = cpic;                    // i)

cpic = &ic;                    // j)

ic = *cpic;                    // k)
```

2.6.2. Schreibschutz für Referenzen

```
const Typ& ref = var;
```

- Referenz `ref` ist schreibgeschützt (*Read only*).
- Über eine konstante Referenz ist keine Veränderung des Inhaltes möglich.
- Eine Referenz, die auf ein konstantes Objekt verweist, muss selbst als konstant deklariert werden.
- Eine als konstant deklarierte Referenz darf auf ein nicht konstantes Objekt verweisen.
- **Hauptsächliche Verwendung bei der Parameter-Übergabe**.

```
void f1(const double& x) {
  cout << x;  //lesender Zugriff
}
void f2(double& y) {
  y = 1.2345;  //schreibender Zugriff
}

double var = 0;
const double PI = 3.1415;

f1( var );  // OK, Uebergabe an const erlaubt
f1( PI );   // OK, f1 aendert nicht
f2( var );  // OK
f2( PI );   // Syntaxfehler, da f2 aendert
f2( 42.0 ); // Syntaxfehler, da f2 aendert
```

Vorsicht: Fallen mit const Referenzen und Zeigern

```
int ival1 = 10;

int ival2 = 11;

int* iPtr = &ival1;

int& iRef = ival2;

int*& refPtr = iPtr;  // OK, Referenz auf einen
                      // Pointer

cout << "iPtr = " << iPtr
     << ", refPtr = " << refPtr << endl;

iPtr = &iRef;        // OK, iPtr zeigt nun auf ival2

cout << "iPtr = " << iPtr
     << ", &ival2 = " << &ival2 << endl;

int*& refPtr2 = &ival1; // Fehler, &ival1 ist eine
                        // constante Adresse, hierauf
                        // kann nur eine Referenz auf einen
                        // konstanten Pointer verweisen

int*const& refPtr3 = &ival1;  // OK, refPtr3 ist Referenz
                              // auf die Anfangsadresse
                              // von ival1

cout << "refPtr3 = " << refPtr3
     << ", &ival1 = " << &ival1 << endl;

int&* ptrRef = &iRef;   // Fehler, es gibt keine Pointer
                        // auf Referenzen
```

Vorsicht Fallen mit const, Referenzen und Zeigern 2

```
const int cival = 10;

int*& refPtr4 = &cival; // Fehler aus 2 Gruenden:
                        // es wird auf eine
                        // konstante Adresse
                        // mit einem konstanten Wert
                        // verwiesen!

const int*& refPtr4 = &cival; // Fehler: es wird mit
                              // einer Refrerenz
                              // vom Typ
                              // nicht konstanter
                              // Pointer verwiesen

int*const& refPtr5 = &cival;    // Fehler es wird mit
                                // einer Refrenz
                                // vom Typ
                                // konstanter Pointer,
                                // der aber nicht
                                // Read-Only ist
                                // verwiesen

const int*const& refPtr6 = &cival; // OK refPtr6
                                   // ist Referenz auf
                                   // die konstante
                                   // Adresse von
                                   // cival, deren
                                   // Inhalt nicht
                                   // veraendert
                                   // werden darf
cout << "refPtr6 = " << refPtr6
     << ", &cival =" << &cival << endl;
```

2.6.3. Schlüsselwort constexpr

```
constexpr Typ var = ausdruck;
```

- Ist ab Version C++11 in C++ vorhanden, gcc benötigt mindestens das Flag `-std=c++11`.
- `constexpr` Objekte sind `const` und werden mit Werten initialisiert, die schon **beim Kompilieren bekannt sind**.
- Sie können überall dort in C++ verwendet werden, wo ein **konstanter Ausdruck** benötigt wird, z.B. bei der Größe von Arrays oder bei ganzzahligen Template-Argumenten.
- Umgekehrt ist nicht jedes `const` Objekt notwendigerweise ein `constexpr` Objekt.

```
int n;
cout << "Eingabe Feldgroesse: ";
cin >> n;

const int feldGroesse1 = n;            // OK

// gekapselte C-Arrays fester Groesse
std::array<double, feldGroesse1> vec1; // Fehler

constexpr int feldGroesse2 = n;        // Fehler

const int M = 10;
const int N = 4;
constexpr int feldGroesse3 = M*N;      // OK
std::array<double, feldGroesse3> vec2; // OK
```

Schlüsselwort constexpr 2

```
constexpr Typ f( ... ) { ... }
```

- `constexpr`-Funktionen können Ergebnisse schon beim Kompilieren ermitteln, wenn Sie mit Argumenten aufgerufen werden, deren Werte ebenfalls alle beim Kompilieren bekannt sind.
- Ihr Ergebnis kann in diesem Fall überall dort verwendet werden, wo ein konstanter Ausdruck notwendig ist.
- Wird eine `constexpr`-Funktion mit einem oder mehreren Argumenten aufgerufen, deren Werte beim Kompilieren noch nicht bekannt sind, verhält sie sich wie eine normale Funktion.

  Ihr Ergebnis kann dann aber nicht dort verwendet werden, wo ein konstanter Ausdruck notwendig ist.

```
int vecSize1(int m, int n ) {
  return m*n;
}
constexpr int vecSize2( int m, int n ) {
  return m*n;
}
int main() {
  int n;
  std::cout << "Eingabe Feldgroesse: ";
  std::cin >> n;
  const int M = 10;
  const int N = 4;
  std::array<double, vecSize1( M, N )> vec3; // Fehler
  std::array<double, vecSize2( M, N )> vec4; // OK
  cout << vecSize2( n, n ) << endl;  // OK, aber
                                     // Ergebnis ist
                                     // keine constexpr
  std::array<double, vecsize2( n, n)> vec5; // Fehler
  return 0;
}
```

2.7. Vereinheitlichte Initialisierung von Objekten

Ab C++11 existieren folgende Möglichkeiten der Initialisierung:

```
int i1 = 42;   // Initialsieren mit =

int i2(42);    // Initialisieren mit runder Klammer

int i3{42};    // Initialisieren mit Braces

int i4 = {42}; // Initialisiern mit = und Braces
```

- Die neue Initialisierung mit den geschweiften Klammern (braces) **(Braced Initialization)** ist in C++11 **einheitlich verwendbar**.

  Einzige Ausnahme:

  - Falls ein **Sequenzkonstruktor** mit einem Parameter vom Typ `initializer_list<T>` vorhanden ist.
  - Dann wird dieser oft verwendet auch in Fällen, in denen man eigentlich einen anderen Konstruktor aufrufen will.
  - Durch den Aufruf des Konstruktors mit runden Klammern kann man dann oft den gewünschten Konstruktor aufrufen.

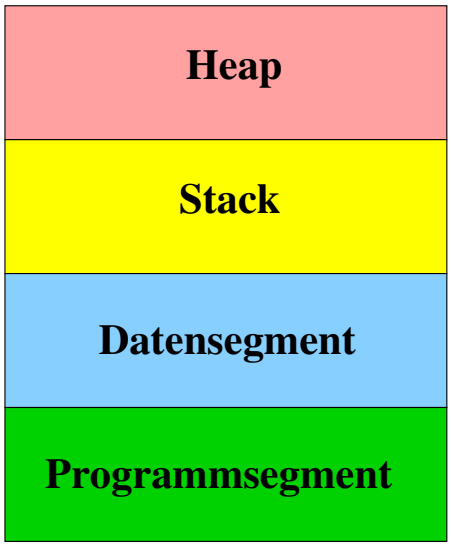
- Die **Braced Initialization** verhindert auch eine **verengende Konvertierung**.

```
double x = 1.1, y = 2.2, z = 3.3;

// Verengende Konvertierung moeglich
int sum1(x+y+z);
int sum2 = x+y+z;

// Verengende Konvertierung wird verboten
// Compiler-Warnung
int sum3{x+y+z};
```

2.8. Dynamische Speicherverwaltung mit new und delete

2.8.1. Aufteilung des Hauptspeichers:

Hauptspeicher



Freispeicher für dynamisch vom Benutzer verwaltete Objekte mit `new/delete`

Automatischer Speicher für lokale Variablen und Parameter

Statischer Speicher für globale Variablen Konstanten, `static-` Variablen

Programmcode

- Der **statische Speicher** ist über die gesamte Programmausführung den Variablen fest zugeordnet.

  Er wird verwendet für globale Variablen und lokale `static`-Variablen.

- Der **automatischer Speicher (Stack)** wird beim Funktionsaufruf automatisch für lokale Variablen und Parameter reserviert.

  Er wird am Ende des Funktionsaufrufs automatisch wieder freigegeben.

- Der **dynamischer Speicher (Heap)** wird verwendet für Objekte, die der Benutzer dynamisch während der Programmausführung angelegt (`new`) und freigegeben hat (`delete`).

  Ein mit `new` erstelltes Objekt existiert unabhängig von Blockgrenzen so lange, bis es mit `delete` zerstört wird.

2.8.2. Dynamischen Speicher reservieren mit new

Ohne Initialisierung:

```
type* ptr = new type;
```

Mit gleichzeitiger Initialisierung:

```
type* ptr = new type(anfangswert);
```

ab C++11:

```
type* ptr = new type{anfangswert};
```

- Reserviert Speicher auf dem Heap für einen Wert vom Typ `type`.
- Kann der Speicher reserviert werden, gibt `new` die Anfangsadresse des reservierten Speicherbereiches zurück.
- Der zurückgegebene Pointer hat schon den richtigen Typ `type*`.

  Ein weiterer Cast wie bei der C-Funktion `malloc()`, die immer einen `void*` Pointer zurück liefert, ist also nicht erforderlich.
- Kann der Speicher wegen Speichermangel nicht allokiert werden, so wird eine `bad_alloc` Exception ausgeworfen.
  Durch die Verwendung von

  ```
  type* ptr  = new (std::nothrow) type;
  ```

  kann im Fehlerfall das Auswerfen einer `bad_alloc` Exception vermieden werden. Es wird stattdessen ein `nullptr` zurück gegeben.

Bsp. new

```
int* ip1 = new int;
int* ip2 = new int(42);
int* ip3 = new int{4711};   // unter C++11
                            // Compiler-Flag:
                            // -std=c++11
                            // Fuer QtCreator:
                            // CONFIG += c++11
                            // in .pro Datei

cout << *ip1 << endl
     << *ip2 << endl
     << *ip3 << endl;

struct Complex{
  double re, im;
} z1 = {1.0, 2.0};

Complex* pc1 = new Complex( z1 );
Complex* pc2 = new Complex{ z1 };

cout << (*pc1).re << ", " << pc1->im << endl;
cout << (*pc2).re << ", " << pc2->im << endl;
```

2.8.3. Speicherfreigabe mit delete

Achtung:

- In C++ gibt es **keine automatische Speicherbereinigung (garbage collection)**!
- Der Programmierer ist selber für eine **saubere Verwaltung der Ressourcen**, insbesondere für die **Vermeidung von memory-leaks**, verantwortlich. In C++ wird hierzu die Technik **Ressourcenbelegung ist Initialisierung (RAII)** angewendet (im Konstruktor in Anspruch genommene Ressourcen werden im Destruktor wieder frei gegeben).
- Die Tools **valgrind** (Linux) oder **heob** (Windows) können helfen, memory-leaks zu entdecken.

```
valgrind ./a.out
==11572== LEAK SUMMARY:
==11572==    definitely lost: 44 bytes in 5 blocks
==11572==    indirectly lost: 0 bytes in 0 blocks
==11572==      possibly lost: 0 bytes in 0 blocks
```

```
delete ptr;
```

```
ptr = nullptr;
```

- Das Objekt, auf den der Zeiger `ptr` zeigt wird zerstört und der reservierte dynamische Speicher wird freigegeben.
- Der Zeiger `ptr` **muß** auf Speicher zeigen, der zuvor mit `new` reserviert wurde.
- `delete` darf nicht ein zweites Mal für dasselbe Objekt aufgerufen werden.
- `delete` darf aber auf einen `NULL`-Pointer angewendet werden, dies ist unschädlich.

Bsp. delete

```
int* ip1 = new int;
int* ip2 = new int(42);
int* ip3 = new int{4711};

cout << *ip1 << endl
     << *ip2 << endl
     << *ip3 << endl;

struct Complex{
  double re, im;
} z1 = {1.0, 2.0};

Complex* pc1 = new Complex( z1 );
Complex* pc2 = new Complex{ z1 };

cout << (*pc1).re << ", " << pc1->im << endl;
cout << (*pc2).re << ", " << pc2->im << endl;

delete ip1; ip1 = nullptr;
delete ip1; // unschaedlich
delete ip2; ip2 = nullptr;
delete ip3; ip3 = nullptr;
delete pc1; pc1 = nullptr;
delete pc2; pc2 = nullptr;
```

```
valgrind ./a.out
==11762== LEAK SUMMARY:
==11762==    definitely lost: 0 bytes in 0 blocks
==11762==    indirectly lost: 0 bytes in 0 blocks
==11762==      possibly lost: 0 bytes in 0 blocks
```

2.8.4. Dynamische Speicherverwaltung für Felder

Speicher für ein Feld reservieren:

```
elementtyp* ptr = new elementtyp[anzahl];
```

- Es wird Speicher für ein Feld mit `anzahl` Elementen vom Typ `elementtyp` reserviert.
- `new[]` liefert die Adresse des ersten Feldelementes zurück.

Dynamisch reservierten Speicher für ein Feld freigeben:

```
delete[] ptr;
```

- Gibt den Speicherbereich frei, auf den `ptr` zeigt.
- Dieser Speicherbereich muss mit `new[]` angelegt worden sein.

Bsp. dynamisch angelegte Felder

```
// Allokation eines Vektors
constexpr int N = 3;
double* pvec = new double[N]{1.0, 2.0, 3.0};
for( int i = 0; i < N; ++i){
   cout << pvec[i] << endl; // Alternative fuer pvec[i]?
}

// Loeschen:
delete[] pvec;
pvec = nullptr;

// Allokation einer NxN Matrix:
// wobei N zur Kompilezeit bekannt ist
// double** mat = new double[N][N]; // Syntax Fehler!!!!
// Statt dessen:
// mat ist ein Pointer auf ein
// Feld mit N Elementen
// welche der beiden Anzahlen in double[N][N]
// kann durch eine andere Konstante ersetzt werden?
double (*mat)[N] = new double[N][N];
int k = 1;
for( int i = 0; i < N; ++i){
  for(int j = 0; j < N; ++j){
    mat[i][j] = k++;
  }
}

for( int i = 0; i < N; ++i){
  for( int j = 0; j < N; ++j){
    cout << std::setw(12) << std::setprecision(2)
         << mat[i][j];  // Alternative fuer mat[i][j]?
  }
  cout << endl;
}

delete[] mat; mat = nullptr;
```

Bsp. dynamisch angelegte Felder 2

```
// Anlegen einer mxn Matrix
// m, n werden eingelesen
int m, n;
cout << "Eingabe m und n: ";
cin >> m >> n;
cout << m << ", " << n << endl;

// Anlegen eines Feldes mit m Pointern auf double
double** mat2 = new double*[m];

// Anlegen der Zeilenvektoren und fuellen
k = 1;
for(int i = 0; i < m; ++i){
  mat2[i] = new double[n];
  for(int j = 0; j < n; ++j) {
    mat2[i][j] = k++;
  }
}

for( int i = 0; i < m; ++i){
  for( int j = 0; j < n; ++j){
    cout << std::setw(12) << std::setprecision(2)
         << mat2[i][j];
  }
  cout << endl;
}

// Speicher freigeben in umgekehrter Reihenfolge
// Loeschen der Zeilen-Vektoren
for(int i = 0; i < m; ++i) {
  delete[] mat2[i]; mat2[i] = nullptr;
}

// Loeschen des Feldes mit den Pointer-Elementen
delete[] mat2; mat2 = nullptr;
```

2.8.5. Dynamische Speicherverwaltung in C

Dynamisch Speicher reservieren in C:

- Durch Verwendung der Funktion

```
void* malloc( size_t size );
```

- Dies erfordert das Inkludieren von `#include <cstdlib>`
- Reserviert Speicherbereich der Größe `size` (in Bytes).
- Liefert `Null`-Zeiger, falls nicht genügend Speicher verfügbar.
- Erfordert einen Cast, z.B. in C-Synatx:

```
double* dp = (double*) malloc(sizeof(double));
int* ip = (int*) malloc(sizeof(*ip));
```

Dynamischen Speicher wieder freigeben in C:

- Durch Verwendung der Funktion

```
void free(void* ptr);
```

- Gibt den Speicherbereich frei, auf den `ptr` zeigt.
- *Wichtig:* nur Speicher freigeben der mit `malloc()` reserviert wurde!
- Bsp.: `free(dp); free(ip);`

Speicher, der mit `new` allokiert wird, ist mit `delete` freizugeben, nie die Arten mischen!
Analog dazu ist Speicher, der mit `malloc()` bzw. `calloc()` allokiert wurde, mit der Funktion `free()` freizugeben.

Bsp.: Dynamische Speicherverwaltung in C

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
using std::cout;
using std::endl;
int main() {

  // nxn Matrix
  int n = 4;
  double** mat = (double**) malloc(n * sizeof(double*));
  int k = 1;
  for(int i = 0; i < n; ++i) {
    mat[i] = (double*) malloc(n * sizeof(double));
    for(int j = 0; j < n; ++j) {
      mat[i][j] = k++;
    }
  }

  // Ausgabe
  for( int i = 0; i < n; ++i){
    for( int j = 0; j < n; ++j){
      cout << std::setw(12) << std::setprecision(2)
           << mat[i][j];
    }
    cout << endl;
  }

  // Speicher freigeben:
  for(int i = 0; i < n; ++i) {
    free(mat[i]);
  }

  free(mat);
  return 0;
}
```

Aufgaben Dynamische Speicherverwaltung

Wo steckt der Fehler?

```
// 1) Aufgabe
int i = 42;
int* ip;
*ip = i;

// 2) Aufgabe
char c = 'A';
char* cp = &c;
cout << *cp << endl;
delete cp;
cout << c << endl;

// 3) Aufgabe
double* pd = new double{13};
cout << *pd << endl;
pd = new double(14);
cout << *pd << endl;

// 4) Aufgabe
void tuWas(int i) {
    int* ptr = new int(i);
    cout << *ptr * *ptr << endl;
}

int main() {
  for( int i = 1; i < 1000; ++i ) {
    tuWas( i );
  }
  return 0;
}
```

5) Aufgabe (Pascal-Dreieck):

Man speichere ein Pascalsches Dreieck der Höhe `h` in einer dynamisch angelegten Dreiecks-Matrix, gebe es **formatiert** aus, und gebe danach den Speicher wieder frei. Man löse dies sowohl mit `new/delete` als auch mit `malloc()/free()`.

2.9. Komplexe Deklarationen lesen

- Als C++ Programmierer sollte man komplexe Deklarationen lesen können.
- Vor allem auch um die Fehlermeldungen des Compilers zu verstehen.
- Hierzu gibt es einfache **Regeln**:
  - Man löst die Deklaration vom Namen der Variable auf. Bei Fehlermeldungen des Compilers fehlt der Name der Variable, es ist nur der Typ wichtig, man spricht von **abstrakten Deklaratoren**. In diesem Fall hilft es herauszufinden, wo die fehlende Variable stehen kann.
  - Man liest dann von **rechts nach links** unter der Beachtung des folgenden Vorranges der Syntax-Elemente:
    1. Runde Klammern `()`, die Teile der Deklaration zusammenfassen.
    2. Runde Klammern `()`, die eine Funktion mit Übergabeparametern, die `...` zurück gibt, anzeigen.
    3. Eckige Klammern `[]`, die ein Array anzeigen.
    4. Sternsymbol `*`, das Zeiger/Pointer auf bedeutet.
- Selbsttest mit dem Tool **cdecl** (`https://cdecl.org/`)

Bsp.:

a) `long* ptr[5];`
`ptr` ist ein Array mit 5 Pointern auf `long`

b) `long (*ptr)[5];`
`ptr` ist ein Pointer auf ein Array mit 5 `long` Elementen.

c) `int** ipp[7];`
`ipp` ist ein Array mit 7 Pointern auf Pointer auf `int`.

2.9.1. Erinnerung Funktionszeiger

- Auch Funktionen haben eine Adresse im Speicher.
- **Funktionszeiger** sind Zeiger, die auf die Speicheradresse einer Funktion zeigen.
- Dabei wird der Funktionsname als konstanter Zeiger auf die Funktion (=Adresse des erzeugten Maschinencodes) betrachtet.
- Ein Funktionszeiger kann an andere Funktion übergeben werden oder in Datenstrukturen gespeichert werden.
- Typische Anwendung für Funktionszeiger: z.B. **Callback-Funktionen**:

  Eine Callback-Funktion nennt man eine Funktion, die einer anderen Funktion als Parameter übergeben wird, und von dieser unter gewissen Bedingungen aufgerufen wird.

  Das erlaubt es, Funktionen allgemein zu definieren und erst beim Aufrufen der Funktion durch Angabe der Callback-Funktion das Verhalten exakt zu bestimmen.

Deklaration eines Funktionszeigers:

```
typ (* funcptr) (Parameterliste);
```

- Deklariert Zeigervariable `funcptr`, die auf eine Funktion mit Parameterliste `(Parameterliste)` und Returntyp `typ` zeigt.
- Die Parameterliste ist wie bei Funktionsprototypen aufgebaut.
- Zum Aufruf wird der Funktionszeiger-Name wie ein Funktionsname verwendet.

Bsp. Funktionszeiger

```
#include <iostream>

using std::cout;
using std::endl;

int add(int x, int y) {
  return x + y;
}

int mult(int x, int y) {
  return x * y;
}

int main() {
  //Funktionszeiger op deklarieren
  int (*op)(int, int);

  op = add;  // oder: op = &add;
  cout << op(3,4) << endl; // Aufruf add

  op = mult;
  cout << op(3,4) << endl; // Aufruf mult
  return 0;
}
```

Aufgabe Funktionszeiger

Trapezregel:

Wie aus der Numerischen Mathematik bekannt, lassen sich mit der summierten Trapezregel bestimmte Integrale näherungsweise berechnen:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(a) + f(b) + 2\sum_{j=1}^{n-1} f(a+jh)\right)$$

mit der Maschenweite $h = \dfrac{(b-a)}{n}$.

Man erstelle eine Funktion `trapez()` mit folgenden Übergabe-Parametern:

- Funktionspointer `f` auf den Integranden.
- Untere Integrationsgrenze `a`
- Obere Integrationsgrenze `b`
- Anzahl Teilintervalle `n`

die einen Näherungswert des obigen bestimmten Integrals mit der Trapezregel berechnet und diesen als Ergebnis zurück gibt.

In einer Hauptfunktion `main()` teste man die erstellte Funktion `trapez()` mit den folgenden Beispielen:

$$\int_1^2 \sin(x)\,dx, \qquad \int_1^2 \frac{\sqrt{x}}{1+2x}\,dx.$$

Bsp. Komplexe Deklarationen lesen 2

d) `char* (*(*seltsam)(double, int))[3]`

`seltsam` ist ein Pointer auf eine Funktion mit einem `double` und einem `int` Argument, die einen Zeiger auf ein Array mit 3 Zeigern auf `char` zurück gibt.

e) `int** w[2][3];`

`w` ist ein Array mit 2 Elementen, die ein Array mit 3 Zeigern auf Zeiger auf `int` sind.

f) `char *(*(**cp[][5])())[] = {0, 0, 0};`

`cp` ist ein Array mit `3` Elementen, die ein Array mit 5 Zeigern auf Zeiger auf eine Funktion sind, die keine Übergabeparameter hat und einen Zeiger auf ein Array mit `*` Elementen, die ein Zeiger auf `char` sind, zurück gibt.

Wo wird diese Deklaration in der Praxis verwendet?

Bsp. Komplexe Deklarationen lesen 3

g) `int (*(*)())()`

Hierbei handelt es sich um einen abstrakten Deklarator, der z.B. in einer Fehlermeldung des Compilers auftreten könnte.

Es gilt zunächst herauszufinden wo eine Variable, wir nennen sie hier `x`, stehen könnte. Mögliche Plazierungen für eine Variable sind:

- Rechts vom Sternsymbol `*` Zeiger auf.
- Links vom eckigen Klammerpaar `[]` Array von.
- Links vom runden Klammerpaar `()` als Funktionsname einer Funktion die etwas zurück gibt.
- Innerhalb des innersten runden Klammerpaares `()`, welches die Auswertungs-Reihenfolge bestimmt.

Im obigen Bsp. bleiben für die Plazierung der Variable die folgenden beiden Möglichkeiten:

`int (*(*x)())()` oder `int (*(*)x())()`

Die 2-te rechte Möglichkeit macht aber das innerste Klammerpaar `(*)` überflüssig.

Somit ist die 1-te linke Möglichkeit zu interpretieren.

`x` ist ein Zeiger auf eine Funktion ohne Übergabeparameter, die einen Zeiger auf eine Funktion ohne Übergabeparameter mit einem `int` Wert als Rückgabewert, zurück gibt.

2.10. Argumentübergabe und Ergebnisrückgabe von Funktionen

- In C++ gilt **Werte-Semantik**!

- Im Unterschied zu Sprachen in denen **Referenz-Semantik** gilt (z.B. Java, Python), werden in C++ Objekte defaultmäßig bei der Argumentübergabe an Funktionen oder bei der Ergebnisrückgabe von Funktionen **kopiert**.

- Essentiell für effizienten C++ Code ist es, das **Kopieren zu vermeiden** oder zumindest die **Anzahl von Kopier-Operationen** zu minimieren.

- Hierfür kann man in C++ **Referenzen** oder **Zeiger/Pointer** verwenden.

- **Nachteil:**
  - Werden aus Effizienz-Gründen Parameter als Referenzen oder als Zeiger an eine Funktion übergeben, um ein Kopieren zu vermeiden,
  - hat man innerhalb der Funktion **vollen Schreibzugriff auf die Parameter**.
  - **Es besteht kein Schutz vor ungewollten Änderungen!**

Deshalb zwingend:

Schutz der Parameter mit const

- Benötigt man in der Funktion nur Lesezugriff, ist es **Pflicht die Parameter mittels** `const` **vor einer unbeabsichtigten Änderung zu schützen (const correctness)**.

- In einem derartigen Fall ist immer die Übergabe als `const` Referenz oder `const` Pointer einer call by value Wertübergabe vorzuziehen.

```
void druckeAuto(const Auto& a) { /*...*/ }

void druckeAuto(const Auto* a) { /*...*/ }
```

- Man vermeidet dadurch auch das **Slicing-Problem**, das entsteht wenn ein Objekt von Typ einer Unterklasse in ein Objekt vom Typ einer Oberklasse Teil-Kopiert wird.

- **Außnahme:** Benötigt eine Funktion nur Lesezugriff auf Parameter, die einen **elementaren Datentyp** (z.B. `int`, `double`, ...) haben, so übergibt man sie auch in C++ defaultmäßig, also call by value.

```
int faku(int n); //normal ueblich

// unnoetig ist
int faku(const int n); // Warum?

// nicht notwendig
int faku(const int& n); // Warum?
```

Bsp. Ergebnisrückgabe von Funktion

Gegebene Definition: *Was fällt auf im Vergleich zu C?*

```
struct Complex{
  double re, im;
  void printMe() const {
    cout << "(" << re << ","
         << im << ")" << endl;
  }
};
```

Aufgabe: Funktion `add42(Complex z)` erstellen, die

- auf `z` **nur lesend** zugreift
- und sowohl zum Realteil als auch zum Imaginärteil von `z` den Wert 42 addiert
- und diese neue komplexe Zahl als Ergebnis zurück gibt.

1) Version: (call by value)

```
Complex plus42Version1(Complex z) {
  Complex erg = {z};
  erg.re += 42;
  erg.im += 42;
  return erg;
}
int main() {
  Complex z{1.0, 2.0};

  Complex erg1 = plus42Version1(z);
  erg1.printMe();
  return 0;
}
```

Nachteil: 2 maliges Kopieren! Sowohl bei der Übergabe von `z` als auch bei der Rückgabe von `erg`.

Bsp. Ergebnisrückgabe von Funktion 2

2) Rückgabe einer Referenz:
Wo steckt der Fehler?

```
Complex& plus42Version2(const Complex& z) {
  Complex erg = {z};
  erg.re += 42;
  erg.im += 42;
  return erg;
}

int main() {
  Complex z{1.0, 2.0};
  Complex erg2 = plus42Version2(z);
  erg2.printMe();
  return 0;
}
```

3) Rückgabe eines Zeigers:
Wo steckt der Fehler?

```
Complex* plus42Version3(const Complex& z) {
  Complex erg = {z};
  erg.re += 42;
  erg.im += 42;
  return &erg;
}

int main() {
  Complex z{1.0, 2.0};
  Complex* erg3 = plus42Version3(z);
  erg3->printMe();
  return 0;
}
```

Bsp. Ergebnisrückgabe von Funktion 3

4) Rückgabe eines Zeigers mit Allokation des Ergebnisses auf Heap:

Was wurde vergessen?

```
Complex* plus42Version4(const Complex& z) {
  Complex* erg = new Complex{z};
  erg->re += 42;
  erg->im += 42;
  return erg;
}

int main() {
  Complex z{1.0, 2.0};
  Complex* erg4 = plus42Version4(z);
  erg4->printMe();
  return 0;
}
```

Nachteil:

- Für die Freigabe des Speichers ist nicht mehr die Funktion selber verantwortlich.
- Die Freigabe muß an einer ganz anderen Stelle im Programm-Code erfolgen.
- Sie wird deshalb leicht vergessen!
- Deshalb die Regel: **Wenn in einer Funktion eine Ressource in Anspruch genommen wird, sollte auch diese Funktion selber für die Freigabe der Ressource verantwortlich sein.**
- Dies ist hier aber nicht möglich, Ausweg?

Bsp. Ergebnisrückgabe von Funktion 4

5) Übergabe der Ergebnis-Variable als Referenz-Parameter:

```
void plus42Version5(const Complex& z,
                    Complex& erg) {
  erg.re = z.re + 42;
  erg.im = z.im + 42;
}

int main() {
  Complex z{1.0, 2.0};
  Complex erg5{0, 0};
  plus42Version5(z, erg5);
  erg5.printMe();
  return 0;
}
```

- Diese Version meckert der Google Style-Checker **cpplint** an:

```
bspFktRueckgabe.cpp:65:
Is this a non-const reference?
If so, make const or use a pointer:
Complex& erg
```

- Denn beim Aufruf dieser Version ist es nicht direkt ersichtlich, dass `erg5` durch die Funktion verändert wird.
- cpplint schlägt deshalb vor, besser einen Zeiger zu verwenden.

Bsp. Ergebnisrückgabe von Funktion 5

6) Übergabe der Ergebnis-Variable als Zeiger-Parameter:

```
void plus42Version6(const Complex& z,
                    Complex* erg) {
  erg->re = z.re + 42;
  erg->im = z.im + 42;
}

int main() {
  Complex z{1.0, 2.0};

  Complex erg6{0, 0};
  plus42Version6(z, &erg6);
  erg6.printMe();

  return 0;
}
```

- Diese Version ist cpplint konform.
- Beim Aufruf ist durch die Verwendung des Adress-Operators ersichtlich, dass die Funktion schreibenden Zugriff auf `erg6` hat.
- Ein weiterer Vorteil dieser Version anstelle eines `return` (Version 4)) besteht darin, dass auch mehrere Objekte zurück gegeben werden können.
- **In dieser Vorlesung wird deshalb immer diese Version verwendet!**

2.11. Deklaration und Definition in C++

2.11.1. Deklaration:

- Bevor ein Name (Bezeichner) in C++ verwendet werden kann, muss er deklariert werden.
- Eine Deklaration in C++ ist eine Anweisung, die einen Namen für eine Entität (Variablen, Funktionen, Typen) einführt und ihm eine Bedeutung gibt.
- Eine Deklaration reserviert aber keinen Speicherplatz für Variablen oder legt für Funktionen die Implementierung des Rumpfes fest.
- Bsp.:

```
extern int a;
extern const double PI;
double f(double);
struct Punkt;
enum Ampel;
double f(double); // Mehrfach-Deklaration
extern in a;      // Mehrfach-Deklaration
```

- Mit dem Schlüsselwort `extern` vor einer Variablen wird für die Variable kein Speicherplatz angelegt. Die Variable wird nur deklariert.
- Eine Entität darf im Programm (auch in mehreren Modulen) **mehrfach deklariert werden**.

2.11.2. Definition

Definition:

- Eine Definition ist auch immer eine Deklaration.
- Eine Definition liefert alles, was für die Verwendung einer Entität benötigt wird (Reservierung von Speicher für Variablen, vollständige Implementierung einer Funktion, vollständige Beschreibung eines Typs/Struktur/Klasse/Enummeration).
- Bsp.:

```
int a = 2;
const double PI = 3.1415;
double f(double x) {
  return x*x + 2;
}
struct Punkt {
   double x, y;
};
enum Ampel { rot, gelb, gruen};
```

- In C++ gilt die **One Definition Rule:**

  Jede Entität (Variable, Konstante, Funktion, Struktur, ...) darf innerhalb eines Programmes (bestehend aus mehreren Modulen) **genau nur ein mal definiert werden**.
- Bei der Strukturierung des Programmes, insbesondere bei der Aufteilung des Codes in Header-Dateien (`*.h` Dateien) und Implementierungs-Dateien (`*.cpp` Dateien) ist dies zu beachten.

2.11.3. Header-Dateien und Implementierungs-Dateien

Bestandteil von Header-Dateien (*.h Dateien):

- reine Deklarationen von Variablen, z.B.: `extern int a;`
- Funktionsprototypen (Schnittstellen)
- Deklaration globaler Konstanten, z.B.: `extern const PI;`
- Definition von Konstanten, die nur in der Übersetzungseinheit sichtbar sind, z.B.: `const int NMAX = 10;`
- Definition von Datentypen wie `struct`, `enum`.
- Deklaration von Klassen (`class`)

Bestandteil von Implementierungsdateien (*.cpp Dateien):

- Definition globaler Variablen
- Funktionsdefinition (Implementierung)
- Definition und Initialisierung globaler Konstanten

2.12. Default Parameter von Funktionen

- In C++ kann man Default-Werte (Standardwerte) für Funktionsparameter festlegen.
- Beim Aufruf müssen dann nicht alle Argumente angegeben werden.
- Der Compiler übernimmt für die fehlenden Aktual-Parameter die vorgegebenen Default-Werte.

Syntax:

```
Trueck f(T1 p1, T2, p2, T3 p3=w3,..., Tn pn=wn);
```

Bsp.: `void func(double r, int a=0, int b=1);`
Mögliche Aufrufe:

```
func(3.14, 3, -8);
func(1.25, 6); // entspricht : func(1.25, 6, 1);
func(-2.5); // entspricht : func(-2.5, 0, 1);
```

- Default-Parameter werden beim Prototyp angegeben.
- Wird für einen Parameter ein Default-Wert festgelegt, **müssen auch alle nachfolgenden Parameter Default-Werte besitzen**.
- Einmal festgelegte Default-Parameter-Werte können in einer weiteren Funktionsvereinbarung (Deklaration oder Definition) nicht umdefiniert werden.
- Auch eine Wiederholung der gleichen Festlegung ist nicht zulässig.

```
void pr(int val, int base=10);
// ...
void pr(int val, int base=16);  // Fehler !
// ...
void pr(int val, int base=10);  // Fehler !
```

Default Parameter von Funktionen 2

- Zulässig ist dagegen die Festlegung von Default-Parametern in einer weiteren Funktionsvereinbarung, wenn diese in vorhergehenden Vereinbarungen derselben Funktion noch nicht enthalten sind.

```
void dr(char* fname, FILE* os);
// ...
// zulaessig ist:
void dr(char* fname, FILE* os = stdout);
```

- Der Initialisierungsausdruck zur Festlegung eines Parameter-Default-Wertes **darf keine lokale Variable sein und keine anderen Parameter enthalten (nur Konstanten und globale Variable)**.

  Er wird in der Umgebung der Funktionsdeklaration festgelegt und bei jedem Funktionsaufruf ausgewertet.

```
int i;
void f() {
  int i;
  // Fehler, lokale Variable !
  extern void g(int x=i);
  // ...
}

int a;
// Fehler, Parameter !
int fehl(int a, int b=a);
```

2.13. Inline-Funktionen

```
inline Treuck f(parameterListe) { //.. }
```

Bsp.:

```
inline int max(int x, int y) {
  return x>y ? x : y;
}
```

- Ziel: Vermeidung des *Aufruf-Overheads*.
- Bei kleinen Funktionen besteht der Gesamtaufwand für die Ausführung einer Funktion zu erheblichen Teilen aus dem Aufruf-Overhead (Speicherung der Rücksprung-Adresse und der lokaler Variablen auf dem Stack).
- Die `inline` Definition ist eine Empfehlung an den Compiler, den Funktionsaufruf durch den Code des Funktionsrumpfes direkt zu ersetzen, d.h. die Funktion wie ein Makro zu expandieren.
- An jeder Aufrufstelle einer Inline-Funktion kann der Compiler den vollständigen Maschinencode des Funktionsrumpfes anstelle des Funktionsaufrufs einfügen.
- Der Compiler kann der Bitte nachkommen oder sie ignorieren.
- Sinnvoll ist die Inline-Expandierung i. A. nur bei sehr kurzen Funktionen, bei denen der Aufruf- und Rückkehr-Mechanismus im Vergleich zum eigentlichen Funktionscode sehr aufwendig ist.
- Im Gegensatz zu *normalen* Funktionen muss vor dem Aufruf einer Inline-Funktion ihre Definition erfolgt sein.
- Wenn der Compiler nur ihre Deklaration kennt, kann er wegen fehlender Kenntnis ihres Codes keine Expandierung vornehmen.
- Deshalb können Inline-Funktionen nicht in andere Module exportiert werden, also nicht die Speicherklasse `extern` haben.

2.14. Überladen von Funktionen

- C++ ermöglicht es im Gegensatz zu C im gleichen Gültigkeitsbereich mehrere Funktionen mit **gleichem Namen aber unterschiedlicher Signatur** zu vereinbaren.
- Man spricht dann vom **Überladen von Funktionen (overlaoding)**.
- *Erinnerung:* Man spricht vom **Überschreiben einer Funktion (overriding)**, wenn in einer Unterklasse eine Funktion, die schon in einer Oberklasse definiert ist, **mit gleicher Signatur** neu definiert wird.
- In C++ besteht die **Signatur** einer Funktion aus:
  - der Anzahl,
  - und der Reihenfolge
  - der Typen

  der Funktions-Parameter.
- *Vorsicht*: **Der Rückgabe-Typ gehört nicht zur Signatur**.

  Ein unterschiedlicher Rückgabe-Typ allein reicht deshalb **nicht** zur Unterscheidung der Funktionen aus und ermöglicht somit kein Überladen von Funktionen.
- Praktisch wird das Überladen von Funktionen angewendet für Funktionen, die **gleiche oder ähnliche Aufgaben** nur mit unterschiedlichen Parameter-Typen bzw. -Anzahl realisieren.
- Der Compiler wählt für einen Funktionsaufruf anhand der Typen der Aktual-Parameter aus, welche der Funktionen am *besten passt*.
- Die Aktualparameter müssen aber eine eindeutige Identifikation zulassen!

Bsp. Überladen von Funktionen

```
int maxi(int a, int b);

// Fehler, da Unterschied nur
// hinsichtlich Referenzen,
// Aufruf maxi( x, y)
// mit int-Variablen x und y
// wird von beiden Funktionen
// akzeptiert
int maxi(int& a, int& b);

// OK
double maxi(double a, double b);

// Vorsicht bei Default-Parametern!
void g(const char* str, int x=0);
void g(const char* str, double x=1.0);

int main() {
  cout << maxi(2, 3) << endl;     // OK

  cout << maxi(2.1, 3.7) << endl; // OK

  cout << maxi( 2, 3.0) << endl;  // Fehler

  g("hallo");                     // Fehler

  g("hallo", 1);                  // OK

  return 0;
}
```

2.15. Datentyp bool in C++

```
bool b1 = (2.3 <= x && x < 4)
          || (6.2 < x && x <= 10.3);
bool b2 = 42;
if (b1) { // tu was  }
```

- Der Datentyp `bool` wurde Ende 1993 in der ANSI/ISO-Sprachspezifikation von C++ eingeführt.
- Er wird als eigenständiger vorzeichenbehafteter Ganzzahl-Typ *(unique signed integral type)* definiert.
- Sein Wertevorrat besteht aus den beiden Wahrheitswerten `false` und `true`
- Die Vergleichsoperatoren oder die logische Verknüpfung mit den Operatoren `&&` und `||` liefern als Ergebnis einen Wert vom Typ `bool`.
- In arithmetischen Ausdrücken kann ein `bool`-Wert automatisch in einen `int`-Wert umgewandelt werden **(integral promotion)**:

  `false` entspricht `0`

  `true` entspricht `1`
- Umgekehrt kann ein numerischer Wert oder ein Pointer-Wert implizit in einen `bool`-Wert umgewandelt werden:

  `== 0` bzw. `== NULL` entspricht `false`

  `!= 0` bzw. `!= NULL` entspricht `true`
- Falls in einem älteren C++-Compiler der Datentyp `bool` noch nicht definiert ist, kann man ihn mittels eines Aufzählungstyps nachbilden:

```
enum bool { false, true };
```

2.16. Der Typplatzhalter auto

- Ab C++11 wird mit dem Typplatzhalter `auto` der Compiler angewiesen, selbst (zur Compilezeit) den tatsächlichen Typ zu ermitteln.
- Der Typplatzhalter auto kann immer dann verwendet werden, wenn der Datentyp implizit festgelegt ist, z. B. durch den Initialisierungswert.
- Hierdurch kann man einerseits bei komplexen Typen, insbesondere bei Templates, die Schreibarbeit verringern und Tippfehler vermeiden.
- Andererseits lassen sich dadurch Probleme mit uninitialisierten Variablen vermeiden.

```
auto preis  = 12.42;   // Variable preis hat den Typ double

auto preis2 = 4.5f;    // Variable preis2 hat den Typ float

int i1 = 2, i2 = 3;
// ...
auto i3 = i1 + i2;     // Variable i3 hat den Typ int

double summe;          // summe ist nicht initialisiert

auto summe2;           // Compilerfehler: Typ von summe2
                       // kann nicht ermittelt werden!

auto summe3 = 0u;      // OK summe3 hat den Typ unsigned
                       // und ist initialisiert

auto i = 0, *ip = &i;  // i hat Typ int und ip
                       //  ist ein Pointer auf int

auto j = 0, pi = 3.14; // Fehler da j und pi nicht den
                       // gleichen Typ haben
```

Der Typplatzhalter auto 2

- Der Typplatzhalter `auto` erstellt automatisch weder:
  - `const/volatile`-Variablen
  - noch Referenz-Variablen.
- Dies muss explizit beim Typplatzhalter mit angegeben werden.

```
int k = 0, &rk = k;

auto k2 = rk;              // k2 hat den Typ int
                           // und NICHT int&

const int ci = 1, &rci = ci;

auto ci2 = ci;             // ci2 hat den Typ int und NICHT
                           // const int

const auto ci3 = ci;       // ci3 hat den Typ const int

auto ci4 = rci;            // ci4 hat den Typ int

auto d = &k;               // d hat den Typ int*

auto e = &ci;              // e hat den Typ const int*

auto &f = ci;              // f hat den Typ const int&

const auto &g = 42;        // OK, g hat den Typ const int&
```

2.17. Bereichsbasierte for-Schleifen für Arrays und Container

- Mit der Hilfe der neuen (ab C++11) **bereichsbasierten for-Schleifen** kann man sicher durch alle Elemente eines Arrays oder Containers laufen.
- Dadurch wird die Gefahr von Bereichsüberschreitungen vermieden, da keine Index-Grenzen angegeben werden müssen.

2.17.1. Bereichsbasierte for-Schleifen für Arrays:

- Die bereichsbasierte `for`-Schleife kann für alle Arrays verwendet werden:

```
int werte[10] {1, 2, 3, 4, 5,
               6, 7, 8, 9, 10};
auto summe = 0;
for (auto element : werte) {
  summe += element;
}
cout << "Summe = " << summe << endl;
```

- Soll innerhalb der Schleife keine Wertkopie des Elements verwendet, sondern direkt mit den Array- bzw. Containerelementen gearbeitet werden, **muss eine Referenz verwendet werden**.

```
for (auto& element : werte) {
  element = element * element;
}
```

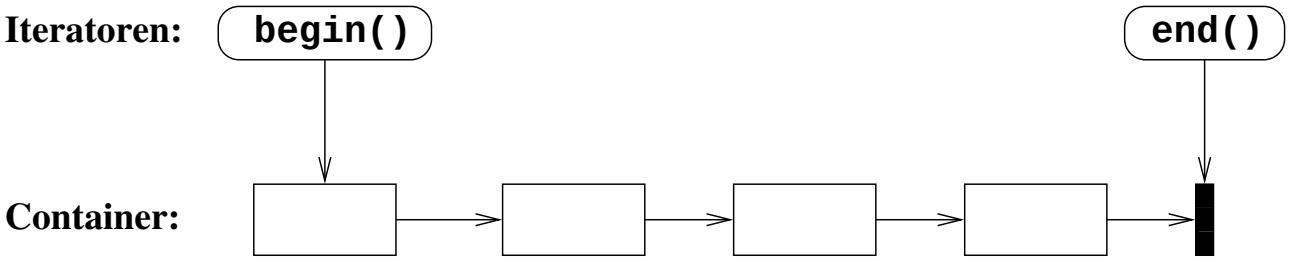
- *Achtung:* Innerhalb der Schleife darf sich die Anzahl der Elemente des Arrays bzw. des Containers nicht ändern, andernfalls kann es zu Speicherfehlern kommen.

2.17.2. Bereichsbasierte for-Schleifen für Container

- Die **C++ Standard Template Library (STL)** ist Bestandteil der C++ Standard-Bibliothek.
- In der STL sind die folgenden Container vorhanden:
  - Sequentielle Container:
    - `vector` (Array, das sich in der Größe anpasst)
    - `list` (doppelt verkettete Liste)
    - `deque` (Double Ended Queue)
    - `forward_list` (einfach verkettete Liste)
  - Containeradapter:
    - `stack` (Stapel, LIFO)
    - `queue` (Queue, FIFO)
    - `priority_queue` (Prioritätswarteschlange)
  - Assoziative Container (Zugriff über Schlüssel)
    - `map` (geordnete Sequenz von (Schlüssel, Wert)-Paaren)
    - `multimap` (doppelte Schlüssel sind erlaubt)
    - `set` (geordnete Menge)
    - `multiset` (doppelte Elemente erlaubt)

Bereichsbasierte for-Schleifen für Container 2

- Weiterhin stellt die STL **Algorithmen** zur Verfügung, z.B.: `find()`, `count()`, `for_each()`, `search()`, `copy()`, `remove()`.
- Die Algorithmen greifen dabei auf die Container nur indirekt über **Iteratoren** zu.
- **Iteratoren** sind Zeiger-ähnliche Objekte, die auf eine bestimmte Position des Containers verweisen.
- Nur über den Iterator kann der Zugriff auf das Element des Containers, auf das der Iterator verweist, erfolgen.
- Sehr oft muss man in den Algorithmen mittels Iteratoren die Anfangsposition (`begin()`, eingeschlossen) und die End-Position (`end()`, ausgeschlossen) angeben.



Bereichsbasierte for-Schleifen für Container 3

- Auch für alle Container der STL kann eine bereichsbasierte Schleife verwendet werden:

```
#include <iostream>
#include <vector>

using std::cout;
using std::endl;

int main() {
  std::vector<int> vec = {1, 2, 3, 4, 5};

  for(auto i: vec) {
    cout << i << endl;
  }
  return 0;
}
```

- Intern wandelt der Compiler diese Schleife folgendermassen um, wobei er Iteratoren verwendet *(dies zeigt auch wie man immer Iteratoren verwendet)*:

```
for( std::vector<int>::iterator it = vec.begin();
     it != vec.end(); ++it) {
  cout << *it << endl;
}
```

- In diesem Fall erleichtert die Verwendung von `auto` die Schreibarbeit erheblich und vereinfacht auch die Syntax:

```
for(auto it = vec.begin(); it != vec.end(); ++it){
    cout << *it << endl;
}
```

Bereichsbasierte for-Schleifen für Container 4

- Benötigt man innerhalb der bereichsbasierten for-Schleife schreibenden Zugriff, muss eine Referenz verwendet werden:

```
for(auto& i: vec) {
  i = i*i;
}

for( auto i: vec) {
  cout << i << endl;
}
```

2.18. Verwendung von nullptr anstelle 0 oder NULL für den Nullpointer

- Das Literal `0` ist kein Zeiger-Typ, sondern stellt einen `int` Wert dar. (Auch wenn jede Repräsentation von `0` in einen beliebigen Typ implizit typgewandelt wird.)
- In C++98 kann dies deshalb insbesondere beim Überladen von Funktionen zu Problemen führen, z.B.:

```
void f(int);
void f(bool);
void f(void*);
// ...
f(0);          // Aufruf von f(int)
f(NULL);       // Aufruf von f(void*)
f(nullptr);    // Aufruf von f(void*)
```

- Bei `NULL` handelt es sich dagegen um eine symbolische Konstante (s. `<stdio.h>` bzw. `<cstdio>`) des Typs `void*` *(generischer Pointer)*.
- Seit C++11 existiert zusätzlich das Schlüsselwort `nullptr`, dessen Wert einen Nullpointer darstellt und keine Headerdatei benötigt.
- `nullptr` kann in jeden typisierten Pointer implizit umgewandelt werden.
- Außerdem garantiert die Einführung des entsprechenden Datentyps `nullptr_t` (s. `stddef.h` bzw. `cstddef`) eine striktere Typprüfung und bessere Lesbarkeit.

```
auto result = findElemen(myList, element);
if (result == nullptr) {
   // ...
}
```

2.19. Zusammenfassende Aufgabe Gray-Code

(für lange Corona Tage/Nächte)

Gray-Code: Bsp. Code-Wort-Länge 3:

Dez	Bin	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

- Code bei dem sich aufeinander folgende Code-Wörter nur in einer einzigen binären Ziffer unterscheiden, die Hamming-Distanz (s. u.) benachbarter Codewörter ist somit 1.
- Die **Hamming-Distanz** von 2 Code-Wörtern ist genau die Anzahl Bit-Positionen, in denen beide Code-Wörter unterschiedliche Werte haben.
- Er findet z.B. Verwendung bei der Übertragung von sich kontinuierlich verändernden Signalen, um Übertragungsfehler korrigieren zu können.
- In der Parallelen Programmierung wird er ebenfalls eingesetzt für die Nummerierung der Rechen-Knoten, wenn sie **Hyper-Cube Topologie** besitzen. Er ermöglicht dann das sehr einfach **E-Cube-Routing**.

Gray Code 2

Der Gray-Code kann rekursiv nach folgendem Schema generiert werden:

- Der Gray-Code mit Bit-Länge 1 besteht aus den beiden Code-Wörtern

  ```
  0
  1
  ```

- Der Gray-Code mit Bit-Länge n kann aus dem Gray-Code mit Bit-Länge $n-1$ konstruiert werden, in dem man zuerst an alle seine Code-Worter das Bit 0 als Vorsilbe anhängt und dann an alle seine Code-Wörter, die man nun **in umgekehrter Reihenfolge durchläuft** das Bit 1 als Vorsilbe anhängt:

  ```
  Vorsilbe    Code der Laenge n-1
        0            0
        0            1
        1            1
        1            0
  ```

Im folgenden werden wir die Code-Wörter eines Gray-Codes in dynamisch erstellten Arrays mit Element-Typ `unsigned int` (32 Bit) speichern.

Dies erlaubt uns rein theoretisch alle Wörter eines Gray-Codes mit einer Bit-Länge von bis zu 32 zu speichern.

Wie viel Speicher würde für die Speicherung alle Code-Wörter mit dieser maximalen Bit-Länge 32 benötigt?

Ist dies auf Ihrem PC/Laptop möglich?

Gray Code 3

Aufgaben:

a) Man erstelle eine Funktion

```
inline void setBit(unsigned int& binValue, int iBit)
```

die in `binValue` das in `iBit` angegebene Bit auf 1 setzt.

Dabei nummerieren wir wie üblich die Bits von rechts (LSB) nach links (MSB) und beginnen die Nummerierung mit 0.

b) Man erstelle eine Funktion

```
inline bool getBit(unsigned int& binValue, int iBit)
```

die den Wert des in `iBit` angegebenen Bits in `binValue` zurück gibt.

c) Man erstelle eine Funktion

```
inline void deleteBit(unsigned int& binValue, int iBit)
```

die in `binValue` das in `iBit` angegebene Bit auf 0 zurück setzt.

d) Man erstelle eine Funktion

```
void binToString(unsigned int binValue,
  int nBits, char* binStr)
```

die die Binärdarstellung von `binValue` als String `binStr` zurück gibt. Dabei sollen nur `nBits` (von rechts gezählt) von `binValue` berücksichtigt werden. Man kann davon ausgehen, dass der Pointer `binStr` auf ein `char` Feld mit `nBits +1` Elementen zeigt, also auch Platz für die Terminierung `'\0'` vorhanden ist.

e) Man erstelle eine Funktion

```
int hammingDistance(unsigned int word1,
  unsigned int word2, int nBits)
```

die die Hamming-Distanz der beiden Code-Wörter `word1` und `word2` zurück liefert und dabei nur `nBits` (von rechts) berücksichtigt.

Gray Code 4

Aufgaben Fortsetzung:

f) Man erstelle eine Funktion

```
inline int grayCodeSize(int nBits)
```

die die Anzahl Code-Wörter, die ein Gray-Code mit Bit-Länge `nBits` hat, zurück gibt.

g) Man erstelle eine **rekursive** Funktion

```
void grayCode(unsigned int* code, int nBits)
```

die über den Pointer `code` alle Code-Wörter eines Gray-Codes mit Bitlänge `nBits` zurück gibt. Man kann davon ausgehen, dass der Pointer auf ein bereits außerhalb der Funktion angelegtes Feld in der richtigen Größe zeigt.

h) Man erstelle eine Funktion

```
void printGrayCode(const unsigned int* code, int nBits)
```

die alle Code-Wörter eines Gray-Codes `code` mit Bit-Länge `nBits` geeignet formatiert ausgibt.

i) Man erstelle eine Funktion

```
bool checkGrayCode(const unsigned int* code, int nBits)
```

die überprüft, ob alle aufeinander folgenden Code-Wörter eines Gray-Codes mit Bit-Länge `nBits` die Hamming-Distanz 1 haben (Rückgabewert `true`), andernfalls ist `false` zurück zu geben.

Gray Code 5

Aufgaben Fortsetzung:

j) man erstelle eine Funktion

```
unsigned int grayEncode(unsigned int binWord)
```

die die Gray-Code Codierung des Binär-Wortes `binWort` zurück liefert. Bekanntlich gilt:

```
grayWord  = binWord ^ (binWord>>1)
```

Man mache sich dies an einem kleinen Beispiel klar.

Ist die Code-Länge `nBits` so bedeutet dies Bit-weise in Pesudo-Code:

```
grayCode[nBits-1] = binWort[nBits-1]
```

und für `i < nBits-1` gilt:

```
grayCode[i] = binWort[i] xor binWort[i+1].
```

k) man erstelle eine Funktion

```
unsigned int grayDecode(unsigned int grayWord,
  int nBits)
```

die das Gray-Code Wort `grayWord` der Bit-Länge `nBits` dekodiert.

Aus der in j) angegebenen Bit-weisen Codierung folgt:

```
binWort[nBits-1] = grayCode[nBits-1]
```

und für `i < nBits-1` erhält man:

```
grayCode[i] xor binWort[i+1]
   = binWort[i] xor binWort[i+1] xor  binWort[i+1]
   = binWort[i].
```

also:

```
binWort[i] = grayCode[i] xor binWort[i+1].
```

l) Man erstelle ein Hauptprogramm `main()` in dem alle erstellten Funktionen geeignet getestet werden.

3. Grundkonzepte der Objektorientierten Programmierung (OOP)

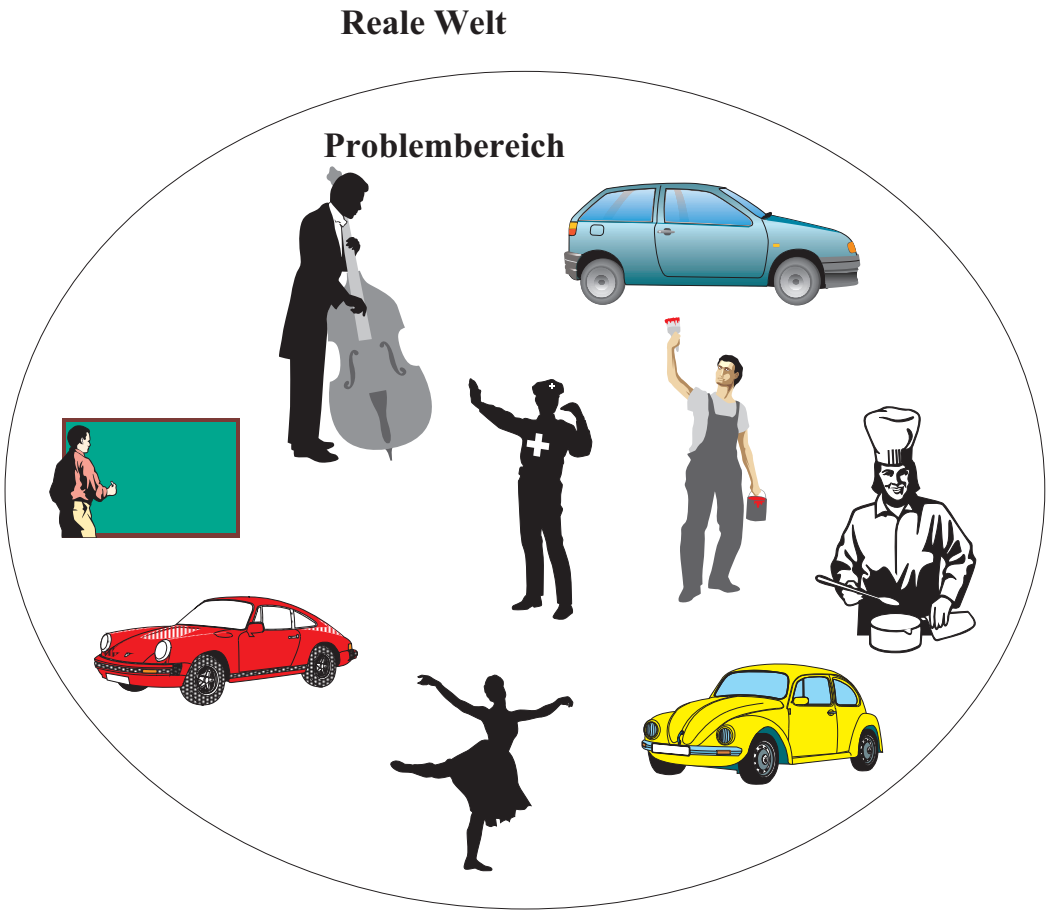
3.1. Objektorientierte Vorgehensweise

- Wurde ausgelöst durch die Softwarekrise 1968, viele große Software-Projekte von Banken, Versicherungen, Behörden scheitern aufgrund ihrer Komplexität.
- Um die Komplexität zu bewältigen wurde meistens die
  - **Strukturierte Programmierung:** Verzicht auf Sprünge mit GOTO, Gliederung des Programms durch Kontrollstrukturen, graph. Darstellung durch **Nassi-Shneiderman-Diagramme**
  - und die **Prozedurale Programmierung:** Programm wird in eine Hierarchie von Prozeduren (Unterprogramme, Funktionen), die kleinere Teilaufgaben lösen, zerlegt.

  als Programmier-Paradigma verwendet.
- Ein **Programmier-Paradigma** ist ein fundamentaler Programmierstil, dem verschiedene Prinzipien und Sichtweisen zugrunde liegen. Es soll den Entwickler bei der Erstellung von *gutem Code* unterstützen, in manchen Fällen sogar zu einer bestimmten Herangehensweise bei der Lösung von Problemen zwingen.
- Die erste objektorientierte Vorgehensweise geht auf die Entwicklung der Programmiersprache **Simula** im Jahr 1967 durch Ole-Johan Dahl und Kristen Nygaard an einem Rechenzentrum an der Universität Oslo zurück.
- Darauf aufbauend wurd in den Jahren 1970 bis 1980 am Palo Alto Research Center der Firma Xerox die Programmiersprache **Smalltalk** entwickelt. Sie ist die erste Programmiersprache, die vollständig das objektorientierte Programmier-Paradigma unterstützt.

Objektorientierte Vorgehensweise 2

Programmieren bedeutet die Nachbildung eines Problembereiches der realen Welt im Computer



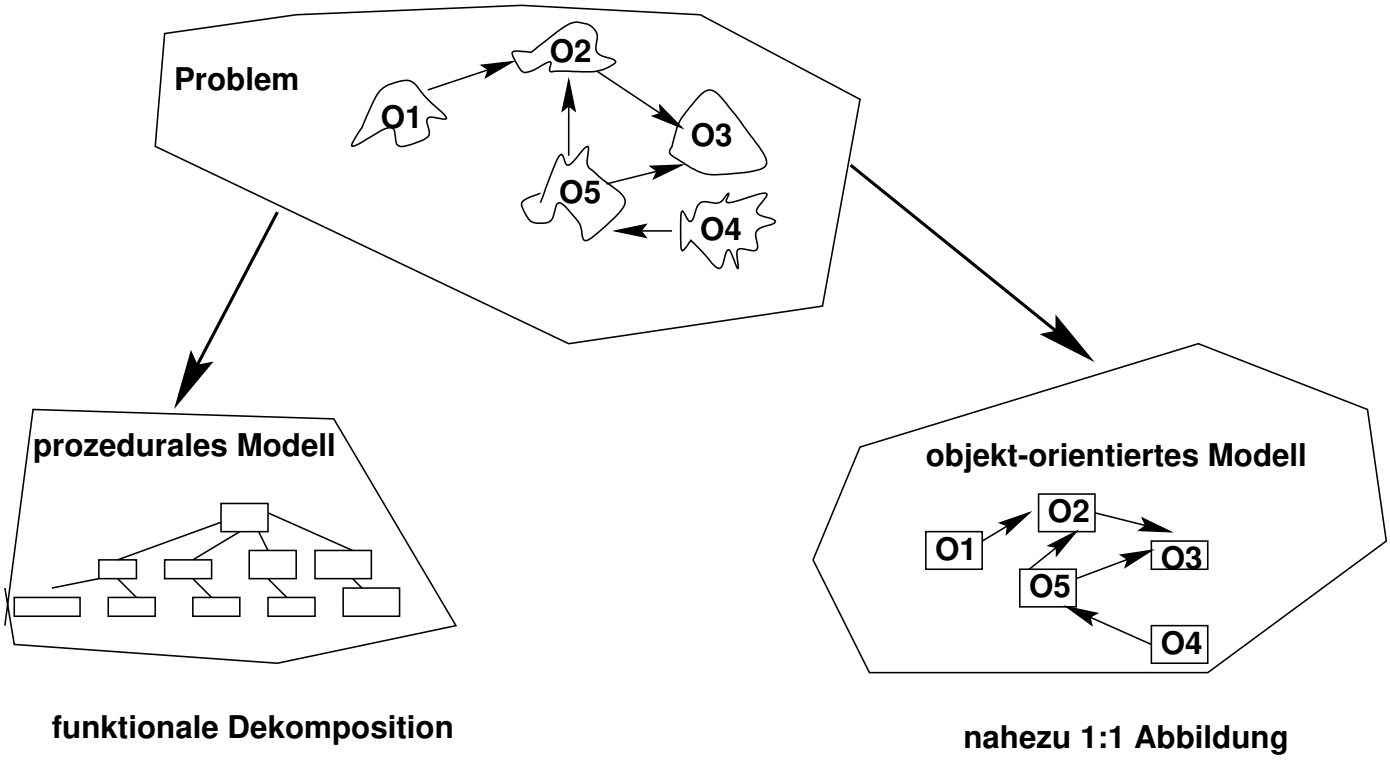
Die **reale Welt besteht**

- nicht aus Datenstrukturen und Prozeduren (Funktionen),
- sondern aus **Objekten**, diese
  - haben bestimmte **Eigenschaften**
  - reagieren mit bestimmten **Verhaltensweisen** auf die Umwelt.

Objektorientierte Vorgehensweise 3

Ansatz:

- natürliche Modellierung des Problembereiches
- durch eine direkte Abbildung der Objekte der realen Welt
- auf Objekte der Computer Welt, diese sind eine vereinfachte Nachbildung der Objekte der realen Welt.



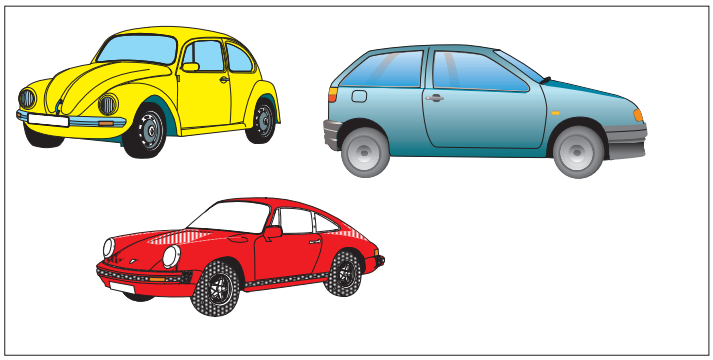
Objektorientierte Vorgehensweise erfordert die Welt als Sammlung miteinander kommunizierender, sich wechselseitig beeinflußender und selbständig agierender Objekte aufzufassen

Objektorientierte Vorgehensweise 4

- Die objektorientierte Vorgehensweise verwendet in allen 3 Phasen der Software-Entwicklung, **Analyse (OOA)**, **Design (OOD)** und **Programmierung (OOP)** die selben einheitlichen Konzepte.
- Die **Grundbausteine sind die Objekte**, das sind genau die stabilsten Komponenten eines Problembereiches, die wenig anfällig für Änderungen sind.
- Dagegen ändern sich funktionale Anforderungen sehr oft.
- Die Objektorientierte Vorgehensweise hat sich seit Anfang der 90 Jahre des letzten Jahrhunderts als Standardmethode der Softwareentwicklung etabliert.
- Sie unterstützt alle folgenden Kriterien für gute Software:
  - **Korrektheit:** Software soll genau die Funktionalitäten besitzen, die von ihren Benutzern erwartet wird.
  - **Benutzerfreundlichkeit:** Software soll einfach und intuitiv zu benutzen sein.
  - **Effizienz:** Software soll mit wenigen Ressourcen auskommen und kurze Antwortzeiten haben.
  - **Beherrschung der Komplexität:** Software sollte in Hinblick auf die Komplexität beherrschbar und leicht testbar sein und auch bleiben.
  - **Wartbarkeit:** Software soll mit wenig Aufwand erweiterbar und änderbar sein.
- Die modernen Programmiersprachen **Java**, **C++**, **JavaScript**, **C#**, **Ruby**, **Python** unterstützen alle das Objektorientierte Programmier-Paradigma.

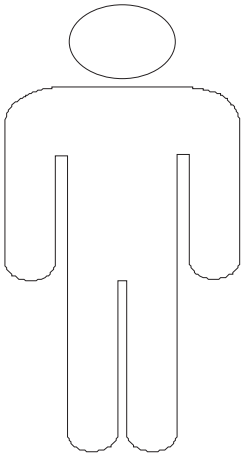
- Aber, die Nutzung der objektorientierten Vorgehensweise und einer objektorientierten Programmiersprache führen nicht automatisch zu objektorientierten und guten Programmen!!

3.2. Klassifizierung/Abstraktion



Klasse Auto

Klasse Person



Die Nachbildung der realen Welt im Computer erfordert:

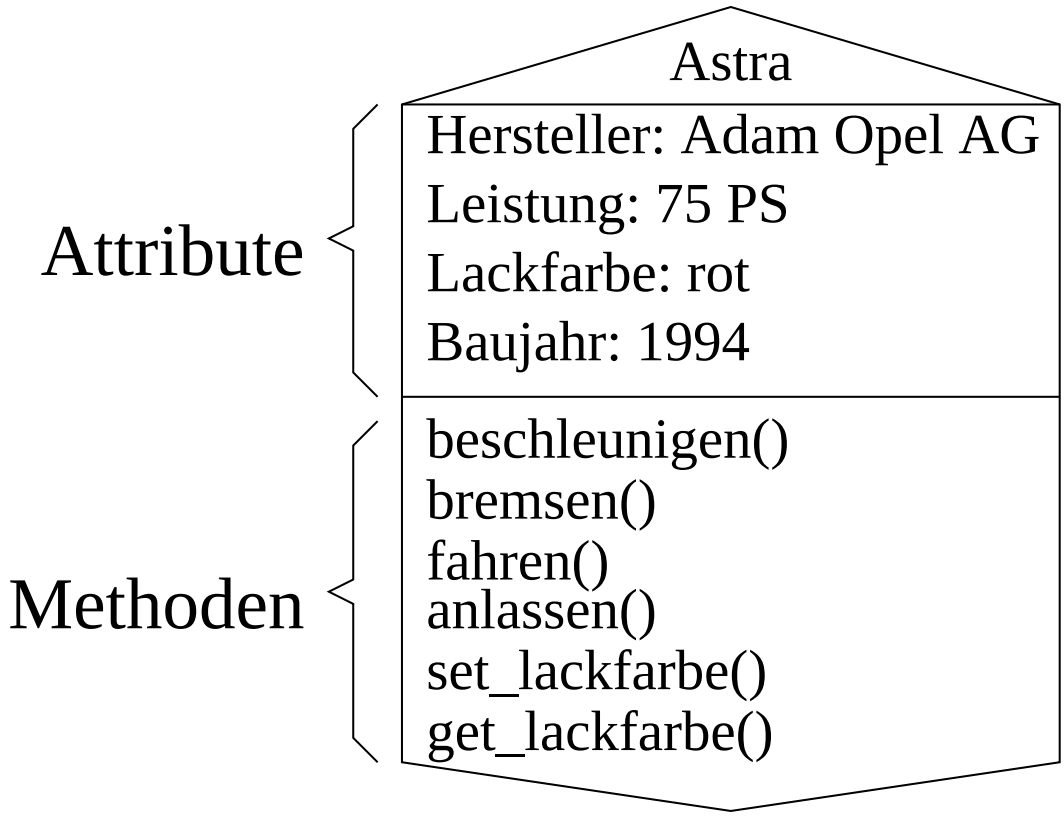
- **Klassifizierung:** Objekte können in **Klassen** (mit gleichen Eigenschaften und Verhaltensweisen) eingeteilt werden. Dabei werden relevante Gemeinsamkeiten von Objekten identifiziert und Objekte mit diesen Gemeinsamkeiten derselben Klasse zugeordnet.
- **Abstraktion:** Konzentration auf die **wesentlichen Eigenschaften** und **Verhaltensweisen** der Objekte bei der Nachbildung im Computer.

3.3. Objekte in der Computer-Welt

Objekt = Daten + Funktionalität

Objekte Besitzen:

1. **Attribute (Daten)**
   beschreiben Eigenschaften (Zustand) des Objektes
2. **Methoden (Funktionen)**
   beschreiben Verhalten des Objektes
   (Verben im Pflichtenheft)
3. **Identität** (z.B. Speicher-Adresse)



Mögliche Kategorien für Objekte

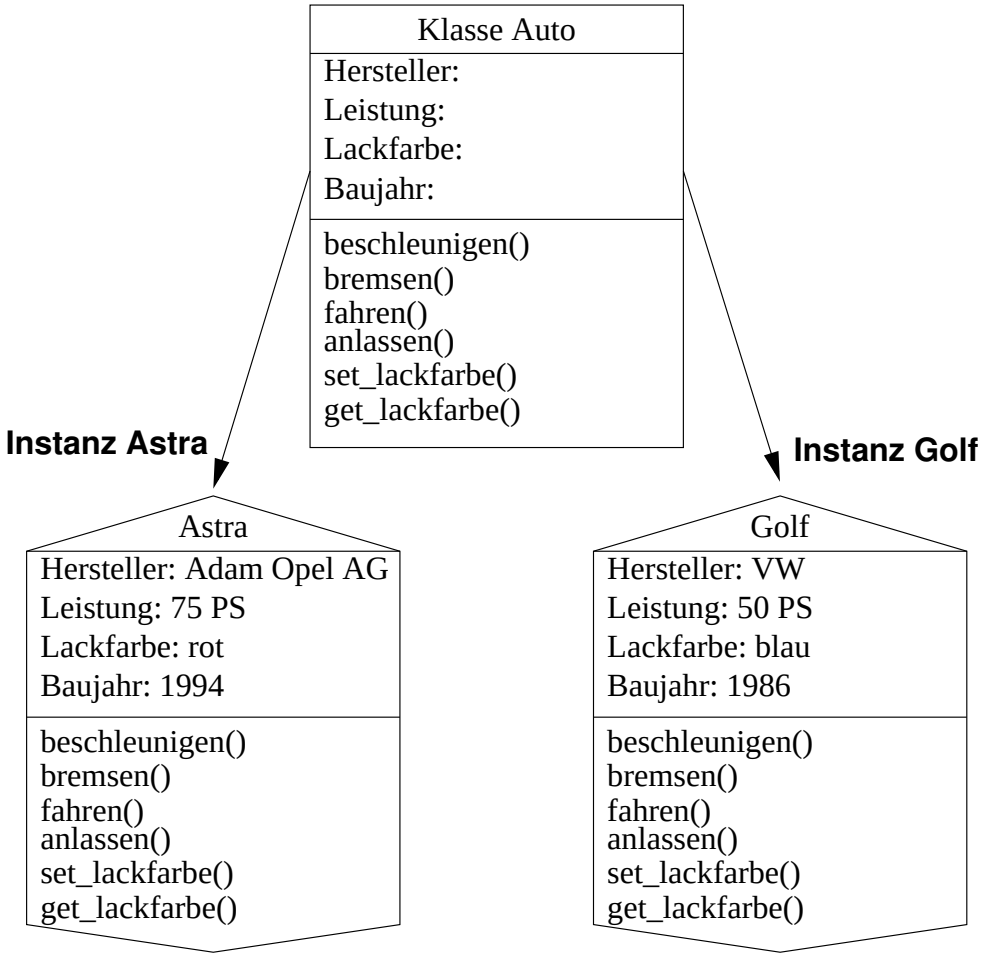
- Substantive im Pflichtenheft (Anforderungsbeschreibung)
- z.B.: Reale physische Dinge, Rollen, Ereignisse, Interaktionen.

3.4. Klasse als Datentyp, Instanz, Objekt

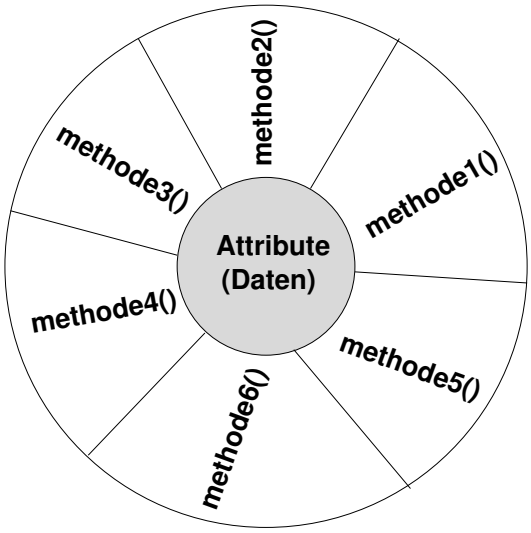
- Objekte sind klassifizierbar (Einteilung in Klassen).
- Eine **Klasse** ist eine abstrakte Beschreibung (Typ) von gleichartigen Objekten. Ihr Typ ist dann genau die Klasse.
- Eine Klasse ist eine Schablone (Bauplan), um gleiche Objekte zu erzeugen,

  Vorstellung: *Prägestempel* zum Ausstanzen von Objekten.
- Ein **Objekt** ist die Ausprägung (**Instanz**) einer Klasse.

Beispiel:



3.5. Datenkapselung (Information Hiding)



- Auf Daten eines Objektes kann nur über seine Methoden zugegriffen werden, **Geheimnisprinzip**, z.B.: `set_lackfarbe()`, `get_lackfarbe()`
- Der Zustand eines Objektes kann sich nur durch den Aufruf von Methoden ändern.
- Das Objekt kann dadurch selbst dafür sorgen, dass die Konsistenz der Daten erhalten bleibt (z.B. Radius eines Kreises ist positiv).
- Die Zusammenfassung aller Methoden eines Objektes stellt eine einheitliche **Schnittstelle** dar, die genau die Funktionalität eines Objektes beschreibt. Sie legt einen **Kontrakt (Vertrag)** für die Verwendung des Objektes fest.
- Die Art der Speicherung der Daten und die Details der Implementierung werden nach außen verborgen.
- Ein Objekt kapselt also die Implementierung der Operationen und trennt so die Schnittstelle des Objekts von der Implementierung.
- Daten sind aktiv, führen Operationen an sich selber aus z.B.: `x.printMe( )` anstelle von `print( x )`

Datenkapselung (Information Hiding) 2

Vorteile:

- Schutz vor ungewollten Zugriffen (erhöhte Sicherheit).
- Die Konsistenz von Daten kann wesentlich einfacher sicher gestellt werden.
- Damit ist es auch einfacher die Korrektheit eines Programms zu gewährleisten. Die Fehlersuche beschränkt sich auf kleine überschabare Programmteile.
- Die Software wird auch leichter testbar.

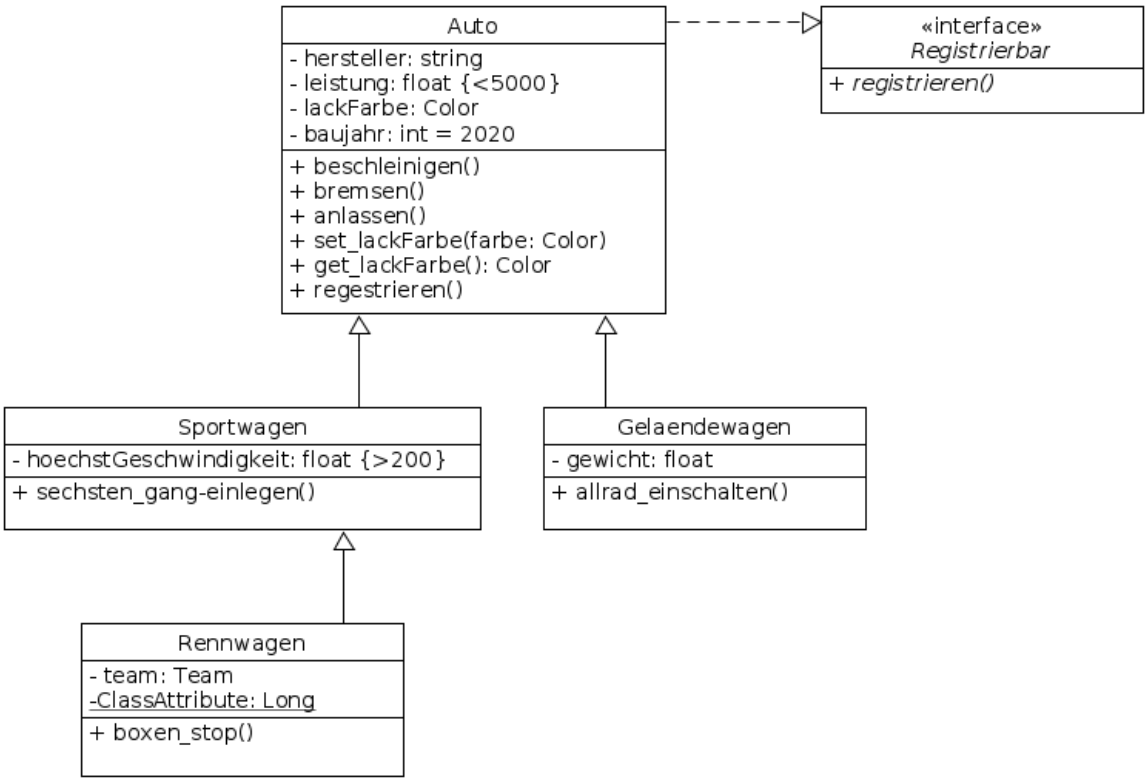
  Hierzu formuliert man auch oft für den Aufruf einer Methode:

  - **Vorbedingungen (Preconditions)**, die vor dem Aufruf einer Methode erfüllt sein müssen.
  - **Nachbedingungen (Postconditions)**, die nach dem Aufruf einer Methode erfüllt sein müssen.
  - **Invarianten (Invariants)**, das sind Eigenschaften eines Objektes, die durch einen Methodenaufruf nicht geändert werden können.

  In der UML können diese Bedingungen mit der Hilfe der **Object Constraint Language (OCL)** formuliert werden.
- Die interne Art der Datenspeicherung und andere Änderungen bezüglich der Innenstruktur eines Objektes können geändert werden und an neue Anforderungen angepasst werden, solange die Schnittstelle des Objektes nicht verändert wird.
- Sie erfordern dann keine weiteren Änderungen oder Anpassungen in anderen Klassen.
- Dadurch reduziert sich die Abhängigkeit zwischen Software-Komponenten (Code-Faktorisierung).
- Die Software wird leicht wartbar, erweiterbar und anpassbar.

3.6. Vererbung (Inheritance)

- Ist **Weitergabe von Attributen und Methoden**
  einer **Basisklasse (Superclass)**
  an eine **Unterklasse (Subclass)**.
- Sie wird nur zur **Spezialisierung** benutzt:
  - Durch das **Überschreiben** von Methoden.
  - Oder durch das Hinzufügen weiterer Attribute und Methoden.
- Meist über mehrere Stufen $\Longrightarrow$ **Klassenhierarchien**



Vorteil:

- Vermeidung von Code-Wiederholungen.
- Leichte Wiederverwendbarkeit von Code (Code-Sharing).
- Einfache Erweiterung der Funktionalität.

3.7. Polymorphie

Nachrichten:

- Der Programmablauf erfolgt durch **Wechselwirkung der Objekte** untereinander.
- Hierzu senden und empfangen Objekte **Nachrichten (messages)**.
- Programmtechnisch entspricht dem Senden einer Nachricht der Aufruf einer Methode eines Objektes.

Polymorphie (Vielgestaltigkeit):

- Das Senden derselben Nachricht an verschiedene Objekte einer Vererbungshierarchie ruft unterschiedliche Reaktionen hervor
  z.B.: Nachricht `x.drucke()` an die Objekte

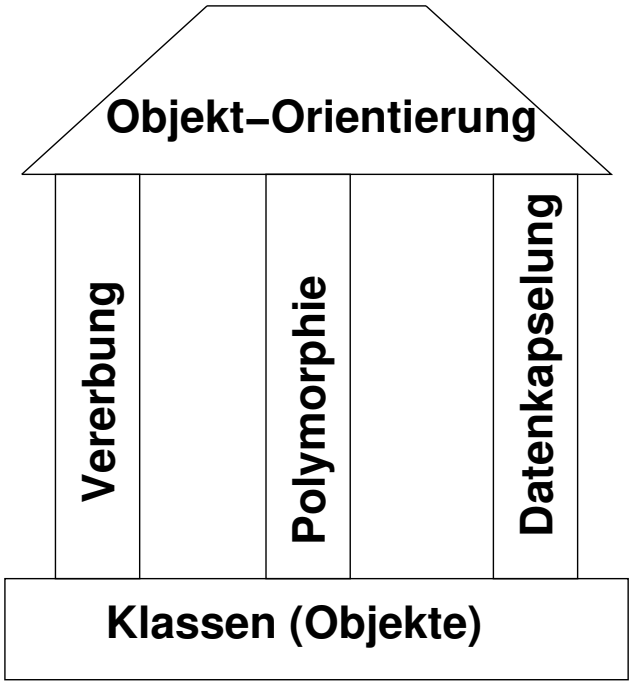
```
Kontoauszug x;
Kundenliste x;
Wechselkurse x;
```

- Erst zur Laufzeit steht i. a. fest, welche Methode verwendet wird (**späte Bindung, dynamische Bindung**).
- Unterschiedliche Objekte können gleichartig behandelt werden, wenn sie die gleiche Schnittstelle (Methode) besitzen.
- Es entfallen komplizierte, fehleranfällige Fallunterscheidungen.
- Polymorphie erhöht die Code Flexibilität und die Wiederverwendbarkeit.
- Nachteil: Polymorphie verschlechtert die Run-Time Performance bedingt durch die dynamische Bindung.

3.8. Zusammenfassung Objektorientierung

- Objekte sind die zentralen Bausteine.
- Ein Objekt ist eine Einheit aus Attributen (Daten) und Methoden.
- Der Zugriff auf die Daten des Objektes ist nur über Methoden möglich Datenkapselung
- Objekte tauschen gegenseitig Nachrichten aus, dadurch erfolgt der Programmablauf.
- Objekte sind die Instanz einer Klasse, eine Klasse ist Blaupause für Objekte gleicher Struktur und ein benutzerdefinierter Datentyp in einer Programmiersprache.
- Es gibt Klassenhierarchien, dabei übernehmen durch Vererbung Unterklassen Attribute und Methoden aus den oberen Klassen oder passen geerbte Methoden an (Überschreiben).
- Das Senden derselben Nachricht an verschiedene Objekte einer Vererbungshierarchie ruft unterschiedliche Reaktionen hervor: Polymorphie

3.8. Haus der Objekt-Orientierung



Fundament: **Objekte**, die Instanzen von Klassen sind

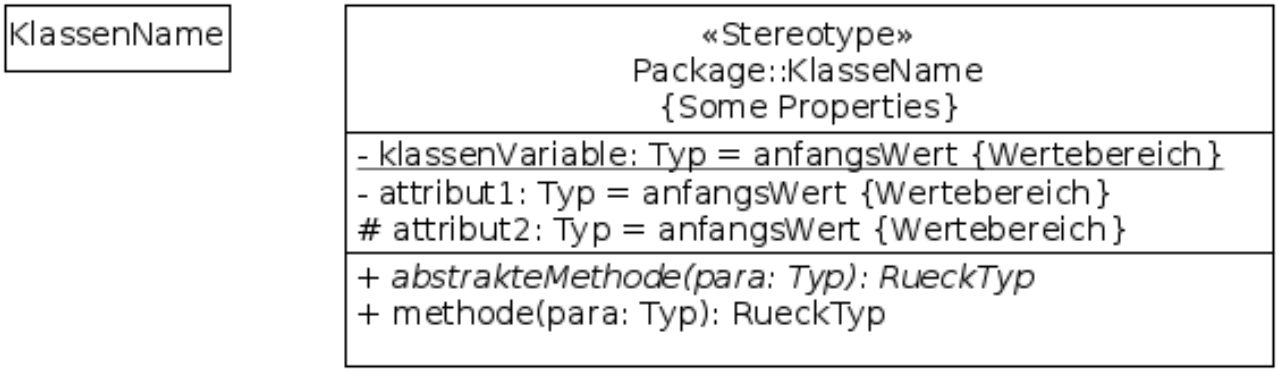
Säulen:

1. **Vererbung**
2. **Polymorphismus**
3. **Datenkapselung**

OOP: Objektorientierte Programmierung bedeutet:

1. Erstellung von Klassen
2. Erzeugung beliebig vieler Objekte (Instanzen)
3. Objekte interagieren miteinander durch Nachrichtenaustausch (Aufruf von Methoden) (Daten und Implementierungsdetails sind verborgen)

3.9. UML Klassendiagramm



- Die **Unified Modeling Language (vereinheitlichte Modellierungssprache), kurz UML** ist eine grafische Modellierungssprache zur Spezifikation, Konstruktion, Dokumentation und Visualisierung von Software und anderen Systemen.
- Mit UML **Klassendiagrammen** kann der Aufbau einer Klasse beschrieben werden.
- Ein Klassendiagramm besteht aus einem Rechteck, das in 3 Teil-Rechtecke unterteilt sein kann.
- Im oberen Rechteck wird der Klassen-Name, im mittleren Teil werden die Attribute und im unteren Teil werden die Methoden eingetragen.
- **Statische Member (Klassen-Variablen, Klassen Methoden)**, die nicht an ein Objekt gebunden sind, werden durch **Unterstreichen** kenntlich gemacht.
- **Abstrakte Methoden**, die nur eine Schnittstelle definieren, aber nicht implementiert werden, kennzeichnet man durch **kursive Schrift**.
- Entsprechend wird auch der Name von **Abstrakten Klassen** und **Schnittstellen (interface)**, die (nur) abstrakte Methoden besitzen, kursiv geschrieben.

UML Klassendiagramm 2

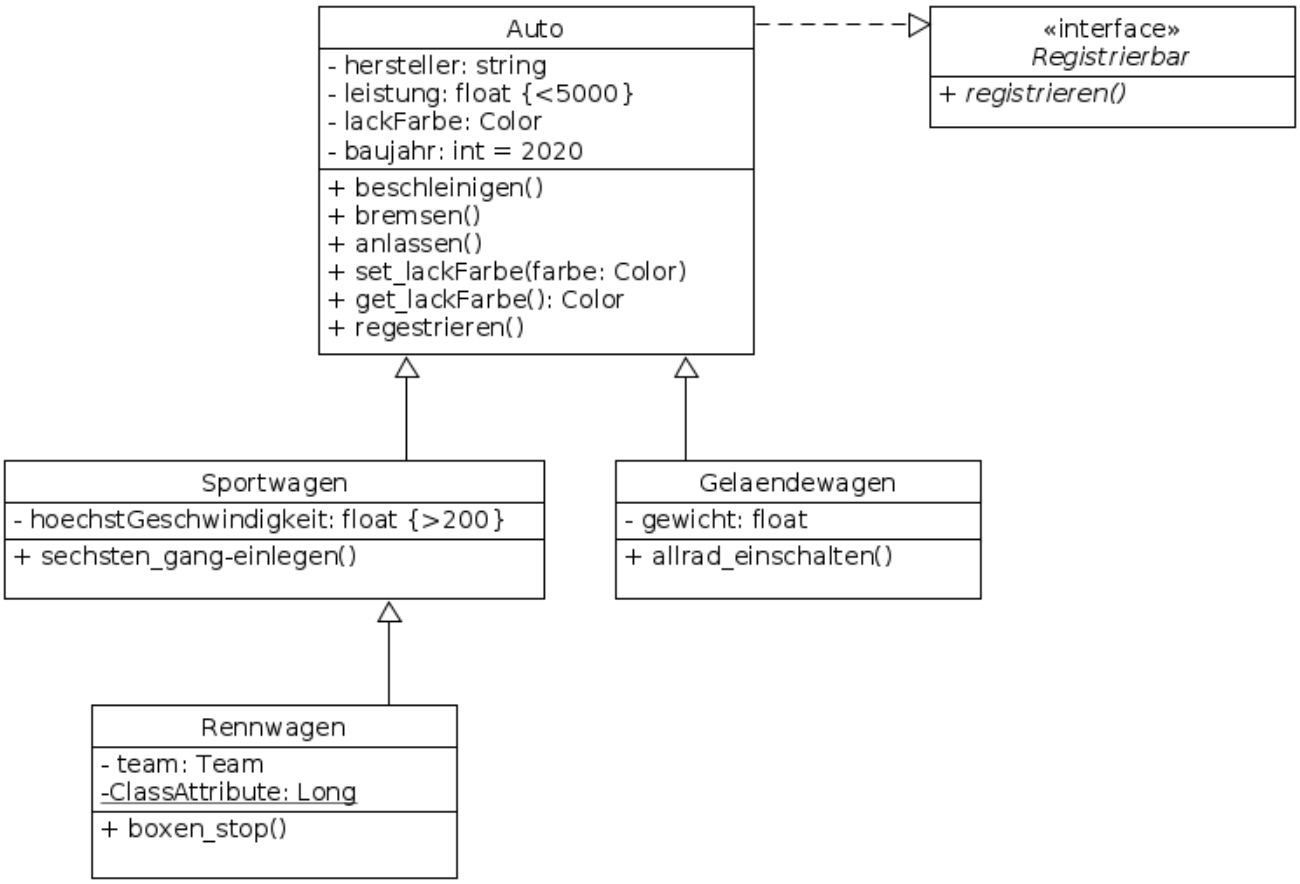
Zugriffs-Rechte werden durch folgende Zeichen angegeben:

- private: –
  (Zugriff nur in der eigenen Klasse)

- protected: #
  (Zugriff in der eigenen Klasse und in allen Unterklassen, s.u.)

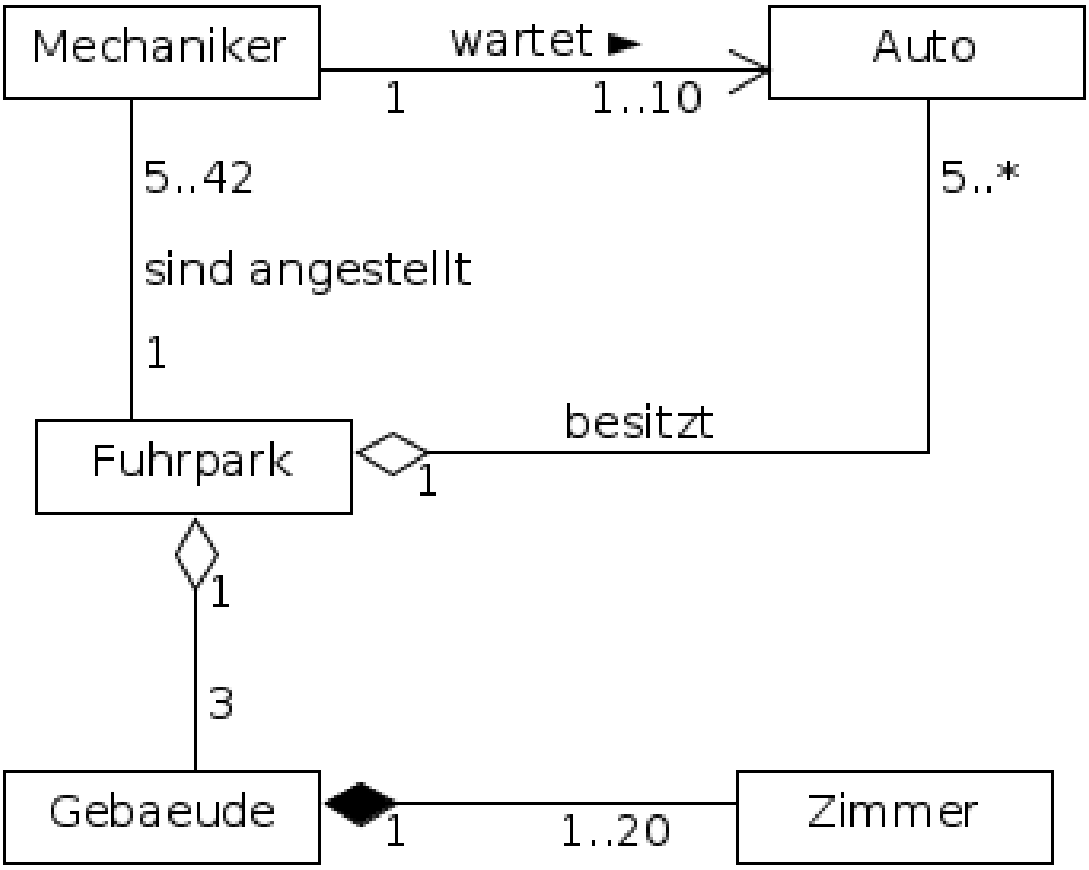
- public: +
  (Zugriff überall)

3.9.1. Vererbung



- Eine Klasse kann von einer anderen Klasse Attribute und Methoden erben.
- Die Klasse, von der geerbt wird, bezeichnet man als **Basisklasse, Oberklasse**.
- Die Klasse, die erbt, bezeichnet man als **Unterklasse, abgeleitete Klasse, Kind Klasse**.
- Den Vorgang des Erbens nennt man meist **Ableitung** oder **Spezialisierung**, die Umkehrung heißt **Generalisierung**.
- Im Klassendiagramm wird die Vererbung durch einen offenen Pfeil von der Unterklasse zur Oberklasse kenntlich gemacht.
- Die Vererbung erhöht die Wiederverwendbarkeit von Code. Typisch sind **Vererbungs-Hierarchien**.

3.9.2. Assoziation

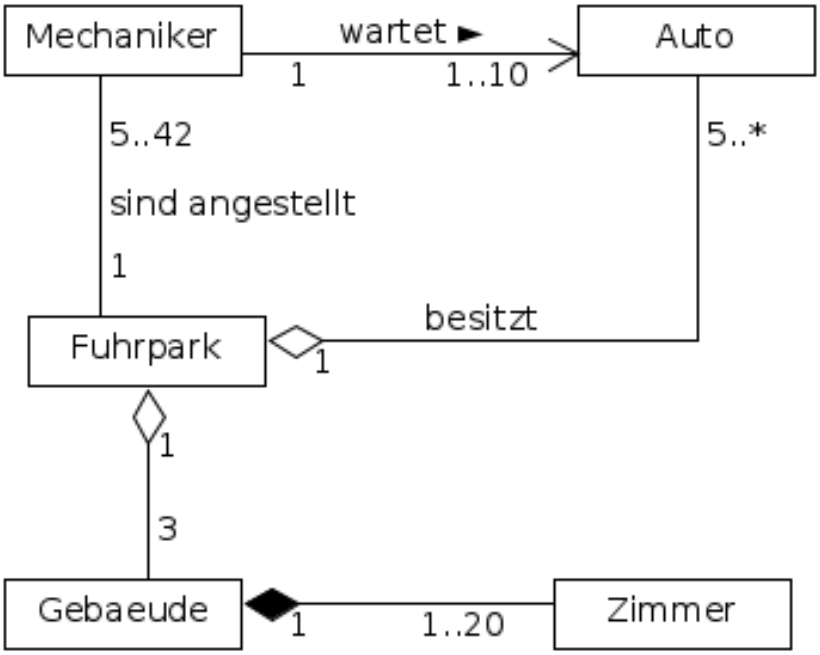


- Eine **Assoziation** beschreibt eine Beziehung zwischen Klassen.

```
mechanikerMike.wartet(lkw23);
```

- In UML-Notation wird eine Assoziation durch eine Verbindungslinie zwischen den Klassen dargestellt.
- Durch einen Pfeil kann die Richtung der Assoziation angegeben werden.
- Die **Multiplizität** gibt an wie viel Objekte aus beiden Klassen miteinander in Beziehung stehen.
  Gültige Notationen sind z.B.: `3`, `0..7`, `*`, `2..*`.

3.9.3. Aggregation, Komposition



- Eine spezielle Form der Assoziation ist die **Aggregation**. Sie liegt vor wenn eine Klasse (das Aggregat) Objekte einer anderen Klasse als Bestandteile hat. Es besteht eine **Teil-Ganzes-Beziehung (has a)**, die im UML-Diagramm durch eine **offene Raute am Aggregat** kenntlich gemacht wird.

```
class Fuhrpark{
  private:
  Auto fahrzeuge[10];
.....
}
```

- Die **Komposition** ist eine strenge Form der Aggregation und liegt dann vor, wenn die **Bestandteil-Objekte nicht ohne das Aggregat eigenständig existieren können**. Sie wird im UML-Diagramm durch eine **ausgefüllte Raute auf der Seite des Aggregates** gekennzeichnet.

3.10. Aufgabe OOA und OOD

Man analysiere die folgenden Anforderungen (Auffinden von Objekten) und erstelle ein objektorientiertes Design, das als Klassendiagramm (z. B. mit Visual Paradigm oder UMlet) zu zeichnen ist.

Man beachte, dass man nur die für die geforderte Anwendung wichtigen Attribute und Methoden berücksichtigt, die Beschreibung ist extra mit viel Kladder (lorem ipsum) aufgefüllt!

Die Software soll so entwickelt werden, dass sie ohne weitere Änderungen auch an andere KFZ-Zulassungsstellen verkauft werden kann.

Die KFZ-Zulassungsstelle im Landkreis Musterland möchte, obwohl sie viele rege, fleißige und kluge Mitarbeiter hat, die Berechnung der KFZ-Steuer nicht mehr händisch sondern über eine Software vornehmen, die es im folgenden zu entwerfen gilt (und die in gespannter Erwartung auch später zu programmieren ist).

Ein Halter hat die Möglichkeit, bei der KFZ-Zulassungsstelle Fahrzeuge anzumelden und abzumelden. Der Einfachheit halber nehmen wir an, dass der Landkreis eine sehr kleine Einwohnerzahl hat und ein Halter eindeutig über seinen Familiennamen identifiziert werden kann.

Da die Bevölkerung im Landkreis sehr wohlhabend ist, gibt es keine Beschränkung hinsichtlich der Anzahl der Fahrzeuge, die ein Halter anmelden kann.

Bei der Anmeldung eines Fahrzeuges soll die Software mittels eines Kennzeichen-Generators ein neues zufälliges Kennzeichen, das dem Fahrzeug zugeordnet wird, generieren.

Ein gültiges Kennzeichen wird aus dem Ortskürzel "`Mu`", 2 zufälligen Großbuchstaben, die nicht aus Umlauten bestehen dürfen, und einer abschließenden zufälligen Nummer z mit $500 \leq z \leq 9999$ gebildet.

Die KFZ-Steuer beträgt für normale Fahrzeuge pro Jahr 5 Euro je 100 ccm des Hubraumes.

Aufgabe OOA und OOD 2

Der dynamische Herr Mustermann, der als Manager einer Versicherung arbeitet, fährt von März bis Oktober auch gerne mit seinem schicken metallic rotem Cabriolet, der mit Alu-Sport-Felgen und einem Carbon-Spoiler ausgestattet ist, einen Motor mit 400 Ps hat und eine Höchstgeschwindigkeit von 320 kmh erreichen kann, bei schönem Wetter durch den Landkreis. Die KFZ-Steuer für dieses Saisonfahrzeug beträgt pro angemeldeten Monat 1 Euro je 200 ccm Hubraum.

Der Unterbrunner Xaver, Landwirt in 8-ter Generation, besitzt neben einem silbergrauen Mercedes Benz Diesel auch einen grünen Traktor der Firma Fendt, Baujahr 1999 mit 7000 Ps. Er hat den Traktor am Stammtisch zu später Stunde dem Rechenberger Sepp für 13 742 Euro abgekauft und ihm dafür versprochen 2 mal im Jahr die Wiese am Dorfteich zu mähen und das angefallene Heu ihm zu überlassen.

Für dieses Nutzfahrzeug berechnet sich die KFZ-Steuer aus dem ganzzahligen Hubraum multipliziert mit der ganzzahligen jährlich zurückgelegten Kilometer-Fahrstrecke multipliziert mit 2 Cents dividiert durch 10 000.

Mit der Software soll es möglich sein die Gesamt-Einnahmen pro Jahr durch die KFZ-Steuer zu berechnen.

Da der Amtsvorsteher Herr Biedermann, der seit 1989 in der KFZ-Zulassungsstelle arbeitet und glücklich mit der Fischers Erika verheiratet ist, gerne einen Gesamtüberblick über die Einnahmen durch die KFZ-Steuer hat, soll es mit der Software auch möglich sein, eine Liste auszudrucken, in der für jeden Halter die KFZ-Steuer für die angemeldeten Fahrzeuge einzeln ausgewiesen wird. Es sollen dabei auch für jedes Fahrzeug die Daten, die für die Berechnung der KFZ-Steuer notwendig sind, mit ausgedruckt werden. Weiterhin soll auch die Summe, die jeder Halter als KFZ-Steuer zu zahlen hat und die Gesamteinnahmen des Landkreises durch die KFZ-Steuer ausgegeben werden.

4. Klassen

4.1. Einführung

- Klassen (`class`) ermöglichen es in C++ **neue benutzerdefinierte Typen** zu erstellen, die genau an den Problembereich angepaßt sind.

- Durch das Überladen von Operatoren (s. Kap. 5)
  (z.B. `+`, `++`, `[]`, `<<`)

  hat man die Möglichkeit, neue Typen zu erstellen, die sich wie Grunddatentypen verhalten
  (auf die diese überladenen Operatoren angewendet werden können)

- Eine Klasse besteht aus einem Satz von **Membern**,
  die gebräuchlichsten sind:

  – **Daten-Member (Attribute)**
  – **Member-Funktionen (Methoden)**

- Eine Klasse bildet einen **eigenen Namensraum (namespace)**, der ihre **Member** enthält.

  Der Name des Namensraumes ist genau der Name der Klasse.

- Klassen sind ein Bauplan für Objekte.

- Durch das Instanzieren kann aus einer Klasse ein Objekt erstellt werden.

- Ein Objekt ist die **Instanz einer Klasse**.

4.2. Definition von Klassen

Man beachte das Semikolon am Ende der Definition!

```
class Klassenname {

private:
  attributDeklarationen
  methodenDeklarationen

protected:
  attributDeklarationen
  methodenDeklarationen

public:
  attributDeklarationen
  methodenDeklarationen

};
```

4.2.1. Zugriffskontrolle durch Modifikatoren

Modifikator	Bedeutung
`public`	öffentlich: Zugriff ist überall erlaubt
`private`	privat: Zugriff ist nur innerhalb der Klasse erlaubt
`protected`	geschützt: Zugriff ist in der Klasse und in Unterklassen erlaubt

- Ein **Modifikator** steuert, wer ein Attribut / eine Methode verwenden darf.
- Die Modifikatoren dürfen in **beliebiger Reihenfolge und mehrfach vorkommen**.
- Ein Modifikator gilt solange, bis er durch einen anderen Modifikator aufgehoben wird.

Zugriffskontrolle durch Modifikatoren 2

- Wenn kein Modifikator angegeben ist gilt in Klassen (`class`) immer der Modifikator `private`.
- Guter Programmierstil ist, **jeden der 3 verschiedenen Modifikatoren nur 1 mal zu verwenden**.
- Und in den Klassen eines Projektes die **Modifikatoren immer in der gleichen Reihenfolge anzuordnen**.
- Über die Reihenfolge der Modifikatoren innerhalb der Klassen-Definition gibt es in der OOP-Gemeinde einen großen Streit.
- Für einen Programmierer einer eigenen Klasse hat die Anordnung `private` vor `public` Vorteile, da er mit den privaten Membern einer Klasse arbeiten muss.
- Um sich Informationen über die privaten (`private`) Member zu verschaffen hat er beim Öffnen einer Header-Datei diese dann sofort im Blick und muss nicht nach unten scrollen.
- Diese Reihenfolge entspricht in der Regel auch der Reihenfolge, die im UML-Diagramm für die Mehrzahl der Member gilt, da Attribute immer `private` sind und Methoden bis auf private Hilfs-Methoden immer öffentlich (`public`) sind.
- Diese Reihenfolge ist deshalb auch für Anfänger einfacher.

Zugriffskontrolle durch Modifikatoren 3

- Ein Benutzer einer Klasse kann dagegen nur mit den öffentlichen (`public`) Membern einer Klasse arbeiten.
- Deshalb bilden die öffentlichen Member die **Schnittstelle einer Klasse zur Außenwelt**.
- Leider kann man in C++ bei der Definition einer Klasse diese Schnittstelle nur sehr aufwendig und umständlich in eine eigene Datei auslagern um sie so von den privaten Membern zu trennen.

  Man müsste dann zu jeder Klasse die man erstellt:

  - Eine Schnittstellen-Klasse (Abstrakte-Klasse) erstellen, in der man die öffentlichen Member einer Klasse deklariert.
  - Und zusätzliche eine Implementierungs-Klasse erstellen, in der alle Member definiert (bei Methoden also implementiert) werden.
- Für Benutzer einer Klasse ist deshalb die Anordnung `public` vor `private` vorteilhaft, da sie sofort die Schnittstelle eine Klasse anzeigt.
- In großen Software-Projekten wird deshalb oft diese Reihenfolge verwendet.

4.2.2. Strukturen struct

- Eine **Struktur** (`struct`) in C++ ist genau so aufgebaut wie eine Klasse.
- Defaultmäßig, also wenn die Modifikatoren fehlen, besitzen aber alle Member einer Struktur die Zugriffs-Beschränkung `public`.

```
struct StrukturName {

private:
  attributDeklarationen
  methodenDeklarationen

protected:
  attributDeklarationen
  methodenDeklarationen

public:
  attributDeklarationen
  methodenDeklarationen

};
```

4.2.3. Daten-Member (Attribute)

- Die Daten-Member (Attribute) einer Klasse sind jeweils immer an ein Objekt der Klasse gebunden.
- Für sie wird erst durch die Definition (Erstellung) eines Objektes Speicher allokiert.
- Sie existieren pro Objekt, jedes Objekt besitzt seine eigenen Attribute.
- Die Werte der Attribute bestimmen den Zustand eines Objektes.
- Ausnahme sind `static`-Member (Klassenvariablen) (s.u.), die nur ein mal pro Klasse existieren.
- `static`-Member (Klassenvariablen) können auch ohne die Erstellung eines Objektes verwendet werden.
- Bsp: Klasse Datum, die zunächst nur Attribute hat.
- Der **Include-Wächter (include guard)** `#ifndef DATUM_H` verhindert das mehrfache Includieren der Klassen-Definition. Nach dem ersten Inkludieren ist die Variable `DATUM_H` gesetzt.

  Datei `datum.h`

```
#ifndef DATUM_H
#define DATUM_H
class Datum {
 private:
  // Attribute der Klasse
  int jahr, monat, tag;

 public:
   // es fehlen noch die Methoden

};
#endif  // DATUM_H
```

4.2.4. Member-Funktionen (Methoden)

- In C++ ist es üblich und sinnvoll, die Deklaration und die Definition (Implementierung) einer Member-Funktion voneinander zu trennen.
- **Deklaration der Member-Funktionen:**
  - beschreibt die **Schnittstelle**, wie ein Benutzer eine Klasse verwenden kann.
  - In der Klassen-Definition, die in einer **Header-Datei** erfolgt, wird deshalb nur der **Prototyp** der Member-Funktion angegeben.
- **Definition (Implementierung) der Member-Funktionen:**
  - Beinhaltet oft viele Details, die vom Benutzer der Klasse nicht benötigt werden.
  - Sie erfolgt deshalb in einer getrennten **Implementierungs-Datei (cpp-Datei)**, die vorübersetzt und ggf. als Bibliothek vom Linker gebunden werden kann.
  - Es muss dann der voll qualifizierte Name angegeben werden:

```
Typ Klassenname::funktionsname(
  parameterliste) {
  // ...
}
```

  - Jeder Member-Funktion wird beim Aufruf der verborgene konstante Pointer `Klassenname* const this;` auf das aktuelle Objekt übergeben.
  - Innerhalb der Member-Funktion kann auf alle Member der Klasse direkt über ihren Namen (ohne `this`) zugegriffen werden.

Bsp. Member-Funktionen (Methoden)

- Klassen-Definition erweitert um die Prototypen der Member-Funktionen:

  Datei `datum.h`

```
#ifndef DATUM_H
#define DATUM_H
class Datum {
 private:
  // Attribute der Klasse
  int jahr, monat, tag;

 public:
   // Member-Funktionen (Methoden)
   void setDatum(int j, int m, int t);
   void printDatum() const;

};
#endif  // DATUM_H
```

- Implementierung der Member-Funktionen in einer `cpp`-Datei:

  Datei `datum.cpp`

```
// hier muss der voll qualifizierte Name
// verwendet werden
void Datum::setDatum(int j, int m, int t) {
      // es ist nicht notwendig
      // (wie in Python self.jahr)
      // den this-Pointer, also
      // z.B. this->jahr = j;
      // zu verwenden
      jahr = j;
      monat = m;
      tag = t;
}
void Datum::printDatum() const {
  std::cout << tag << "." << monat << "."
            << jahr << std::endl;
}
```

4.2.5. inline Funktionen/Methoden

```
inline Treuck f(parameterListe) { //.. }
```

Bsp.:

```
inline int max(int x, int y) {
  return x>y ? x : y;
}
```

- Ziel: Vermeidung des *Aufruf-Overheads*.
- Bei kleinen Funktionen besteht der Gesamtaufwand für die Ausführung einer Funktion zu erheblichen Teilen aus dem Aufruf-Overhead (Speicherung der Rücksprung-Adresse und der Übergabe-Parameter auf dem Stack).
- Die `inline` Definition ist eine Empfehlung an den Compiler, den Funktionsaufruf durch den Code des Funktionsrumpfes direkt zu ersetzen, d.h. die Funktion wie ein Makro zu expandieren.
- An jeder Aufrufstelle einer Inline-Funktion kann der Compiler den vollständigen Maschinencode des Funktionsrumpfes anstelle des Funktionsaufrufs einfügen.
- Der Compiler kann der Bitte nachkommen oder sie ignorieren.
- Sinnvoll ist die Inline-Expandierung i.A. nur bei sehr kurzen Funktionen, bei denen der Aufruf- und Rückkehr-Mechanismus im Vergleich zum eigentlichen Funktionscode sehr aufwendig ist.
- Im Gegensatz zu *normalen* Funktionen muss vor dem Aufruf einer Inline-Funktion ihre Definition erfolgt sein.
- Wenn der Compiler nur ihre Deklaration kennt, kann er wegen fehlender Kenntnis ihres Codes keine Expandierung vornehmen.
- Deshalb können Inline-Funktionen nicht in andere Module exportiert werden, also nicht die Speicherklasse `extern` haben.

inline Funktionen/Methoden 2

- Sehr kleine Member-Funktionen, die häufig aufgerufen werden, können innerhalb der Klassen-Definition definiert (implementiert) werden.
- Dadurch werden diese Member-Funktionen als inline deklariert.
- Das Schlüsselwort `inline` ist nicht notwendig.

Bsp. Klassen-Definition mit inline Methode addJahr():

Datei `datum.h`

```
#ifndef DATUM_H
#define DATUM_H
class Datum {
 private:
  // Attribute der Klasse
  int jahr, monat, tag;

 public:
   // Member-Funktionen (Methoden)
   void setDatum(int j, int m, int t);
   void printDatum() const;

   // inline Methode
   Datum addJahr(int n) {
    jahr += n;
    return *this;
  }

};
#endif  // DATUM_H
```

4.2.6. Konstante Member-Funktionen

- Eine Member-Funktion kann mit dem Zusatz `const` vereinbart werden.
- Dadurch wird sichergestellt, dass der Zustand des aktuellen Objektes nicht innerhalb der konstanten Member-Funktion versehentlich verändert werden kann.
- Hierfür muß das Schlüsselwort `const` sowohl bei der Funktionsdeklaration als auch bei der Funktionsdefinition im Funktionskopf nach der Parameterliste angegeben werden.

```
Typ funktionsName( Parameterliste ) const;
```

- Beim Aufruf einer konstanten Member-Funktion wird dann der verborgene konstante Pointer `this` auf das aktuelle Objekt sogar als **Read-Only Pointer** übergeben:

```
const Klassenname* const this;
```

- **Über konstante Objekte dürfen nur konstante Member-Funktionen** aufgerufen werden.
- Innerhalb einer konstanten Member-Funktion ist es nicht möglich, nicht konstante Member-Funktionen aufzurufen.
- Verändert eine Memberfunktion den Zustand des aktuellen Objektes nicht, so sollte sie immer als konstante Member-Funktion vereinbart werden **(const-correctness)**.
- Im obigen Bsp. ist die Methode `printDatum()` eine konstante Member-Funktionen.

4.3. Statische Klassen-Member

- Klassen-Member können auch mit dem zusätzlichen Schlüsselwort `static` vereinbart werden.
- Statische Daten-Member **(Klassenvariablen)** existieren nur ein mal pro Klasse und nicht pro Objekt.
- Statische Daten-Member werden innerhalb der Klassendefinition nur deklariert.
- Ihre Definition (Speicherplatzallokation) **muss außerhalb der Klassendefinition auf der globalen Ebene erfolgen**. Bei der Definition entfällt das Schlüsselwort `static`:

```
class Klassenname {

   static typ varName;

   // ....
};


// In .cpp Datei:
typ Klassenname::varName = wert;
```

- Ab C++17 entfällt die Notwendigkeit, Klassenvariablen außerhalb der Klasse zu definieren.
- Eine Member-Funktion kann ebenfalls mit dem Schlüsselwort `static` innerhalb der Klassen-Definition deklariert werden. Bei der Definition außerhalb der Klassen-Definition entfällt wiederum das Schlüsselwort `static`.
- Statische Member-Funktionen haben Zugriff auf Klassen-Variablen, aber keinen Zugriff auf die nicht statischen Attribute.

Bsp. statische Klassen-Member

Erweiterung der Klasse `Datum` um statische Member:

```
class Datum {
 private:

 // statische Klassenvariablen
  static const int jahrDefault = 1970,
                   monatDefault = 1,
                   tagDefault = 1;

  // private statische Hilfs-Methoden
  static bool checkTag(int m, int t);

  // private statische inline Hilfs-Methode
  static bool checkDatum(int j, int m, int t) {
    return j >= 0 && 1 <= m && m <= 12
           && checkTag(m, t);
  }
  // Attribute
  int jahr, monat, tag;

  // private Member-Funktion
  void setDefault();

 public:
  // jetzt als inline Methode
  void setDatum(int j, int m, int t) {
    if (checkDatum(j, m, t)) {
      jahr = j;
      monat = m;
      tag = t;
    } else {
      setDefault();
    }
  }
  // ...
};
```

Bsp. statische Klassen-Member 2

Definition der statischen Member in der cpp-Datei:

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

// Initialisierung der Klassen-Variablen
// ab C++17 kann dies auch direkt in der
// Klassen-Definition erfolgen
const int Datum::jahrDefault = 1970,
          Datum::monatDefault = 1,
          Datum::tagDefault = 1;

// Definition der statischen Methode checkTag()
// das Schluesselwort static entfaellt
bool Datum::checkTag(int m, int t) {
  bool erg = false;
  switch (m) {
    case 2:
      // Schaltjahre werden ignoriert
      erg = (1 <= t && t <= 28);
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      erg = (1 <= t && t <= 30);
      break;
    default:
      erg = (1 <= t && t <= 31);
      break;
  }
  return erg && (t > 0);
}
```

Bsp. statische Klassen-Member 3

Fortsetzung der cpp-Datei mit der Definition der nicht statischen Member:

```
void Datum::printDatum() const {
  std::cout << tag << "." << monat
            << "." << jahr << std::endl;
}

void Datum::setDefault() {
  jahr = jahrDefault;
  monat = monatDefault;
  tag = tagDefault;
}
```

4.4. Instanzen von Klassen (Objekte)

- Ein durch eine Klassen-Definition eingeführter Klassen-Name kann als Typ verwendet werden.
- Objekte vom Typ einer Klassen können:

  i) **statisch auf dem Stack** gespeichert werden:

  ```
  Klassenname variable; // statisch

  variable.methode();
  ```

  Der Zugriff auf die Member erfolgt dann mittels **Punkt-Notation**.

  ii) oder **dynamisch auf dem Heap** gespeichert werden:

  ```
  Klassenname* ptr = new Klassenname;

  (*ptr).methode();
  // oder einfacher
  ptr->methode();
  ```

  Der Zugriff auf die Member erfolgt entweder durch:

  a) das Dereferenzieren mit dem * Operator und der Punkt-Notation (da der . Operator eine höhere Priorität als der * Operator hat, sind die runden Klammern notwendig)
  b) oder einfacher über die abkürzende Schreibweise mittels ->.

- In beiden Fällen wurde hier der Standard-Konstruktor, der automatisch vom Compiler erzeugt wird wenn kein anderer Konstruktor definiert wurde (s.u.), verwendet.
- Falls andere Konstruktoren in der Klasse definiert wurden, können auch diese zum Instanzieren verwendet werden.

Bsp. Instanzen von Klassen

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

int main() {

  // Statische Instanz
  // (Speicherung auf Stack)
  // Verwendung des Standard-Konstruktors
  // der vom Compiler automatisch erstellt wird
  // der aber die Attribute nicht initialisiert
  Datum heute;

  heute.printDatum();  // deshalb Muell!!

  // nun Initialisierung der Attribute
  heute.setDatum(2019, 10, 18);

  heute.printDatum();  // Ausgabe: 18.10.2019

  // Dynamische Instanz (Heap)
  Datum* pd = new Datum;
  (*pd).setDatum(1900, 1, 1);
  pd->printDatum();
  delete pd;
  pd = nullptr;

  // Verwendung von Smart-Pointern
  // ab C++14
  // die Freigabe des Speichers erfolgt
  // dann automatisch
  std::unique_ptr<Datum> smartPointerD
    = std::make_unique<Datum>();

  smartPointerD->setDatum(2014, 1, 1);
  smartPointerD->printDatum();
  return 0;
}
```

4.4.1. Operationen mit Objekten

1. Initialisierung eines Objektes bei der Erstellung durch ein bestehendes anderes Objekt vom gleichen Typ. Hierfür wird der vom Compiler automatisch erzeugte **Copy-Konstruktor** (s. Kap. 5) verwendet, der eine **bitweise, flache Kopie** erstellt.

```
Datum heute;
heute.setDatum(2019, 10, 18);
// .....
// Copy-Konstruktor
Datum naechstesJahr = heute.addJahr(1);
```

2. **Zuweisung eines Objektes an ein anderes Objekt:**
   - Beide Objekte müssen derselben Klasse angehören.
   - Die Zuweisung bewirkt eine **bitweise Kopie**, d.h. eine **flache Kopie**.
   - Hierfür wird der vom Compiler erstellte **Copy-Zuweisungs-Operator** verwendet (s. Kap. 5).
   - Für eine **tiefe Kopie** muss dieser geeignet Überladen werden, (s. Kap. 5, Rule of Three/Five).

```
Datum d2;
// Copy Zuweisungs-Operator
d2 = naechstesJahr.addJahr(2);
```

3. **Übergabe eines Objektes als Parameter an eine Funktion:**
   - Es findet Wertübergabe statt, d.h das Objekt wird kopiert.
   - Der formale Parameter ist ein neues, temporäres Objekt, das mit dem Wert des aktuellen Parameters initialisiert wird.

     Hierfür wird wiederum der vom Compiler automatisch erzeugte **Copy-Konstruktor** (s. Kap. 5) verwendet.

Operationen mit Objekten 2

4. **Rückgabe eines Objektes als Funktionswert:**

   – Hierbei wird wieder bitweise ein temporäres Objekt kopiert,

   – z.B. Methode `addJahr()` der Klasse Datum:

```
class Datum {
 private:
   // ......
  // Attribute der Klasse
  int jahr, monat, tag;

 public:
   // Member-Funktionen (Methoden)
   void setDatum(int j, int m, int t);
   void printDatum() const;

   // inline Methode
   // bei der Rueckgabe wird
   // durch Kopie ein
   // temporaeres Objekt erstellt
   Datum addJahr(int n) {
    jahr += n;
    return *this;
  }
  // .....
};
```

   – Mittels **Return Value Optimization (RVO), copy elision** vermeiden manche Compiler das Kopieren von temporären Objekten.

   – Diese kann z.B. für das Debuggen ausgeschaltet werden mit dem g++ Compiler-Flag `-fno-elide-constructors`.

   – Im Qt-Creator fügt man in der `.pro` Datei folgendes hinzu:

```
QMAKE_CXXFLAGS += -fno-elide-constructors
```

4.5. Konstruktoren

- Ein Konstruktor ist eine spezielle Member-Funktion einer Klasse, die bei der Erzeugung (statisch oder dynamisch) eines Objekts dieser Klasse automatisch aufgerufen wird.
- Hauptzweck eines Konstruktors ist es die Daten-Member mit gültigen Werten zu Initialisieren.
- Konstruktoren haben immer den gleichen Namen wie die Klasse und besitzen keinen Rückgabetyp.
- Konstruktoren dürfen andere Member-Funktionen der Klasse aufrufen.
- Konstruktoren können auch überladen werden.
- Konstruktoren können auch Default-Parameter besitzen.
- Besitzt ein Konstruktor Parameter, so sind bei jeder Objekt-Erzeugung (Definition bzw Allokation) entsprechende aktuelle Parameter anzugeben.
- Für die Objektdefinition existieren hierbei folgende Syntaxformen:

```
Klassenname objektname =
  Klassenname(aktpar_liste);

// Alternative
Klassenname objektname(aktpar_liste);

// ab C++11 vereinheitlichte
// Initialisierung mitels {}
Klassenname objektname{aktpar_liste};
```

- Die Angabe von zu wenig oder gar keinen Parametern ist fehlerhaft.

4.5.1. Resssourcenbelegung ist Initialisierung (RAII)

- Zu den wichtigsten Aufgaben eines nicht trivialen Programms gehört die Verwaltung von Ressourcen.
- Eine **Ressource** ist etwas, was durch ein Programm angefordert und verwendet wird und danach auch wieder freigegeben werden muss.
- Beispiele für Ressourcen sind z.B.:
  - dynamischer Speicher auf dem Heap,
  - Datei-Handles,
  - Sockets bei der Kommunikation über das Netz,
  - Sperren (locks, Mutex, Semaphore) bei der Thread-Programmierung.
- Bei lang laufenden Programmen kann eine nicht rechtzeitig freigegebene Ressource (ein **Leck**, z.B. memory-leak) zu ernsthaften Leistungseinbußen und möglicherweise zu einem kläglichem Absturz führen.

Resssourcenbelegung ist Initialisierung (RAII) 2

- In C++ wird zur Verwaltung von Ressourcen die Technik

  Ressourcenbelegung ist Initialisierung, Resource Acquisition Is Initialization (RAII)

  angewendet:

  – **Ressorcen werden im Konstruktor, also bei der Initialisierung eines Objektes, in Anspruch genommen**.

  – **Im Destruktor werden alle vom Objekt in Anspruch genommenen Ressourcen wieder frei gegeben**.

- Wird diese Technik immer sauber angewendet, sind Objekte immer selber für die Verwaltung der Ressourcen, verantwortlich. Der Benutzer von Objekten muss sich hierum nicht kümmern.

- Ein Objekt gilt als konstruiert, wenn sein Konstruktor ohne Auslösung einer Exception ausgeführt wurde, also ggf. Ressourcen in Anspruch genommen werden.

- Dann und nur dann wird bei der Stack-Abwicklung (Stack Unwinding) der Destruktor für ein lokales Objekt aufgerufen, um ggf. auch die Ressourcen wieder frei zu geben.

4.5.2. Default-Konstruktor

- Ein Konstruktor ohne Parameter heißt **Default-Konstruktor (Standard-Konstruktor)**.
- Er wird aufgerufen, wenn ein Objekt ohne eine explizite Initialisierung definiert wird.

```
Klassenname var;

// oder ab C++11
Klassenname var{};
```

- *Vorsicht:* **Die folgende Anweisung bewirkt dagegen nicht den Aufruf des Default-Konstruktors!**

```
Klassenname var();
```

  Vielmehr wird hierdurch eine Funktion mit dem Namen `var`, die keine Übergabeparameter hat und die den Rückgabetyp `Klassenname` hat, deklariert!

- Ein Default-Konstruktor setzt normalerweise alle Member-Daten auf Standardwerte.
- Falls für eine Klasse kein Konstruktor definiert ist, erzeugt der Compiler einen **Standard-Default-Konstruktor**, der aber die Daten-Member **nicht initialisiert**.
- Enthält dagegen eine Klasse **mindestens einen Konstruktor, so erstellt der Compiler keinen Standard-Default-Konstruktor**.
- Ein Default-Konstruktor muß dann explizit definiert werden, wenn er zur Verfügung stehen soll.

Bsp. Default-Konstruktor

```
class Datum {
 private:
  static const int jahrDefault = 1970,
                   monatDefault = 1,
                   tagDefault = 1;

 // Attribute
 int jahr, monat, tag;

 public:
  // Vom Programmierer erstellter
  // Default-Konstruktor
  Datum() : jahr{jahrDefault},
            monat{monatDefault},
            tag{tagDefault  { }

  // ...
};
```

<u>Aufruf des Default-Konstruktors:</u>

```
Datum heute;
heute.printDatum();

Datum* pd = new Datum;
pd->printDatum();
```

4.5.3. Konstruktoren mit Member-Initialisierungsliste

- Beim Aufruf eines Konstruktors wird noch vor der Ausführung der Anweisungen im Konstruktor-Rumpf zunächst Speicherplatz für die Attribute reserviert.
- Ohne weitere Angaben wird für jedes Attribut dabei automatisch deren Default-Konstuktor ausgeführt.

  Dadurch werden die Attribute zunächst mit Defaultwerten initialisiert.

  Ihre endgültigen Werte erhalten die Attribute erst bei der Ausführung der Anweisungen im Rumpf des Konstruktors.
- Mit der Hilfe einer **Member-Initialisierungsliste (member initializer list)** gibt es die Möglichkeit, beide Vorgänge in einem Schritt zusammenzufassen, was besonders bei größeren oder vielen Objekten Laufzeitvorteile bringt.
- Eine Member-Initialisierungsliste ist **im Funktionskopf der Konstruktor-Definition (nicht jedoch in der Deklaration) anzugeben**:

```
Klassenname::Klassenname(paraliste)
  : attr1(wert1),
    attr2(wert2), ... {
  // Anweisungen im Rumpf
}
```

Konstruktoren mit Member-Initialisierungsliste 2

- oder ab C++11 gibt es die einheitliche Initialisierung mit geschweiften Klammern:

```
Klassenname::Klassenname(paraliste)
  : attr1{wert1}, attr2{wert2}, ... {
  // Anweisungen im Rumpf
}
```

- Die Einträge in der Initialisierungsliste führen zum Aufruf des Konstruktors des jeweiligen Attributes.
- Die Reihenfolge der Konstruktor-Aufrufe wird durch die **Attribut-Reihenfolge in der Klassen-Definition und nicht durch die Reihenfolge in der Initialisierungsliste bestimmt!**
- Die Attribut-Konstruktoren werden vor der Ausführung der Anweisungen im Rumpf des Konstruktors aufgerufen.

```
class Datum {
 private:
  int jahr, monat, tag;
 public:
  // Konstruktor mit
  // Member-Initialisierungsliste
  Datum(int j, int m, int t)
      : jahr{j}, monat{m}, tag{t} {
    if (!checkDatum(j, m, t)) {
      setDefault();
    }
  }
  // ....
};

Datum d911{2001, 9, 11};
d911.printDatum();
```

4.5.4. Notwendige Existenz eines Default-Konstruktors

```
class Datum {
 private:
  int jahr, monat, tag;
 public:
  Datum(int j, int m, int t)
      : jahr{j}, monat{m}, tag{t} { //....}
};

Datum tagX;
// Compiler Fehler
// Fehler: no matching constructor
// for initialization of 'Datum'
```

- Da im obigen Beispiel in der Klasse `Datum` ein Konstruktor definiert ist, erzeugt der Compiler keinen Default-Konstruktor, auch wenn er bei der Erstellung eines Objektes benötigt wird.
- In diesem Fall muss der Programmierer selber einen z.B. leeren Default-Konstruktor erstellen:

  ```
  class Datum {
   private:
    int jahr, monat, tag;
   public:
    // leerer Default-Konstruktor
    Datum() {}
    Datum(int j, int m, int t)
        : jahr{j}, monat{m}, tag{t} {//....}
  };

  // Nun ok
  Datum tagX;
  ```

- Ab C++11 kann man auch mittels `=default` Standardmethoden aktivieren (s.u.).

4.5.5. Standard-Methoden aktivieren, Methoden deaktivieren

- Ab C++11 können Standard-Methoden (die ggf. vom Compiler automatisch erstellt werden können) aktiviert und Methoden deaktiviert werden:
- **Standard-Methode aktivieren:**

  Dazu genügt es am Ende der Deklaration der Standard-Methode die Auszeichnung `=default` einzufügen.

```
class Datum {
 private:
  int jahr, monat, tag;

  // ......

 public:
  // Default-Konstruktor wird
  // nun erstellt vom Compiler
  Datum() = default;

  Datum(int j, int m, int t)
      : jahr{j}, monat{m}, tag{t} {//.....}
  //....
};

// Nun ok
Datum tagX;
```

Methoden deaktivieren

- **Methoden deaktivieren:**

  Dazu genügt es am Ende der Deklaration der Standard-Methode die Auszeichnung `=delete` einzufügen.

- Z.B. um die Erstellung von Objekten mit dem Standard-Konstruktor zu verhindern (wenn der Compiler ihn automatisch erstellen würde),

- oder das Kopieren mit dem Copy-Konstruktor zu verbieten.

```
class Datum {
 private:
  int jahr, monat, tag;

 public:
  // Default-Konstruktor deaktivieren
  Datum() = delete;

  // Kopieren verbieten
  // durch deaktivieren des
  // Copy-Konstruktors
  Datum(const Datum& rhs) = delete;

  Datum(int j, int m, int t)
      : jahr{j}, monat{m}, tag{t} {//....}
 //....
};

Datum tagX = Datum(2222, 1, 1);
// Fehler: call to deleted
// constructor of 'Datum'
```

4.5.6. Delegierende Konstruktoren

- Wenn verschiedene Konstruktoren einer Klasse dieselben Aufgaben erledigen müssen kann man entweder:
  - diese Aufgaben in eine Member-Funktion auslagern,
  - oder in einem Konstruktor einen anderen Konstruktor Aufrufen, man spricht dann von einem **delegierenden Konstruktor**.
- **Vorsicht:** Der Aufruf des anderen Konstruktors kann **nur in der Member-Initialisierungsliste erfolgen**:

```
class Datum {
 private:
  int jahr, monat, tag;
 public:
  // Default-Konstruktor als
  // delegierender Konstruktor
  Datum() : Datum(2000, 1, 1) {}
  Datum(int j, int m, int t)
      : jahr{j}, monat{m}, tag{t} {//...}
};
```

- Ab C++11 ist der Aufruf auch mit geschweiften Klammern möglich:
  `Datum() : Datum{2000, 1, 1} {}`
- Der folgende **fehlerhafte Versuch** den anderen Konstruktor im Konstruktor-Rumpf aufzurufen erzeugt in Wirklichkeit nur ein **temporäres Objekt**, ohne dabei das Objekt auf das `this` verweist zu initialisieren.

```
  Datum() { Datum(2222, 1, 1); }
```

- Bei der Delegation von Konstruktoren besteht die **Gefahr der unendlichen Rekursion**:

```
class MyClass {
  public:
     MyClass(char c) : MyClass((int)c) {}
     MyClass(int i) : MyClass((char)(i % 128)) {}
};
```

4.5.7. Konvertierende Konstruktoren

- Ein Konstruktor, der **mit einem einzelnen Parameter** aufgerufen werden kann, definiert auch eine Konvertierung des Parameter-Typs in den Klassen-Typ des Konstruktors.
- Ein derartiger Konstruktor kann somit auch zur Realisierung impliziter (durch den Compiler) bzw. expliziter (Cast-Operator, `static_cast`) Typkonvertierungen aufgerufen werden.
- Oft auch in Fällen, in denen die damit mögliche Typkonvertierung wenig sinnvoll ist oder gar vollkommen ungewollt ist.

```
// Bsp. Konvertierender Konstruktor
class Datum {
 private:
  int jahr, monat, tag;

 public:

  // Konvertierender Konstruktor
  Datum(int j)
   : jahr{j}, monat{1}, tag{1} {}

  // ...
};
```

<u>Aufruf:</u> (ohne Fehlermeldung durch Compiler!)

```
Datum d3 = 19.80; // Z.B. Tippfehler, zusaetzlicher Punkt
d3.printDatum();  // Ausgabe: 1.1.19
```

Nicht konvertierende Konstruktoren

- Der implizite Aufruf eines Konstruktors und damit seine versehentliche Verwendung für eine ggf. wenig sinnvolle Typkonvertierung kann verhindert werden, indem man den Konstruktor mit dem vorangestellten Funktions-Spezifizierer `explicit` versieht.
- Der Funktions-Spezifizierer `explicit` ist nur bei der Konstruktor-Deklaration innerhalb der Klassendefinition anzugeben.
- In einer eventuellen Definition außerhalb der Klassendefinition ist er nicht zu wiederholen.
- Ein nicht-konvertierender Konstruktor kann nur explizit verwendet werden, allerdings auch für explizite Typkonvertierungen.

```
// Nicht-Konvertierender
// Konstruktor
explicit Datum(int j)
  : jahr{j}, monat{1},
    tag{1} {}
```

Aufruf:

```
// Compiler-Fehler
Datum d3 = 19.80; // Tippfehler wird erkannt!
d3.printDatum();
// OK
Datum d4(19.80); // hier aber nicht
d4.printDatum();

// Compiler Fehler
Datum d5{19.80}; // Tippfehler wird erkannt!
d5.printDatum();
```

4.5.8. Konstruktoren mit Default-Parametern

- Konstruktoren können auch Default-Parameter besitzen.
- Hierdurch wird es möglich bei der Objekt-Erstellung eine unterschiedliche Anzahl von Parametern zu verwenden.
- Ein Konstruktor kann dadurch zu einem konvertierenden Konstruktor (einem Konstruktor mit nur einem Parameter) werden.

  Ggf. sollte er dann auch mit dem Funktions-Spezifizierer `explicit` versehen werden, um unbeabsichtigte Konvertierungen zu verhindern.
- Sind alle Parameter eines Konstruktors Default-Parameter, so ersetzt dieser Konstruktor auch einen Default-Konstruktor, dieser muss (darf) dann nicht zusätzlich definiert werden.

```
class Datum {
 private:
  int jahr, monat, tag;

 public:
  // Konstruktor mit Default-Parametern
  // es muss /darf zusaetzlich kein
  // Default-Konstruktor definiert werden
  explicit Datum(int j=2000, int m=1, int t=1)
      : jahr{j}, monat{m}, tag{t} { //.... }

  //....
};

Datum heute;   // Verwendung aller Default-Parameter
heute.printDatum();  // Ausgabe: 1.1.2000
```

4.5.9. Initialisierungslisten-Konstruktoren

- Soll ein Objekt mit einer unterschiedlichen Anzahl von Werten initialisiert werden (z.B. Felder, Listen mit unterschiedlicher Element-Anzahl), so bietet sich die Verwendung eines **Initialisierungslisten-Konstruktors, (Sequenzkonstruktors)** an.
- Alle Container der STL besitzen einen derartigen Konstruktor.
- In diesem Fall hat der Konstruktor einen einzigen Parameter vom Typ `std::initializer_list`.
- Die Erstellung eines Initialisierungslisten-Konstruktors erfordert die Include-Anweisung: `#include <initializer_list>`
- Ein Initialisierungslisten-Konstruktor erstellt Objekte mit der Hilfe einer `{}`-Liste als Initialisierungswert.

```
// Aufruf des Initialisierungs-Listen-Konstruktors
Datum d1{2020, 4, 28};
```

- Zur Auflösung von Mehrdeutigkeiten gelten die folgenden Regeln:
  - Wenn sich sowohl ein Standard-Konstruktor als auch ein Initialisierungslisten-Konstruktor aufrufen läßt, dann wird der Standard-Konstruktor verwendet.
  - Wenn sich sowohl ein Initialisierungslisten-Konstruktor als auch ein *normaler* Konstruktor mit Parametern aufrufen lässt, dann **wird immer der Initialisierungslisten-Konstruktor verwendet**.
- Um im letzten Fall den *normaler* Konstruktor mit Parametern aufzurufen, *kann in diesem einzigen Fall nicht die einheitliche Initialisierung mit den geschweiften Klammern verwendet werden*. Man muss nun diesen Konstruktor mit runden Klammern aufrufen: `Datum d2(2020, 4, 28);`

Initialisierungslisten-Konstruktoren 2

Bsp.: Eigentlich erfordert die Klasse Datum keinen Initialisierungslisten-Konstruktors, man kann aber mit ihr erklären wie die Mehrdeutigkeit bei der Objekt-Erstellung aufgelöst wird (s.o.).

```
#ifndef DATUM_H
#define DATUM_H
#include <initializer_list>
#include <iostream>
class Datum {
 private:
  int jahr, monat, tag;
 public:
  explicit Datum(int j = 2000, int m = 1, int t = 1)
      : jahr{j}, monat{m}, tag{t} {//....}
  Datum(std::initializer_list<int> list) {
    // Verarbeitung der Elemente mit Iterator it
    auto it = list.begin();
    if (it != list.end()) {
      jahr = *it; ++it;
    } else {
      jahr = 1900;
    }
    if (it != list.end()) {
      monat = *it; ++it;
    } else {
      monat = 12;
    }
    if (it != list.end()) {
      tag = *it; ++it;
    } else {
      tag = 31;
    }
    if (!checkDatum(jahr, monat, tag)) {
      setDefault();
    }
  }  // ....
};
```

4.6. Destruktoren

- Ein Destruktor ist das Gegenstück zum Konstruktor.
- Er ist eine weitere spezielle Member-Funktion einer Klasse.
- Er wird automatisch bei Beendigung der Lebensdauer eines Objekts der Klasse aufgerufen.
- Sein Zweck besteht im Rückgängigmachen von Initialisierungen, die im Konstruktor durchgeführt wurden.
- Insbesondere müssen im Destruktor Ressourcen (z.B. Speicher, Dateien) freigegeben werden.
- Stichwort: **Ressourcenbelegung ist Initialisierung (resource acquisition is initialization, RAII)** (s.o. und Kap. 5).
- Der Name eines Destruktors besteht aus dem Klassen-Namen mit mit vorangestelltem ~ *(er ist das Komplement des Konstruktors)*.
- Ein Destruktor hat keinen Rückgabetyp und keine Parameter.
- Ein Destruktor kann somit auch nicht überladen werden.
- Destruktoren dürfen nicht `static` sein.
- Im Destruktor dürfen andere Member-Funktionen aufgerufen werden, die z.B. Aufräum-Arbeiten übernehmen.
- Der Aufruf von Destruktoren erfolgt in umgekehrter Reihenfolge der zugehörigen Konstruktoraufrufe.

  Z.B. wird der Destruktor einer Klasse vor den eventuellen Destruktoren von Klassen-Komponenten aufgerufen.

```
~Datum() {
  std::cout << "Aufruf Destruktor: ";
  printDatum();
  std::cout << "stirbt" << std::endl;
}
```

Aufgabe Klasse Uhr

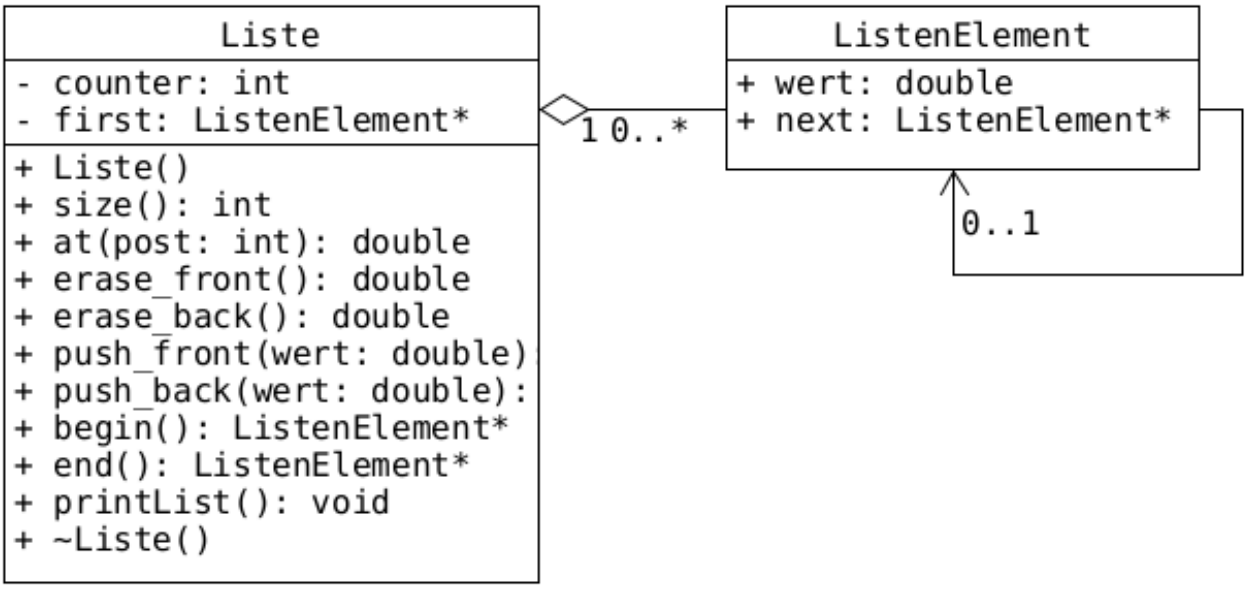
Man implementiere gemäß des folgenden UML-Klassendiagramms die Klasse `Uhr`. Da UML Programmiersprachen-unabhängig sein sollte, ist eine C++ typische `const` Deklaration nicht angegeben.

```
Uhr
---------------------------------------------
- SDEFAULT: int=42
- stunden: int
- minuten: int
- sekunden: int
---------------------------------------------
+ getSDefault(): int
- checkAttributes()
+ displayClock()
+ setTime(h: int, m: int=0, s: int=0)
+ tick()
+ Uhr()
+ Uhr(h: int, m: int=0, s: int=0)
+ Uhr(rhs: const Uhr&)
+ Uhr(liste: std::initializer_list<int>)
+ operator=(rhs: const Uhr&): Uhr&
+ ~Uhr()
```

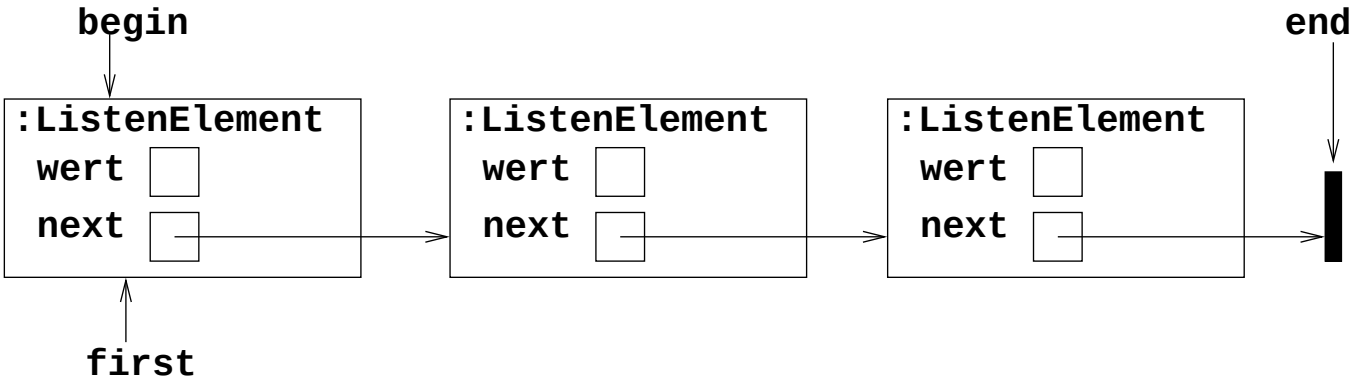
- Der Default-Konstruktor soll als delegierender Konstruktor implementiert werden und alle Attribute auf 0 setzen.
- Die private Hilfs-Methode `checkAttributes()` überprüft ob alle Attribute gültige Werte besitzen. Falls ein Attribut keinen gültigen Wert hat, ist es auf 0 zu setzen.
- Die Methode `tick()` erhöht die Sekunden um eine Sekunde.
- `displayClock()` druckt die Uhrzeit in *ansprechender Form* aus.
- Überlegen Sie ob etwas im Destruktor erledigt werden muß. Wenn Ihre Antwort Nein ist, drucken Sie einfach die Uhrzeit und den Text `"Objekt stirbt"` aus.
- Testen Sie Ihre Implementierung in einem Hauptprogramm.

Aufgabe Klasse Liste

Man implementiere gemäß des folgenden UML-Klassendiagramms die Klasse `Liste`.



Die Liste soll eine einfach verkettete Liste sein, in der `double` Werte gespeichert werden können.



- Welche Initialisierung ist im Konstruktor sinnvoll?
- `counter` zählt die Anzahl `double`-Werte, die in der Liste gespeichert sind.
- Die Methode `size()` gibt genau diese Anzahl zurück.

Aufgabe Klasse Liste 2

- `at()` liefert das Element zurück, das an der Position `pos` gespeichert ist. wie in der Informatik üblich hat das erste Element in der Liste die Position 0.

- `erase_fornt()` bzw. `erase_back()` löschen das Element am Anfang bzw. am Ende der Liste und geben es zurück.

- `push_front()` bzw. `push_back()` fügt `wert` am Anfang bzw. am Ende der Liste ein.

- `begin()` liefert einen Pointer auf das `ListenElement`-Objekt am Anfang der Liste zurück, wohingegen `end()` immer `nullptr` zurück liefert.

  `printList()` druckt alle in der Liste gespeicherten Elemente in *ansprechender Form* aus.

- Überlegen Sie was nach RAII alles im Destruktor erledigt werden muss.

5. Überladung von Operatoren

5.1. Grundlegende Konzepte

- Die folgenden Operatoren sind defaultmäßig auch für benutzerdefinierte Klassen definiert:
  - Zuweisungsoperator = (liefert flache (bitweise) Kopie)
  - Adressoperator `&`
  - Elementoperator `.`
  - Komponentenauswahloperator `.*` (für Pointer-Member)
  - Bereichsoperator (Scope Resolution Operator) `::`
  - Größenermittlungsoperator `sizeof`
  - Objekt-Erzeugungsoperator `new`
  - Objekt-Zerstörungsoperator `delete`
- Auch bei der Arbeit mit Objekten von benutzerdefinierten Klassen ist eine **prägnante Notation** sinnvoll und sehr bequem, z.B.:

```
Komplex z1{1, 3}, z2{-2, 1}, z3{4, -2}, z4;
z4 = -z1 * z2 + z3; // anstelle von
z4 = operator+(operator*(z1.operator-(),
                         z2),z3);
```

- Deshalb ermöglicht C++ das Überladen der folgenden Operatoren:

`->`	`[]`	`()`	`++`	`--`
`!`	`~`	`*` (unär)	`+` (unär)	`-` (unär)
`&` (unär)	`new`	`delete`	`->`	
`/`	`%`	`*` (binär)	`+` (binär)	`-` (binär)
`<<`	`>>`	`<`	`<=`	`>`
`>=`	`==`	`!=`	`&` (binär)	`^`
`|`	`&&`	`||`	`=`	`*=`
`/=`	`%=`	`+=`	`-=`	`<<=`
`>>=`	`&=`	`^=`	`|=`	`,`

Grundlegende Konzepte 2

- Man kann keine neuen zusätzlichen Operator-Symbole, die noch nicht in der obigen Tabelle aufgeführt sind, erstellen.
- Man kann weiterhin die folgenden Operatoren nicht überladen:
  - Scope-Operator `::`
  - Member-Zugriffs-Operator `.`
  - `sizeof` Operator
  - Bedingte Bewertung `? :`
  - Logische Operatoren `&&` und `||` sollten nicht überladen werden, da man sonst die *short circuit* Eigenschaft verliert.
- Durch das Überladen kann nicht die Standard-Priorität und die Standard-Assoziativität eines Operators geändert werden.
- Für interne Grunddatentypen dürfen die Operatoren nicht überladen werden.
- Um einen Operator zu Überladen, muss eine Operator-Funktion (s.u.) erstellt werden.
- Da der Rückgabe-Typ einer Funktion in C++ nicht Bestandteil der Signatur ist, kann eigentlich jeder beliebige Typ als Rückgabe-Typ einer Operator-Funktion gewählt werden.
- Es ist allerdings guter Stil, dass der Rückgabe-Typ ähnlich zum Rückgabe-Typ für die internen Grunddatentyp-Operatoren ist, z.B. sollten Vergleichsoperatoren ein Ergebnis vom Typ `bool` zurück liefern.
- Wenn möglich sollten Objekte als Referenzen zurückgegeben werden.
- Erfordert das Ergebnis eines Operators die Erstellung eines neuen Objektes, so kann dieses nur als Wert-Kopie zurück gegeben werden.

5.2. Operatorfunktionen

```
type operatorSymbol(Parameterliste)
```

Bsp.:

```
Komplex& operator+=(const Komplex& k);

Komplex operator+(const Komplex& a,
                  const Komplex& b);
```

- Zur Überladung eines Operators wird eine Operatorfunktion erstellt.
- Der Name einer Operatorfunktion setzt sich zusammen aus dem Schlüsselwort `operator` und dem Operator-Symbol.
- Ein überladener Operator sollte sich ähnlich zum Operator für einen Grunddatentyp verhalten.
- Eine Operatorfunktion kann entweder als **Member-Funktion** einer Klasse oder als **freie Funktion** definiert werden.
- **Member-Funktionen** sollten als `const` deklariert werden, wenn durch den Operator das eigene Objekt nicht verändert wird.
- Interne Datentypen werden als Wert übergeben.
- Objekte sollten dagegen immer als Referenz übergeben werden.
- Erfordert die Überladung des Operators keinen schreibenden Zugriff auf ein übergebenes Objekt, sollte das Objekt sogar als `const`-Referenz übergeben werden.

5.2.1. Freie Operatorfunktion

- Als **freie Funktionen** sollten grundsätzlich immer definiert werden:
  - Die Stream-Operatoren >> und <<.
  - (Kommutative) binäre Operatoren wie +, *, −, /
- Im Fall einer freien Funktion muss mindestens ein Parameter vom Typ einer benutzerdefinierten Klasse sein.

  Folgender Prototyp ist z.B. als freie Operatorfunktion ungültig:

  ```
  int operator+(int a, int b);
  ```

  Dies verhindert die Veränderung von Operatoren, die für interne Grunddatentypen definiert sind.
- Eine Operatorfunktion, die als freie Funktion definiert ist, hat **genau so viele Parameter, wie der Operator Operanden hat**:

  Ist z.B. @ ein unärer Präfix-Operator (Operator hat 1 Operanden), so ist

  ```
  @a
  ```

  äquivalent zum Aufruf von

  ```
  operator@(a).
  ```

  Ist z.B. @ ein binärer Operator (Operator hat 2 Operanden), so ist

  ```
  a @ b
  ```

  äquivalent zum Aufruf von

  ```
  operator@(a, b).
  ```

5.2.2. Freie Operatorfunktion

Als Bsp. wird in diesem Kapitel eine Klasse `Komplex` verwendet, die extra den Realteil und den Imaginärteil in einem dynamisch auf dem Heap gespeicherten Feld (Pointer `pk`) speichert, dies erfordert es dann auch die Big-Five (s.u.) selber zu erstellen.

```
class Komplex {
 private:
  double* pk;

 public:
  // unaeres Minus (Vorzeichen) als freie Funktion
  friend Komplex operator-(const Komplex& a) {
    Komplex erg{a};
    erg.pk[0] *= -1;  // Realteil
    erg.pk[1] *= -1;  // Imaginaerteil
    return erg;
  }
  // binaeres Minus als freie Funktion
  friend Komplex operator-(const Komplex& a,
                           const Komplex& b) {
    Komplex erg{a};
    erg.pk[0] -= b.pk[0];
    erg.pk[1] -= b.pk[1];
    return erg;
  }
};
```

<u>In main:</u>

```
Komplex z1{4, 5};
Komplex z2{1, 3};
Komplex z3, z4;
z3 = -z1;  // z3 = operator-(z1);
z4 = z1 - z2; // z4 = operator-(z1, z2);
cout << z3 << endl;
cout << z4 << endl;
```

5.2.2. Das friend Konzept in C++

```
friend Typ funktionsName(parameterliste);
```

```
friend class KlassenName;
```

```
class A {

  friend int freieFkt(A& a);

  friend void B::method1(const A& a);

  friend class C;
};
```

- Mit dem Schlüsselwort `friend` innerhalb der Klassen-Definition (im `private` oder `public` Bereich, kein Unterschied) kann eine Klasse
  - einer freien Funktionen oder
  - einer Methode einer anderen Klasse oder
  - einer anderen Klasse

  (sog. *Freunden*) den Zugriff auf alle privaten Datenelemente und Methoden erlauben.
- Nur die Klasse selbst kann bestimmen, wen sie als Freund (`friend`) haben möchte!
- Dadurch wird allerdings die Datenkapselung aufgeweicht (**schlechter Programmierstil!**)
- Deshalb ist dies ein sehr umstrittenes Konzept in C++.
- Die Verwendung ist nur akzeptabel, falls ansonsten die benötigten Datenzugriffe, z.B. über Setter/Getter, sehr aufwendig sind.

5.2.3. Operatorfunktionen als Member-Funktionen

- Eine Operatorfunktion, die als Member-Funktion definiert ist, hat immer **einen Parameter weniger als der Operator Operanden hat**.
- Denn in diesem Fall ist `this` der erste (linke) Operand.
- Ist z.B. `@` ein unärer Präfix-Operator, so ist `@a` äquivalent zum Aufruf von `a.operator@()`.
- Ist z.B. `@` ein binärer Operator, so ist `a @ b` äquivalent zum Aufruf von `a.operator@(b)`.

```
class Komplex {
 private:
  double* pk;
 public:
  // unaeres Minus als Member-Funktion
  Komplex operator-() const {
    Komplex erg{-1 * pk[0], -1 * pk[1]};
    return erg;
  }
  // binaeres Minus als Member-Funktion
  Komplex operator-(const Komplex& a) const {
    Komplex erg{*this};
    erg.pk[0] -= a.pk[0];
    erg.pk[1] -= a.pk[1];
    return erg;
  }
};
```

Aufruf in main:

```
Komplex z1{4, 5}, z2{1, 3}, z3, z4;
z3 = -z1; // z3 = z1.operator-();
z4 = z1 - z2; // z4 = z1.operator-(z2);
cout << z3 << endl;
cout << z4 << endl;
```

Operatorfunktionen als Member-Funktionen 2

- Als Member-Funktionen müssen immer die folgenden Operatorfunktionen definiert werden:
  - Copy/Move Zuweisungs-Operator =
  - Zugriffs-Operator `[]`
  - Aufruf-Operator `()` (Funktions-Objekte, Funktoren)

- Gewöhnlich definiert man auch immer die zusammengesetzten Operatoren z.B.: +=, −=, ⋆=, /=, als Member-Funktionen.

- Ebenfalls sollte man Operatoren, die den Zustand eines Objektes verändern immer als Member-Funktionen definieren, z.B. Inkrement-Operator ++, Dekrement-Operator −−, Dereferenzierungs-Operator ⋆.

- Kommutative Operatoren, die mit gemischten Typen verwendet werden können, definiert man als freie Funktionen, z.B. binäre Rechenoperationen.

  Indem man für ihre Implementierung die zusammengesetzten Operatoren verwendet, kann man vermeiden, sie als `friend` Funktionen zu deklarieren.

- Operanden dürfen nicht als Zeiger übergeben werden, denn dies würde z.B.: die Syntax `z3 = &z1 + &z2;` erfordern. Kleine Objekte (bis zu einer Größe von 4 Byte) übergibt man per Wert, größere Objekte per Referenz oder als `const` Referenz.

5.3. Ausgabe-Operator

```
class Komplex {
 private:
  double* pk;

 public:
  void output(std::ostream& os) const {
    os << "(" << pk[0] << ", " << pk[1] << ")";
  }
};

// Ueberladen des Ausgabe-Operators als
// freie Funktion
inline std::ostream& operator<<(std::ostream& os,
  const Komplex& z) {
  z.output(os);
  return os;
}
```

- Es sollte immer eine Referenz auf den Ausgabe-Strom zurückgegeben werden, so dass eine Verkettung möglich ist.

  ```
  Komplex z1{4, 5}, z2{1, 3};
  std::cout << z1 << z2 << std::endl;
  ```

- Als Parameter werden der Ausgabe-Strom als Referenz und das auszugebende Objekt als `const` Referenz übergeben.
- Durch die Verwendung einer Member-Funktion `output()` vermeidet man eine `friend` Deklaration.
- Alle Daten-Member des Objektes werden mit einer **minimalen Formatierung** ausgegeben.

  Insbesondere wird in der Regel die Ausgabe **nicht** mit einem Zeilenumbruch beendet.

5.4. Indizierungs-Operator

```
#include <stdexcept>

class Komplex {
 private:
  double* pk;

 public:
  double& operator[](int i) {
    if (0 <= i && i <= 1) {
      return pk[i];
    } else {
      throw std::out_of_range("ERROR: falscher Index");
    }
  }

  const double& operator[](int i) const {
    if (0 <= i && i <= 1) {
      return pk[i];
    } else {
      throw std::out_of_range("ERROR: falscher Index");
    }
  }
};
```

- Da der Indizierungs-Operator `[]` sowohl als L-Wert als auch als R-Wert auftreten kann, sollte er immer eine Referenz zurück geben.
- Weiterhin sollte er immer in 2 Versionen vorliegen.

  Eine Version sollte immer eine `const`-Member-Funktion sein, die eine `const` Referenz zurück liefert. Diese Version kann dann auch für `const` Objekte verwendet werden.
- Im obigen Beispiel kann der Indizierungs-Operator als Getter/Setter für den Realteil und den Imaginärteil verwendet werden.

5.5. Arithmetik mit gemischten Typen

```
class Komplex {
 private:
  double* pk;
 public:
  Komplex() : pk{new double[2]{0, 0}}  {}

  explicit Komplex(double re, double im)
    : pk{new double[2]{re, im}} {}

  Komplex& operator+=(const Komplex& a) {
    pk[0] += a.pk[0];
    pk[1] += a.pk[1];
    return *this;
  }

  Komplex& operator+=(const double a) {
    pk[0] += a;
    return *this;
  }
};

Komplex operator+(const Komplex& a, const Komplex& b);

Komplex operator+(const Komplex& a, double b);

Komplex operator+(double a, const Komplex& b);
```

Arithmetik mit gemischten Typen 2

Implementierung der freien Operator-Funktionen in einer cpp-Datei:

```
Komplex operator+(const Komplex& a,
                  const Komplex& b) {
  Komplex sum = a;
  sum += b;
  return sum;
}

Komplex operator+(const Komplex& a, double b) {
  Komplex sum{a[0] + b, a[1]};
  return sum;
}

Komplex operator+(double a, const Komplex& b) {
  Komplex sum{a + b[0], b[1]};
  return sum;
}
```

Arithmetik mit gemischten Typen 3

Aufruf:

```
Komplex z1{4, 5};
Komplex z2{1, 3};
Komplex z3, z4, z5;
z3 = z1 + z2;
z4 = 2.0 + z1;
z5 = z1 + 3.0;
cout << z3 << z4 << z5 << endl;
```

Durch die Verwendung eines **konvertierenden Konstruktors** (indem man `explicit` weg lässt) ist nur noch eine Version für += und + notwendig:

```
class Komplex {
 private:
  double* pk;

 public:
 Komplex(double re=0, double im=0)
    : pk{new double[2]{re, im}} {}

  Komplex& operator+=(const Komplex& a) {
    pk[0] += a.pk[0];
    pk[1] += a.pk[1];
    return *this;
  }
};

inline Komplex operator+(const Komplex& a,
                         const Komplex& b) {
  Komplex sum = a; sum += b;
  return sum;
}
```

5.6. Inkrement-Operator

- Der Inkrement-Operator ++ (und entsprechend der Dekrement-Operator −−) existiert in der:
  - **Präfix-Form**: `++obj`,
    Rückgabe des veränderten Operanden
    ist ein L-Wert
  - **Postfix-Form**: `obj++`
    Rückgabe des alten (unveränderten) Operanden
    ist ein R-Wert
- Zur Unterscheidung erhält die Postfix-Form einen ganzzahligen Dummy Parameter.

  Beim Aufruf der Operatorfunktion wird für ihn i. A. der Wert 0 übergeben.
- Rückgabetyp:
  - Präfix-Form: Referenz
    `Typ& operator++();`
  - Postfix-Form: Werte-Kopie
    `Typ operator++(int dummy);`

```
++obj; // entspricht obj.operator++();

obj++; // entspricht obj.operator++(0);
```

Bsp. Inkrement-Operator

```
class Komplex {
 private:
  double* pk;

 public:
  // ............

  // Praefix-Form
  Komplex& operator++() {
    ++pk[0];
    ++pk[1];
    return *this;
  }

  // Postfix-Form
  Komplex operator++(int dummy) {
    Komplex tmp{*this};
    ++*this;
    return tmp;
  }
};
```

Aufruf in main:

```
Komplex z5{1, 2};
cout << ++z5 << endl;
// aequivalent zu
cout << z5.operator++() << endl;

cout << z5++ << endl;
// aequivalent zu
cout << z5.operator++(0) << endl;

cout << z5 << endl;
```

5.7. Kopieren und Zuweisen

- Objekte können standardmäßig kopiert werden, man spricht in diesem Zusammenhang auch von **Copy-Semantik** oder **Wertsemantik**.
- Dies wird ermöglicht durch die folgenden sogenannten speziellen Member-Funktionen, die in C++ **Big-Three** genannt werden:
  - **Copy-Konstruktor**: `X(const X& rhs);`
  - **Copy-Zuweisungs-Operator**: `X& operator=(const X& rhs);`
  - **Destruktor**: `~X();`
- Bei Bedarf werden sie vom Compiler automatisch erstellt.
- Verwaltet ein Objekt Ressourcen (z.B. Heap-Speicher, Dateien, Netzwerkverbindungen), so werden diese einfach nur **flach kopiert** (d.h. Kopie der Adressen), wenn der vom Compiler erstellte Copy-Konstruktor oder Copy-Zuweisungs-Operator verwendet wird.
- Dies führt dann zu Problemen, da dann mehrere Objekte gleichzeitig für die selbe Ressource verantwortlich sind!
- In C++ gilt deshalb die **Dreier-Regel (Rule of Three):**

  Wird einer der Big-Three selber implementiert, sollten auch immer die beiden anderen selber implementiert werden.
- Z. B.: Implementiert man selber einen Destruktor, in dem Heap-Speicher freigegeben wird, so sollte man immer auch den Copy-Konstruktor und den Copy-Zuweisungs-Operator derart selber implementieren, dass in ihnen eine **tiefe Kopie** erzeugt wird.

5.7.1. Bsp. Verletzung der Dreier-Regel

```
class Komplex {
 private:
  double* pk;

 public:
  Komplex(double re=0, double im=0)
    : pk{new double[2]{re, im}} {}

  ~Komplex() {
    delete[] pk;
    pk = nullptr;
  }
  // ...
};
```

```
int main() {
  int i;
  Komplex z1{1, 2}, z2;
  Komplex z3 = z1;  // Copy-Konstruktor
  z2 = ++z1;        // Copy-Zuweisung
  cout<<"z1: "<<z1<<endl; cout<<"z2: "<<z2<<endl;
  cout << "z3: " << z3 << endl;
  z2[1] = -42;
  cout << "z1: "<<z1<<endl; cout<<"z2: "<<z2<<endl;
  cout << "z3: " << z3 << endl;
  cin >> i;
}
```

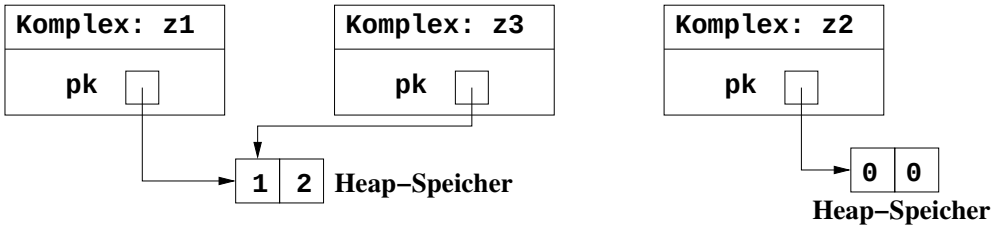
Ausgabe: Nach Eingabe einer ganzen Zahl Absturz des Programms!!

```
z1: (2, 3)
z2: (2, 3)
z3: (2, 3)        // sollte eigentlich (1, 2) sein!
z1: (2, -42)      // sollte eigentlich (2, 3) sein!
z2: (2, -42)
z3: (2, -42)      // sollte eigentlich (1, 2) sein!
```

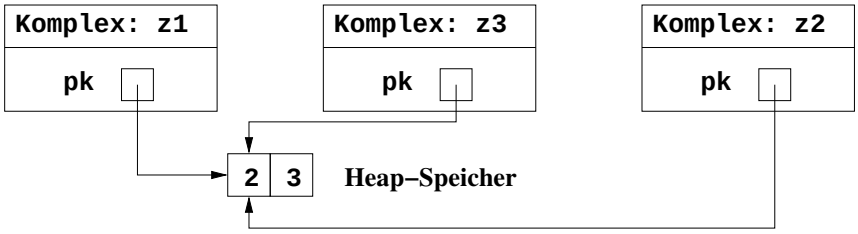
Denn im Destruktor wird der gleiche Heap-Speicher 3 mal gelöscht!

Bsp. Verletzung der Dreier-Regel 2

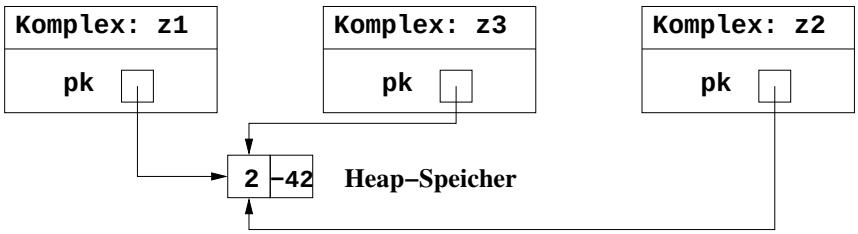
Situation im Speicher nach der Ausführung der Konstruktoren



Situation im Speicher nach der Ausführung von z2 = ++z1;



Situation im Speicher nach der Ausführung von z2[1] = -42;



- Alle Objekte nutzen aufgrund der flachen Kopie die gleiche Ressource (das 2-dim. Feld auf dem Heap)
- Durch den 3-maligen Aufruf des Destruktors (jeweils für die 3 Objekte) wird versucht, diesen Heap-Speicher 3 mal zu löschen!
- **Ausweg:** Umsetzung der Dreier-Regel (Rule of Three).

5.7.2. Bsp. Erfüllung der Dreier-Regel

```
class Komplex {
 private:
  double* pk;

 public:
  // ...

  // Copy Konstruktor, erstellt tiefe Kopie
  Komplex(const Komplex& rhs) {
    pk = new double[2];
    pk[0] = rhs.pk[0];
    pk[1] = rhs.pk[1];
  }

  // Copy Zuweisungs-Operator, erstellt tiefe Kopie
  Komplex& operator=(const Komplex& rhs) {
    if (this != &rhs) {
      delete[] pk;
      pk = new double[2];
      pk[0] = rhs.pk[0];
      pk[1] = rhs.pk[1];
    }
    return *this;
  }
};
```

- Copy-Konstruktor und Copy-Zuweisungs-Operator erstellen eine **tiefe Kopie**.
- Im Copy-Zuweisungs-Operator sollte auch immer überprüft werden ob **ein Objekt sich selbst zugewiesen wird!**
- Bleibt diese Prüfung aus, löscht man in diesem Fall genau den Speicher, dessen Inhalte man kopieren muss!

Bsp. Erfüllung der Dreier-Regel 2

```
int main() {
  int i;
  Komplex z1{1, 2}, z2;
  Komplex z3 = z1;  // Copy-Konstruktor
  z2 = ++z1;        // Copy-Zuweisung
  cout<<"z1: "<<z1<<endl; cout<<"z2: "<<z2<<endl;
  cout << "z3: " << z3 << endl;
  z2[1] = -42;
  cout << "z1: "<<z1<<endl; cout<<"z2: "<<z2<<endl;
  cout << "z3: " << z3 << endl;
  cin >> i;
}
```

Ausgabe:

Mit dem selber implementierten Copy-Konstruktor und Copy-Zuweisungs-Operator, die eine **tiefe Kopie** erstellen, erfolgt kein Programmabsturz!

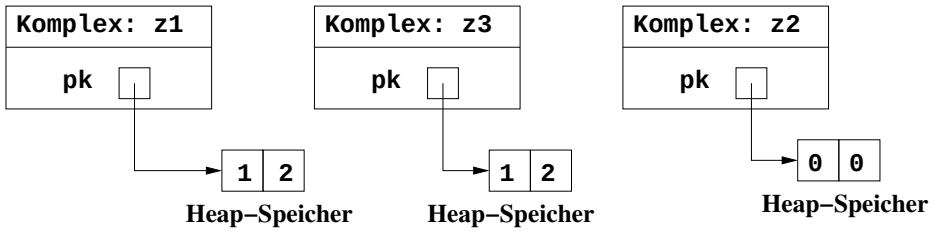
```
z1: (2, 3)
z2: (2, 3)
z3: (1, 2)
z1: (2, 3)
z2: (2, -42)
z3: (1, 2)
```

Es sind nun die beiden Kriterien, die für eine **wertorientierte Programmierung** notwendig sind, erfüllt:

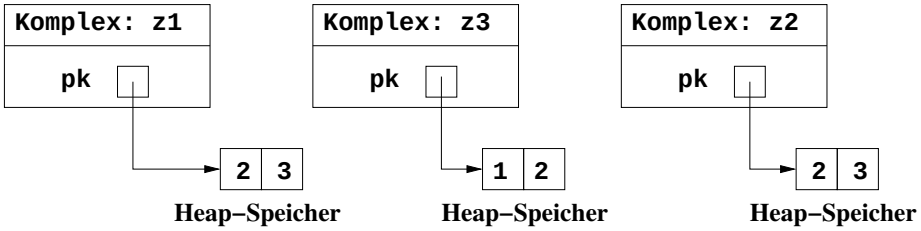
- **Äquivalenz:** Nach `x = y` sollten Operationen auf `x` und `y` das gleiche Ergebnis liefern, insbesondere sollte dann gelten `x == y` und `f(x) == f(y)` für jede Funktion `f()`.
- **Unabhängigkeit:** Nach `x = y` sollten Operationen auf `x` den Zustand von `y` nicht verändern. Insbesondere ändert `f(x)` den Wert von `y` nicht, wenn `f()` eine bel. Funktion ist.

Bsp. Erfüllung der Dreier-Regel 3

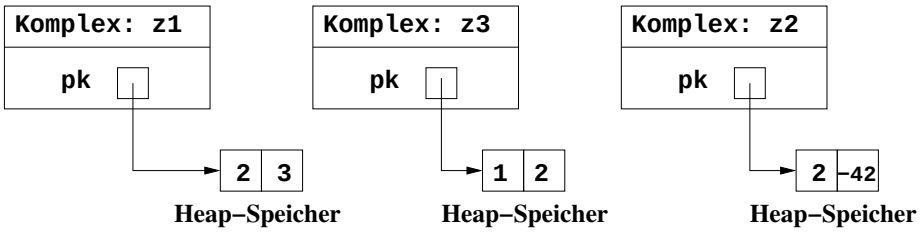
Situation im Speicher nach der Ausführung der Konstruktoren



Situation im Speicher nach der Ausführung von z2 = ++z1;



Situation im Speicher nach der Ausführung von z2[1] = -42;



- Alle Objekte nutzen ihre eigene Ressource (ein 2-dim. Feld auf dem Heap)
- Es sind nun die beiden Kriterien, die für eine wertorientierte Programmierung notwendig sind erfüllt:

5.8. Verschieben anstelle von Kopieren

- Ziel: Aufwand für wiederholtes `new` und `delete` für temporäre Objekte (R-Werte) vermeiden.

```
Komplex z1{1,2}, z2{-1,1}, z3{42,43}, z4;
z4 = z1 + z2 + z3;
```

- Indem man bereits erstellten temporären Objekten ihren allozierten Speicher stiehlt.
- Seit C++11 gibt es die Möglichkeit dieser **Verschiebesemantik (Move-Semantik)**.
- Bildlich gesprochen: *Anstelle eines Kopierens und Einfügens wendet man ein Ausschneiden und Einfügen an.*
- Die **Big-Three** werden ergänzt um
  - **Verschiebe-Konstruktor (Move-Konstruktor)**
    `X(X&& rhs);`
  - **Verschiebe-Zuweisungs-Operator (Move-Zuweisung)**
    `X& operator=(X&& rhs);`

  zu den **Big-Five**.

  `const` entfällt nun beim Übergabe-Parameter, warum?
- Der neue Typ `X&&` ist eine **Referenz auf einen R-Wert**.

  Normale Referenzen können nur an L-Werte gebunden werden.

  R-Wert Referenzen können dagegen auch an R-Werte gebunden werden.
- Die Dreier-Regel (Rule of Three) wird ergänzt zur **<u>Rule of Five:</u>**

  Wenn die Big-Three selber implementiert werden, soll man sie immer zu den Big-Five ergänzen.

5.8.1. Bsp. R-Werte-Referenz

```
#include <iostream>
#include <string>

using std::cout;
using std::endl;
void printNachricht(std::string& nach) {
  cout << nach << endl;
}
void printNachricht(std::string&& nach) {
  cout << "R-Value-Reference: "
      << nach << endl;
}
int main() {
  std::string s1 = "Hello";
  std::string s2 = "World";
  printNachricht(s1);
  printNachricht(s1 + s2);  // R-Value
  printNachricht("Hello");  // R-Value
  return 0;
}
```

Ausgabe:

```
Hello
R-Value-Reference: HelloWorld
R-Value-Reference: Hello
```

- Beim 2-ten und 3-ten Aufruf der überladenen Funktion `printNachricht()` ist der Aktualparameter `s1 + s2` bzw. `"Hello"` ein temporäres Objekt, also ein R-Wert.
- Deshalb wird in diesem Fall die Funktion

  `void printNachricht(std::string&& nach)`

  ausgeführt, also genau die Version der überladenen Funktion verwendet, deren Parameter eine R-Wert-Referenz ist.

Bsp. Verschieben

```
class Komplex;
inline std::ostream& operator<<(std::ostream& os,
  const Komplex& z);

class Komplex {
  // ....

 public:
  // Move-Konstruktor
  Komplex(Komplex&& rhs) {
    std::cout << "Move-Konstruktor: " << rhs
              << std::endl;
    pk = rhs.pk;
    rhs.pk = nullptr;
  }

  // Move-Zuweisungs-Operator
  Komplex& operator=(Komplex&& rhs) {
    std::cout << "Move-Zuweisungs-Operator: " << rhs
              << std::endl;
    if (this != &rhs) {
      delete[] pk;
      pk = rhs.pk;
      rhs.pk = nullptr;
    }
    return *this;
  }

  Komplex& operator+=(const Komplex& a) {
    pk[0] += a.pk[0];
    pk[1] += a.pk[1];
    return *this;
  }

  // ...
};
```

Bsp. Verschieben 2

```
Komplex operator+(const Komplex& a,
                  const Komplex& b) {
  Komplex sum = a;
  sum += b;
  return sum;
}
```

```
int main() {
  Komplex z1{1, 2}, z2{3, 4};
  // std::move() erzeugt R-Wert Referenz
  Komplex z3 = std::move(z1 + z2);
  Komplex z4;
  z4 = z1 + z2;
  cout << "z3: " << z3 << endl;
  cout << "z4: " << z4 << endl;
}
```

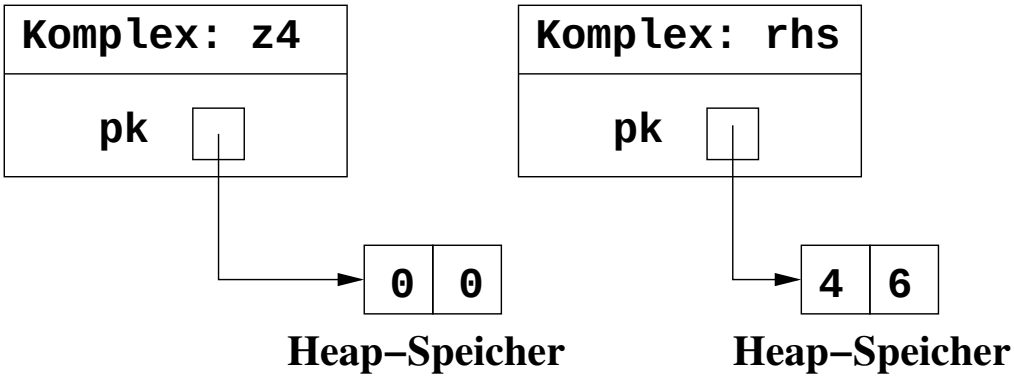
Ausgabe:

```
Copy-Konstruktor: (1, 2)
Move-Konstruktor: (4, 6)
Move-Konstruktor: (4, 6)
Copy-Konstruktor: (1, 2)
Move-Konstruktor: (4, 6)
Move-Zuweisungs-Operator: (4, 6)
z3: (4, 6)
z4: (4, 6)
```

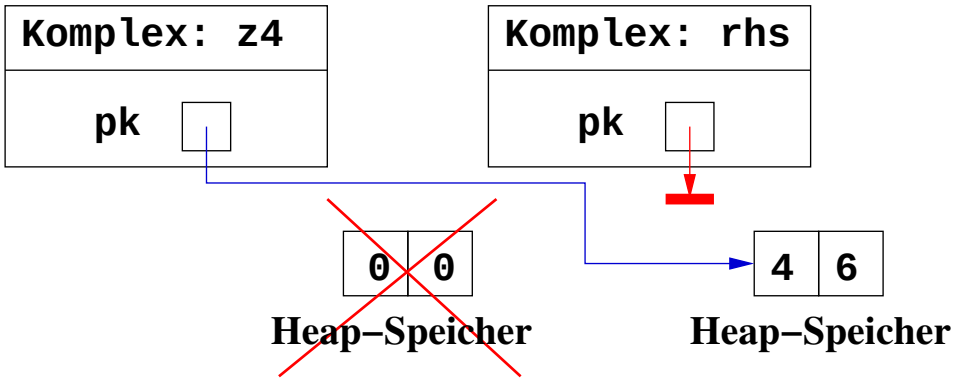
- Der binäre + Operator benötigt jeweils am Anfang den Aufruf eines Copy-Kunstruktors und bei der Rückgabe den Move-Konstruktor.
- Dann erfolgt durch Verwendeung des Move-Konstruktors die Initialisierung von `z3`.
- Die Zuweisung an `z4` erfolgt wiederum nach dem Ausführen des binären + Operators durch den Move-Zuweisungs-Operator.

Bsp. Verschieben 3

Situation im Speicher vor der Move-Zuweisung z4 = z1 + z2;



Situation im Speicher nach der Move-Zuweisung z4 = z1 + z2;



- Dem temporären Objekt `z1 + z2`, auf das über die R-Wert-Referenz `rhs` im Move-Zuweisungs-Operator zugegriffen wird, wird die Ressource Heap-Speicher genommen. Sie wird vom temporären Objekt zum Objekt `z4` verschoben (move).
- Man vermeidet dadurch das Anlegen von neuem Heap-Speicher und das Kopieren der einzelnen Elemente.

Aufgabe Klasse Vec

Man erstelle eine Klasse `Vec` mit folgenden Membern:

a) Attribute `int n` (Elementanzahl) und
   `double* elem` (Zeiger auf dynamisch angelegtes Feld mit `n` Elementen.

b) Standard-Konstruktor:
   `n=5`, alle Elemente werden auf `0.0` gesetzt.

c) **nicht-konvertierender** Konstruktor `Vec(int size)`:
   `n = size`, alle Elemente werden auf `0.0` gesetzt.

d) **konvertierender** Konstruktor `Vec(double wert)`:
   `n=5`, alle Elemente werden auf `wert` gesetzt.

e) **delegierender Konstruktor**
   `Vec(int size, double wert)`, der an den Konstruktor in Teil c) delegiert.

f) Einen **Sequenzkonstruktor**.

g) Überladung des Ausgabe-Operators << als freie Funktion.

h) Überladung des Indizierungs-Operators `[]`.

i) Überladung des += Operators.

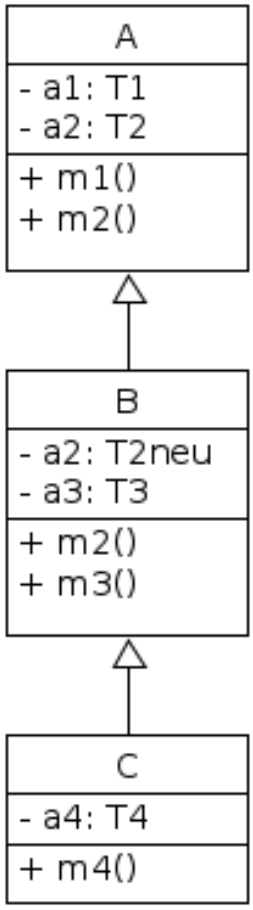
j) Überladung des + Operators.

k) Implementierung der **Big-Five**.

l) Man teste alle implementierten Methoden in einem Hauptprogramm.

6. Vererbung

6.1. Grundlegende Konzepte der Vererbung



- Wenn eine Klasse die Member (Attribute/Methoden) einer anderen Klasse übernimmt, spricht man von **Vererbung (engl. inheritance)**.
- Die Klasse die erbt nennt man **abgeleitete Klasse (Unterklasse, Subklasse, Kindklasse)**.
- Die Klasse von der geerbt wird nennt man **Basisklasse (Oberklasse, Superklasse, Elternklasse)**.
- Gemeinsame Eigenschaften und Verhaltensweisen *faktorisiert man aus* in eine Basisklasse, man spricht dann auch von **Generalisierung**.
- Eine **Spezialisierung** erfolgt durch die Erzeugung einer neuen abgeleiteten Klasse, die die Eigenschaften und Verhaltensweisen der Basisklasse erbt.
- Kennzeichen der Vererbung ist die **is-a Beziehung**. Ein Objekt der abgeleiteten Klasse ist ein Objekt der Basisklasse mit zusätzlichen Eigenschaften und Verhaltensweisen.
- In der abgeleiteten Klasse können geerbte Member angepasst werden oder neue Member hinzugefügt werden.
- Die Abgeleitete Klasse kann wiederum Basisklasse einer anderen abgeleiteten Klasse sein. Dadurch können tiefe **Klassenhierarchien** entstehen.

6.1.1. Vorteile der Vererbung

- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen.
- Durch Vererbung werden Code-Wiederholungen vermieden.
- In den neue Klassen müssen nur noch Erweiterungen und Anpassungen vorgenommen werden.
- Durch Vererbung erhält man einen sehr wichtigen Konsistenzvorteil, da gemeinsame Eigenschaften und Verhaltensweisen nicht über mehrere Stellen im Code verstreut sind, sondern auf eine einzige Stelle konzentriert sind. Die Konsistenz muss deshalb nur an einer Stelle geprüft werden.
- Genauso müssen Änderungen im Code nur an einer einzigen Stelle vorgenommen werden und nicht wiederholt an vielen Stellen.
- Die Vererbung ist ein Grundkonzept der objektorientierten Programmierung.
- Die Vererbung ist die Voraussetzung für die **Polymorphie**:

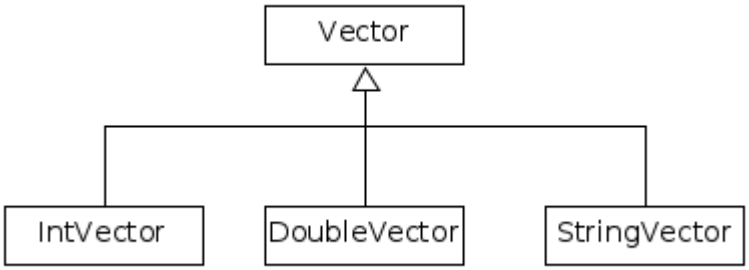
  Verschiedenartige Objekte können zeitweise unter einem gemeinsamen Oberbegriff gleichartig behandelt werden, ohne dass sie dabei ihre unterschiedlichen Eigenschaften verlieren.

6.1.2. Vererbung oder Templates

- Um Code-Wiederholungen zu vermeiden gibt es in C++ neben der **Vererbung** auch noch die Möglichkeit **Templates** einzusetzen.
- **Templates** sollten immer dann verwendet werden, wenn sich der Code der unterschiedlichen Klassen nur hinsichtlich des Types von einigen Attributen unterscheidet, z.B.:

  `vector<int>`, `vector<double>`, `vector<string>`

- Der Einsatz von Templates ist dann sowohl bezüglich Laufzeit-Performance als auch bezüglich Codier-Aufwand sehr effizient.
- Es ist dann viel aufwendiger und ineffizienter Vererbung zu verwenden in der Art:

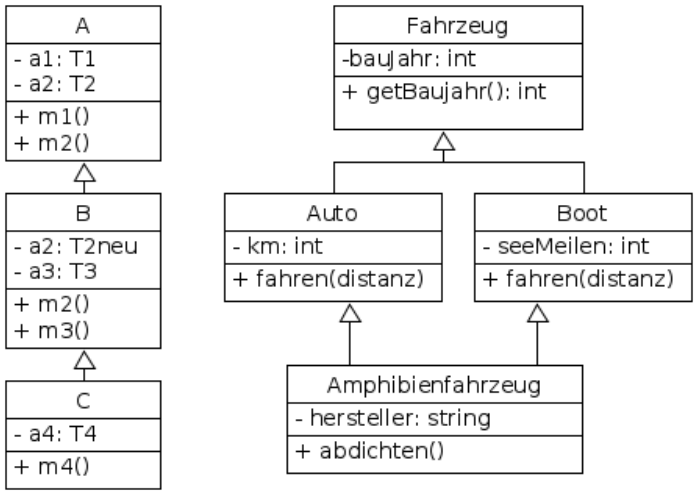


- In diesem Fall sind Templates die viel bessere Lösung.
- Vererbung wird man in C++ immer dann verwenden, wenn die Verwendung von Templates nicht oder nur sehr schwer möglich ist, also wenn die verschiedenen Klassen sich in einigen grundlegenden Eigenschaften/Verhaltensweisen gleichen aber unter den unterschiedlichen Klassen auch ganz verschiedene Attribute bzw. Methoden gibt.

6.1.3. Kardinalität der Vererbung

1) Einfach-Vererbung

- Eine abgeleitete Klasse erbt genau von einer Oberklasse.
- Das UML-Diagramm ist ein Baum (jeder Sohn hat genau einen Vater).



2) Mehrfach-Vererbung

- Eine abgeleitete Klasse erbt von mehreren Oberklassen.
- Das UML-Diagramm ist ein gerichteter Graph.
- Die Mehrfach-Vererbung kann zu einigen Problemen führen (Namenskonflikte, mehrfache Speicherung von geerbten Attributen)
- In vielen Programmiersprachen (z.B. Java) ist deshalb die Mehrfach-Vererbung verboten.
- C++ unterstützt die Mehrfach-Vererbung.
- Namenskonflikte lassen sich in manchen Fällen in C++ mit der Hilfe des Scope-Operators `::` beheben.
- Die mehrfache Speicherung von Attributen kann in C++ durch virtuelle Basisklassen vermieden werden.
- In dieser Vorlesung wird keine Mehrfach-Vererbung verwendet.

6.2. Definition abgeleiteter Klassen in C++

```
class C : public B1, public B2, ..., public Bn {
    // .....
};
```

- Die Klassendefinition wird durch eine **Abstammungsliste (base-list)** ergänzt, in der die Basisklassen, von denen geerbt werden soll, aufgelistet sind.
- Die Klasse `C` wird von den Basisklassen `B1`, `B2`, ..., `Bn` abgeleitet.
- Durch das Zugriffsattribut `public` wird der Zugriff auf die Member der Basisklasse geregelt.
- Grundsätzlich gilt, dass die abgeleitete Klasse keinen direkten Zugriff auf die `privaten` Member der Basisklasse hat.
- Standard ist das Zugriffsattribut `public`. Durch diese sogenannte **public-Vererbung** werden alle öffentlichen Member aus der Basis-Klasse auch in der abgeleiteten Klasse öffentlich gemacht.
- Dadurch wird die öffentliche Schnittstelle der Basis-Klasse von der abgeleiteten Klasse voll übernommen.
- In diesem Fall gilt das **Liskovsche Substitutionsprinzip** (s.u):

  Es kann immer ein Objekt der abgeleiteten Klasse angegeben werden
  (z.B. bei einer Zuweisung oder als aktueller Parameter),
  wenn ein Basisklassen-Objekt erwartet wird.
- Das Objekt der abgeleiteten Klasse wird dann implizit in ein Basisklassen-Objekt konvertiert.
- Die is-a Beziehung ist erfüllt: ein Objekt der abgeleiteten Klasse is-a Objekt der Basisklasse.

6.2.1. Zugriff auf Basisklassen-Member

Der Zugriff auf die nicht privaten Member der Basisklassen wird durch Zugriffsattribute in der Abstammungsliste festgelegt.
Betrachtet sei die Situation dass:

- Klasse `D` öffentlich von Klasse `C` erbt
- und Klasse `C` von Klasse `B` erbt.

Zugriffsattribut in Abstammungsliste von `C` bezüglich `B`			
	`public`	`protected`	`private`
In Methode oder in Friend von `C`	Zugriff auf `public` und `protected` Member von `B`	Zugriff auf `public` und `protected` Member von `B`	Zugriff auf `public` und `protected` Member von `B`
In Methode oder in Friend von `D`	Zugriff auf `public` und `protected` Member von `B`	Zugriff auf `public` und `protected` Member von `B`	**kein** Zugriff auf `public` und `protected` Member von `B`
über Objekt vom Typ `C`	Zugriff nur auf `public` Member von `B`	**kein** Zugriff auf `public` Member von `B`	**kein** Zugriff auf `public` Member von `B`

- Bei der `protected` Vererbung werden die `public` und `protected` Member der Basisklasse zu `protected` Membern der abgeleiteten Klasse.
- Bei der `private` Vererbung werden die `public` und `protected` Member der Basisklasse zu `private` Membern der abgeleiteten Klasse.
- Nur in ganz seltenen Fällen ist die Verwendung von `protected` oder `private` anstelle von `public` als Zugriffsattribut in der Abstammungsliste sinnvoll (z.B. Einschränkung der öffentlichen Schnittstelle der Basisklasse).

6.2.2. Ableitung unterbinden mit final

```
class C final : public B1, public B2, ..., public Bn {
    // .....
};
```

- Manchmal ist es sinnvoll keine weitere Spezialisierung einer Klasse zuzulassen.
- In C++ kann man dann eine solche Klasse mit dem Schlüsselwort `final` markieren.

```
class Direktor final : public Manager {

  // ...............

};
```

- Von der Klasse `Direktor` kann dann nicht mehr abgeleitet werden.

6.3. Member der abgeleiteten Klasse

Namenssuche des Compilers

- Der Compiler sucht den Namen eines Elementes (Attribut/Methode) zunächst in der abgeleiteten Klasse.
- Findet er ihn dort nicht geht er in der Klassenhierarchie Stufe um Stufe höher und sucht dort nach dem passenden Element.

In der abgeleiteten Klasse können Attribute:

- Von der Basisklasse geerbt werden.
- Neu hinzugefügt werden.
- Durch die Verwendung des gleichen Namens in der abgeleiteten Klasse überschrieben werden, man spricht dann auch von **Redefinition**.

  Durch die Namenssuche des Compilers führt dies allerdings dazu, dass dadurch das Element in der Basisklasse **verdeckt wird**.

  Dies ist weniger sinnvoll und oft ein unbeabsichtigter Fehler.

In der abgeleiteten Klasse können Methoden:

- Von der Basisklasse geerbt werden
- Neu hinzugefügt werden
- Durch Verwendung der gleichen Signatur in der abgeleiteten Klasse **überschrieben (redefiniert)** werden.

  Das Überschreiben von Methoden ist oft sinnvoll, um in der abgeleiteten Klasse eine Methode an neue Funktionalitäten anzupassen.

  Auch in diesem Fall wird die Methode aus der Basisklasse verdeckt.

  Der Zugriff auf die Basisklassen-Methode muss dann mit dem Scope-Operator erfolgen: `Basisklasse::methode();`

Bsp. Überschreiben von Methoden

```
#ifndef FAHRZEUG_H
#define FAHRZEUG_H
#include <iostream>
class Fahrzeug {
 private:
  int herstellungsJahr;

 public:
  Fahrzeug(int jahr) : herstellungsJahr{jahr} {}
  void output(std::ostream& os) const {
    os << "Fahrzeug, hergestellt im Jahr: "
       << herstellungsJahr << std::endl;
  }
  ~Fahrzeug();
};

#endif  // FAHRZEUG_H
```

```
#ifndef KFZ_H
#define KFZ_H
#include "fahrzeug.h"

class KFZ : public Fahrzeug {
 private:
  double ps;

 public:
  KFZ(int jahr, double p) : Fahrzeug{jahr}, ps{p} {}

  void output(std::ostream& os) const {
    Fahrzeug::output(os);//Aufruf Basisklassen-Meth.
    os << "Spezieller Typ KFZ mit: " << ps
       << " PS" << std::endl;
  }
  ~KFZ();
};

#endif  // KFZ_H
```

6.4. Konstruktoren und Vererbung

- Konstruktoren werden nicht vererbt.
- Für das Anlegen eines Objektes vom Typ der abgeleiteten Klasse ist ein Konstruktor der abgeleiteten Klasse zuständig.
- Da die abgeleitete Klasse alle Attribute der Basisklasse enthält, müssen auch diese angelegt und initialisiert werden.
- Hierzu ruft der Konstruktor der abgeleiteten Klasse automatisch den Default-Konstruktor der Basisklasse auf, bevor der Code des Konstruktors selbst ausgeführt wird.
- Besitzt die Basisklasse einen Konstruktor mit Parametern, so ist es sinnvoll, diesen explizit aufzurufen (**Basisinitialisierer**).
- Der explizite Aufruf des Basisinitialisieres kann aber nur in der Member-Initialisierungsliste eines Konstruktors der abgeleiteten Klasse erfolgen.

```
class Fahrzeug {
 private:
  int herstellungsJahr;

 public:
  Fahrzeug(int jahr) : herstellungsJahr{jahr} {}
  // ....
};
```

```
class KFZ : public Fahrzeug {
 private:
  double ps;

 public:
  KFZ(int jahr, double p) : Fahrzeug{jahr}, ps{p} {}
  // ......
};
```

6.5. Aufbau und Abbau von Objekten

Aufbau von Objekten:

- Auch in einer Vererbungshierarchie wird in einem Konstruktor einer Unterklasse explizit immer nur der Konstruktor der direkten Basisklasse in der Memberinitialisierungs-Liste aufgerufen.
- Dieser ruft dann ggf. wiederum explizit den Konstruktor oder den Default-Konstruktor der nächsten direkten Basisklasse auf.
- Zu aller erst wird dann tatsächlich das Teilobjekt angelegt, das den Typ der obersten Basisklasse hat.
- Es folgt dann das Anlegen der (Teil)-Objekte der nachfolgenden abgeleiteten Klassen.
- Erst zum Schluss wird dann der Konstruktor ausgeführt, der zum anlegen eines Objektes vom Typ einer abgeleiteten Klasse aufgerufen wurde.

Abbau von Objekten

- Zum Ende der Lebensdauer eines Objekts wird der klasseneigene Destruktor automatisch aufgerufen.
- Danach werden die Destruktoren der Basisklassen automatisch aufgerufen.
- Die Reihenfolge der Destruktoren ergibt sich ausgehend von der abgeleiteten Klasse in Richtung der Basisklasse und deren Basisklassen, usw. bis zur obersten Basisklasse.

Bsp. Aufbau und Abbau von Objekten

```
class A {
 private:
  int a;

 public:
  A(int aa) : a{aa} { cout << "Konstruktor A" << endl; }
  ~A() { cout << "Destruktor A" << endl; }
};

class B : public A {
 private:
  int b;

 public:
  B(int aa, int bb) : A{aa}, b{bb} {
    cout << "Konstruktor B" << endl;
  }
  ~B() { cout << "Destruktor B" << endl; }
};

class C : public B {
 private:
  int c;

 public:
  C(int aa, int bb, int cc) : B{aa, bb}, c{cc} {
    cout << "Konstruktor C" << endl;
  }
  ~C() { cout << "Destruktor C" << endl; }
};

int main() {
  C* pc = new C{1, 42, 4711};
  cout << "Objekt vom Typ C erstellt" << endl;
  delete pc;
  return 0;
}
```

Bsp. Aufbau und Abbau von Objekten2

Ausgabe:

```
Konstruktor A
Konstruktor B
Konstruktor C
Objekt vom Typ C erstellt
Destruktor C
Destruktor B
Destruktor A
```

7. Polymorphie

Polymorphie (Vielgestaltigkeit)
beschreibt in der Objektorientierung die Fähigkeit, dass die gleiche Methode für verschiedenartige Objekte aufgerufen werden kann und dies zu unterschiedlichen Auswirkungen (die ganau zum Objekt-Typ passen) führt.

- Verschiedenartige Objekte aus inhomogenen Mengen können dadurch zeitweise unter einem gemeinsamen Oberbegriff gleichartig behandelt wreden.
- Man vermeidet dadurch hässliche Fallunterscheidungen mit Typ-Prüfungen: *z.B. (Pseudo-Code):*

```
double flaeche;
if (x.type() == "Kreis") {
  flaeche = M_PI * x.getR()*x.getR();
}
if (x.type() == "Rechteck") {
  flaeche = x.getA() * x.getB();
}

 cout << "Flaeche = " << flaeche << endl;
```

- In C++ gibt es die Möglichkeit Polymorphie zu implementieren durch:

  a) **Templates** (später), man spricht dann von **statischer, ungebundener Polymorphie**.

  b) **Vererbung** und **virtuellen Methoden**, man spricht dann von **dynamischer, gebundener Polymorphie**. Die Typen sind in diesem Fall an einen gemeinsamen Basistyp gebunden.

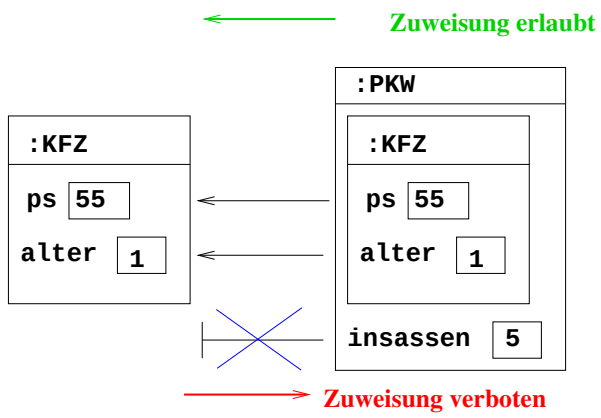
7.1. Liskovsches Substitutionsprinzip

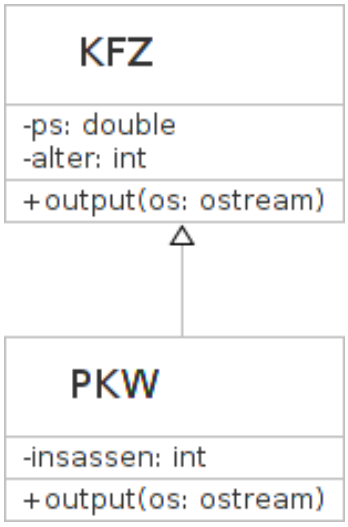
Sei Klasse `B` von Klasse `A` abgeleitet, also `B` erbt von `A`.

- Es kann immer ein `B`-Objekt angegeben werden (z.B. bei einer Zuweisung oder als Aktual-Parameter), wenn ein `A`-Objekt erwartet wird.
- Das `B`-Objekt wird dann implizit in ein `A`-Objekt konvertiert.
- Die Umkehrung gilt nicht!

  Jeder PKW ist ein Fahrzeug, aber nicht jedes Fahrzeug ist ein PKW, da es z.B. ein Fahrrad oder ein LKW sein kann.
- Bei der Zuweisung erfolgt das sog. **Slicing**:
  - Ein Objekt einer abgeleiteten Klasse kann an ein Objekt vom Typ einer direkten oder indirekten Basisklasse zugewiesen werden.
  - Bei der Zuweisung werden nur die Member-Elemente der Basisklasse komponentenweise kopiert.
  - Analoges gilt für die Parameterübergabe oder Konvertierungen.
- Ein Pointer / eine Referenz vom Typ der Oberklasse kann auf Objekte vom Typ einer Unterklasse verweisen.

```
KFZ kfz;
PKW pkw{55, 1, 5};
kfz = pkw;   // OK
pkw = kfz;   // Compiler Fehler
```





7.2. Cast-Operatoren

7.2.1. C-Stil Cast-Operatoren

```
double preis = 12.34;
int euro;


euro = (int)preis;

// oder in der Konstruktor-Schreibweise
euro = int(preis);
```

- Sind gefährliche Operationen,
- da sie die Typ-Kontrolle des Compilers unterlaufen.
- In der Regel geht Information verloren.

Ratschlag:

- Scheinbar notwendige Typ-Umwandlungen sollte man immer hinterfragen.
- Oft sind sie nur aufgrund eines Fehlers im Design erforderlich.
- Sollten sie trotzdem notwendig sein, ist es ratsam immer die neuen Typkonvertierungs-Operatoren zu verwenden.
- Diese haben eine einheitliche Form und sind durch ihre Namen leicht optisch und syntaktisch im Code zu erkennen.
- Ihr Name beschreibt ihren spezialisierten Einsatz-Zweck.

7.2.2. Static Cast

```
T var = static_cast<T>(ausdruck);
```

- Wird verwendet zum Konvertieren der Standard-Typen
- oder zum Konvertieren in Klassentypen die einparametrige Konstruktoren besitzen.
- Für einen **Down-Cast** kann der `static_cast` Operator nur verwendet werden, wenn zur **Compile-Zeit feststeht**, dass der Basisklassen-Zeiger auf ein Objekt einer abgeleiteten Klasse verweist.

```
class A {   // Klasse mit einparametrigem
 private:   // Konvertierungs-Konstruktor
  int a;
 public:
  A(int i): a{i} {}
};
void funcA(const A& obj) { cout << "Aufruf funcA" << endl;}
class Basis {
 public:
  virtual void output() const {cout << "Basis" << endl;}
};
class Kind : public Basis {
 public:
  virtual void output() const {cout << "Kind" << endl;}
  void kindMethode() const {cout << "Kind-Methode" << endl;}
};
int main() {
  double preis = 12.34; int euro;
  euro = static_cast<int>(preis);
  funcA(static_cast<A>(euro));
  Kind kind;
  Basis* basisPtr = &kind;
  Kind* kindPtr = static_cast<Kind*>(basisPtr);
  kindPtr->kindMethode();
}
```

7.2.3. Dynamic Cast

```
T var = dynamic_cast<T>(ausdruck);
```

- Typ-Umwandlungen, die in einer Klassenhierarchie *nach oben* gehen werden als **Up-Cast** bezeichnet.

  Sie sind immer möglich und sicher und werden deshalb implizit vorgenommen.

- Typ-Umwandlungen, die in einer Klassenhierarchie *nach unten* gehen werden als **Down-Cast** bezeichnet.

  Ein Down-Cast kann in C++ nur **explizit** durch eine Cast-Anweisung erfolgen.

  Sie ist nur sicher, falls das durch den Basisklassen-Zeiger adressierte Objekt tatsächlich den Typ der abgeleiteten Klasse hat.

- Um einen Down-Cast stets sicher durchzuführen, wird der `dynamic_cast`-Operator verwendet.

- Die Typprüfung kann auch zur Laufzeit erfolgen, falls sie zur Compile-Zeit nicht möglich ist.

- Der Typ `T` muss ein Zeiger oder eine Referenz auf eine Klasse sein.

- Falls `ausdruck` einen Zeiger liefert, der nicht auf ein Objekt vom Typ `T` oder einen von `T` abgeleiteten Typ zeigt, ist das Ergebnis ein Null-Zeiger vom Typ `(T*)nullptr`.

  Dies kann für eine **Typprüfung** verwendet werden (s. u.).

- Falls `ausdruck` eine Referenz ist, die nicht auf ein Objekt vom Typ `T` oder abgeleitet von `T` verweist, wird eine Ausnahme vom Typ `bad_cast` ausgeworfen.

Bsp. dynamic Cast

```
// Vererbungshierarchie mit Basis Klasse und 2 Kind-Klasse
class Basis {
 public:
  virtual void output() const {cout << "Basis" << endl;}
};
class Kind : public Basis {
 public:
  virtual void output() const {cout << "Kind" << endl;}
  void kindMethode() const {cout << "Kind-Methode" << endl;}
};

class Nesthaekchen : public Basis {
 public:
  virtual void output() const {
    cout << "Class Nesthaekchen" << endl; }
};
int main() {
  Kind k;
  Nesthaekchen n;
  Basis* pb = &k;
  pb->output();
  // Fehler ein Downcast ist notwendig
  // pb->kindMethode();
  dynamic_cast<Kind*>(pb)->kindMethode();

  // falls cast nich Moeglich erhaelt man nullptr zurueck
  if (dynamic_cast<Nesthaekchen*>(pb) == nullptr) {
    cout << "downcast zu Nesthaekchen nicht moeglich"
         << endl;
  }
  // Falls bei Referenzen cast nicht moeglich
  // wird eine bad_cast Exception ausgeworfen
  Basis& rb = n;
  Kind& rk = dynamic_cast<Kind&>(rb);
  rk.kindMethode();
  return 0;
}
```

7.2.4. Const Cast

```
T var = const_cast<T>(ausdruck);
```

- Dieser Operator ist der einzige Operator, mit dem die `const`-Eigenschaft eines Objektes entfernt werden kann.
- **Er sollte eigentlich nicht eingesetzt werden, da eine `const`-Deklaration immer einen Sinn hat.**
- Er wird benötigt, wenn für eine Methode, die das eigene Objekt nicht verändert, die `const` Markierung vergessen wurde.

```
class SillyVec {
 private:
  int size = 10;
 public:
  int getSize() { return size; } // const vergessen!!
};
void funcVec(const SillyVec* pcv) {
  int s;
  // Fehler, mit const Pointer darf Methode, die
  // keine const Markierung hat, NICHT aufgerufen werden
  // s = pcv->getSize();
  // deshalb muss const-Markierung von pcv entfernt werden
  SillyVec* pv = const_cast<SillyVec*>(pcv);
  s = pv->getSize();
  cout << s << endl;
}
int main() {
  const int i = 100;
  const int* icp = &i;
  int* ip = const_cast<int*>(icp);
  *ip = 42;
  SillyVec sv;
  funcVec(&sv);
}
```

7.2.5. Reinterpret Cast

```
T var = reinterpret_cast<T>(ausdruck);
```

- Dieser Cast-Operator kann die `const`-Eigenschaft eines Objektes nicht ändern, aber ansonsten ist jede Typumwandlung möglich.
- Die Typumwandlung findet zur Compile-Zeit statt.
- Der Bit-Inhalt wird im Sinne des angegebenen Typs `T` re-interpretiert.
- Das Ergebnis ist Hardware-abhängig.
- **Die Verwendung dieses Cast-Operators ist ein sehr fragwürdiger Hack!!**
- Er wird oft verwendet für Strukturen, die nicht miteinander in einer Vererbungshierarchie stehen.
- Er kann sinnvoll sein bei der Konvertierung von Zeigern von und nach `void*`.

```
#include <cstdlib>
int main() {
  void* p = std::malloc(10 * sizeof(int));
  int* ip = reinterpret_cast<int*>(p);
  *ip = 1;
  return 0;
}
```

7.3. Laufzeit-Typinformationen

Die Verwendung von Typinformationen zur Laufzeit bezeichnet man konventionell als **Laufzeittypinformationen**,
oft abgekürzt als **RTTI (Run-Time Type Information)**.

7.3.1. Typinformationen über Dynamic Cast

- Der oben beschriebene `dynamic_cast`-Operator führt vor der Umwandlung eine Typüberprüfung zur Laufzeit durch.
- Ist die Umwandlung nicht möglich, ist das Ergebnis der Umwandlung der `nullptr`.
- Mit seiner Hilfe kann also festgestellt werden, ob ein unbestimmtes Objekt in einen bestimmten Typ `T` umgewandelt werden kann.
- Hieraus folgt, dass das unbestimmte Objekt vom Typ `T` oder vom Typ einer von `T` abgeleiteten Klasse ist.
- Diese einfache Laufzeit-Typinformation ist in vielen Fällen ausreichend, wie z.B. im folgenden *extra total hässlich codierten* Bsp..

  In einer nachfolgenden Aufgabe werden wir mittels Polymorphie diesen total hässlichen Code eliminieren!

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>

using std::cout;
using std::endl;

class X {
  virtual void output() const {cout << "X" << endl; }
};
```

Bsp. Typinformation mit Dynamic Cast

```
class Kreis : public X {
 private:
  double r;

 public:
  Kreis(double aR) : r{aR} {}
  double getR() const { return r; }
};

class Rechteck : public X {
 private:
  double a, b;

 public:
  Rechteck(double aA, double aB) : a{aA}, b{aB} {}
  double getA() const { return a; }
  double getB() const { return b; }
};
```

Bsp. Typinformation mit Dynamic Cast 2

```
int main() {
  Kreis k{2};
  Kreis* kp;
  Rechteck r{2, 3};
  Rechteck* rp;
  X* xp;
  std::srand(static_cast<unsigned int>(time(nullptr)));
  double zufall = static_cast<double>(rand()) / RAND_MAX;
  if (zufall >= 0.5) {
    xp = &k;
  } else {
    xp = &r;
  }

  double flaeche;
  kp = dynamic_cast<Kreis*>(xp);
  rp = dynamic_cast<Rechteck*>(xp);
  if (kp != nullptr) {
    // unbekanntes Objekt hat den Typ Kreis
    flaeche = M_PI * kp->getR()*kp->getR();
  }
  if (rp != nullptr) {
    // unbekanntes Objekt hat den Typ Rechteck
    flaeche = rp->getA() * rp->getB();
  }

  cout << "Flaeche = " << flaeche << endl;
  return 0;
}
```

7.3.2. Typinformationen über typeid-Operator

```
const type_info& typeid(ausdruck)
```

- Gelegentlich kann es erforderlich sein, den genauen Typ eines Objektes zu kennen, z.B. den genauen Klassennamen oder den Aufbau der Klasse.
- Hierfür ist in C++ der `typeid()` Operator vorgesehen.
- Er liefert als Ergebnis eine Referenz auf ein Objekt vom Typ `type_info` (definiert in `<typeinfo>`), das den Typ des Übergabeparameters darstellt.
- Der Parameter `ausdruck` kann entweder der Name eines vollständig definierten Typs sein oder ein Ausdruck.
- Im Fall das `ausdruck` ein Ausdruck ist:
  - wird eine Ausnahme vom Typ `std::bad_typeid` ausgelöst, wenn sein Wert ein `nullptr` ist.
  - wird der Typ zur Compile-Zeit ohne Auswertung des Ausdruckes ermittelt, wenn `ausdruck` einen nicht polymorphen Typ hat oder kein L-Wert ist.

    Ein Typ (Klasse) ist genau dann **polymorph**, wenn es mindestens eine virtuelle Methode (s.u.) in der Definition des Types existiert.
  - wird der Typ zur Laufzeit durch die Auswertung von `ausdruck` ermittelt, wenn `ausdruck` einen polymorphen Typ hat.

Typinformation mit typeid-Operator 2

Die Definition von `type_info` sieht im Wesentlichen wie folgt aus:

```
class type_info {
  // daten

 public:
   virtual ~type_info(); // virtueller Destruktor
                         // damit ist type_info polymorph

   // Vergleiche sind moeglich
   bool operator==(const type_info&) const noexcept;
   bool operator!=(const type_info&) const noexcept;

   // Sortierung fuer assoziative Container (map)
   // ohne Zusammenhang zur Vererbungshierarchie
   bool before(const type_info&) const noexcept;
   size_t hash_code() const noexcept;

   // Name des Typs
   const char* name() const noexcept;

   // Kopieren der Objekte ist verboten
   type_info(const type_info&) = delete;
   type_info& operator=(const type_info&) = delete;
};
```

- Es wird nicht garantiert, dass es nur ein `type_info`-Objekt für jeden Typ im Programm gibt.
- Deshalb sollte man nie Pointer auf `type_info`-Objekte auf Gleichheit (Adressen-Gleichheit) testen, sondern immer die `type_info`-Objekte selber auf (Inhalts)-Gleichheit mit den bereitgestellten Operatoren überprüfen.

Bsp. Typinformation mit typeid-Operator

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;
class W {
  // nicht polymorphe Klasse
};
class X : public W {
  virtual void output() const { cout << "X" << endl; }
};
class Kreis : public X {
 private:
  std::string typ = "Kreis";
  double r;

 public:
  Kreis(double aR) : r{aR} {}
  std::string type() const { return typ; }
  double getR() const { return r; }
};

class Rechteck : public X {
 private:
  std::string typ = "Rechteck";
  double a, b;

 public:
  Rechteck(double aA, double aB) : a{aA}, b{aB} {}
  std::string type() const { return typ; }
  double getA() const { return a; }
  double getB() const { return b; }
};
```

Bsp. Typinformation mit typeid-Operator 2

```
int main() {
  Kreis k{2};
  Kreis* kp;
  Rechteck r{2, 3};
  Rechteck* rp;
  X* xp;
  std::srand(static_cast<unsigned int>(time(nullptr)));
  double zufall = static_cast<double>(rand()) / RAND_MAX;
  if (zufall >= 0.5) {
    xp = &k;
  } else {
    xp = &r;
  }

  double flaeche;
  const std::type_info& ti = typeid(*xp);
  cout << "Objekt hat den Typ: " << ti.name()
       << ", Hash-Code: " << ti.hash_code() << endl;
  if (ti == typeid(Kreis)) {
    kp = dynamic_cast<Kreis*>(xp);
    flaeche = M_PI * kp->getR()*kp->getR();
  }
  if (ti == typeid(Rechteck)) {
    rp = dynamic_cast<Rechteck*>(xp);
    flaeche = rp->getA() * rp->getB();
  }
  cout << "Flaeche = " << flaeche << endl;

  // Nicht polymorpher Typ
  W* wp = xp;
  cout << "Ergebnis mit nicht polymorphen Pointer Typ: "
       << typeid(*wp).name() << endl;
  return 0;
}
```

7.4. Virtuelle Methoden

Nicht virtuelle Methoden:

- Standard-Verhalten für Methoden in C++.
- Der Typ des Pointers / der Referenz bestimmt welche Methode ausgeführt wird.
- Das Binden erfolgt schon zur Compile-Zeit, man spricht von **früher Bindung** oder **statischer Bindung**.

Virtuelle Methoden:

```
virtual T methode(parameterListe);
```

- Die Methode muss bei ihrer Deklaration mit dem Spezifizierer `virtual` versehen sein.
- Der Typ des Objektes, auf das ein Zeiger/Referenz verweist, bestimmt welche Methode ausgeführt wird.
- Die Bindung erfolgt erst zur Laufzeit, man spricht von **dynamischem Binden** oder **spätem Binden**.
- Nur nicht statische Member-Funktionen können virtuell sein.
- Konstruktoren können nicht virtuell sein.
- Destruktoren können virtuell sein (s.u.).
- Besitzt eine Klasse mindestens eine virtuelle Methode, so wird sie **polymorphe Klasse** oder **polymorpher Typ** genannt.
- Bei der Redefinition in einer abgeleiteten Klasse muss `virtual` nicht mehr angegeben werden *(virtuell bleibt virtuell)*.
- Eine Methode der Basisklasse wird nicht nachträglich virtuell, wenn sie in der abgeleiteten Klasse als virtuell deklariert wird.

Virtuelle Methoden 2

- Die Redefinition einer virtuellen Methode in einer Unterklasse **muss mit genau der gleichen Signatur und dem gleichen Rückgabetyp wie in der Basisklasse** erfolgen, ansonsten findet überladen und nicht überschreiben statt!

  Ab C++11 hilft die Kennzeichnung mit dem Schlüsselwort `override` am Ende der Methoden-Deklaration in der Unterklasse.

- Überschriebene Methoden sollten die gleichen Default-Argumente wie ihre Basisklassen-Version besitzen.

  Denn Default-Werte werden der statischen Bindung entnommen, eine Redefinition der Default-Werte wird also unwirksam!

- Die Deklaration einer Methode als `virtual` bewirkt, dass Objekte dieser Klasse indirekt die Information über den Objekt-Typ als Pointer auf ein Objekt vom Typ `type_info` enthalten.

- Weiterhin enthalten Objekte eines polymorphen Typs einen versteckten Zeiger `vptr` auf die **virtuelle Methodentabelle (virtual method table, VMT)**.

  Die VMT enthält zu jeder virtuellen Methode die Anfangsadresse des Maschinencodes für die Methodenimplementierung.

  Bei einem Aufruf einer virtuellen Methode wird dann in der VMT des Objekts nachgeschaut, wo der Code liegt, der ausgeführt werden soll.

  Deshalb erfordert der Aufruf virtueller Methoden etwas mehr Laufzeit.

- Besitzt ein Objekt einen polymorphen Typ, ist es bedingt durch die VMT auch geringfügig größer.

Aufgabe Verwendung von Polymorphie

- Man eliminiere die hässlichen Typ-Fallunterscheidungen im obigen Bsp. Typinformation mit Dynamic Cast durch die Verwendung von Polymorphie.
- Für eine gemeinsame Basisklasse wähle man einen besseren Namen als `X`.
- Auf welche virtuelle Methode wird man zurückgreifen?
- Welche Probleme hat man bei der Implementierung dieser Methode in der Basis-Klasse.
- Tipp: Man kopiere den Beispiel-Code aus der PDF-Datei.
- Einfachheitshalber verzichten wir auf die Auslagerung aller Klassen in eigene Header-Dateien und `cpp`-Dateien.

7.5. Abstrakte Klassen

- Die Basisklasse einer Vererbungshierarchie sollte möglichst sehr allgemein gehalten werden, so dass ohne Änderungen viele verschiedenartige Klassen von ihr abgeleitet werden können.
- Dies erleichtert die einfache Erweiterbarkeit des Programm-Codes.
- Eine sinnvolle Implementierung ihrer Methoden ist aber deshalb oft nicht möglich, wie die obige Aufgabe am Beispiel der Klasse `Figur` und der Implementierung der Methode `flaeche()` zeigt.
- C++ ermöglicht es in diesem Fall die Methode als **rein virtuelle Methode (pure virtual method)** zu deklarieren:

```
virtual T methode(parameterListe) = 0;
```

- Die Methode muss dann in dieser Klasse nicht mehr implementiert (also definiert) werden.
- Dies dient als Festlegung einer gemeinsamen Schnittstelle.
- Besitz eine Klasse mindestens eine rein virtuelle Methode, so können von dieser Klasse keine Objekte erstellt werden.
- Man bezeichnet sie deshalb als **abstrakte Klasse (Abstract Base Class, ABC)** im Gegensatz zu **konkreten Klassen**, von denen Objekte erzeugt werden können.

Abstrakte Klassen 2

Abstrakte Klasse	Konkrete Klasse
enthält mindestens eine rein virtuelle Methode.	alle Methoden sind implementiert jede geerbte rein virtuelle Methode muß implementiert sein
Es können keine Objekte der Klasse erstellt werden, allerdings können Pointer und Referenzen vom Typ der Klasse deklariert werden.	Das Instanzieren der Klasse ist möglich.
Zweck: Festlegung einer gemeinsamen Schnittstelle	Zweck: Speicherung eines Objektes

Aufgabe: Man ergänze die obige Aufgabe Polymorphismus:

- Man wandele die Methode `flaeche` der Klasse `Figur` in eine rein virtuelle Methode um.
- Jede Figur soll weiterhin die Koordinaten (`x, y`) eines Referenz-Punktes speichern.
- Der Referenz-Punkt eines Kreises soll der Kreis-Mittelpunkt sein.
- Der Referenzpunkt eines Rechteckes soll die obere linke Ecke sein.
- Wo wird man diese Koordinaten am günstigsten speichern?
- Man nehme entsprechende Änderungen vor.
- Weiterhin soll es möglich sein alle Figur-Objekte mittels Überladen des Operators << zu drucken.

7.6. Schnittstellen-Programmierung

Schnittstelle (interface)
Eine abstrakte Klasse ohne Attribute deren Methoden alle rein virtuell sind außer einem virtuellem Destruktor bezeichnet man in der Objektorientierung als **Schnittstelle** oder manchmal als **Signatur-Klasse**.

- Schnittstellen sind ein sehr wichtiges Mittel, um in großen Programmen Module und Programm-Teile zu entkoppeln.
- Da eine Klasse, die von einer Schnittstelle abgeleitete ist, keine Implementierung von Methoden erbt, spricht man auch von **Schnittstellenvererbung** im Gegensatz zur **Implementierungsvererbung**.
- Der Zweck einer Schnittstelle ist die Definition eines Vertrages **(design by contract)** zwischen Objekten, die bestimmte Dienstleistungen erbringen und Objekten, die diese Dienstleistungen in Anspruch nehmen.
- Ist eine Schnittstelle einmal festgelegt können die Klassen für verschiedene Objekt-Arten unabhängig voneinander entwickelt werden.
- Insbesondere formuliert man sehr oft Algorithmen auf der Basis von Schnittstellen. Sie sind dann anwendbar auf alle Objekt-Typen, die die von dieser Schnittstelle abgeleitet sind **(Schnittstellen-Programmierung)**.
- In C++ ist es üblich, dass der Name von Schnittstellen mit einem großem `I` (für Interface) beginnt.

Aufgabe Schnittstellen-Programmierung

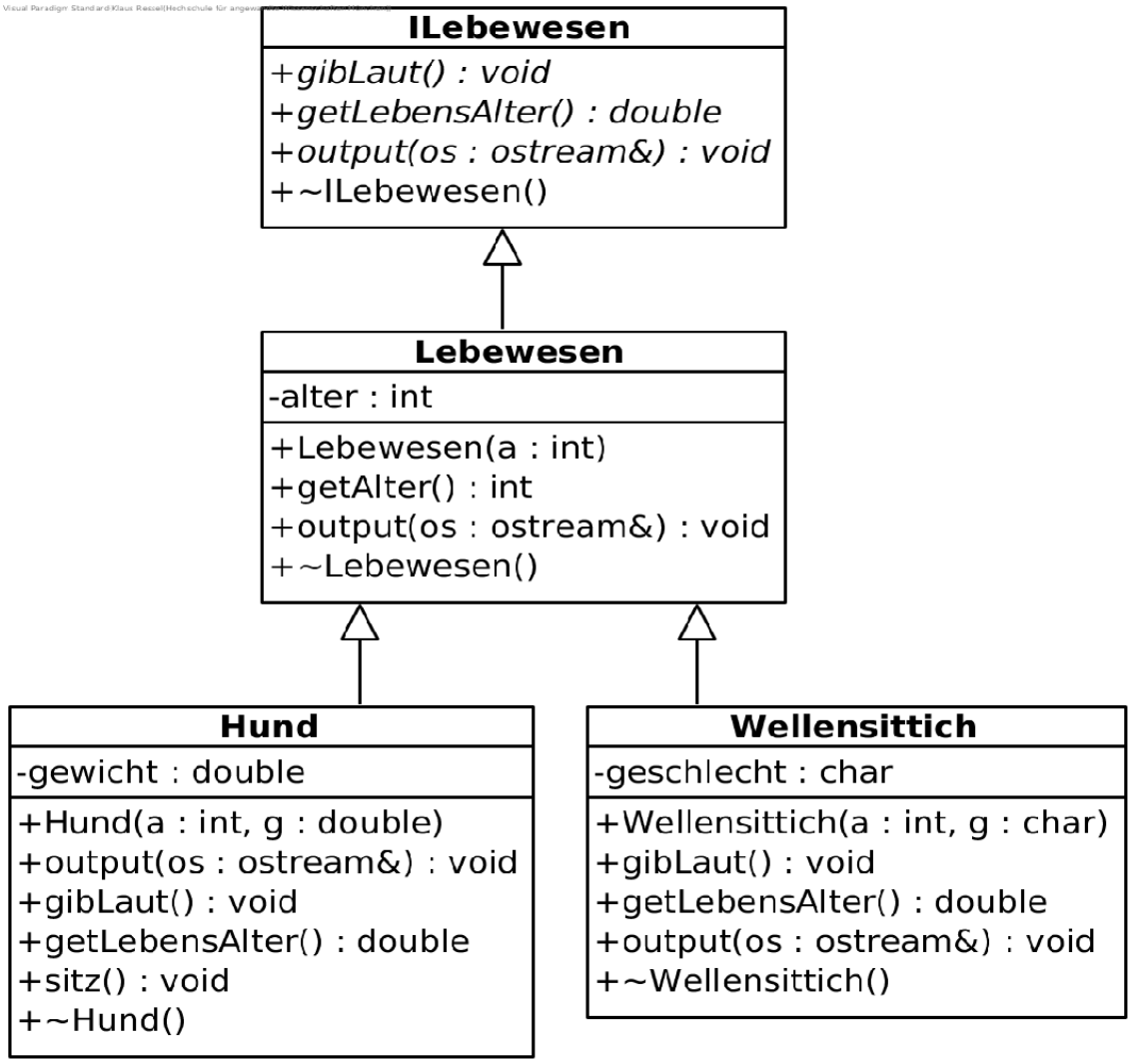
1. Man ergänze die Klassen in der obigen Aufgabe Polymorphie durch die beiden Schnittstellen
   a) `IFlaechenberechenbar`
   b) `IDruckbar`.
2. Weiterhin formuliere man unter Verwendung dieser Schnittstellen die Algorithmen:
   a) `IFlaechenberechenbar* maxFlaeche(`
      `std::vector<IFlaechenberechenbar*>)`
   b) `void druckeAlleFiguren(std::vector<IDruckbar*>)`

Bemerkung:
Hinsichtlich Templates (generischer Typen) spricht man von:

- **Invariation:** `B` erbt von `A` aber `T<B>` erbt dann nicht von `T<A>`
  Dies gilt für C++ Templates.
- **Covariation:** `B` erbt von `A` so folgt `T<B>` erbt dann von `T<A>`.
- **Contravariation:** `B` erbt von `A` so folgt `T<A>` erbt dann von `T<B>`.

Aufgabe Schnittstellen-Programmierung 2

Auch das Design der Klasse `Lebewesen` aus der 4) Vorlesung läßt sich verbessern, indem man gemäß des unten dargestellten UML-Diagrammes eine Schnittstelle einführt.
Man implementiere nun dieses neue Design.



- Ein Hund-Objekt gibt Laut durch die Ausgabe von `"Wau Wau"`, eine Wellensittich-Objekt gibt Laut durch die Ausgabe von `"Piep Piep"`.
- Das Lebens-Alter eines Hundes berechnet sich über `alter * 5.5`, Das Lebens-Alter eines Wellensittichs über `alter * 12.9`.

7.7. Virtuelle Destruktoren

- Der Destruktor einer Klasse, die als Basisklasse für andere Klassen dienen kannt, sollte **immer virtuell deklariert werden**.
- Dies ist notwendig um sicherzustellen, dass auch wirklich der Destruktor der Unterklasse aufgerufen wird, wenn mit einem Basisklassen-Pointer / einer Basisklassen-Referenz gearbeitet wird.
- Denn falls durch statische Bindung nur der Destruktor der Basisklasse ausgeführt wird kann es zu memory-leaks oder zu einer anderen Art von Ressourcen-Verlust kommen, wenn im Destruktor der Unterklasse Ressourcen freigegeben werden müssen.

Bsp.:

```
#ifndef X_H_
#define X_H_
#include <iostream>
using std::cout;
using std::endl;
class X {
 private:
  int* p;
 public:
  X() {
    p = new int[2];
    cout << "Konstruktor X()" << endl;
  }
  ~X();
};
#endif  // X_H_

X::~X() {
  delete[] p;
  cout << "Destruktor ~X()" << endl;
}
```

Bsp. Virtueller Destruktor

```
#define Y_H_
#include <iostream>
#include "x.h"

class Y : public X {
 private:
  int* q;

 public:
  Y() {
    q = new int[1024];
    cout << "Konstruktor Y(), Adresse von q: "
         << q << endl;
  }
  ~Y();
};

#endif  // Y_H_

#include "y.h"

Y::~Y() {
  delete[] q;
  cout << "Destruktor ~Y()" << endl;
}
```

Bsp. Virtueller Destruktor 2

```
#include "y.h"
#include <iostream>
using std::cout;
using std::endl;
int main() {
  cout << "Verwendung der Klasse Y" << endl;
  cout << sizeof(int) << endl;
  for (int i = 0; i < 8; ++i) {
    X* px = new Y;
    delete px;
  }
  return 0;
}
```

Ausgabe:

```
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x2082080
Destruktor ~X()
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x2083090
Destruktor ~X()
 ....
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x20880e0
Destruktor ~X()
```

`valgrind` liefert die Ausgabe:

```
LEAK SUMMARY:
==14445==    definitely lost: 32,768 bytes in 8 blocks
==14445==    indirectly lost: 0 bytes in 0 blocks
==14445==      possibly lost: 0 bytes in 0 blocks
```

Da der Destruktor der Unterklasse `Y` nicht aufgerufen wird, gehen $8 \cdot 1024 \cdot 4 = 32768$ Bytes verloren!

Bsp. Virtueller Destruktor 3

Ausweg: Destruktor der Basisklasse `X` virtuell deklarieren:

```
virtual ~X();
```

Ausgabe:

```
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x7e3070
Destruktor ~Y()
Destruktor ~X()
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x7e3070
Destruktor ~Y()
Destruktor ~X()
...
Konstruktor X()
Konstruktor Y(), Adresse von q: 0x7e3070
Destruktor ~Y()
Destruktor ~X()
```

`valgrind` liefert dann die Ausgabe:

```
LEAK SUMMARY:
==14567==    definitely lost: 0 bytes in 0 blocks
==14567==    indirectly lost: 0 bytes in 0 blocks
==14567==      possibly lost: 0 bytes in 0 blocks
```

7.8. Aufgabe KFZ-Steuer OOP

- In dieser Aufgabe ist nun der Entwurf aus der Aufgabe OOA und OOD aus Kapitel 3 S. 154 zu implementieren (OOP).
- Nach der Implementierung Ihres Entwurfes sollte das unten angegebene `main`-Programm ausführbar sein und die unten dargestellte Ausgabe liefern.

```
#include <iostream>
#include "halter.h"
#include "nutzfahrzeug.h"
#include "saisonfahrzeug.h"
#include "zulassungsstelle.h"

int main() {
  Zulassungsstelle zs("Musterland", "Mu");
  Halter h1("Mustermann");
  Fahrzeug f1(3000);
  zs.anmelden(&h1, &f1);
  Fahrzeug f2(5000);
  zs.anmelden(&h1, &f2);
  Saisonfahrzeug sf1(6000, 3, 10);
  zs.anmelden(&h1, &sf1);
  Halter h2("BauerXaver");
  Fahrzeug f3(4000);
  zs.anmelden(&h2, &f3);
  Nutzfahrzeug nf1(7000, 10000);
  zs.anmelden(&h2, &nf1);
  zs.output(std::cout);
  zs.abmelden(&h1, &f2);
  zs.abmelden(&h2, &f3);
  zs.output(std::cout);
  return 0;
}
```

Aufgabe KFZ-Steuer OOP 2

Ausgabe:

```
Zulassungsstelle im Landkreis: Musterland
=========================================
Name: Mustermann
--------------------------------
Kennzeichen: Mu MR 694, Hub: 3000
KFZ-Steuer: 150
--------------------------------
Kennzeichen: Mu SK 7392, Hub: 5000
KFZ-Steuer: 250
--------------------------------
Kennzeichen: Mu GA 2174, Hub: 6000
KFZ-Steuer: 240
Start-Monat: 3, Ende-Monat: 10
Gesamt-KFZ-Steuer: 640
=========================================
Name: BauerXaver
--------------------------------
Kennzeichen: Mu GK 4896, Hub: 4000
KFZ-Steuer: 200
--------------------------------
Kennzeichen: Mu GP 6806, Hub: 7000
KFZ-Steuer: 140
Jahres-km: 10000
Gesamt-KFZ-Steuer: 340
Gesamtes KFZ-Steuer Aufkommen: 980
```

Aufgabe KFZ-Steuer OOP 3

Ausgabe Fortsetzung:

```
Zulassungsstelle im Landkreis: Musterland
==========================================
Name: Mustermann
--------------------------------
Kennzeichen: Mu MR 694, Hub: 3000
KFZ-Steuer: 150
--------------------------------
Kennzeichen: Mu GA 2174, Hub: 6000
KFZ-Steuer: 240
Start-Monat: 3, Ende-Monat: 10
Gesamt-KFZ-Steuer: 390
==========================================
Name: BauerXaver
--------------------------------
Kennzeichen: Mu GP 6806, Hub: 7000
KFZ-Steuer: 140
Jahres-km: 10000
Gesamt-KFZ-Steuer: 140
Gesamtes KFZ-Steuer Aufkommen: 530
```

8. Entwurfsmuster (Design Patterns)

Entwurfsmuster (Design Pattern)
Ein Entwurfsmuster ist eine **anpassbare Lösungs-Schablone** für ein **nicht triviales** und **immer wiederkehrendes Problem** in der **Software-Architektur**.

- Ein objektorientiertes Programm besteht aus einer Vielzahl miteinander wechselwirkender Objekte.
- Für viele Probleme stellt sich die nicht triviale Frage, wie Klassen, Schnittstellen, Vererbungshierarchien zu wählen sind, damit eine leicht erweiterbare, einfach zu testende, ausgereifte, elegante Software entsteht.
- Die **Gang of Four (GOF)** (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) haben entdeckt, dass für viele Probleme immer gleiche anpassbare Lösungs-Schablonen verwendet werden können.
- In ihrem Buch *Design Patterns, Elements of Reusable Object-Oriented Software*, 1994 (über 1 Million Auflage!) haben sie die folgenden **23 Lösungs-Schablonen (Design Patterns)** erkannt, formalisiert und vorgestellt:

  a) **Erzeugungsmuster:** Abstract Factory, Builder, Factory Method, Prototype, Singleton.

  b) **Strukturmuster:** Adapter, Bridge, Decorator, Fascade, Flyweight, Composite, Proxy.

  c) **Verhaltensmuster:** Command, Observer, Visitor, Interpreter, Iterator, Memento, Template Method, Strategy, Mediator, State, Chain of Responsibility.

- Dies hat in der objektorientierten Software-Entwicklung eine Revolution ausgelöst.

Entwurfsmuster 2

- Mittlerweile sind viele weitere Entwurfsmuster hinzu gekommen.
- Entwurfsmuster bilden ein sehr wichtiges **Begriffssystem** für Entwickler.

  Sie sind die Sprache, in der sich Entwickler über den Entwurf von Software unterhalten.
- Die Kunst ist es, zu erkennen dass für eine Problemstellung Entwurfsmuster eingesetzt und ggf. kombiniert werden können (erfordert viel Erfahrung).
- Oft müssen sie an die jeweilige Problemstellung angepasst werden.
- Die eigene Software sollte immer wieder auf die Verwendung von Entwurfsmustern überprüft werden.
- Es kann sehr hilfreich sein, sogenannte **Anti-Pattern** zu studieren.
- Diese sind das Gegenstück zu Entwurfsmustern und zeigen fehlerhaftes, ungünstiges Design auf.
- Im Web findet man viele Beispiele.

8.1. Singleton

Zweck:

- Stellt sicher, dass es von einer Klasse nur genau ein Objekt gibt.
- Stellt eine globale Zugriffsmöglichkeit auf dieses Objekt bereit.

Struktur: (klassische Version von GoF)

- Es werden statische Klassen-Member verwendet, die mit dem Schlüsselwort `static` deklariert wurden.
- Diese existieren nur ein mal pro Klasse.
- Der private Konstruktor verhindert die Instanzierung außerhalb der Klasse.
- Es wurde vergessen auch den Copy-Konstruktor und den Copy-Zuweisungs-Operator mit private zu schützen.
- Diese Version ist nicht Thread-save!

Verwendung:

- Verwaltung der Konfiguration einer Software.
- Protokollierung (Logging).
- Verwaltung von einmalig vorhandenen Ressourcen, z.B. Drucker-Spooler.

8.1.1. Bsp. Singleton (klassisch)

Speicherung einer Konstanten in einem Singleton anstelle der Verwendung einer globalen Variable.

```
#ifndef SINGLETON_H
#define SINGLETON_H
#include <iostream>

class Singleton {
 private:
  static Singleton* instance;
  Singleton() {}
  ~Singleton() { destroyInstance(); }

 public:
  static Singleton* getInstance() {
    if (instance == nullptr) {
      instance = new Singleton;
    }
    std::cout << &instance << std::endl;
    return instance;
  }
  void destroyInstance() {
    if (instance != nullptr) {
      delete instance;
      instance = nullptr;
    }
  }
  int doOperation() const { return 42; }
};

#endif  // SINGLETON_H
```

Initialisierung der Klassenvariablen außerhalb des Namensbereiches der Klasse (ab C++11 nicht mehr notwendig)

```
#include "singleton.h"
Singleton* Singleton::instance = nullptr;
```

Bsp. Singleton (klassisch) 2

```
#include <iostream>
#include "singleton.h"
using std::cout;
using std::endl;

int main() {
  Singleton* sp1 = Singleton::getInstance();
  int i42 = sp1->doOperation();
  Singleton* sp2 = Singleton::getInstance();
  int j42 = sp1->doOperation();
  cout << i42 << ", " << j42 << endl;
  sp2->destroyInstance();
  sp1->destroyInstance();
  return 0;
}
```

Ausgabe:

```
0x602078
0x602078
42, 42
```

8.1.2. Bsp. Singleton (Scott Meyers)

```
#ifndef SINGLETON_H
#define SINGLETON_H
#include <iostream>
class Singleton final {
 private:
  Singleton() = default;
  Singleton(const Singleton&) = delete;
  Singleton(Singleton&&) = delete;
  Singleton& operator=(const Singleton&) = delete;
  Singleton& operator=(Singleton&&) = delete;
 public:
  static Singleton& getInstance() {
    static Singleton instance{};
    std::cout << &instance << std::endl;
    return instance;
  }
  int doOperation() { return 42; }
};
#endif  // SINGLETON_H
```

- Durch `final` wird die Vererbung unterbunden. Dadurch können auch nicht über Unterklassen mehrere Objekte erstellt werden die ein Singleton als Teilobjekt besitzen.
- Mittels `delete` für den Copy/Move-Konstruktor und den Copy/Move-Zuweisungs-Operator wird die Erstellung einer Kopie unterbunden.
- Die einzige Instanz der Sigleton-Klasse wird als lokale statische Variable gespeichert, ihre Erstellung ist nach §6.7 ISOC++11 thread sicher.

Bsp. Singleton (Scott Meyers) 2

```
#include <iostream>
#include "singleton.h"
using std::cout;
using std::endl;
int main() {
  int i42 = Singleton::getInstance().doOperation();
  int j42 = Singleton::getInstance().doOperation();
  cout << i42 << ", " << j42 << endl;
  return 0;
}
```

Ausgabe:

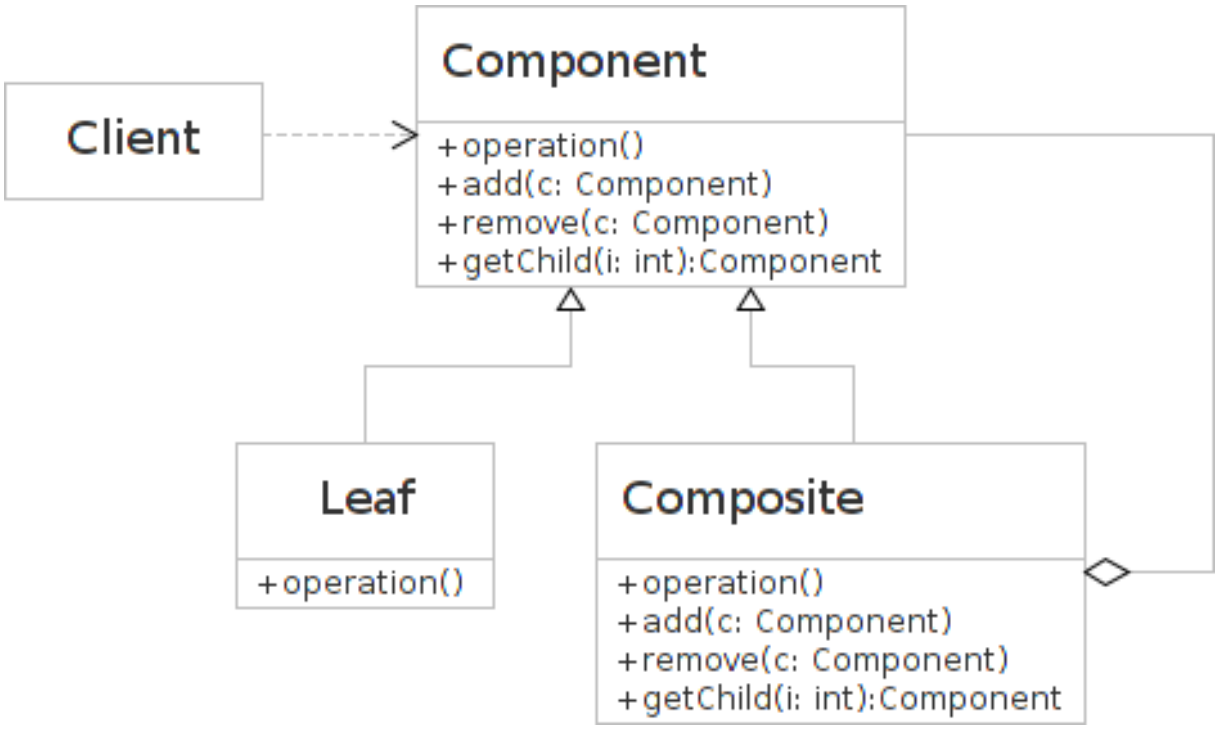
```
0x602078
0x602078
42, 42
```

8.2. Composite

Zweck:

- Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren.
- Es ermöglicht einem Client einzelne Objekte als auch Composites einheitlich zu behandeln.

Struktur



- Die Component ist eine Schnittstelle / abstrakte Klasse, die die Methode `operation()` deklariert, über die alle Elemente der Baumstruktur gleichartig vom Client behandelt werden können.
- Die Verwaltungsoperationen (`add()`, `remove()`, `getChild()`) für die Kindobjekte können auch ganz in das Composite verschoben werden.

8.2.1. Bsp. Impedanzberechnung

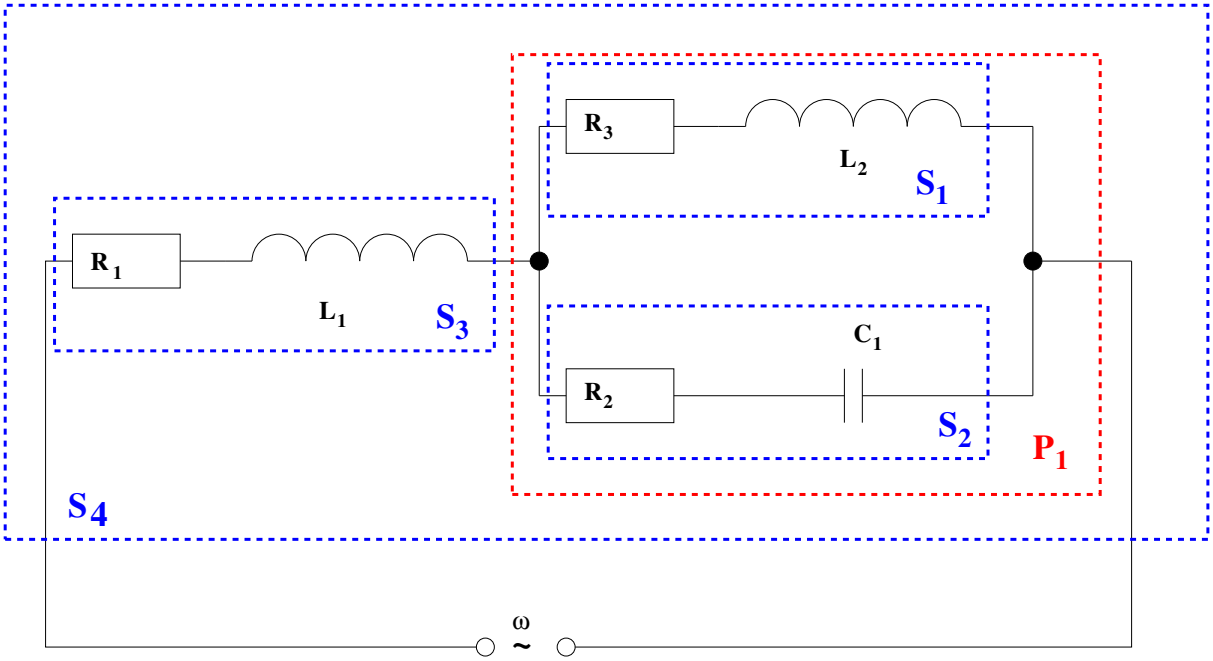
Problem:

Berechnung der Impedanz (komplexer Widerstand) von Zweipol-Schaltungen

$$Z_R = R + 0j, \qquad Z_C = \frac{-1}{\omega C}\, j, \qquad Z_L = \omega L\, j$$

Winkelgeschwindigkeit $\omega = 2\pi f$

Bsp.:



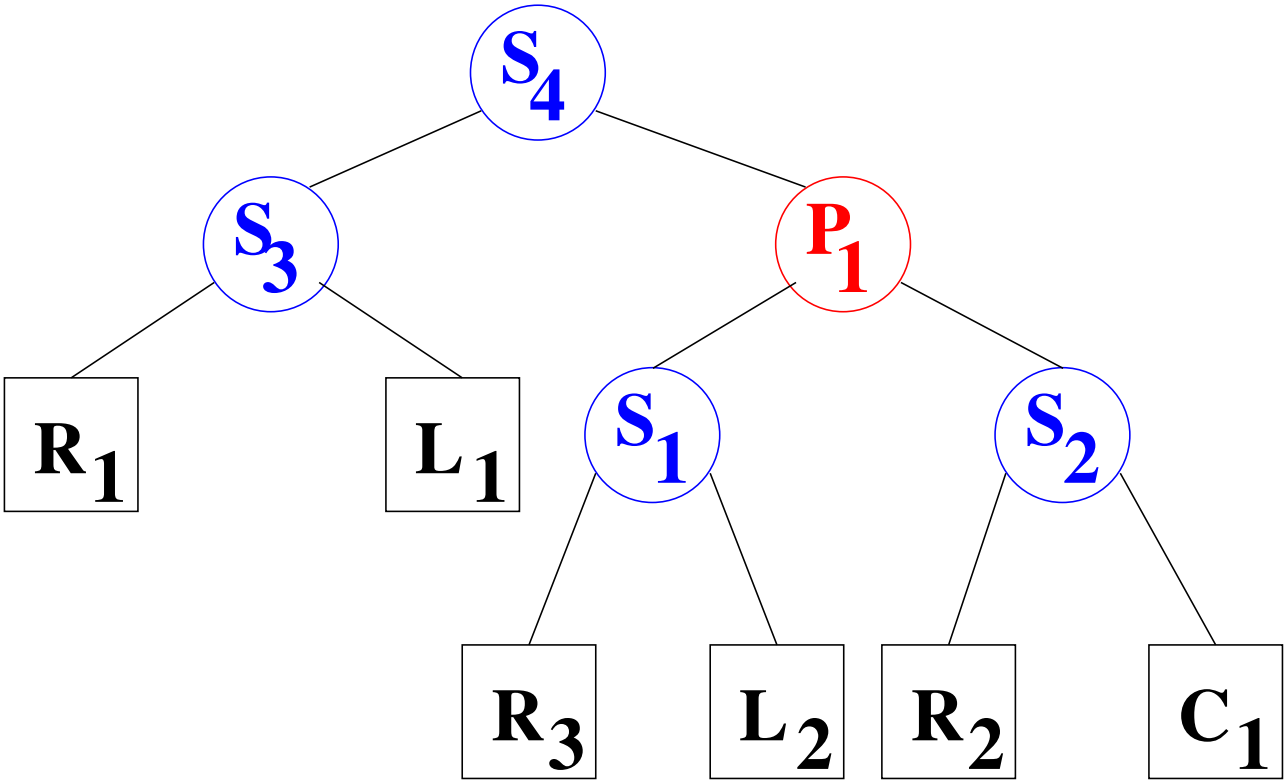
$$R_1 = 50\ \Omega, \qquad R_2 = 300\ \Omega, \qquad R_3 = 20\ \Omega$$

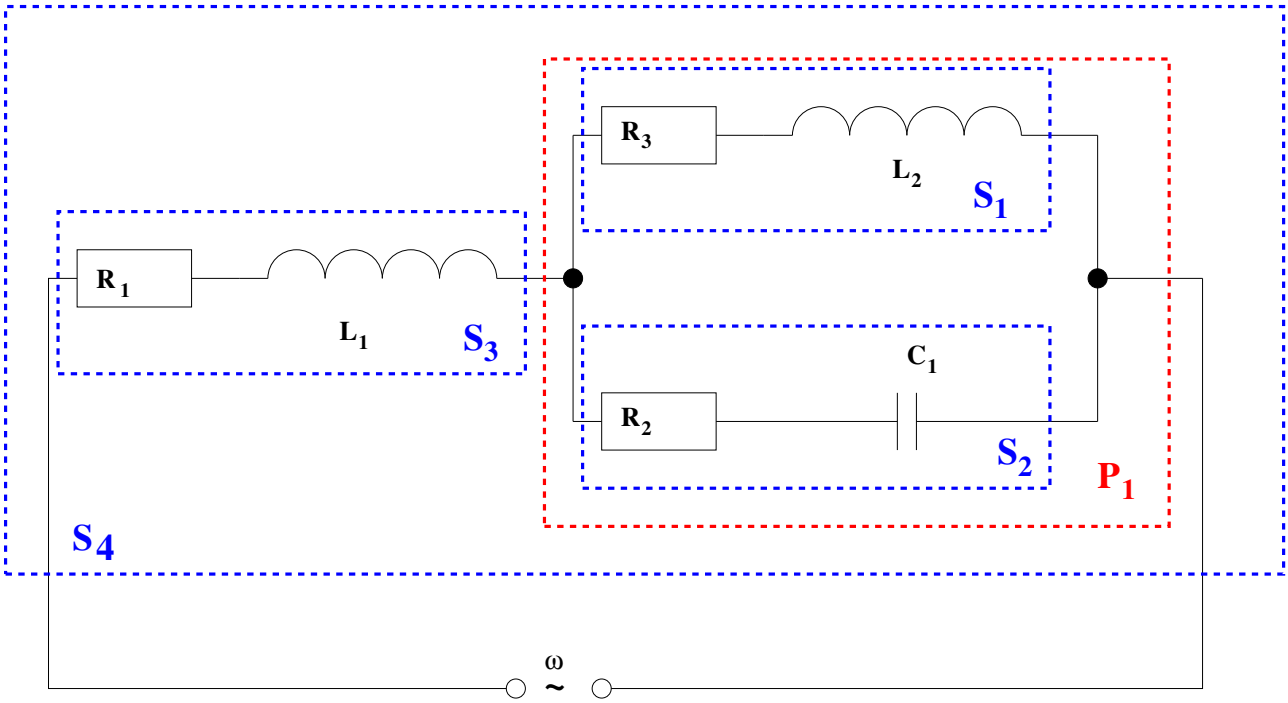
$$L_1 = 1\ H, \qquad L_2 = 1.5\ H, \qquad C_1 = 10\mu F$$

$$\omega = 300\ s^{-1}$$

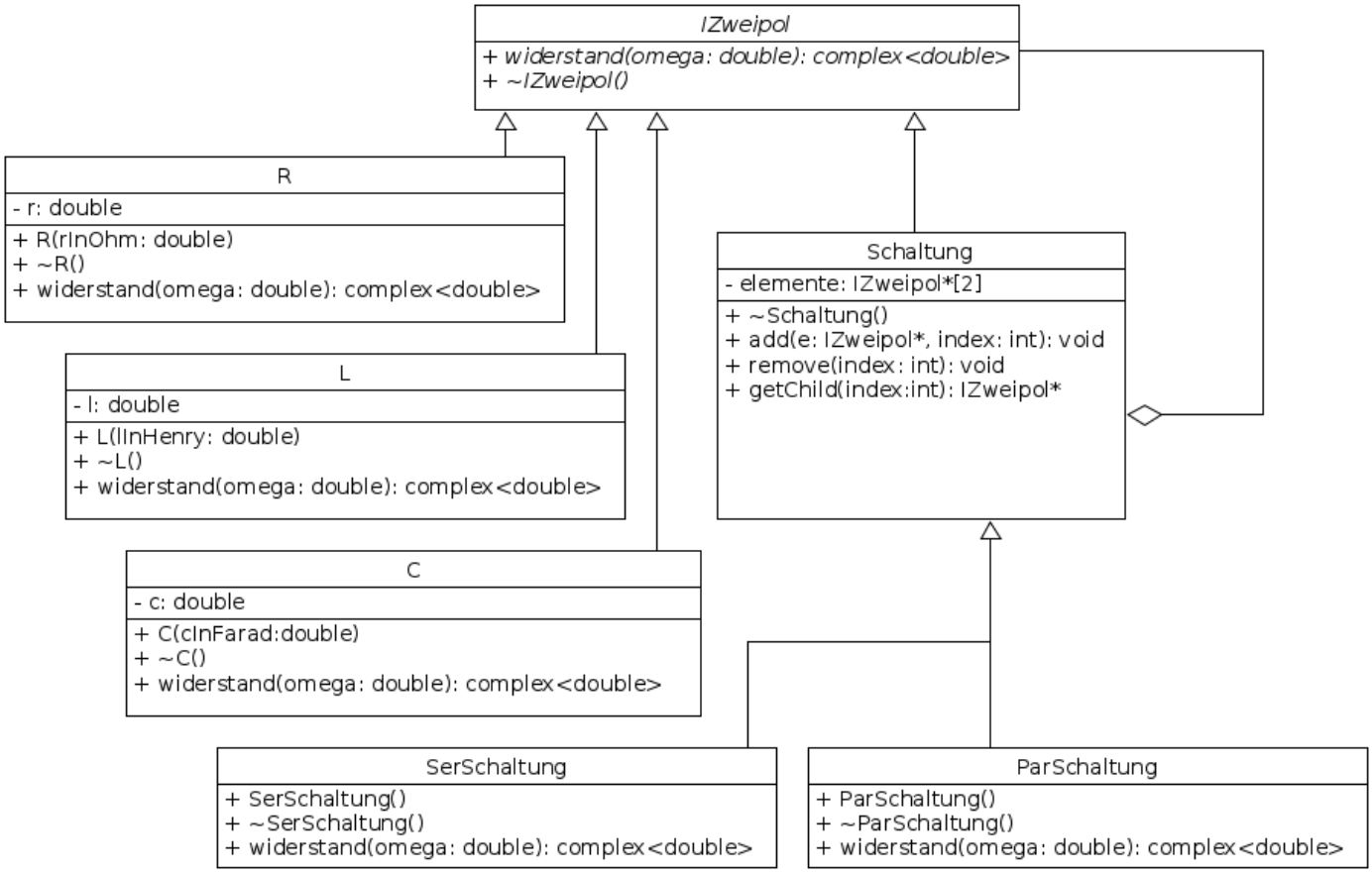
Wie kann dies mit dem Composite Pattern gelöst werden?

Zweipolschaltung als Baum





Impedanzberechnung mit Composite



Aufgabe:

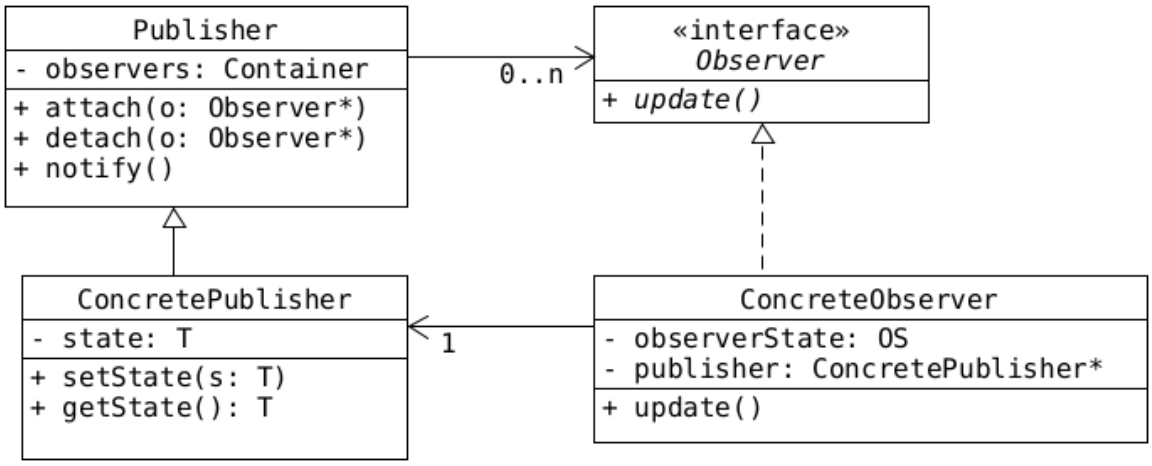
Man implementiere diese Lösung.

8.3. Observer

Zweck:

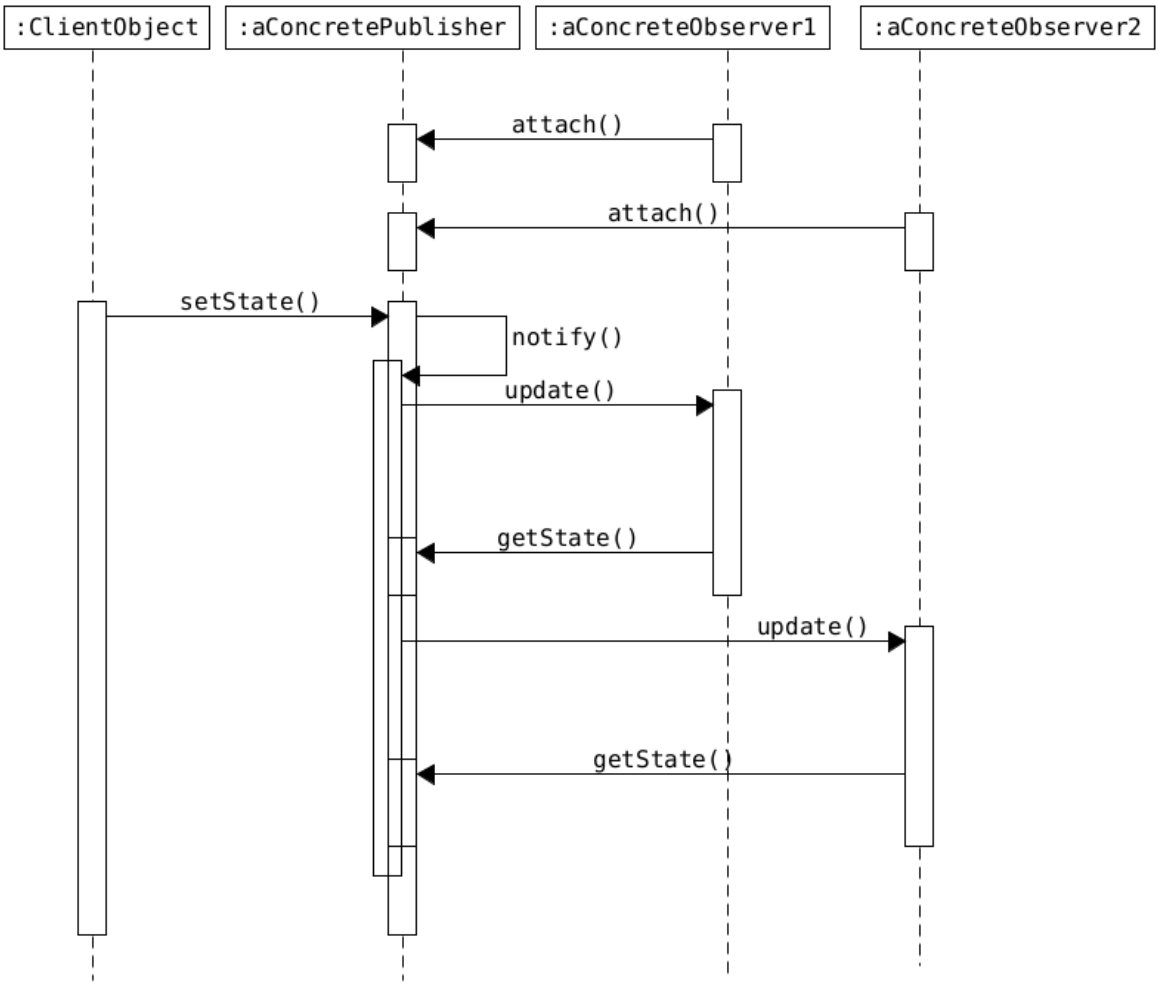
- Definiert eine 1 zu n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustandes eines Objektes (**Publisher, Subjekt**) dazu führt, dass alle abhängigen Objekte (**Beobachter**) benachrichtigt und automatisch aktualisiert werden.
- Diese Art der Interaktion wird auch als **Publish-Subscribe** (publiziere und abonniere) Kommunikations-Muster bezeichnet.

Struktur



- Die konkreten Beobachter können sich über die Methoden `attach()` bzw. `detach()` beim Publisher an bzw. abmelden.
- Ändert sich der Zustand des konkreten Publishers durch den Aufruf der Methode `setState()` wird automatisch die Methode `notify()` aufgerufen.
- In der Methode `notify()` werden alle angemeldeten Beobachter durch den Aufruf ihrer `update()` Methode über die Änderung informiert.

8.3.1. Pull und Push Modell



- Das obige Sequenz-Diagramm zeigt den Ablauf beim **Pull-Modell**.
  Der Publisher schickt an die Beobachter nur die minimale Information, dass sich sein Zustand geändert hat. Die Beobachter müssen dann entscheiden ob sie sich die Information über den neuen Wert des Zustandes selber verschaffen.
- Beim **Push-Modell** schickt der Publisher die detaillierte Information über die Änderung gleich über die `update()` Methode mit.

8.3.2. Bsp. Observer

- Ein Temperatur-Sensor wird von einer Dezimal-Anzeige und einer LED-Signal-Anzeige beobachtet.
- Die Dezimal-Anzeige gibt die neuen Temperaturwerte aus.
- Die LED-Signal-Anzeige leuchtet grün, wenn für den Temperatur-Wert T gilt: $T \in [16, 25]$, andernfalls rot.

```
#include <iostream>
#include <list>

using std::cout;
using std::endl;

class Observer {
 public:
  virtual void update() = 0;
};

class Publisher {
 private:
  std::list<Observer*> observers;

 public:
  void attach(Observer* obs) {
    observers.push_back(obs);
  }
  void detach(Observer* obs) {
    observers.remove(obs);
  }
  void notify() const {
    for (auto o : observers) {
      o->update();
    }
  }
};
```

Bsp. Observer 2

```
class TemperaturSensor : public Publisher {
 private:
  double temperatur;

 public:
  void setTemperatur(double t) {
    temperatur = t;
    notify();
  }

  double getTemperatur() const { return temperatur; }
};

class DezimalAnzeige : public Observer {
 private:
  double temperatur;
  TemperaturSensor* publisher;

 public:
  DezimalAnzeige(TemperaturSensor* pub)
    : temperatur{pub->getTemperatur()},
      publisher{pub} {
    publisher->attach(this);
  }
  ~DezimalAnzeige() { publisher->detach(this); }
  void display() const {
    cout << "Temperatur: "
         << temperatur << endl;
  }
  void update() override {
    temperatur = publisher->getTemperatur();
    display();
  }
};
```

Bsp. Observer 3

```
class LEDSignalAnzeige : public Observer {
 private:
  static int greenMin;
  static int greenMax;
  enum class MyColor { green, red };
  MyColor color;
  TemperaturSensor* publisher;

 public:
  LEDSignalAnzeige(TemperaturSensor* pub)
      : color{MyColor::green}, publisher{pub} {
    publisher->attach(this);
  }
  ~LEDSignalAnzeige() { publisher->detach(this); }
  void display() const {
    cout << "LED-Anzeige: "
    << ((color == MyColor::green) ? "green" : "red")
    << endl;
  }
  void update() override {
    double t = publisher->getTemperatur();
    if (greenMin <= t && t <= greenMax) {
      color = MyColor::green;
    } else {
      color = MyColor::red;
    }
    display();
  }
};

int LEDSignalAnzeige::greenMin = 16;
int LEDSignalAnzeige::greenMax = 25;
```

Bsp. Observer 4

```
int main() {
  TemperaturSensor ts;
  DezimalAnzeige da{&ts};
  LEDSignalAnzeige lsa{&ts};
  ts.setTemperatur(22);
  ts.setTemperatur(15.5);
  ts.detach(&da);
  ts.setTemperatur(21.3);
  ts.setTemperatur(25.4);
  return 0;
}
```

Ausgabe:

```
Temperatur: 22
LED-Anzeige: green
Temperatur: 15.5
LED-Anzeige: red
LED-Anzeige: green
LED-Anzeige: red
```

9. Ausnahmebehandlung

9.1. Laufzeitfehler/Logische Fehler

- Bisher wurde eine perfekte Welt angenommen.
- In der rauen Wirklichkeit sind **Laufzeitfehler** (unvorhersehbare Fehler zur Programmlaufzeit)
  - Fehlerhafte Benutzereingabe.
  - Datei nicht vorhanden.
  - Instabile Netzwerkverbindung.
  - Fehlschlag bei dem Anlegen von Heap-Speicher.

  und **logische Fehler (Programmierfehler):**
  - Index-Fehler, Division durch 0.
  - Bereichsüberschreitung durch Wahl des falschen Datentyps.
  - Fehlerhafte Spezifikation im Pflichtenheft.

  nicht zu vermeiden.
- **Ziel:** kontrollierte Reaktion der Software auf Fehler:
  - Benutzer bei Fehler benachrichtigen.
  - Möglicherweise Behebung des Fehlers ohne Programmabbruch.
  - Programm auf sanfte Weise beenden (gesamte bisherige Arbeit des Benutzers speichern).
- Dies ist sehr wichtig für kommerzielle Software.
- Ein Kunde wendet sich von einer Software für immer ab, wenn:
  - Die Software häufig abstürzt.
  - Eine Fehlbedienung zum Verlust von Daten führt.
  - Der Grund für einen Programmabsturz nicht ersichtlich ist.

9.1.1. Historische Programmfehler

- **Verlust der Mariner 1 Sonde 1962**

```
      DO 5 K = 1. 3
      T(K) = W0
      Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
      D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
     *(B0**2-X*B0*B1))/Z
      E(K) = H**2*93.2943*W0/SIN(W0)*Z
      H = D(K)-E(K)
    5 CONTINUE
```

  Punkt statt Komma! Schaden: 18.5 Millionen USD.

- **F-16 Jagdflugzeug, ca. 1978:**

  Erstes Jagdflugzeug der USA mit Computer-Steuerung.

  Beim Überfliegen des Äquators: Das Flugzeug stellte sich auf den Kopf.

  Ursache: Vorzeichenfehler im Algorithmus des Programms bei der Berücksichtigung der geographischen Breite.

  Glück: Fehler wurde in der Simulation gefunden.

- **B 737, ca. 1980:**

  Der automatische Vortriebsregler schaltete sich manchmal während des Startvorgangs bei 60 kn ab.

  Ursache: Software legte nur fest, was unter und was über 60 kn passieren sollte.

- **Autofabrik General Motors, Michigan, ca. 1985:**

  Alle schwarzen Autos verließen die Montagehalle ohne Windschutzscheibe.

  Die für das Einsetzen vorhandenen Roboter erkannten die Farbe schwarz nicht.

Historische Programmfehler 2

- **Therac-25: Linearbeschleuniger zur Tumor-Behandlung ca. 1983:**
  Software errechnet und steuert Bestrahlungs-Stärke entweder als Elektronenstrahlung oder als Röntgenstrahlung (100-facher Wert der Elektronenstrahlung) Elektronenstrahl brannte Loch durch die Schulter eines Patienten. Tod etlicher Patienten durch Überdosis. Grund: Operator gab den Strahltyp falsch ein.
- **New York Stock Exchange, November 1992:**
  Zum ersten Mal kam eine Aktie auf einen Wert von über 10 000 USD. Das führte zu einem Fehler im Börsen-Computer, der 5-stellige Zahlen nicht vorsah.
- **Landtagswahl Schleswig-Holstein, April 1992:**
  Programm der Landeswahlleitung hatte den 4.973%-Anteil der Grünen auf 5.023% aufgerundet und somit das Überspringen der 5%-Hürde vorgetäuscht,
- **Jungfernflug der Ariane 5 Juni 1996:**
  37 Sekunden nach Zünden der Rakete erreichte Ariane 5 eine Horizontal-Geschwindigkeit von 32768.0 (interne Einheiten). Dieser Wert lag etwa fünfmal höher als bei Ariane 4. Die Umwandlung in eine ganze Zahl führte daher zu einem Überlauf, der jedoch nicht abgefangen wurde.
- **Y2K: Year 2000, Jahreswechsel, Januar 2000:**
  Viele Fehler traten schon bei den Tests vor 2000 auf!

Viele viele mehr, z.B. unter

`http://www.sarahandrobin.com/ingo/swr/index.html`

9.1.2. Fehlerhäufigkeit und Programmgröße

Fehlerhäufigkeit:

Normale Software	25 Fehler	pro 1000 Zeilen
Wichtige Software	2-3 Fehler	pro 1000 Zeilen
Medizinische Software	0.2 Fehler	pro 1000 Zeilen
Space Shuttle Software	< 0.1 Fehler	pro 1000 Zeilen

Programmgröße:

TeX 82	14 000 Zeilen
Erste Handys	200 000 Zeilen
Hubble Bodensoftware	1 Millionen Zeilen
Luftraum-Überwachung	2 Millionen Zeilen
Space Shuttle	3 Millionen Zeilen
B-2 Stealth Bomber	4 Millionen Zeilen
Windows 95	10 Millionen Zeilen
Windows NT 4.0	16 Millionen Zeilen
Windows 2000	27 Millionen Zeilen
Windows 7	40 Millionen Zeilen
Microsoft Office2013	45 Millionen Zeilen
Facebook	61 Millionen Zeilen
Debian 5.0	68 Millionen Zeilen
Mac OS Tiger	86 Millionen Zeilen
SDI (Schätzung)	25 - 100 Millionen Zeilen
google	2 Milliarden Zeilen

9.2. Klassische Fehlerbehandlung mit Fehler-Nummern

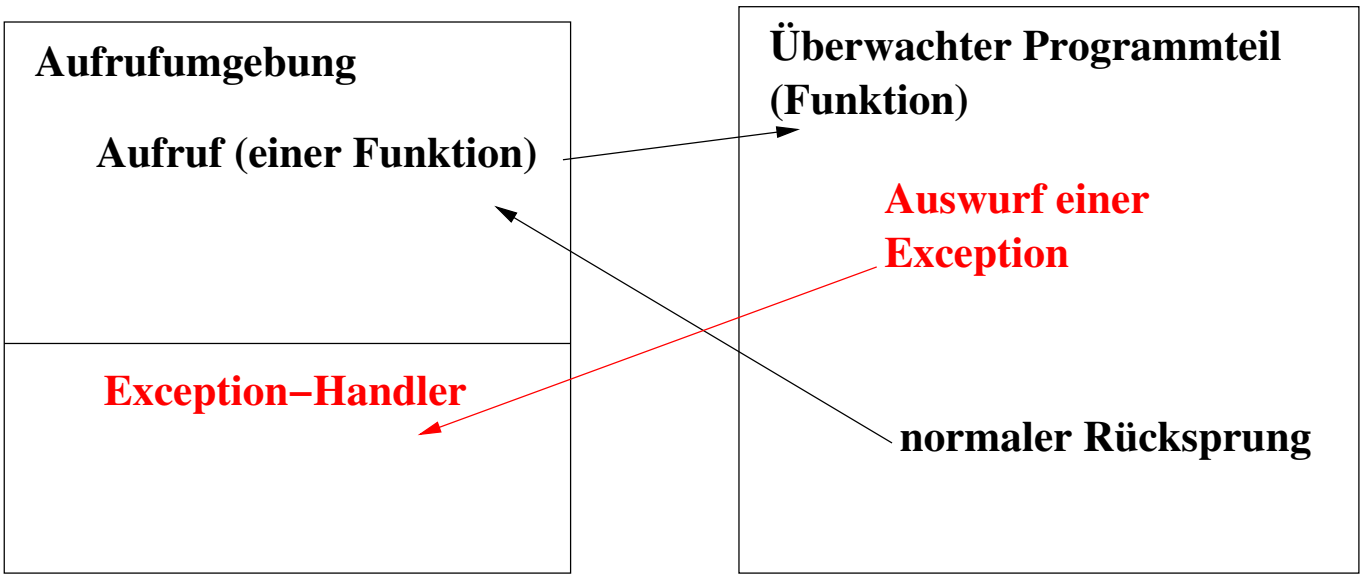
```
int main() {

  if( f1() != 0){
    // Fehler-
    // behandlung
  } else {
    // ok
    ...
  }
  ...
  return 0;
}
```

```
int f1() {
  ...
  if( f2() != 0 ){
    // Fehler
    return -1;
  } else {
    // ok
    ...
    return 0;
  }
}
```

```
int f2() {
  ...
  if( katastrophe ){
    // Fehler
    return -1;
  } else {
    // ok
    ...
    return 0;
  }
}
```

- Fehler in einer Funktionen werden durch spezielle return-Werte (**Fehler-Nummern/Fehler-Codes**) angezeigt.
- Diese Rückgabewerte werden nach jedem Funktionsaufruf abgefragt.
- **Nachteile:**
  - Starke Verschmutzung des Programm-Codes.
  - Der Rückgabe-Wert kann nicht für die Rückgabe eines normales Ergebnisses verwendet werden.
  - Erfordert Laufzeit, wirkt sich deshalb sehr negativ auf die Performance aus.

9.3. Konzept der Exceptions



- Das Exception-Handling in C++ hat das Ziel, den normalen Programmfluß von der Fehlerbehandlung zu trennen,
- dabei den Code wenig zu verschmutzen und keinen Overhead zur Laufzeit zu verursachen.
- An der Stelle im Programm-Code, an der ein Fehler entdeckt wird, wird eine Ausnahme **(Exception)** mittels `throw` ausgeworfen.
- In der Regel liegt die Stelle, an der dieser Fehler sinnvoll behandelt werden kann, an einer ganz anderen Stelle des Programm-Codes.
- Deshalb wird der normale Programmablauf deshalb unterbrochen.
- Vor dem Verlassen des überwachten Programmteils (`try`-Block) werden alle Änderungen des Stacks, die seit dem Eintritt in den überwachten Programmteil stattgefunden haben, wieder rückgängig gemacht.

  Insbesondere werden alle lokalen, nicht statischen Objekte durch den Aufruf ihrer Destruktoren zerstört. Man spricht von **Stack-Unwinding**.

Konzept der Exceptions 2

```
int main() {
  try{
     f1();
  } catch(E1& e1){
     ...
  } catch(E2& e2){
     ...
  }

  ...

   return 0;
}
```

```
int f1() {
  try{
    f2();
    if(katastrophe1){
      throw E1("K1");
    }
    ....
  } catch(E3& e3){
     ....
  }
  ....
  return 0;
}
```

```
int f2() {
  ...
  if(katastrophe2){
    throw E2("K2");
  } else if(katastrope3) {
    throw E3("K3");
  } else {
    // ok
    ...
  return 0;
  }
}
```

- Danach wird in der Aufrufumgebung nach einem passenden Exception-Handler (`catch`-Block), der den Fehler behandeln kann, gesucht.
- Nach der Ausführung eines Handlers wird das Exception-Objekt zerstört und das Programm mit der ersten Anweisung fortgesetzt, die den `catch`-Blöcken folgt.
- Findet die Aufrufumgebung in der aktuellen Stufe keinen geeigneten Exception-Handler wird die Ausnahme an die nächst höhere Stufe (Funktion im Aufruf-Pfad) weiter geleitet.
- Kann auf dem Weg durch den Aufruf-Pfad bis zur obersten Aufruf-Stufe kein passender Exception-Handler gefunden werden, wird das Programm durch den Aufruf von

  `std::terminate()`

  beendet.

9.4. Exception Auswerfen mit throw

```
throw exceptionObject{inits};
```

- In C++ kann der Typ eines Exception-Objektes ein beliebiger Typ außer `void` sein, **der sich kopieren oder verschieben läßt**.
- Insbesondere ist auch der Auswurf von Basisdatentypen oder C-Strings möglich.

  ```
  throw int(-42);
  throw "schwerer Fehler";
  ```

- Wirklich sinnvoll ist es aber, sich eigene **Fehlerklassen** zu erstellen, die genau an einen speziellen Fehlertyp angepasst sind.

  Hierzu kann man ggf. diese Fehlerklassen von den Fehlerklassen, die in der Standardbibliothek vorhanden sind, z.B. `std::exception`, ableiten (s.u.).
- Der Typ der Fehlerklasse ist wichtig, da anhand des Typs die entsprechende Fehlerbehandlung ausgewählt wird.
- Aufgrund des Stack-Unwindings kann es manchmal hilfreich sein, die Fehlerklasse um weitere Datenelemente zu erweitern, in denen man temporäre Werte/Objekte sichern kann, die sonst verloren gehen. Z.B. den Wert eines Indexes bei einer Index-Überschreitung.
- Da Exception-Objekte ggf. aber sehr oft kopiert werden, sollten keine sehr großen Datenmengen in Fehlertypen vorhanden sein.
- In C++ gilt: **Throw by value, catch by const reference**.
- Im Gegensatz zu anderen Sprachen (z.B. Java) sollten Exception-Objekte nie im Heap angelegt werden, **unsinnig ist also:** `throw new MeinFehlerTyp{inits};`

9.5. try-catch Block

```
try{
  // ueberwachter kritischer Programmteil
  ...
} catch(const E1& e1) {
  // behandle Fehler vom Typ E1
  ...
} catch(const E2& e2) {
  // behandle Fehler vom Typ E2
  ...
} ...
...
} catch(...) {
  // behanlde alle sonstigen Fehler
}
```

- Einen Programmteil, in dem Exceptions ausgelöst werden können (überwachter kritischer Programmteil), schließt man in einen `try`-Block ein.
- Die Exception-Handler für verschiedenen Typen von Exceptions folgen unmittelbar dem `try`-Block in `catch`-Blöcken.
- Es muss mindestens ein catch-Block existieren.
- Jeder catch-Block ist ein Exception-Handler, der Exceptions entsprechend seiner Deklaration in `( )` behandeln kann.
- Die Exception-Deklaration mit 3 Punkten (sogen. Ellipse) `catch ( ... ){}` kann verwendet werden, um alle übrigen Exceptions abzufangen.
- Vorsicht: **man achte auf die richtige Reihenfolge der Handler!**

try-catch Block 2

```
try{
  throw E{};
} catch(H e){
  // wann kommen wir hier her?
}
```

Der Handler `catch(H e)` wird aufgerufen, wenn:

a) `H` den gleichen Typ wie `E` hat.

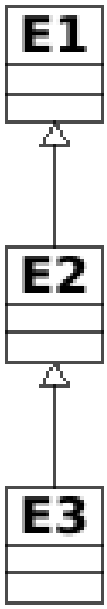
b) `H` eine direkte Oberklasse von `E` ist oder `H` eine Oberklasse von `E` in einer Vererbungshierarchie ist.

c) Wenn `H` und `E` Zeigertypen sind und a) oder b) für die Typen gilt, auf die sie verweisen.

d) Wenn `H` eine Referenz ist und a) oder b) für den Typ gilt, auf den `H` verweist.

<u>Wo steckt der Fehler?</u>

```
try{
   ...
} catch(const E1& e1) {
  ...
} catch(const E2& e2) {
  ...
} catch(const E3& e3) {
  ...
} catch(...) {
  ...
}
```



9.6. Exception erneut auswerfen

`throw;`

- Hat ein Handler eine Exception abgefangen, kann es sein, dass er den Fehler nicht vollständig behandeln kann.
- In diesem Fall erledigt der Handler normalerweise nur die Dinge, die er lokal behandeln kann.
- Er löst danach die Ausnahme erneut mit `throw;` (ohne Angabe eines Exception-Typs) erneut aus.
- Der Fehler kann dann auf einer höheren Aufrufebene endbehandelt werden.

```
void h() {
  try {
    ...
  } catch(const E1& e) {
    if (can_handle_it_completely) {
      // Fehler vollst. behandeln
      return;
    } else {
      // behandele was hier moeglich ist
      throw; // Ausnahme erneut auswerfen
    }
  }
}
```

9.7. Exception Spezifikationen für Funktionen

- In älteren C++ Versionen konnten am Ende einer Funktionsdeklaration Exceptions, die von einer Funktion geworfen werden, aufgelistet werden:

```
// f kann nur Exceptions vom Typ E1
//   bis En auswerfen
void f() throw(E1, E2, ..., En);

// f kann keine Exceptions auswerfen
void f() throw();
```

- Diese **alte Form der Exception-Spezifikation ist in C++17 ganz entfallen!**
- In der neuen Form der Exception-Spezifikation wird nur noch das Schlüsselwort `noexcept` verwendet:
  - Die Funktion kann beliebige Ausnahmen auswerfen:

    ```
    void f();      // oder

    void f() noexcept(false);
    ```

  - Die Funktion wirft keine Ausnahmen aus:

    ```
    void f() noexcept; // oder

    void f() noexcept(true);
    ```

    Der Compiler kann sich dann die Arbeit für das Stack-Unwinding sparen und es hilft ihm auch bei anderen Optimierungen.

9.8. Bsp. Stack Unwinding

```
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>

using std::cerr;
using std::endl;
using std::exception;
using std::string;

void f1();
void f2();

int main() {
  try {
    f1();
  } catch (const exception& e) {
    cerr << "Exception gefangen" << endl;
    return -1;
  }
  return 0;
}
```

Bsp. Stack Unwinding 2

```
void f1() {
  string s1;
  string* s2 = new string();
  f2();
  delete s2;
}
void f2() {
  std::ofstream fs;
  fs.open("test.txt");
  fs << "Hello World" << endl;
  throw exception();
  fs.close();
}
```

- Aufgrund des Auswurfes der Exception in `f2()` wird der Heap-Speicher für `s2` nicht frei gegeben, es entsteht ein Memory-Leak! Ausgabe von `valgrind`:

```
==8629== LEAK SUMMARY:
==8629==    definitely lost: 32 bytes in 1 blocks
==8629==    indirectly lost: 0 bytes in 0 blocks
==8629==      possibly lost: 0 bytes in 0 blocks
==8629==    still reachable: 72,704 bytes in 1 blocks
```

- In C++ gibt es keinen `finally` Block wie in Python.
- In C++ wird dieses Problem mittels RAII (*Resource Acquisition Is Initialization*) gelöst.
- Da `fs` auf dem Stack angelegt wurde, und in der C++ Bibliothek RAII strikt eingesetzt wird, wird aufgrund des Stack Unwindings `fs` durch den automatischen Aufruf des Destruktors geschlossen.

9.9. RAII anstelle von finally

Wenn eine Funktion eine Ressource (Heap-Speicher, Datei, Mutex) belegt, sollte auch sicher gestellt werden, dass die Ressource immer freigegeben wird.

Naiver Versuch, der nicht fehlertolerant ist!

```
void useFile(const char* fn) {
  FILE* f = fopen(fn, "r");

  // f verwenden ...

  fclose(f);
}
```

Ein Versuch Fehlertoleranz herzustellen:

```
void useFile(const char* fn) {
  FILE* f = fopen(fn, "r");
  if (f != nullptr) {
    try {
      // f verwenden ...
    } catch(...) {
      fclose(f);
      f = nullptr;
    }
    if (f) {
      fclose(f);
      f = nullptr;
    }
  }
}
```

Diese Lösung ist umständlich, und potentiell teuer.

RAII anstelle von finally 2

Weiterhin wird der Code erheblich komplexer wenn mehrere Ressourcen freizugeben sind. Dann hat man folgendes Problem:

```
void belege_ressourcen() {

  // Ressource 1 belegen
  ...
  // Ressource n belegen

  // Ressourcen verwenden

  // Ressource n freigeben
  ...
  // Ressource 1 freigeben
}
```

- Dies kann automatisch mithilfe lokaler Objekte über RAII (Ressource Acquisition Is Initialization) gelöst werden:
- **Ressourcen werden im Konstruktor belegt und im Destruktor wieder frei gegeben**.
- Ein Objekt gilt als konstruiert, wenn sein Konstruktor ohne Auswurf einer Exception ausgeführt wird. Nur dann wird beim Stack-Unwinding auch der Destruktor für das Objekt aufgerufen.
- Wird die Ausführung des Konstruktors durch den Auswurf einer Exception unterbrochen, werden nur die Destruktoren für die Teilobjekte, die schon erstellt wurden, beim Stack-Unwinding aufgerufen.

RAII anstelle von finally 3

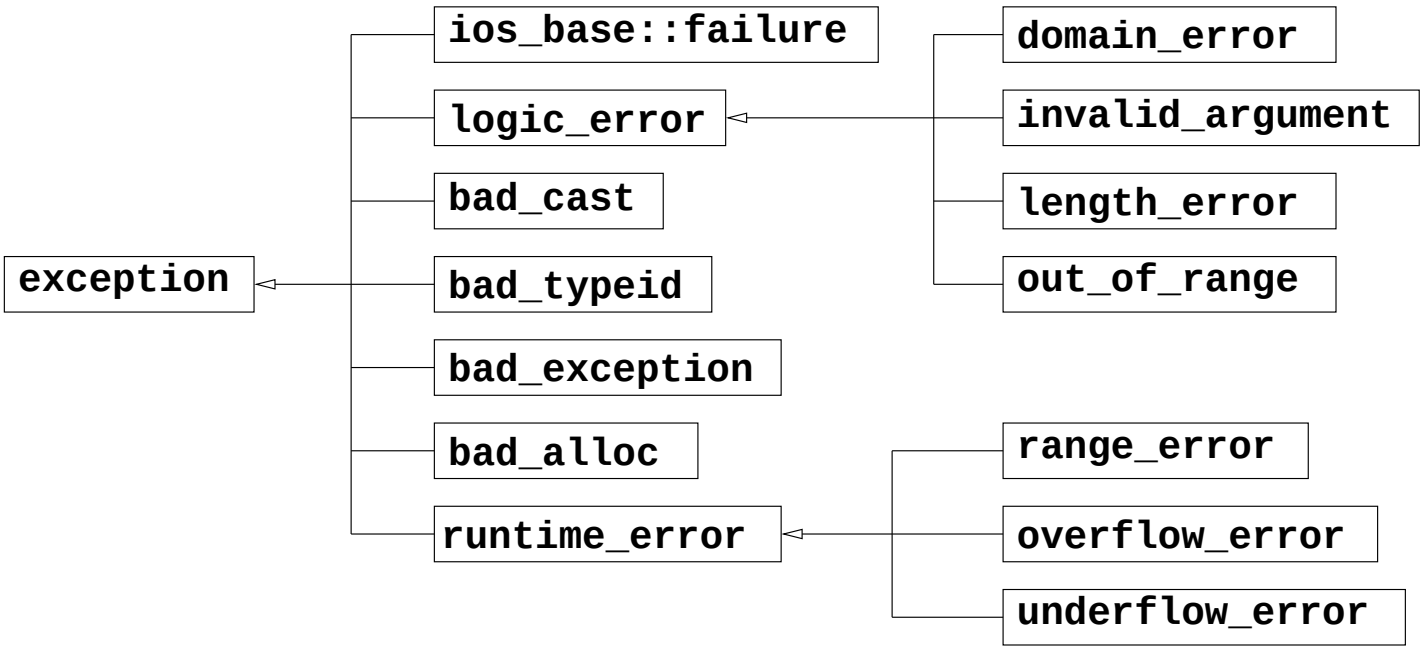
RAII Lösung:

```
class File_ptr {
 private:
  FILE* fp;

 public:
  File_ptr(const char* fn, const char* mode)
    : fp{fopen(n,a)} {
    if (fp == nullptr) {
      throw runtime_error("Fehler beim oeffnen");
    }
  }

  ~File_ptr() { fclose(fp); }

  FILE* operator(){ return fp; }
}
```

Fehlertolerante Implementierung von useFile():

```
void useFile(const char* fn) {
  File_ptr f{fn, "r"};
  // f verwenden
}
```

9.10. Fehlerklassen der Std.-Bibliothek



- Von der Basisklasse `exception` sind die beiden wichtigen Unterklassen `logic_error` und `runtime_error` abgeleitet.
- Die Fehlerklasse `logic_error` dient zur Darstellung von **logischen Fehlern (Fehlern in der Programmlogik)**, deren Ursache Programmierfehler sind. Diese Fehler sind eigentlich prinzipiell vermeidbar.
  - `domain_error`: Operanden einer Operation liegen außerhalb des definierten Bereiches.
  - `invalid_argument`: Ein Parameter beim Funktionsaufruf ist ungültig und wird nicht akzeptiert.
  - `length_error`: Die maximal erlaubte Größe für die Erzeugung eines Objektes wird überschritten.
  - `out_of_range`: Argumente liegen außerhalb des definierten Bereiches.
- `runtime_error`: Darstellung von **Laufzeitfehlern**, die nicht vorhersehbar sind und durch datenabhängige Fehler oder fehlende äußere Ressourcen verursacht werden.

9.10.1. Basisklasse exception

```
class exception {
  public:
   exception();
   exception(const exception&);
   exception& operator=(const exception&);
   virtual ~exception();
   virtual const char* what() const;
};
```

- Ist die Basisklasse für alle Fehlerklassen, die in der Standard-Bibliothek verwendet werden.
- Alle Klassen der Standard-Bibliothek erfüllen die **Basisgarantie**, dass keine Ressourcenlecks entstehen und keine Klasseninvariante verletzt wird, selbst wenn eine Exception ausgelöst wird.
- Alle Fehlerklassen werden in der Header-Datei

  `<stdexcept>`

  definiert.
- Die Methode `what()` liefert einen String, der etwas über den Fehler aussagt, zurück.
- Die Unterklassen `logic_error` und `runtime_error` besitzten auch jeweils einen Konstruktor mit dem
  - Formal-Parameter `const std::string& what_arg`
  - oder mit dem Formal-Parameter `const char* what_arg`

  mit denen dieser String gesetzt werden kann.

9.10.2. Unterklasse `ios_base::failure`

- Ab C++11 ist diese Klasse eine Unterklasse der Klasse `system_error`, die wiederum eine Unterklasse von `runtime_error` ist.
- Bei der Eingabe/Ausgabe mit Streams können Fehler auftreten.
- Ein Fehler wird durch das Setzen eines Status-Bits des Aufzählungstyps `ios::iostate` markiert.

```
enum iostate {
  goodbit = 0x00,  // alles ok
  eofbit  = 0x01,  // Ende des Streams
  failbit = 0x02,  // letzter IO fehlerhaft
  badbit  = 0x04   // ungueltige Operation
};
```

- Die Bits `badbit`, `failbit` und `eofbit` bilden das Status-Wort.
- Das `goodbit` ist eine Ausnahme, denn es entspricht eigentlich keinem einzelnen Bit eines BitMusters (des Status-Wortes) sondern es ist ein eigenständiges einzelnes Flag.

  Es hat den Wert 0 wenn im Status-Wort kein Bit gesetzt ist, das Status-Wort hat dann genau den Wert `goodbit`.
- Zusätzlich kann über eine **Exception-Maske**, die ebenfalls vom Typ `iostate` ist, festgelegt werden, ob eine Exception vom Typ `ios_base::failure` geworfen wird, wenn ein Status-Bit gesetzt wird.

  Defaultmäßig ist nur das `goodbit` in der Exception-Maske gesezt, d.h. es werden keine Exceptions im Fehlerfall ausgeworfen.

Unterklasse `ios_base::failure` 2

```
enum iostate {
  goodbit = 0x00,  // alles ok
  eofbit  = 0x01,  // Ende des Streams
  failbit = 0x02,  // letzter IO fehlerhaft
  badbit  = 0x04   // ungueltige Operation
};
```

- Mit der Methode

  `iostate exceptions() const`

  kann man die aktuelle Exception-Maske erhalten.

- Mit der Methode

  `void exceptions (iostate except)`

  kann man die Exception-Maske auf die Maske `except` setzen.

- Die `except` Maske kann man mittels bitweisem OR zusammen setzten, z.B.:

  ```
  cin.exceptions(cin.exceptions()
                 | ios_base::failbit
                 | ios_base::badbit);
  ```

  In diesem Fall werden Exceptions vom Typ `ios_base::failure` ausgeworfen, wenn das `failbit` oder das `badbit` gesetzt wird.

9.11. Zusammenfassendes Beispiel

- Der Benutzer soll eine ganze Zahl `x` eingeben.

  Es ist dann `y = 42 / (x-1);` zu berechnen (Integer-Division ist beabsichtigt).

- Bei einer syntaktisch fehlerhaften Eingabe soll eine `ios_base::failure` Exception ausgeworfen werden.

- Falls `x < 0` wird eine `out_of_range("negativer Wert!")` Exception ausgeworfen.

- Falls `x == 1` wird eine

  `std::domain_error("es droht eine Division durch Null")`

  Exception ausgeworfen.

- Falls `x == 99` wird `MyException("Eingabe von 99!!", 99)` Exception ausgeworfen.

```
#include <iostream>
#include <stdexcept>
using std::cin;
using std::cout;
using std::endl;
class MyException : public std::out_of_range {
 private:
  int myErrorNumber;
 public:
  MyException(const char* m, int eN)
      : std::out_of_range(m), myErrorNumber{eN} {}
  void printMyMessage() const {
    cout << what() << " Fehler-Nummer: "
         << myErrorNumber << endl;
  }
};
```

Zusammenfassendes Beispiel 2

```
void func() {
  int x, y;
  bool done = false;
  // Ausnahme-Ausloesung fuer cin einschalten
  cin.exceptions(std::ios_base::failbit);
  while (!done) {
    try {
      cout << "Eingabe einer positive Zahl x: ";
      cin >> x;

      if (cin.fail()) {
        throw std::runtime_error("keine ganze Zahl");
      }

      if (x == 1) {
        throw std::domain_error(
          "es droht eine Division durch Null");
      }
      if (x < 0) {
        throw std::out_of_range("negativer Wert!");
      }
      if (x == 99) {
        throw MyException("Eingabe von 99!!", 99);
      }
      y = 42 / (x - 1);
      cout << y << endl;
      if (y == 42) {
        done = true;
      }
    } catch (const std::ios_base::failure& e) {
      // cin zureuck setzen und Puffer leeren
      cin.clear();
      // fehlerhaftes Zeichen entfernen
      std::string trash;
      cin >> trash;
      cout << e.what() << endl;
      cout << trash << " keine ganze Zahl!" << endl;
```

Zusammenfassendes Beispiel 3

```
    } catch (const MyException& e) {
      e.printMyMessage();
      // Exception weiter leiten
      throw;
    } catch (const std::runtime_error& e) {
      cout << e.what() << endl;
      cin.clear();
      // fehlerhaftes Zeichen entfernen
      std::string trash;
      cin >> trash;
    } catch (const std::domain_error& e) {
      cout << e.what() << endl;
    } catch (const std::out_of_range& e) {
      cout << e.what() << endl;
    } catch (...) {
      throw;
    }
  }
}

int main() {
  cout << "Exception-Demo" << endl;
  func();
  return 0;
}
```

9.12. Exception Missbrauch zur Ablaufsteuerung

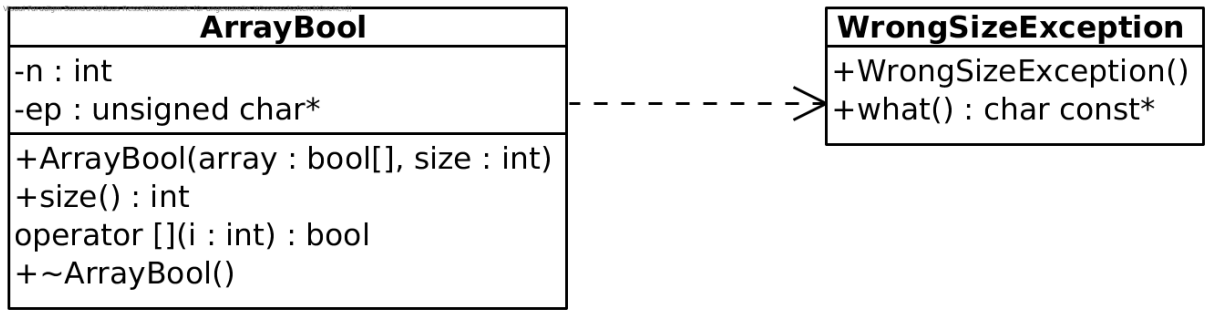
```
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
using std::cout;
using std::endl;
using std::string;

using namespace std;
class KundeNichtGefundenException : public exception {
 public:
  const char* what() const noexcept override {
    return "Kunde nicht gefunden!";
  }
};

class Kunde {
 private:
  string name;

 public:
  Kunde(string n) : name{n} {}
  string getName() const { return name; }
};

Kunde findeKunde(vector<Kunde>& kundenListe, string name) {
  for (auto k : kundenListe) {
    if (k.getName() == name) {
      return k;
    }
  }
  throw KundeNichtGefundenException{};
}
```

Exception Missbrauch zur Ablaufsteuerung 2

```
int main() {
  vector<Kunde> kundenListe;
  kundenListe.push_back(Kunde{"Meier"});
  kundenListe.push_back(Kunde{"Mueller"});
  kundenListe.push_back(Kunde{"Schulze"});
  try {
    // Kunde k = findeKunde(kundenListe, "Schmitz");
    Kunde k = findeKunde(kundenListe, "Schulze");
    cout << k.getName() << " in Liste gefunden" << endl;
  } catch (const KundeNichtGefundenException& e) {
    cout << e.what() << endl;
  } catch (...) {
    cout << "eine wirkliche Exception" << endl;
  }
  return 0;
}
```

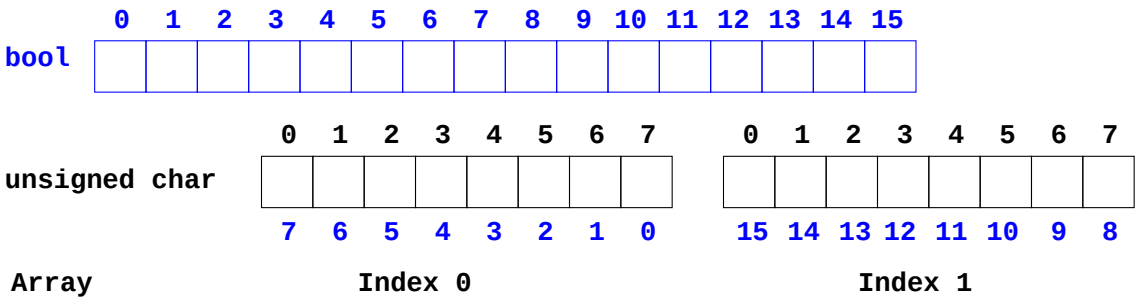
- **Sehr schlechter Programmierstil!!!**
- Das nicht Auffinden von Daten in einem Container ist keine Exception.
- Diese Lösung widerspricht der strikten Trennung des normalen Programmablaufes und der Fehlerbehandlung.
- Exceptions werden hier für die Steuerung des Programmablaufes missbraucht.

9.13. Aufgabe ArrayBool



- Man erstelle gemäß des obigen Klassendiagramms eine Klasse `ArrayBool`.
- Die Klasse `ArrayBool` soll ein Array mit `bool` Elementen als Bit-Feld (Array vom Typ `unsigned char`, das auf dem Heap angelegt ist) speichern und bietet darauf lesenden aber keinen schreibenden Zugriff.

  Die Zuordnung der Bits zu den Elementen des `bool`-Arrays ist in der unteren Abbildung beispielhaft dargestellt.



Bezeichnet `i` einen Index bezüglich des `bool` Arrays, so gilt:

```
Index = i / 8
```

und man muss um

```
i % 8
```

Bits nach links verschieben um an die entsprechende Position im `unsigned char` Feld zu kommen.

Aufgabe ArrayBool 2

- Dabei soll die Elementanzahl des `bool`-Arrays glatt ohne Rest durch 8 teilbar sein. Andernfalls soll eine Exception vom Typ `WrongSizeException` ausgeworfen werden.

  Hierzu muss ebenfalls die Klasse `WrongSizeException`, die von der Klasse `std::domain_error` erben soll, erstellt werden.

- Falls die Gößе des `bool`-Arrays größer als 32 ist, soll eine `std::length_error` Exception ausgeworfen werden.

- Falls der Index im Operator `[]` ungültig ist, soll eine `std::invalid_argument` Exception ausgeworfen werden.

- Man teste die Klasse in einem Hauptprogramm, in dem die oben angegebenen Exceptions behandelt werden, indem einfach ihre Fehlermeldung ausgegeben wird.

10. Templates

- Templates ermöglichen es in C++ **generisch** oder **generativ** zu Programmieren.
- **Generisch** bezeichnet die Anpassbarkeit eines Moduls, einer Komponente oder eines Software-Bausteines an unterschiedliche Gegebenheiten und Anforderungen.
- **Generativ** dagegen bedeutet die Möglichkeit etwas **automatisch erzeugen zu lassen**.
- Die generische oder die generative Programmierung sind eine Technologie um die **Wiederverwendbarkeit von Code** zu erreichen und **Code-Wiederholungen zu vermeiden**.
- Es kommt häufig vor, dass dieselben Algorithmen oder Klassen für **verschiedene Datentypen** benötigt werden:
  - Sortieralgorithmen, Suchalgorithmen, etc. für verschiedene Datentypen.
  - Container (Stack, Vektor, Liste, Baum) für verschiedene Datentypen.
- Mit der Hilfe von Templates kann eine Mehrfachimplementierung vermieden werden.
- Templates sind parametrisierte **Code-Schablonen**, die vom Compiler benutzt werden um Code für Funktionen **(Funktions-Template)** oder Klassen **(Klassen-Template)** zu erzeugen.
- Aus Laufzeitgründen basiert die Standardbibliothek von C++ sehr stark auf einer sehr eleganten Verwendung von Templates. Polymorphie wird sehr wenig verwendet.

10.1. Funktions-Templates

```
template <class T>
rueckgabeTyp funktionsName ( paraListe ){
    ...
}
```

- Anstelle von `template<class T>` kann man auch `template<typename T>` schreiben.
- Der Parameter T ist ein Typname, der durchgängig in der Definition wie ein normaler Typ (z.B. als Rückgabe-Typ oder als Typ für die Formal-Parameter, oder als Typ für lokale Variablen) verwendet werden kann.
- In den spitzen Klammern können auch mehere Typ-Parameter, die mit einem Komma zu trennen sind, angegeben werden.
- Mit der Definition eines Funktions-Templates wird noch keine konkrete Funktion erzeugt.

Bsp:

```
template <class T>
T max(T a, T b) {
  return a > b ? a : b;
}
int main() {
  int i = 2, j = 3;
  double x = 42.3, y = -2.9;
  float f = -1.5f;
  cout << max(i, j) << endl;
  cout << max(x, y) << endl;
  cout << max<double>(f, y) << endl;
  return 0;
}
```

10.1.1. Aufruf von Funktions-Templates

```
cout << max(i, j) << endl;
cout << max(x, y) << endl;
cout << max<double>(f, y) << endl;
```

- Erst beim Aufruf eines Funktions-Templates wird vom Compiler der entsprechende Code erzeugt, indem der Typ-Parameter durch einen konkreten Typ ersetzt wird.
- Der Typ-Parameter `T` wird entweder anhand der Funktionsargumente automatisch abgeleitet (deduziert).

  Hierfür müssen alle Parameter eindeutig aus dem Programmkontext abgeleitet werden können. Es finden keine impliziten Typ-Umwandlungen statt **(implizite Template Instanzierung)**.
- Oder der Typ-Parameter wird explizit angegeben, indem man den gewünschten Typ in die spitzen Klammern hinter den Funktionsnamen schreibt.

  Es findet dann auch eine implizite Typ-Umwandlung statt.
- Bei einem Aufruf eines Funktions-Templates benötigt der Compiler die Definition des Funktions-Templates. Nur aus einer Deklaration eines Funktions-Templates kann er keinen Code für das Instanzieren erzeugen.
- Bestimmte Fehler werden u.U sehr spät vom Compiler erkannt, die dann auch eine endlos lange Fehlermeldung verursachen

  ```
  std::complex<double> c1{1, 2}, c2{3, 4};
  cout << ::max<std::complex<double> >(c1, c2) << endl;
  ```
- Eine **explizite Template Instanzierung** kann durch folgende Anweisung erfolgen:

  ```
  template int max(int, int);
  template float max<float>(float, float);
  ```

10.1.2. Spezialisierung von Funktions-Templates

- Manchmal kann die allgemeine Lösung, die ein Funktions-Template fest legt, nicht für bestimmte Datentypen verwendet werden (da z.B. Operatoren nicht existieren) oder sie liefert falsche Ergebnisse.

  ```
  const char* name1 = "Xaver";
  const char* name2 = "Anton";
  cout << max(name1, name2) << endl;
  ```

  Liefert die falsche Ausgabe Anton, da nur die Anfangsadressen des ersten Zeichens verglichen werden.

- Dann kann eine Spezialisierung eines Funktions-Templates separat definiert werden, indem
  - `template<>` dem Funktions-Namen vorangestellt wird.
  - der Typ-Parameter durch einen konkreten Typ ersetzt wird.

Beispiel:

```
template <class T>
T max(T a, T b) {
  return a > b ? a : b;
}

// Spezialisierung
template <>
const char* max(const char* a, const char* b) {
  return std::strcmp(a, b) > 0 ? a : b;
}
```

10.1.3. Nicht-Typ-Parameter

- Neben Typ-Parametern können Templates auch Nicht-Typ-Parameter besitzen.
- Die folgenden Nicht-Typ-Parameter sind zugelassen:
  - Ganzzahlige Konstanten und Aufzählungen.
  - Zeiger (Pointer) auf Objekte, auf Funktionen, auf Attribute und auf Methoden von Klassen-Typen.
  - Referenzen auf Objekte und Funktionen.
  - `std::nullptr_t`, der Typ von `nullptr`.

Bsp.:

```
#include <cstring>
#include <iomanip>
#include <iostream>
#include <typeinfo>

using std::cout;
using std::endl;

template <int n = 1, class T>
T nTimes(T x) {
  return x * n;
}

// Spezialisierung
template <int n = 1>
const char* nTimes(const char* s) {
  std::size_t len = strlen(s);
  char* erg = new char[len * n + 1];
  for (int i = 0; i < n; ++i) {
    strncat(&erg[i * len], s, len);
  }
  return erg;
}
```

Bsp. Nicht-Typ-Parameter 2

```
int main() {
  cout << nTimes(2) << ", Type: "
       << typeid(nTimes(2)).name() << endl;

  cout << nTimes<42>(2) << ", Type: "
       << typeid(nTimes<42>(2)).name() << endl;

  cout << std::fixed << std::setprecision(3)
       << nTimes<42>(2.0)
       << ", Type: " << typeid(nTimes<42>(2.0)).name()
       << endl;

  cout << std::setprecision(3) << nTimes<42, double>(2)
       << ", Type: "
       << typeid(nTimes<42, double>(2)).name() << endl;

  const char* s = "Hello";
  auto erg = nTimes<3>(s);
  cout << erg << ", Type: " << typeid(erg).name()
       << endl;
  delete[] erg;
  erg = nullptr;
  return 0;
}
```

Ausgabe:

```
2, Type: i
84, Type: i
84.000, Type: d
84.000, Type: d
HelloHelloHello, Type: PKc
```

```
>c++filt -t i
int
> c++filt -t d
double
> c++filt -t PKc
char const*
```

Die Typ-Namen sind g++ Compiler spezifische sogenannte *name mangling* Namen, die mit dem GNU-Tool `c++filt` in die üblichen Typen gewandelt werden können:

10.1.4. Aufgabe Bubble-Sort

Man wandele die folgenden Funktionen in Funktions-Templates um. Es soll auch möglich sein, Arrays mit `const char*` Elementen zu sortieren.

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}

bool less(int x, int y) { return x < y; }

void bubbleSort(int* v, int n) {
  for (int i = 1; i < n; ++i) {
    for (int j = 0; j < n - i; ++j) {
      if (less(v[j + 1], v[j])) {
        swap(v[j], v[j + 1]);
      }
    }
  }
}

void printVec(int* v, int n) {
  for (int i = 0; i < n; ++i) {
    cout << v[i] << endl;
  }
}

int main() {
  int v[5] = {10, 8, 3, 2, 7};
  bubbleSort(v, 5);
  printVec(v, 5);
  return 0;
}
```

10.2. Klassen-Templates

```
template <class T>
class Klassenname {
  ...
};
```

- Alternativ kann statt `template<class T>` auch `template<typename T>` verwendet werden.
- Der Typ-Parameter `T` kann durchgängig in der Definition wie ein normaler Typ verwendet werden.
- Das Schlüsselwort `class` bzw. `typename` ist anzugeben, auch wenn `T` ein primitiver Datentyp ist.
- Es dürfen auch weitere Typ-Parameter und weitere Nicht-Typ-Parameter verwendet werden.
- Die Definition einer generischen Klasse erzeugt noch keine konkrete Klasse. Erst die Verwendung der Klasse mit einem konkreten Typ führt zur Erzeugung von Quellcode.
  - **implizite Instanzierung:**

    ```
    Klassenname<int> ki;
    KlassenName<Kunde*> kKunde;
    ```

  - **explizite Instanzierung:** (mit Schlüsselwort `template`)

    ```
    template class Klassenname<int>;
    template class Klassenname<Kunde*>;
    ```

10.2.1. Klassen-Name

- **Innerhalb einer Klasse** kann bei Eindeutigkeit anstelle von `Klassenname<T>` auch nur `Klassenname` geschrieben werden.

```
template<class T>
class Set{
   ...
   // hier ersetzt der Compiler
   // Set durch Set<T>
   Set(const Set&);

   // der Typparameter T ist hier
   // nicht notwendig s.o.
   Set<T>& operator=(const Set<T>&);
   ...
};
```

- Außerhalb einer Klasse muss dagegen immer der Klassenname mit dem Typ-Parameter angegeben werden:

```
template<class T>
int Set<T>::size(){...}
```

10.2.2. Aufteilung des Quellcodes

Da beim Instanzieren eines Klassen-Templates dem Compiler nicht nur die Definition des Klassen-Templates sondern auch die vollständige Implementierung der Methoden bekannt sein muss, hat man unter anderem die beiden folgenden Möglichkeiten:

1) cpp-Datei und nur explizite Instanzierung:

- Die Implementierung der Methoden kann in eine `cpp`-Datei, die im Projekt eingebunden ist, ausgelagert werden.
- Es ist dann aber **nur die explizite Instanzierung der Klassen-Templates in dieser Datei** möglich.
- Dies kommt z.B. für Bibliotheken nicht in Frage, da der Entwickler nie weiß, für welche Typen ein Anwender das Klassen-Template benötigt.

2) Implementierung alle Methoden in der Header-Datei:

- Bei diesem sogenannten (**Inklusions-Modell**) werden die Methoden ebenfalls in der Header-Datei implementiert.
- Nachteil: Durch das Inkludieren von Header-Dateien, die für die Implementierung der Methoden benötigt werden, z.B. `<iostream>` oder andere Standard-Header Dateien, kann die Header-Datei sehr groß werden.
- Eine Methode kann entweder `inline` innerhalb der Definition des Klassen-Templates definiert werden, oder außerhalb nach der Klassen-Template Definition.
- Die Klassen-Template Definition und die Implementierung der Methoden können dabei auf 2 Include-Dateien verteilt werden.
- Die Header-Datei für die Definition kann die Endung `.h` behalten, manchmal ist auch die Endung `.hcc` oder `.tcc` üblich.

Bsp. cpp-Datei u. expl. Instanzierung

Header-Datei `array.h`:

```
#ifndef ARRAY_H
#define ARRAY_H

template <class T>
class Array {
 private:
  int n;
  T* data;

  void copy_data(const Array& rhs) {
    for (int i = 0; i < rhs.n; ++i) {
      data[i] = rhs.data[i];
    }
  }

 public:
  Array(int n_) : n{n_}, data{new T[n]} {}

  Array(const Array& rhs) {
    n = rhs.n;
    data = new T[n];
    copy_data(rhs);
  }

  ~Array();

  int size() const { return n; }

  Array& operator=(const Array& rhs);

  T& operator[](int i);
};

#endif  // ARRAY_H
```

Bsp. cpp-Datei u. expl. Instanzierung 2

cpp-Datei `array.cpp`: (ist Übersetzungseinheit des Projektes)

```
#include <stdexcept>
#include "array.h"

template <class T>
Array<T>::~Array() {
  delete[] data;
  data = nullptr;
  n = 0;
}

template <class T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  if (this != &rhs) {
    n = rhs.n;
    delete[] data;
    data = new T[n];
    copy_data(rhs);
  }
  return *this;
}

template <class T>
T& Array<T>::operator[](int i) {
  if (0 <= i && i < n) {
    return data[i];
  } else {
    throw std::invalid_argument("Ungueltiger Index");
  }
}


// Explizite Instanzierung
template class Array<int>;
template class Array<double>;
template class Array<const char*>;
```

Bsp. cpp-Datei u. expl. Instanzierung 3

cpp-Datei `main.cpp`**:**

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

using std::cout;
using std::endl;

int main() {
  Array<int> vec{3};
  for (int i = 0; i < vec.size(); ++i) {
    vec[i] = i;
  }
  vec[1] = 42;
  for (int i = 0; i < vec.size(); ++i) {
    cout << i << ", " << vec[i] << endl;
  }

  Array<int> vec_neu{4};
  vec_neu = vec;
  for (int i = 0; i < vec_neu.size(); ++i) {
    cout << i << ", " << vec_neu[i] << endl;
  }

  Array<const char*> teilnehmer{2};
  teilnehmer[0] = "Max";
  teilnehmer[1] = "Hans";
  for (int i = 0; i < teilnehmer.size(); ++i) {
    cout << i << ", " << teilnehmer[i] << endl;
  }

  return 0;
}
```

Es wird auch der Code für `Array<double>` erstellt, obwohl er (in `main()`) nicht verwendet wird.

Bsp. Implementierung in Header-Datei

Header-Datei `array.h`**:**

```
#ifndef ARRAY_H
#define ARRAY_H
#include <stdexcept>

template <class T>
class Array {
 private:
  int n;
  T* data;
  void copy_data(const Array& rhs) {
    for (int i = 0; i < rhs.n; ++i) {
      data[i] = rhs.data[i];
    }
  }

 public:
  Array(int n_) : n{n_}, data{new T[n]} {}
  Array(const Array& rhs) {
    n = rhs.n;
    data = new T[n];
    copy_data(rhs);
  }
  ~Array();
  int size() const { return n; }
  Array& operator=(const Array& rhs);
  T& operator[](int i);
};

// Die folgenden Implementierungen koennen auch
// mittels #include hier eingefuegt werden
template <class T>
Array<T>::~Array() {
  delete[] data;
  data = nullptr;
  n = 0;
}
```

Bsp. Implementierung in Header-Datei 2

Fortsetzung Header-Datei `array.h`:

```
template <class T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  if (this != &rhs) {
    n = rhs.n;
    delete[] data;
    data = new T[n];
    copy_data(rhs);
  }
  return *this;
}
template <class T>
T& Array<T>::operator[](int i) {
  if (0 <= i && i < n) {
    return data[i];
  } else {
    throw std::invalid_argument("Ungueltiger Index");
  }
}

#endif  // ARRAY_H
```

- Die Implementierung der Methoden kann auch in eine Header-Datei mit der Endung `.cpp` ausgelagert werden, die dann in der Header-Datei, in der die Definition des Klassen-Templates erfolgt, am Ende inkludiert wird.
- Die `.cpp` Datei darf aber **nicht dem Projekt als zu übersetzende Einheit hinzugefügt werden**.
- Die Datei `main.cpp` aus dem ersten Beispiel kann ohne Änderungen auch hier verwendet werden.

Aufteilung in 2 Header-Dateien

- Ein anderer Ansatz ist die Aufteilung in 2 Header-Dateien.
- Die Definition des Klassen-Templates erfolgt in der Header-Datei `klassenname.h`.
- Die Implementierung der Methoden erfolgt in der Header-Datei mit dem Namen `klassenname_impl.h`.

  Am Anfang dieser Datei muss die Header-Datei `klassenname.h` inkludiert werden.

  Ein mehrfaches Inkludieren der Header-Datei `klassenname_impl.h` muss mit Include-Guards verhindert werden.
- In den Übersetzungseinheiten, in denen Instanzen des Klassen-Templates erstellt werden sollen, muss nur die Header-Datei `klassenname_impl.h` inkludiert werden.

10.2.3. Methoden-Templates

- Eine Methode eines Klassen-Templates kann auch ein Funktions-Template sein, das mit einem anderen Typ-Parameter parametrisiert ist.
- Anwendung: Es soll möglich sein, z. B. Arrays mit Typ-Paremeter `int` in Arrays mit Typ-Parameter `double` zu kopieren.

Header-Datei `array.h`:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <stdexcept>

template <class T>
class Array {
 private:
  int n;
  T* data;
  void copy_data(const Array& rhs) {
    for (int i = 0; i < rhs.n; ++i) {
      data[i] = rhs.data[i];
    }
  }

 public:
  Array(int n_) : n{n_}, data{new T[n]} {}
  Array(const Array& rhs) {
    n = rhs.n;
    data = new T[n];
    copy_data(rhs);
  }
```

Bsp Methoden-Templates

Fortsetzung Header-Datei `array.h`:

```
  // Methoden-Template
  template <class E>
  Array(const Array<E>& rhs) {
    // n = rhs.n funktioniert nicht warum?
    n = rhs.size();
    data = new T[n];
    for (int i = 0; i < n; ++i) {
      // data[i] = rhs.data[i] funktioniert nicht,
      // warum?
      data[i] = rhs[i];
    }
  }

  ~Array();

  int size() const { return n; }

  Array& operator=(const Array& rhs);

  // Methoden-Template
  template <class E>
  Array& operator=(const Array<E>& rhs);

  T& operator[](int i);

  T operator[](int i) const {
    if (0 <= i && i < n) {
      return data[i];
    } else {
      throw std::invalid_argument("Falscher Index");
    }
  }
};
```

Bsp Methoden-Templates 2

Fortsetzung Header-Datei `array.h`:

```
template <class T>
template <class E>
Array<T>& Array<T>::operator=(const Array<E>& rhs) {
  // Test auf Selbstzuweisung nicht notwendig!!
  n = rhs.size();
  delete[] data;
  data = new T[n];
  for (int i = 0; i < n; ++i) {
    data[i] = rhs[i];
  }
  return *this;
}

template <class T>
Array<T>::~Array() {
  delete[] data;
  data = nullptr;
  n = 0;
}

template <class T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  if (this != &rhs) {
    n = rhs.n;
    delete[] data;
    data = new T[n];
    copy_data(rhs);
  }
  return *this;
}
```

Bsp Methoden-Templates 3

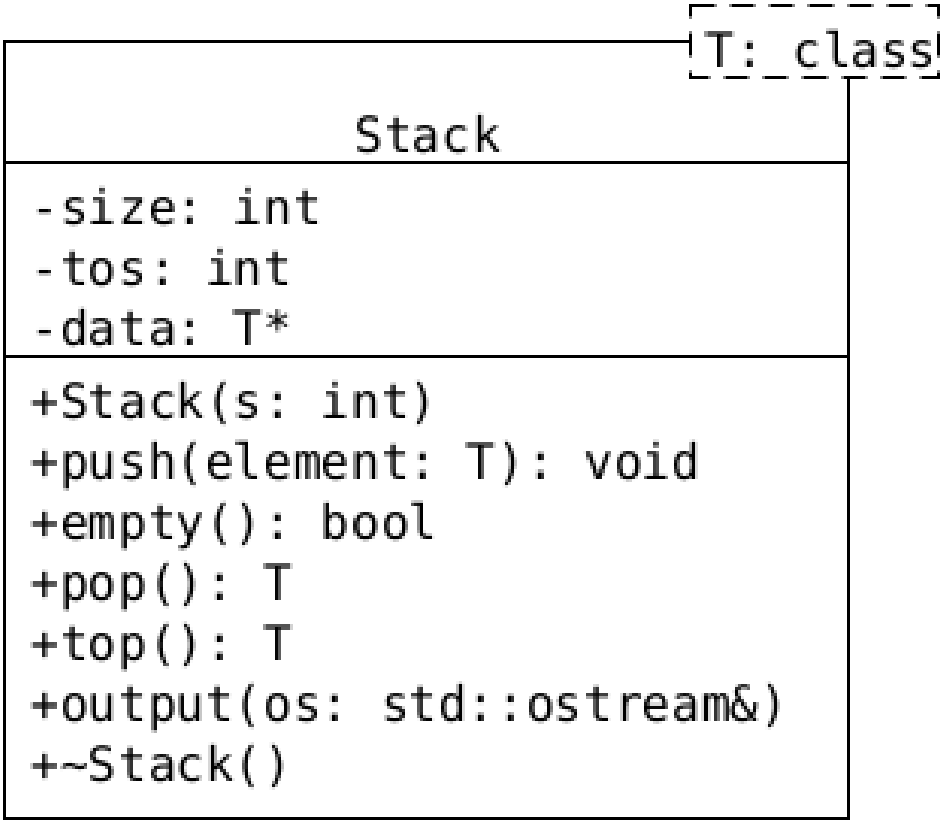
Fortsetzung Header-Datei `array.h`:

```
template <class T>
T& Array<T>::operator[](int i) {
  if (0 <= i && i < n) {
    return data[i];
  } else {
    throw std::invalid_argument("Ungueltiger Index");
  }
}

#endif  // ARRAY_H
```

Hauptprogramm `main.cpp`:

```
#include <iostream>
#include "array.h"
using std::cout;
using std::endl;
int main() {
  Array<int> vec{3};
  for (int i = 0; i < vec.size(); ++i) {
    vec[i] = i;
  }
  vec[1] = 42;
  Array<double> vecD{vec};
  vecD[0] = 3.1415;
  for (int i = 0; i < vecD.size(); ++i) {
    cout << i << ", " << vecD[i] << endl;
  }
  Array<long> vecL{4};
  vecL = vec;
  vecL[0] = 4711;
  for (int i = 0; i < vecL.size(); ++i) {
    cout << i << ", " << vecL[i] << endl;
  }
}
```

10.2.4. Aufgabe Template Stack



a) Man implementiere gemäß des obigen Klassendiagramms das Template `Stack`.

Aufgabe Template Stack 2

b) Mit Hilfe eines Stack lassen sich **vollständig geklammerte Ausdrücke** (für jeden Operator im Ausdruck existiert ein Klammer-Paar) auswerten.

```
(((15 - 3) * 4) + (6 /(2 + 1)))
```

Zunächst überführt man den Ausdruck von der **Infix-Form** in die **Postfix-Form**, indem man durch alle Zeichen im Ausdruck geht und:

- eine öffnende Klammer löscht,
- einen Operator auf einen Stack legt
- einen Operanden in das Ergebnis schreibt
- bei einer schließenden Klammer einen Operator vom Stack löscht und diesen in das Ergebnis schreibt.

Man erstelle eine Funktion

```
void infixToPostfix(const char* infix, char* postfix)
```

die diesen Algorithmus umsetzt.

c) Einen Ausdruck in der Postfix-Form kann man auswerten, indem man durch alle Zeichen im Postfix-Ausdruck geht und:

- Operanden auf den Stapel legt.
- bei einem Operator holt man die beiden letzten Operanden vom Stack und verknüpft sie in der umgekehrten Reihenfolge mit dem Operator. Das Ergebnis legt man wieder auf den Stack.

Man erstelle eine Funktion

```
double postfixToValue(const char* postfix)
```

die diesen Algorithmus umsetzt.

11. Verwendung der Standardbibliothek

11.1. Allgemeines zur ISO-C++ Standard-Bibliothek

- Diese umfasst Komponenten, die im ISO-C++-Standard spezifiziert sind und Bestandteil jeder C++-Implementierung mit identischem Verhalten (Portabilität) sind.
- Spezifikation des reinen C++ Teils umfasst ca. 785 Seiten.

  Die Beschreibung der ISO-C-Bibliothek, die ebenfalls zur C++-Standard-Bibliothek gehört, besteht nochmal aus 139 Seiten.

  Die C++-Sprachspezifikation ist nur 398 Seiten lang.
- Der final publizierte Standard kann als PDF-Datei für ca 30 Dollar erworben werden.

  Öffentlich zugänglich sind immer die *working drafts*, die über Git-Hub verwaltet werden:

  `https://github.com/cplusplus/draft`
- Das letzte *working draft* vor der finalen Veröffentlichung kann man oft im Web finden und legal herunterladen, z.B. zu C++11:

  `www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`
- Eine gute Online-Referenz ist:

  `www.cplusplus.com/reference`
- Die Bibliothek stützt sich aus Laufzeit-Gründen sehr stark auf Templates, weniger auf dynamische Polymorphie.
- Aus Gründen der Portabilität, Laufzeitperformance, Wartbarkeit und der Gewähr der Fehler-Freiheit gilt die Empfehlung:

  Immer Verwendung der C++-Standard-Bibliothek, wenn dies möglich ist.

11.1.1. Design-Aspekte

- **Allgemeingültigkeit:**

  Muss frei von Anwendungsfall-spezifischen Verfahrensweisen sein.

- **Effizienz:**

  Muss eine echte Alternative zu Hand-codierten Funktionen, Klassen und Templates trotz Allgemeingültigkeit sein.

- **Sicherheit:**

  Muss ausreichend sicher für den allgemeinen Anwendungsfall sein.

- **Komfortable Verwendbarkeit:**

  Sollte für jeden Einsteiger als auch für jeden professionellen Programmierer einfach zu verwenden sein.

- **Vollständigkeit:**

  Muss das, was sie leistet, vollständig leisten.

- **Basis für andere Bibliotheken:**

  Bietet ein Framework zur Erweiterung der angebotenen Hilfsmittel (Konventionen, Hilfstechniken).

11.1.2. Header-Dateien

- Die Verwendung von Komponenten der Standardbibliothek erfordert das Inkludieren von Header-Dateien.
- Werden die Header-Dateien ohne die Extension in der `#include`-Anweisung angegeben, so sind alle deklarierten Namen im Namensraum `std`.

  Eine Anweisung

  ```
  #include <iostream>
  ```

  wird vom Präprozessor umgewandelt in

  ```
  namespace std {
    #include <iostream.h>
  }
  ```

- Aus Kompatibilitätsgründen können Header-Dateien auch mit der Extension inkludiert werden.

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

  Die deklarierten Namen befinden sich dann aber nicht mehr im Namensraum `std` sondern im globalen Namensraum.

11.2. Komponenten der Standard-Bibliothek

Die **Container**, **Iteratoren** und die **Algorithmen** Bibliotheken bilden zusammen die **Standard Template Library (STL)**. Sie ermöglichen die effiziente Verwaltung großer Datenmengen.

Allgemeine Utilities:

`<utility>`	Operatoren und Paare
`<tuple>`	Tupel
`<type_traits>`	Typ-Traits
`<typeindex>`	Einen `type_info` als Schlüssel oder Hashcode verwenden
`<functional>`	Funktionsobjekte
`<memory>`	Ressourcenverwaltungszeiger
`<scoped_allocator>`	Allokatoren mit begrenztem Gültigkeitsbereich
`<ratio>`	Rationale Arithmetik für die Übersetzungszeit
`<chrono>`	Hilfsfunktionen für Zeit
`<ctime>`	Datum und Uhrzeit im C-Stil
`<iterator>`	Iteratoren und Unterstützung für Iteratoren

Container:

`<vector>`	Eindimensionales Array, in der Größe veränderbar
`<deque>`	Deque (Double-ended queue)
`<forward_list>`	Einfach verkettete Liste
`<list>`	Doppelt verkettete Liste
`<map>`	Assoziatives Array auch `multimap`
`<set>`	Menge, auch `multiset`
`<unordered_map>`	Assoziatives Array mit Hashtabelle
`<unordered_set>`	Menge mit Hashtabelle
`<queue>`	Warteschlange
`<stack>`	Stack
`<array>`	Eindimensionales Array fester Größe
`<bitset>`	Array von `bool`-Typen

Algorithmen:

`<algorithm>`	Allgemeine Algorithmen
`<cstdlib>`	`bsearch()`, `qsort()`

Komponenten der Standard-Bibliothek 2

Diagnostik:

`<exception>`	Ausnahmeklasse
`<stdexcept>`	Standardausnahmen
`<cassert>`	Assert-Makro
`<cerrno>`	Fehlerbehandlung im C-Stil
`<system_error>`	Unterstützung für Systemfehler

Strings und Zeichen:

`<string>`	C++ Strings
`<cctype>`	Zeichenklassifizierung
`<cwctype>`	WideChar-Klassifizierung
`<cstring>`	String-Funktionen von C
`<cwchar>`	WideChar-String-Funktionen von C
`<cstdlib>`	Allokationsfunktionen von C
`<cuchar>`	Mehrbytezeichen im C-Stil
`<regex>`	reguläre Ausdrücke

Ein-/Ausgabe:

`<iosfwd>`	Vorwärtsdeklarationen für Ein-/Ausgabe
`<iostream>`	Standard-`iostream`-Objekte und -Operationen
`<ios>`	`iostream`-Basisklassen
`<streambuf>`	Stream-Puffer
`<istream>`	Eingabe-Stream-Template
`<ostream>`	Ausgabe-Stream-Template
`<iomanip>`	Manipulatoren
`<sstream>`	Streams zu/von Strings
`<fstream>`	Streams von/zu Dateien
`<cstdio>`	`printf()` -Familie der Ein-/Ausgabe
`<cwchar>`	`printf()`-Ein-/Ausgabe von Wide-Zeichen

Komponenten der Standard-Bibliothek 3

Lokalisierung (Nationale Besonderheiten):

`<locale>`	kulturelle nationale Unterschiede
`<clocale>`	kulturelle nationale Unterschiede im C-Stil
`<codecvt>`	Facetten für Codekonvertierung

Sprachunterstützung:

`<limits>`	Numerische Limits
`<climits>`	Makros im C-Stil für numerische Limits bei Skalaren
`<cfloat>`	Makros im C-Stil für numerische Limits bei Gleitkommazahlen
`<cstdint>`	standardmäßige Typnamen für Ganzzahlen
`<new>`	dynamische Speicherverwaltung
`<typeinfo>`	Unterstützung für Laufzeittypidentifizierung
`<exception>`	Unterstützung für Ausnahmenbehandlung
`<initializer_list>`	Initialisierungsliste
`<cstddef>`	Sprachunterstützung für C-Bibliothek
`<cstdarg>`	Argumentlisten für Funktionen variabler Länge
`<csetjmp>`	Stack-Abwicklung im C-Stil
`<cstdlib>`	Programmbeendigung
`<ctime>`	Systemuhr
`<csignal>`	Signalverarbeitung im C-Stil

Numerik:

`<complex>`	Komplexe Zahlen und Operationen
`<valarray>`	Numerische Vektoren und Operationen
`<numeric>`	verallgemeinerte numerische Operationen
`<cmath>`	mathematische Standardfunktionen
`<cstdlib>`	Zufallszahlen im C-Stil
`<random>`	Zufallszahlengeneratoren

Komponenten der Standard-Bibliothek 4

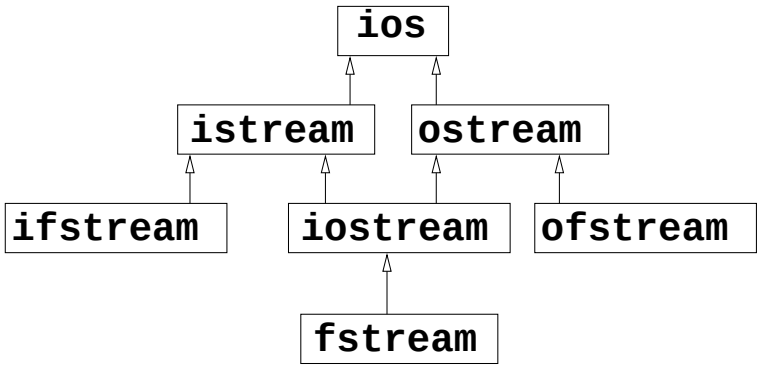
Nebenläufigkeit:

<atomic>	Atomare Typen und Operationen
<condition_variable>	Warten auf eine Aktion
<future>	Asynchrone Task
<mutex>	Mutex-Klassen
<thread>	Threads

C-Kompatibilität:

<cinttypes>	Alias für allgemeine Ganzzahltypen
<cstdbool>	C-bool
<ccomplex>	<complex>
<cfenv>	Umgebung für Gleitkommazahlen
<cstdalign>	C-Ausrichtung
<ctgmath>	typgenerische mathematische C-Makros
	<complex> und <cmath>

11.3. Dateiverarbeitung mit Streams



- Um Daten, Objekte permanent über den Programmabbruch hinaus zu speichern müssen sie auf externen Speichermedien abgelegt werden, man spricht von **Persistenz**.
- In C++ können sowohl die in C üblichen Verfahren zur Datei und Geräte Eingabe/Ausgabe (`<cstdio>`) als auch die mächtigen Verfahren, die die C++ Standard-Bibliothek bietet, verwendet werden.
- Wie in C findet auch in C++ jede Eingabe/Ausgabe über Streams statt.
- Ein **Stream** bezeichnet eine Folge von ASCII-Zeichen oder Bytes.
- Streams werden sowohl der Standard-Eingabe (Tastatur), der Standard-Ausgabe (Bildschirm) als auch Dateien oder anderen Geräten zugeordnet.
- Die von der C++ Standard-Bibliothek zur Verfügung gestellten Stream-Klassen wandeln Objekte beliebiger Typen in einen Strom aus ASCII-Zeichen (oder Bytes) und umgekehrt um.

  Sie stellen die Schnittstelle des Programms zum Dateisystem des Rechners dar.
- Der Anfangsbuchstabe `i` kennzeichnet Eingabe-Streams (input), der Anfangsbuchstabe `o` kennzeichnet Ausgabe-Streams (output).

11.3.1. Funktionalität der Stream-Klassen

- Schreiben und Lesen ähnlich zur Verwendung von `cin` und `cout`:
  - Methoden für unformatiertes Schreiben und Lesen einzelner Zeichen/Datenblöcke.
  - Formatiertes Schreiben und Lesen mit den Operatoren << und >>
  - Methoden und Manipulatoren zur Formatierung von Zeichenfolgen
  - Statusabfrage
- Methoden zum Datei-Handling:
  - Datei öffnen
  - Fehlerbehandlung
  - Datei schließen
- Methoden für blockweises Lesen und Schreiben:
  - `read()`
  - `write()`
  - File-Pointer positionieren

  Festplatten arbeiten Block-orientiert, Daten werden in Blöcken, deren Größe ein Vielfaches von 512 Byte sind, abgelegt.

  Für eine effiziente Übertragung werden die Daten deshalb in **Zwischenspeichern (Puffern)** solange gesammelt, bis die Blockgröße erreicht ist.

  Dann wird der ganze Block übertragen.

  Der Zugriff auf die Daten kann entweder **sequentiell** oder **wahlfrei** (Positionierung des File-Pointers an eine bestimmte Stelle) erfolgen.

11.3.2. Vorgehensweise

1. Inkludieren der benötigten Header-Dateien:

`<iostream>`	Definition der grundlegenden Stream-Klassen, parameterlose Manipulatoren außer File-Stream und String-Stream
`<fstream>`	Definition der File-Streams `ifstream` zum Lesen aus der Datei `ofstream` zum Schreiben in die Datei `fstream` zum Lesen und Schreiben
`<strstream>`	Definition der String-Streams
`<iomanip>`	Manipulatoren mit Parametern

2. Variablendeklaration zum Ansprechen der Datei.
3. Datei öffnen (erfolgt ggf. schon im Konstruktor).
4. Status-Abfragen und Arbeiten mit der Datei.
5. Datei schließen (erfolgt bei Objekt-Zerstörung in der Regel auch im Destruktor, RAII-Konzept).

Streams können nicht kopiert werden.

11.3.3. Klasse ofstream

Konstruktoren (C++11):

```
ofstream();
explicit ofstream (const char* filename,
  ios_base::openmode m=ios_base::out);
explicit ofstream (const string& filename,
  ios_base::openmode m=ios_base::out);
ofstream (const ofstream&) = delete;
ofstream (ofstream&& x);
```

Öffnungs-Modi:

Bitmaske	Bedeutung
`ios::out`	Schreibmodus
`ios::app`	Append (Anhängemodus, Schreiben an das Dateiende)
`ios::ate`	AtEnd automatische Positionierung an das Dateiende
`ios::in`	Lesemodus
`ios::trunc`	Truncate (der alte Dateiinhalt wird gelöscht)
`ios::binary`	Binärmodus (Systemabhängig)
`ios::nocreate`	die Datei muss bereits existieren
`ios::noreplace`	bestehende Dateiteile dürfen nicht überschrieben werden

- Der Standard-Konstruktor erzeugt noch keine Bindung zu einer Datei, diese muss dann mittels `open()` erfolgen, es wird aber schon ein Puffer-Speicher angelegt.
- Die Konstruktoren mit Parameter `filename` binden das Stream-Objektes an eine Datei. Der Konstruktor übernimmt gleichzeitig das Öffnen der Datei. Die Datei muss existieren. Die Pfadangabe ist optional. Falls die Pfadangabe weggelassen wird, muss die Datei im aktuellen Verzeichnis liegen. Die aktuelle Position nach dem Öffnen ist der Dateianfang.
- Andere Öffnungs-Modi können mit Bit-weisem oder (`|`) kombiniert werden:

  ```
  ofstream  outFile("data.dat", ios::out | ios::app);
  ```

Klasse ofstream 2

Wichtige Methoden:

- Öffnen einer Datei, falls erfolgreich wird `ios::goodbit` gesetzt, im Fehlerfall wird `ios::failbit` gesetzt.

```
void open (const char* filename,
  ios_base::openmode m=ios_base::out);

void open (const string& filename,
  ios_base::openmode mode=ios_base::out);
```

- Test ob Stream geöffnet ist: `bool is_open() const;`
- Schließen einer Datei: `void close();`
- Für viele interne Datentypen ist der Operator << überladen.
- Pointer zum internen `filebuf` Objekt:
  `filebuf* rdbuf() const;`
- Ein Zeichen in den Stream schreiben:
  `ostream& put (char c);`
- Datenblock in Stream schreiben:
  `ostream& write (const char* s, streamsize n);`

- Position des File-Pointers abfragen: `streampos tellp();`
- File-Pointer auf bestimmte Position setzen:

```
ostream& seekp (streampos pos);
ostream& seekp (streamoff off,
  ios_base::seekdir way);
```

- Pufferspeicher löschen, alle Daten werden in den Stream geschrieben

  `ostream& flush();`

Klasse ofstream 3

Im Fehler-Fall werden die folgenden Bits von `iosstate` gesetzt:

`ios::badbit`	unerwarteter schwerer Fehler
`ios::failbit`	unerwarteter Fehler
`ios::eofbit`	Ende der Eingabe erreicht (End Of File)
`ios::goodbit`	alles ist OK

Der Status kann mit den folgenden von der Klasse `ios` geerbten Methoden abgefragt und gesetzt werden:

`bool bad() const;`	Abfrage von `badbit`
`bool fail() const;`	Abfrage von `failbit`
`bool eof() const;`	Abfrage von `eofbit`
`bool good() const;`	Abfrage von `goodbit`
`iostate rdstate() const;`	Bitmuster der Fehler-Flags
`void setstate (iostate state);`	Fehler-Flags neu setzen
`void clear (iostate state = goodbit);`	Fehler-Flag löschen

Aufgabe:

Man schreibe in die Text-Datei `jaeger.txt` die *10 Jaegerlein*:

```
10 Jaegerlein
9  Jaegerlein
................
1 Jaegerlein
```

Danach probiere man ein Append mit dem gleichen Text.
Es ist zu überprüfen ob die Datei jeweils geöffnet werden konnte.

11.3.4. Klasse ifstream

Konstruktoren (C++11):

```
ifstream();

explicit ifstream (const char* filename,
  ios_base::openmode mode=ios_base::in);

explicit ifstream (const string& filename,
  ios_base::openmode mode=ios_base::in);

ifstream (const ifstream&) = delete;

ifstream (ifstream&& x);
```

Gleiche Verwendung wie bei der Klasse `ofstream`.

Wichtige Methoden:

- Öffnen einer Datei, falls erfolgreich wird `ios::goodbit` gesetzt, im Fehlerfall wird `ios::failbit` gesetzt.

  ```
  void open (const char* filename,
    ios_base::openmode mode=ios_base::in);

  void open (const string& filename,
    ios_base::openmode mode=ios_base::in);
  ```

- Test ob Stream geöffnet ist `bool is_open() const;`
- Schließen einer Datei: `void close();`
- Methoden zur Status-Abfrage wie in der Klasse `ofstream`
- Für viele interne Datentypen ist der Operator >> zum formatierten Lesen überladen.

Klasse ifstream 2

Wichtige Methoden:

- Unformatiertes Lesen:

```
int get();
istream& get (char& c);
istream& get (char* s, streamsize n);
istream& get (char* s, streamsize n,
              char delim);
istream& get (streambuf& sb);
istream& get (streambuf& sb,
              char delim);
```

- Lesen einer Zeile bis zum Terminierungs-Symbol `delim` oder bis Zeilenende `'\n'`.

```
istream& getline (char* s, streamsize n );
istream& getline (char* s, streamsize n,
                  char delim);
```

- In der Header-Datei `<string>` sind diese Methoden als freie Funktionen für die Verwendung von `string`-Objekten überladen.

```
istream& getline (istream&  is, string& str,
                  char delim);
istream& getline (istream&& is, string& str,
  char delim);
istream& getline (istream&  is, string& str);
istream& getline (istream&& is, string& str);
```

- Blockweises Lesen:

```
istream& read (char* s, streamsize n);
```

Aufgabe Arbeiten mit ifstream

1. Man lese die Datei `jaeger.txt` formatiert ein und gebe die gelesenen Daten aus.
2. Man lese die Datei `jaeger.txt` zeilenweise ein und gebe die gelesenen Zeilen aus.
3. Man kopiere die bereitgestellte CSV-Datei `csWithRating.txt` in das `build`-Verzeichnis des Projektes, (nicht in die Unterverzeichnisse `debug` oder `release` unter Windows). Man lese nun diese Textdatei zeilenweise und gebe die eingelesenen Zeilen aus.
4. Aus jeder eingelesenen Zeile extrahiere man die Daten Autor, Titel, Jahr sowie Bewertung und gebe sie aus.
5. Man füge die bereitgestellte Klasse Buch dem Projekt hinzu und speichere alle Daten in einer Variablen `bib` vom Typ `std::vector<Buch>`.

   Man gebe für jedes in `bib` gespeicherte Buch den Autor aus.
6. Es sollen nun mehrfach vorkommende Autoren nur einmal ausgegeben werden, welcher Container bietet sich hierfür an? Man gebe alle Autorennamen aus.
7. Die Bücher sollen nun sortiert nach Erscheinungsjahr in einem `map` gespeichert werden. Hierzur verwendet man das Erscheinungsjahr als Schlüssel und als zugehörigen Wert einen Container vom Typ `std::vector<Buch>`.

   Man gebe für alle vorkommenden Jahre die Anzahl der in diesem Jahr erschienen Bücher aus.
8. Für das Jahr 2015 sortiere man die Bücher nach absteigender Bewertung. Die Vergleichsfunktion soll sowohl als **Funktionspointer** als auch als **Funktionsobjekt** als auch als **Lambda-Ausdruck** übergeben werden (s. u.).

11.4. Algorithmen der STL

- In

  `<algorithm>`

  sind rund 80 Standard-Algorithmen definiert.

- Sie arbeiten auf Sequenzen, die durch ein Iterator-Paar (**Halb-Offenes-Intervall**)

  `[b: e)`

  für Eingaben, oder durch einen einzelnen Iterator `b2` für Ausgaben angegeben werden.

- Viele Algorithmen gibt es in 2 Versionen:

  – Eine einfache Version, die ihre Aktionen mit der Hilfe von konventionellen Operatoren wie < und == ausführt.

  ```
  int iv[4] = {4, 7, 1, 2};
  // aufsteigend Sortieren
  std::sort(std::begin(iv), std::end(iv));
  ```

  – Eine Version, die Schlüsseloperationen als Funktions-Argument übernimmt.

  ```
  bool greater(const int& a, const int& b) {
    return a > b;
  }
  ...
  void myBsp(){
    int iv[4] = {4, 7, 1, 2};
    // absteigend Sortieren
    std::sort(std::begin(iv), std::end(iv), greater);
  }
  ```

Algorithmen der STL 2

- Oft wird durch die Nachsilbe `_if` darauf hingewiesen, dass ein Algorithmus ein **Prädikat** (Funktion, die einen boolschen Wert zurück liefert) benötigt:

```
bool isEven(const int& a) {
  return a % 2 == 0;
}
...
void myBsp() {
  std::array<int, 5> iarr = {2, 4, 7, 1, 2};

  cout << "Anzahl gerader Elemente: "
       << std::count_if(iarr.begin(), iarr.end(),
                        isEven)
       << endl;
}
```

  Ohne Nachsilbe `_if` wird ein übergebenes Element gesucht:

```
void myBsp() {
  std::array<int, 5> iarr = {2, 4, 7, 1, 2};

  cout << "Haefigkeit von 2: "
     << std::count(iarr.begin(), iarr.end(), 2)
     << endl;
}
```

- Die Ergebnisse vieler Algorithmen werden als Iterator-Position zurückgegeben. Ein nicht gefundenes Element wird durch die Iterator-Position `end()` angezeigt.

```
auto it = std::find(iarr.begin(), iarr.end(), 3);
if (it == iarr.end()) {
   cout << "Element 3 nicht gefunden!" << endl;
}
```

11.4.1. Funktionsübergabe

- In der **Funktionalen Programmierung** versteht man unter einer **First-Class-Funktion (first class function)** eine Funktion, die:
  - einer Variablen zugewiesen werden kann,
  - in einer Datenstruktur gespeichert werden kann,
  - zur Laufzeit erzeugt werden kann,
  - von einer anderen Funktion als Wert zurückgegeben werden kann,
  - einer anderen Funktion als Parameter übergeben werden kann.
- Eine **Funktion höherer Ordnung (Higher Order Function)** bezeichnet in der funktionalen Programmierung eine Funktion, der über Parameter andere Funktionen übergeben werden können oder die eine andere Funktion als Ergebnis zurück liefert.
- Viele Algorithmen der STL verhalten sich also wie Funktionen höherer Ordnung.
- Ihnen können First-Class-Funktionen übergeben werden als:

  a) **Funktionspointer**.

  b) **Funktionsobjekt**.

  c) **Lambda-Ausdruck (anonyme Funktion)**.

11.4.2. Funktionsübergabe als Funktionspointer

Bsp:

```
bool greater(const int& a, const int& b) {
  return a > b;
}

void myBsp() {
  int iv[4] = {4, 7, 1, 2};

  bool (*fp)(const int&, const int&) = greater;
  sort(std::begin(iv), std::end(iv), fp);
}
```

Ab C++11 gibt es auch das in `<functional>` definierte Klassen-Template

```
template<class R, class... Args>
class function<R(Args...)>;
```

Es ist ein polymorpher Allzweck-Funktionswrapper. Instanzen von `std::function` können jedes aufrufbare Ziel speichern, kopieren, und aufrufen. Hierdurch vereinfacht sich die Syntax zu:

```
void myBsp() {
  int iv[4] = {4, 7, 1, 2};

  std::function<bool(const int&, const int&)> fp = greater;
  sort(std::begin(iv), std::end(iv), fp);
}
```

11.4.3. Funktionsübergabe als Funktionsobjekt

- Ein Objekt, in dessen Klasse der Funktionsaufrufoperator `T operator()(paras)` überladen wird, heißt **Funtionsobjekt (Funktor)**.
- Funktionsobjekte lassen sich wie Funktionen verwenden.

  Zusätzlich kann in ihnen ein Zustand gespeichert werden.
- Funktionsobjekte, mit denen man die Bedeutung von wichtigen Operationen eines allgemeinen Algorithmus festlegen kann werden auch **Richtlinienobjekte (policy object)** genannt.
- Funktionsobjekte ohne Übergabe-Parameter werden auch **Generator** genannt.

```
// Funktionsobjekt (Funktor)
class Greater {
 public:
  bool operator()(const int& a, const int& b) const {
    return a > b;
  }
};


void myBsp() {
  int iv[4] = {4, 7, 1, 2};

  Greater g{};
  cout << std::boolalpha
       << g(4711, 4712) << endl;  // Ausgabe false


  sort(std::begin(iv), std::end(iv), g);
}
```

Bsp. Fibonacci Generator

Funktionsobjekt das in seinen Attributen einen Zustand speichert

```
class FibonacciGenerator {
 private:
  int iold;
  int i;
  int inew;

 public:
  FibonacciGenerator() : iold{0}, i{0}, inew{0} {}
  int operator()() {
    int erg = 0;
    if (iold == 0) {
      iold = 1;
      erg = iold;
    } else if (i == 0) {
      i = 1;
      erg = i;
    } else {
      inew = iold + i;
      iold = i;
      i = inew;
      erg = inew;
    }
    return erg;
  }
};
```

Bsp. Fibonacci Generator 2

```
template<class C>
class Output {
 public:
  void operator()(C c) {
    cout << "[ ";
    bool isFirst = true;
    for (auto e : c) {
      if (isFirst) {
        cout << e;
        isFirst = false;
      } else {
        cout << ", " << e;
      }
    }
    cout << "]" << endl;
  }
};

int main() {
 Output<std::vector<int> > out{};
  std::vector<int> vFib(10, 0);
  FibonacciGenerator fibG{};
  std::generate(vFib.begin(), vFib.end(), fibG);
  out(vFib);
  return 0;
}

// Ausgabe:
// [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

11.4.4. Funktionsübergabe mittels Lambda-Ausdruck

- Ein Lambda-Ausdruck ermöglicht es ein anonymes Funktionsobjekt zu definieren und zu erstellen.
- Lambda-Ausdrücke werden in der Regel direkt an der Stelle definiert, an der sie einer Funktion höherer Ordnung übergeben werden.

```
[] () { }
```

- Die **Erfassungsliste (capture list)** `[]` leitet einen Lambda-Ausdruck ein und legt fest, welche Variablen aus der Definitionsumgebung im Rumpf des Lambda-Ausdruckes verwendet werden können (s.u.).
- Die **Parameterübergabeliste** `()` legt die Parameter fest, die der anonymen Funktion übergeben werden.
- Der **Rumpf (Textkörper)** `{}` beinhaltet den auszuführenden Programm-Code.

```
sort(std::begin(iv), std::end(iv),
       [](const int& a, const int& b) {
         return a > b; });
```

Erfassungsliste eines Lambda-Ausdruckes

- `[]`: Leere Erfassungsliste: keine lokalen Variablen aus der umgebenden Definitionsumgebung können im Rumpf verwendet werden.
- `[&]`: Implizite Erfassung als Referenz: Alle lokalen Variablen können im Rumpf verwendet werden, der Zugriff erfolgt als Referenz.
- `[=]`: Implizite Erfassung als Wert: Alle lokalen Variablen können im Rumpf verwendet werden, der Zugriff erfolgt über eine Wert-Kopie.
- `[&v1, v2, ...]` Explizite Erfassung: Die in der Liste angeführten Variablen können im Rumpf verwendet werden. Variablen, denen ein `&` vorangestellt ist werden als Referenz erfasst, alle anderen als Wert. Die Liste kann auch `this` enthalten.
- `[&, v1, v2, ...]` Implizite Erfassung als Referenz bis auf die in der folgenden Liste angeführten Variablen.
- `[=,&v1, &v2, ...]` Implizite Erfassung als Wert bis auf die in der folgenden Liste aufgeführten Variablen.

```
void printModulo(const std::vector<int>& v,
  std::ostream& os, int m) {
  // print v[i] falls v[i]%m == 0
  auto modp = [&os, m](int x) {
    if (x % m == 0) {
      os << x << endl;
    }
  };
  std::for_each(std::begin(v),
    std::end(v), modp);
}
```

Äquivalentes Funktionsobjekt

```
void printModulo(const std::vector<int>& v,
  std::ostream& os, int m) {
  // print v[i] falls v[i]%m == 0
  auto modp = [&os, m](int x) {
    if (x % m == 0) {
      os << x << endl;
    }
  };
  std::for_each(std::begin(v),
    std::end(v), modp);
}
```

Das zu diesem Lambda-Ausdruck äquivalente Funktionsobjekt lautet:

```
class ModPrint {
  private:
   std::ostream& os;
   int m;

  public:
   ModPrint(std::ostream& _os, int _m)
     : os{_os}, m{_m} {}
   void operator()(int x) const {
     if (x % m == 0) {
       os << x << endl;
     }
   }
};
```

<u>Man beachte:</u> Standardmäßig ist der überladene Aufrufoperator `const`.

mutable und Rückgabetyp

Der Lambda-Ausdruck muß als `mutable` deklariert werden, wenn das äquivalente Funktionsobjekt durch den überladenen Aufrufoperator modifiziert wird.

```
void fillBackwards(std::vector<int>& v) {
  int n = v.size();
  std::generate(v.begin(), v.end(),
    [n]() mutable -> int { return --n; });
}
```

Im äquivalenten Funktionsobjekt wird durch den konstanten Aufrufoperator das Attribut `n` verändert.

```
class CountBack {
  private:
   mutable int n;

  public:
  CountBack(int _n) : n{_n} {}
  int operator()() const { return --n; }
};
```

Durch das Voranstellen von `mutable` bei der Deklaration eines Attributes wird es ermöglicht, das Attribut auch in `const` Methoden der Klasse zu ändern.

Der Typ `R` des Rückgabe-Wertes eines Lambda-Ausdruckes kann in der folgenden From angegeben werden:

```
[]() -> R {}
```

Ab C++14 kann er entfallen wenn der Compiler den Typ aus dem Ergebnis ableiten kann.

11.5. Funktionen höherer Ordnung als Template

- Durch die Definition einer Funktion höherer Ordnung als Template ist es möglich, First-Class-Funktionen sowohl als Funktionspointer, als auch als Funktionsobjekt sowie als Lambda-Ausdruck zu übergeben.
- Dies wird am Beispiel der folgenden Funktionen höherer Ordnung demonstriert, die jede Programmiersprache, die das Funktionale Programmier-Paradigma unterstüzt, üblicherweise besitzt:

Funktional	C++
`map`	`std::transform`
`filter`	`std::remove_if`
`reduce` (`fold`)	`std::accumulate`

```
template<class C, typename F>
C myMap(typename C::const_iterator b,
  typename C::const_iterator e, F f)  {
  C erg;
  while (b != e) {
    erg.push_back(f(*b));
    ++b;
  }
  return erg;
}

template<class C>
void printCout(C v) {
  for(auto e:v) {
     cout << e << ", ";
  }
  cout << endl;
}
```

Funktionen höherer Ordnung als Template 2

```
template<class C, typename F>
C myFilter(typename C::const_iterator b,
  typename C::const_iterator e, F f) {
  C erg;
  while (b != e) {
    if (f(*b)) {
      erg.push_back(*b);
    }
    ++b;
  }
  return erg;
}

template<class C, typename F>
typename C::value_type myReduce(
  typename C::const_iterator b,
  typename C::const_iterator e, F f) {
  typename C::value_type erg = 0;
  if (b != e) {
    erg = *b;
  }
  ++b;
  while (b != e) {
    erg = f(erg, *b);
    ++b;
  }
  return erg;
}
```

Funktionen höherer Ordnung als Template 3

```
template<class T, int n>
class AbsGreatherThanN{
public:
  bool operator()(T e) const {
    return abs(e) > n;
  }
};

int main(){
  std::vector<int> v(12, 0);
  int i = 0, vz = 1;
  for(auto& e: v){
    ++i;
    e = vz * i;
    vz *= -1;
  }
  printCout(v);
  // 1, -2, 3, -4, 5, -6, 7, -8, 9, -10, 11, -12,
  std::vector<int> v1 = myMap<std::vector<int> >(
    v.begin(), v.end(), [](int x){return x*x*x;});
  printCout(v1);
  // 1, -8, 27, -64, 125, -216, 343, -512, 729, -1000,
  //  1331, -1728,
  std::vector<int> v2 = myFilter<std::vector<int> >(
    v1.begin(), v1.end(),
    AbsGreatherThanN<int,50>{} );
  printCout(v2);
  // -64, 125, -216, 343, -512, 729, -1000, 1331, -1728,
  int erg = myReduce<std::vector<int> >(v2.begin(),
    v2.end(), [](int a, int b) {
      return a + b;
    } );
  cout << erg << endl;
  // -992
  return 0;
}
```

11.6. Aufgaben Funktionale Programmierung

1. Man erstelle ein Template-Funktions-Objekt `DruckeContainer` mit Typ-Parameter `T`, mit dessen Hilfe der Inhalt beliebiger Container ausgegeben werden kann. Übergabe-Parameter des überladenen Aufruf-Operators sind die beiden Iteratoren `typename T::const_iterator b` und `typename T::const_iterator e`.

   Dabei soll z. B. ein Container, in dem die `int`-Zahlen $1, 2, \ldots, 10$ gespeichert sind wie folgt ausgegeben werden:

   ```
   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
   ```

2. Man erstelle ein Funktions-Template

   ```
   template <class T, class F1, class F2>
   void verkettung(typename T::iterator b,
                   typename T::iterator e,
                   F1 f1, F2 f2)
   ```

   das jedes im Container vom Typ `T` gespeicherte Element `x` ersetzt durch `f1(f2(x))`.

3. Man erstelle ein Funktions-Template

   ```
   template <class T, class C>
   bool allTrue(typename T::const_iterator b,
                typename T::const_iterator e, C c)
   ```

   das nur dann den Wert `true` zurück liefert, wenn der Comparator `c` für alle im Container vom Typ `T` gespeicherten Elemente das Ergebnis `true` liefert.

Aufgaben Funktionale Programmierung 2

4. Man erstelle ein Template-Funktions-Objekt `PlusN` mit Typ-Parameter `T` und Nicht-Typ-Parameter `int n`. Darin soll der überladene Aufruf-Operator den Übergabe-Parameter `T x` haben und als Ergebnis `n + x` zurück liefern.

5. Man erstelle ein Template-Funktions-Objekt `MalN` mit Typ-Parameter `T` und Nicht-Typ-Parameter `int n`. Darin soll der überladene Aufruf-Operator den Übergabe-Parameter `T x` haben und als Ergebnis `n * x` zurück liefern.

6. Man erstelle ein Template-Funktions-Objekt `GroesserAlsN` mit Typ-Parameter `T` und Nicht-Typ-Parameter `int n`. Darin soll der überladene Aufruf-Operator den Übergabe-Parameter `T x` haben und als Ergebnis `x > n` zurück liefern.

7. Man teste nun die Implementierten Klassen und Funktionen indem man als Container `std::vector<int> vec;` verwendet und:

   a) Diesen mit dem Inhalt `[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]` befüllt und danach ausdruckt.

   b) Auf jedes Element des Containers die Verkettung `f1(f2(x))` anwendet mit `f1(x) = 42 + x` und `f2(x) = 2*x` und danach das Ergebnis ausgibt.

   c) Dann überprüft ob für alle Elemente x des Containers gilt $x > 50$ bzw. $x > 0$.

8. Anstelle der Funktions-Objekte ersetze man im Test unter 7. diese durch Lambda-Ausdrücke.

9. Man verwende nun für die Tests in 7. und 8. als Container `std::list<double> lst;`.

12. Einführung in Qt

12.1. Graphische Benutzeroberflächen

- engl.: Graphical User Interface (GUI).
- Bezeichnet die Form einer Benutzerschnittstelle für eine Anwendungssoftware (Programm).
- Sie ermöglicht die Bedienung des Programmes mittels grafischer Symbole, die **Steuerelemente** oder auch **Widgets** genannt werden.
- Die Bedienung geschieht bei Computern meistens mittels einer Maus als Steuergerät, mit der die grafischen Elemente bedient oder ausgewählt werden können.
- Bei Smartphones, Tablets und Kiosksystemen erfolgt die Bedienung durch die Berührung eines Sensorbildschirms mit den Fingern oder mit einem Stift.
- Für kommerzielle Anwendungen ist eine Bedienung mit einem intuitivem GUI heutzutage Pflicht.

12.1.1. Ablauf Konsolen-Anwendung

```
qtcreator_process_stub
Mehrwertsteuerberechnung
Eingabe Preis: 100
19
Noch eine Berechnung? Eingabe J/N: J
Mehrwertsteuerberechnung
Eingabe Preis: 10
1.9
Noch eine Berechnung? Eingabe J/N: N
Betätigen Sie die <RETURN> Taste, um das Fenster zu schließen...
```

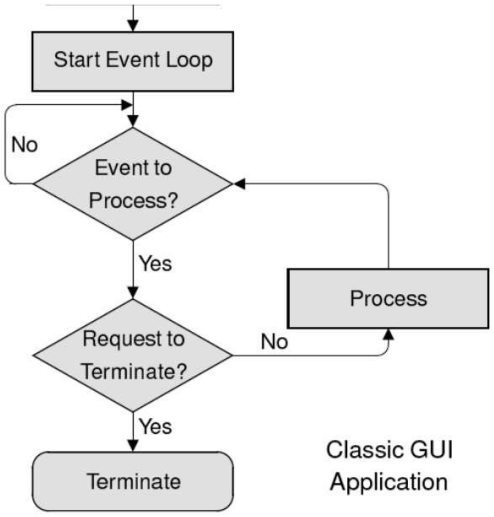
- Der Programm-Ablauf erfolgt nach dem **EVA-Modell**:

  Eingabe, Verarbeitung, Ausgabe

- Konsolen-Anwendungen werden von allen üblichen Programmiersprachen unterstützt.
- Das Programm bestimmt was passiert.
- Man spricht von **ablaufgesteuerter Programmierung**.

12.1.2. Ablauf GUI-Anwendung

- Der Benutzer kann durch die Bedienung der Steuerelemente **Ereignisse (Events) auslösen**.
- Das Programm reagiert auf den Eintritt eines Ereignisses in bestimmter Weise, indem der dem Ereignis zugeordnete Programmcode (**call-back**) ausgeführt wird.
- Danach wartet das Programm auf weitere Ereignisse.
- Eine **Ereignisschleife (event loop)** wird während der gesamten Lebensdauer des GUIs ausgeführt.



- Der Programmablauf wird vom Benutzer maßgeblich bestimmt und nicht dem Benutzer vom Programm diktiert (er ist oft zufällig **asynchron**).
- Man spricht von einer **Ereignisorientierten Programmierung**.

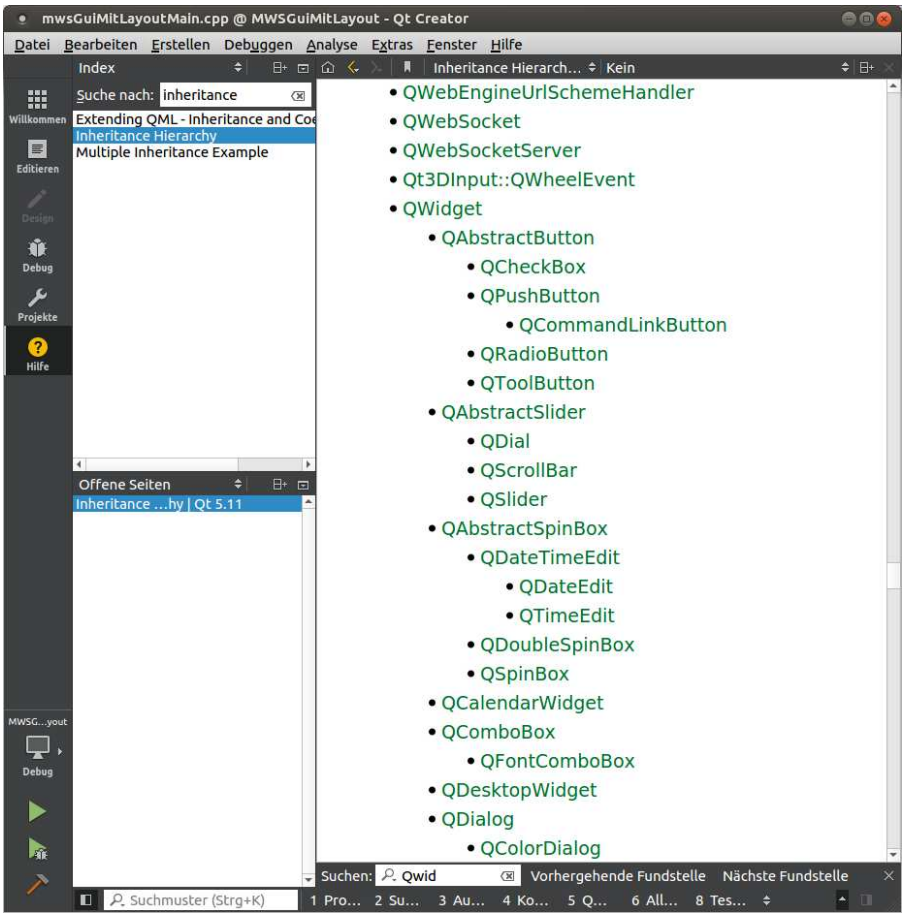
12.2. Toolkits

- In Standard C++ ist keine Bibliothek für die Erstellung von GUIs vorhanden, hierfür müssen externe GUI-Toolkits verwendet werden.
- **GUI-Toolkits** sind Bibliotheken, mit deren Hilfe sich GUIs erstellen lassen.
- Sehr bekannt sind die Plattform-abhängigen Toolkits **Microsoft Foundation Classes (MFC)** und deren Nachfolger **Windows Forms in .Net** für Windows.
- **GTK+** (the Gimp Toolkit) wurde ursprünglich für Linux Systeme, speziell für das Bildbearbeitungs-Programm GIMP und den GNOME-Desktop entwickelt. Heute ist es eingeschränkt auch unter Windows und MacOs nutzbar.
- **Tk** wurde als Erweiterung für die Scriptsprache **Tcl (Tool Command Language)** speziell für UNIX X11 entwickelt. Es ist heutzutage cross-Plattform kompatibel, z.B. für Linux, Windows, MacOS, aber sehr speziell auf die Programmierung mit Tcl zugeschnitten, man spricht dann von **Tcl/Tk**.
- **wxWidgets** ist ein in C++ geschriebenes cross-Plattform kompatibles GUI-Toolkit.
- **Qt** ist ein sehr umfangreiches und ausgereiftes GUI-Toolkit, das von vielen professionellen Anwendungen (Adobe Photo Shop, Google Earth, Skype, Spotify, Mathematica) verwendet wird.

  Qt ist auch die Grundlage auf der **KDE (K Desktop Environment)** und viele damit einhergehende Open-Source-Anwendungen erstellt wurden. Die Ereignisorientierte Programmierung wird in Qt mit Hilfe eines sehr mächtigen **Signal-Slot-Konzeptes** umgesetzt.

12.3. Bestandteile von Qt

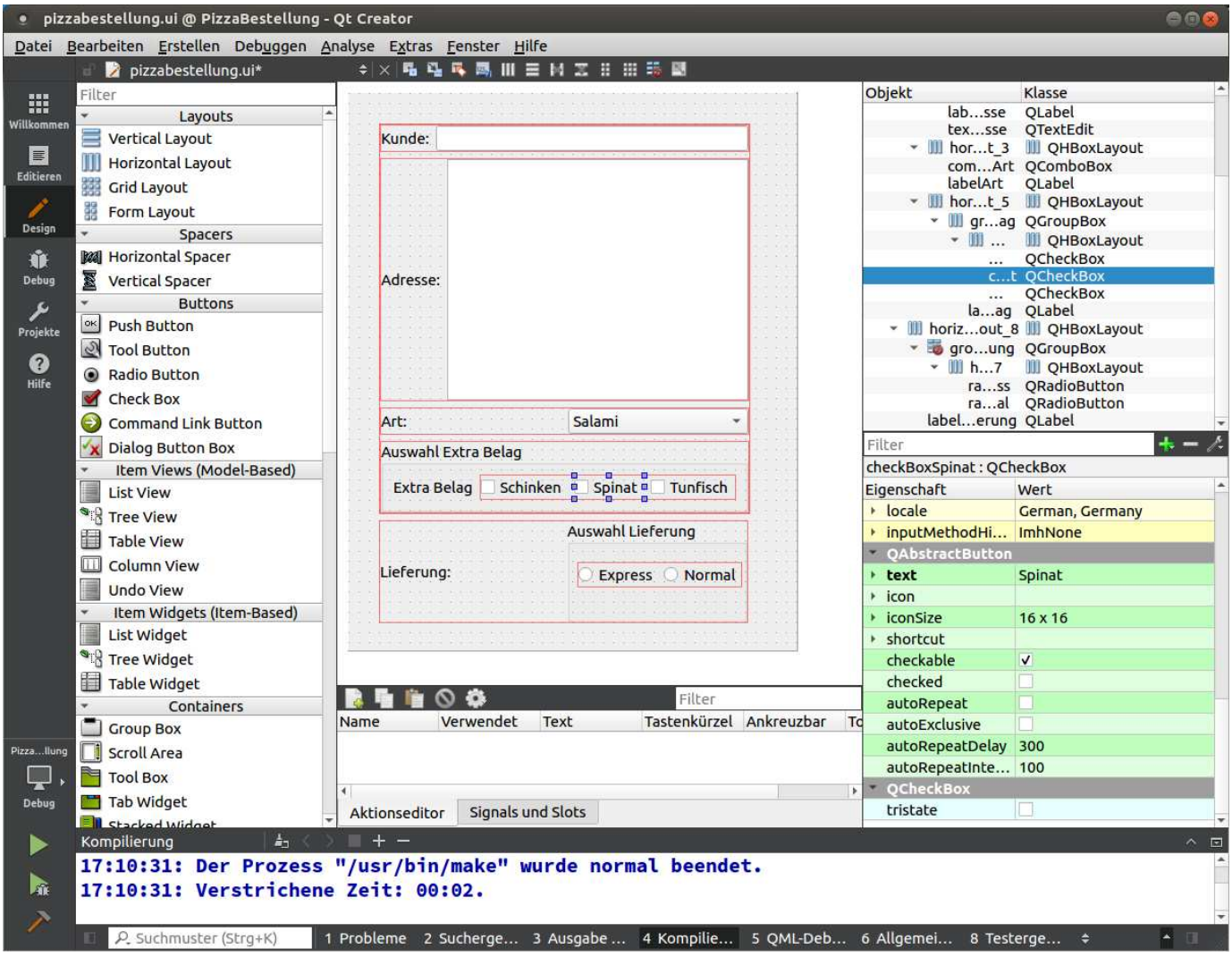
- Qt besteht aus über **1500 Klassen** die neben GUIs auch eine beachtliche Schnittmege zur C++ Standard-Bibliothek beinhalten (z.B.: `QString`, `QFile`).
- Mit dem **Qt Creator** wird eine eigene IDE zur Verfügung gestellt, mit der Qt-Anwendungen entwickelt werden können.
- Das Programm **Qt Assistant** (Aufruf mit `assistant`) ist ein umfangreiches Hilfe-System für Qt, das im Qt Creator mit integriert ist.



- Zusätzlich steht unter `https://doc.qt.io/qt-5/reference-overview.html` eine sehr ausführliche Online-Dokumentation mit vielen Beispielen zur Verfügung.

Bestandteile von Qt 2

- Das Programm **Qt Linguist** unterstützt die Anpassung von Anwendungen, die international nutzbar sein sollen.
- Das Programm **Qt Designer** ist ein CAD-ähnliches Werkzeug, mit dessen Hilfe GUIs mittels Drag-and-Drop erstellt werden können.



Es wird dabei eine **Formulardatei** mit der Endung `.ui` erstellt, die den Aufbau des GUIs im XML-Format speichert.

Bei der Erstellung des Projektes erzeugt der **UI-Compiler (User Interface Compiler, UIC)** aus der Formulardatei eine Header-Datei mit dem Namen `ui_formularDeiteiName.h`.

12.3.1. Wichtige Qt Module (Qt Essentials)

1. **Qt Core**:
   Beinhaltet Kern-Klassen ohne GUI-Funktionslität, die von allen anderen Modulen verwendet werden,

   z.B.: `QObject`, `QByteArray`, `QDate`, `QFile`, `QList`, `QPoint`, `QSize`, `QString`.

2. **Qt Gui**:
   Beinhaltet die wichtigsten Klassen zum Zeichnen von GUIs.

   z.B.: `QBrush`, `QColor`, `QFont`, `QPainter`, `QPen`, `QPolygon`, `QPolygonF`,

3. **Qt Widgets**:
   Beinhaltet die Klassen für die wichtigsten Steuerelemente (Widgets).

   z.B.: `QAbstractButton`, `QAction`, `QCheckBox`, `QComboBox`, `QGridLayout`, `QLabel`, `QLineEdit`, `QPushButton`, `QRadioButton`, `QWidget`.

4. **Qt Multimedia** und **Qt Mulimedia Widgets**:
   Beinhaltet Klassen für Audio-, Video-, Radio-, und Kamera-Funktionalitäten auf einer niedrigeren Ebene.

5. **Qt Network**:
   Klassen für die Netzwerk-Programmierung.

6. **Qt SQL**:
   Klassen für die Datenbankintegration.

7. **Qt Test**:
   Klassen zum Testen von Qt-Programmen.

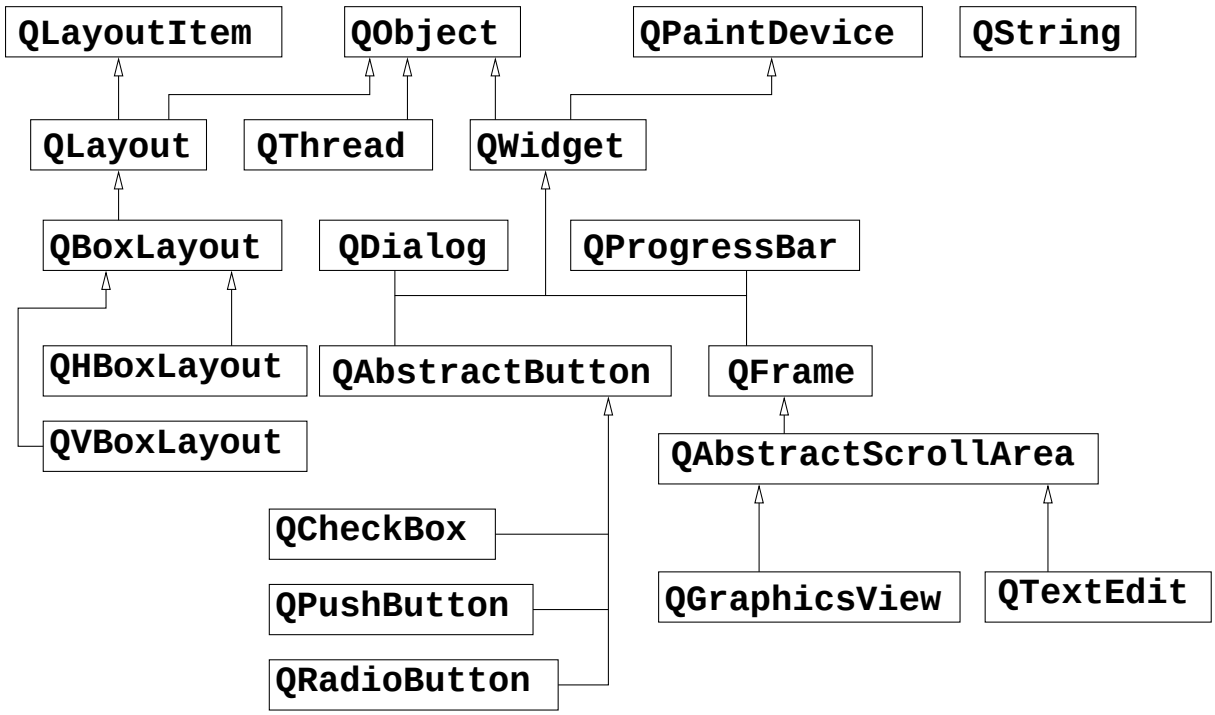
Wichtige Qt Module (Qt Essentials) 2

- Diese Module sind für alle gängigen Plattformen verfügbar.
- Sie bilden das Fundament des Qt Frameworks.
- Weiterhin gehören noch zu den Qt Essentials die Module **Qt QML** und **Qt Quick**, die es auch mit anderen Sprachen (XML bzw. JavaScript) als C++ ermöglichen, Qt-Programme zu entwickeln.
- Darauf aufbauend gibt es eine Fülle von sogenannten **Qt Add Ons** die für Spezialanwendungen gedacht sind und möglicherweise nicht auf allen Plattformen verfügbar sind.

  z.B.: **Active Qt**, **Qt 3d**, **Qt Concurrent**, **Qt Sensors**, **Qt WebEngine**, **Qt WebSockets**, **Qt XML**, **Qt XML Patterns**.
- Im Qt Assistenten kann man sich alle Qt Module anzeigen lassen, indem man im Index nach *All Modules* sucht.

12.3.2. Klassenhierarchie von Qt

- Das Qt Framework basiert auf einer **Vererbungs-Hierarchie**.
- Im Qt Assistenten kann man sich über diese Vererbungs-Hierarchie informieren, indem man nach *Inheritance-Hierarchy* sucht.
- Die Klasse **QObject** ist die Basisklasse von allen Klassen, die das Signal-Slot-Konzept und die automatische Speicherverwaltung eines Objektbaumes durch den Wurzel-Vater verwenden.
- Klassen die nicht das Signal-Slot-Konzept und die automatische Speicherverwaltung verwenden, sind nicht von **QObject** abgeleitet, so z.B. die Klasse **QString**.
- Alle üblichen Steuerelemente erben von der Klasse **QWidget**.



12.4. Objektbäume und Besitz

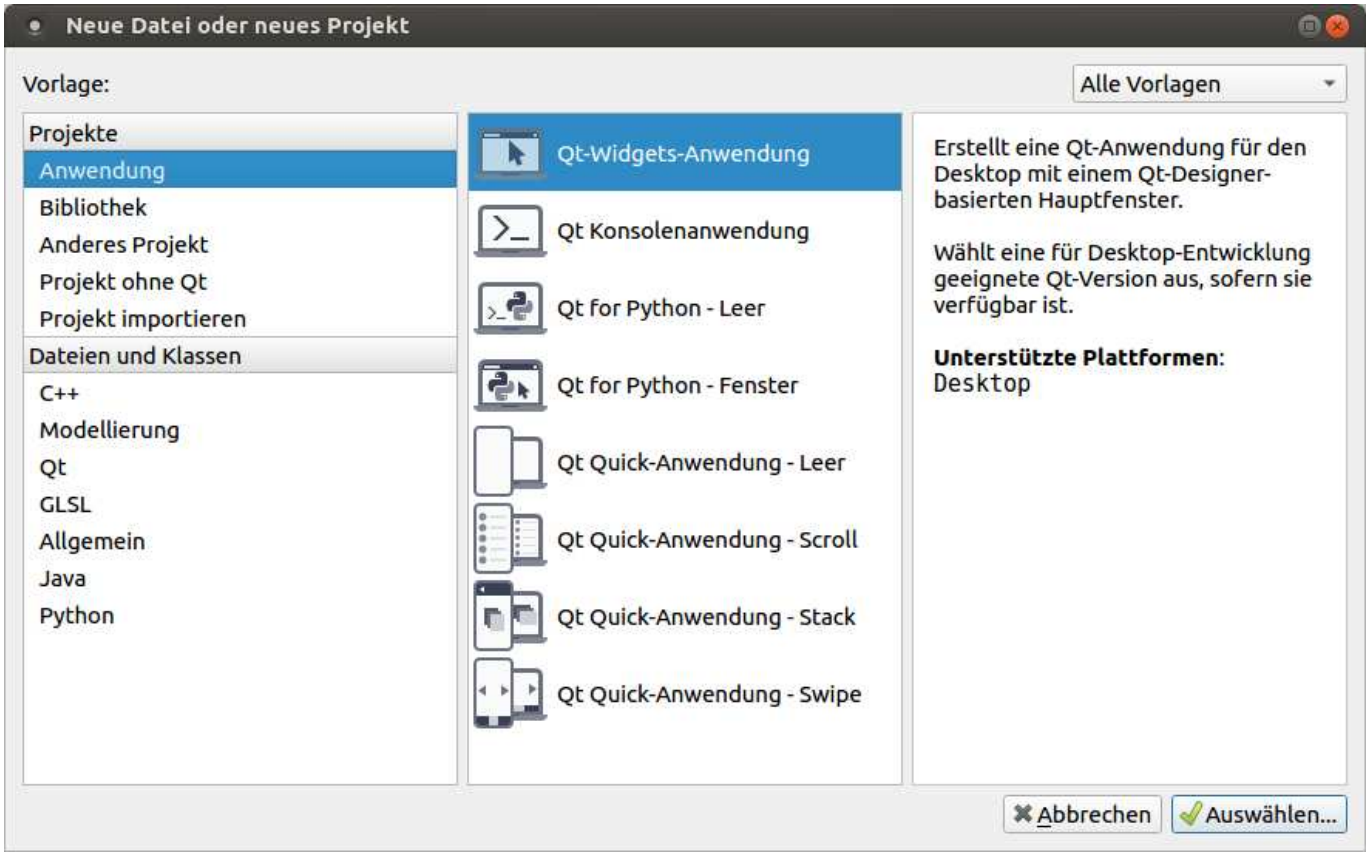
- Die Basisklasse **QObject** hat die wichtige Eigenschaft, dass jede ihrer Instanzen einen **Vater** (`parent`) und eine **Liste von Kindern** (`children`) hat.
- Wird ein Objekt vom Typ `QObject` erstellt mit der Angabe eines Vaters im Parameter des Konstruktors, so wird dieses Objekt automatisch zu der Liste der Kinder des Vaters hinzugefügt.
- Dadurch ist es möglich einen ganzen Objekt-Baum zu erstellen.
- Insbesondere können auf jedes `QWidget`-Objekt andere `QWidget`-Objekte als Kinder abgelegt werden.
- Dabei werden **Kinder und Vater in Qt immer als Zeiger übergeben** und nie als Wertparameter.
- Genau dies ermöglicht eine dynamische Polymorphie und die Anwendung des **Composite-Entwurfs-Musters**.
- Muss das Vater-Objekt neu gezeichet werden (z.B. weil es verschoben wurde oder wieder sichtbar ist) leitet es automatisch auch das neue Zeichnen für alle seine Kinder ein.
- Weiterhin ist eine automatische Speicherverwaltung für die Kinder eingerichtet:

  Wird ein Vater-Objekt per Destruktor-Aufruf zerstört, dann:

  - Löscht der Vater automatisch alle seine Kinder.
  - Benachrichtigt er seinen Vater, dass dieser ihn aus dessen Liste der Kinder entfernt.
  - Abschließend er sich selber zerstört.

12.5. Erstellung eines GUIs mit Qt Creator

1. Im Menue `Datei->Neu` wählt man unter `Anwendung` den Typ `Qt-Widgets-Anwendung` aus.



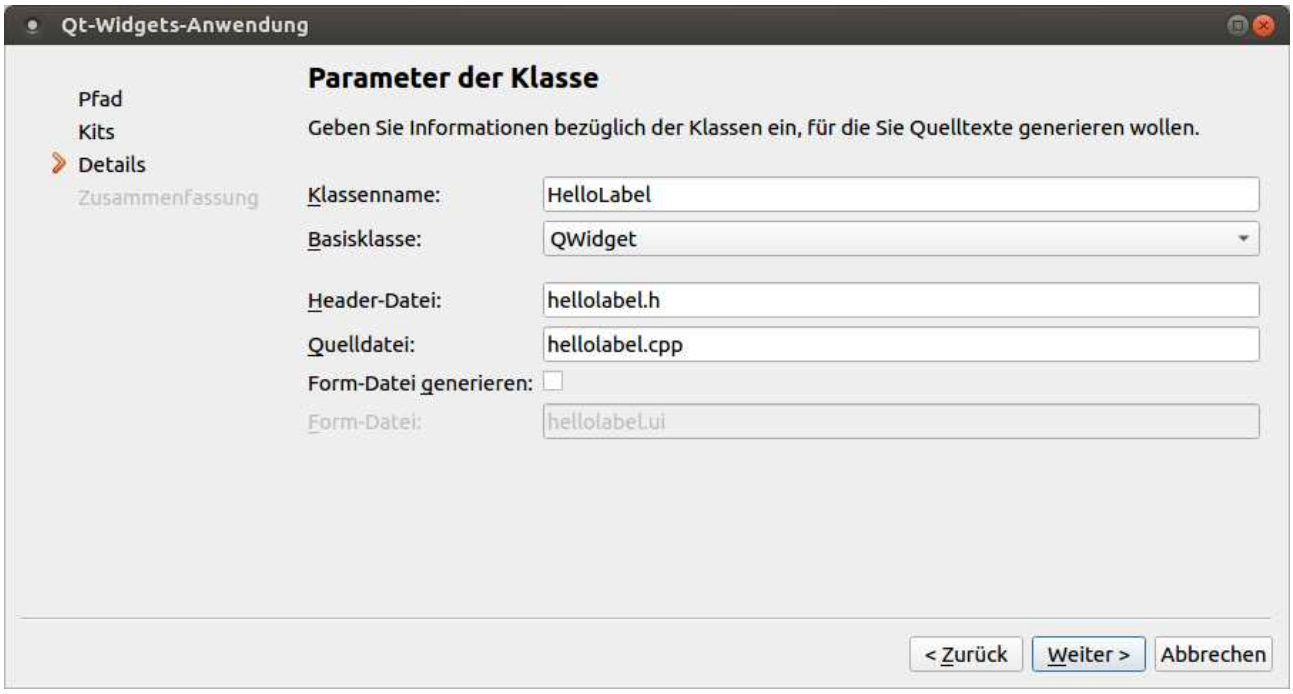
Erstellung eines GUIs mit Qt Creator 2

2. Im Dialog `Parameter der Klasse` legt man den Klassennamen fest.

   Weiterhin kann man auswählen, von welcher Basisklasse geerbt werden soll:

   – `QWidget`: Für sehr einfache GUIs.
   – `QDialog`: GUI zur Information oder für kurze Anwendereingaben.
   – `QMainWindow`: Hauptfenster mit Menue-Struktur, Toolbar, Dock, Zentralbereich und Statusbar (s.u).

   Da wir als *Programmier-Profis* nicht den Qt Designer (Kindergarten-Programmierung für Designer) verwenden, wird der Haken bei `Form-Datei generieren:` entfernt.



12.5.1. Eigene GUI Klasse

Klasse die von einer Klasse, die `QWidget` als Basisklasse besitzt, abgeleitet wird, hier z.B. von `QLabel` und das Makro `Q_OBJECT` enthält.

Header-Datei:

```
#ifndef HELLOLABEL_H
#define HELLOLABEL_H
#include <QLabel>

class HelloLabel : public QLabel {
  Q_OBJECT

 public:
  HelloLabel(QWidget* parent = 0);
  ~HelloLabel();
};

#endif  // HELLOLABEL_H
```

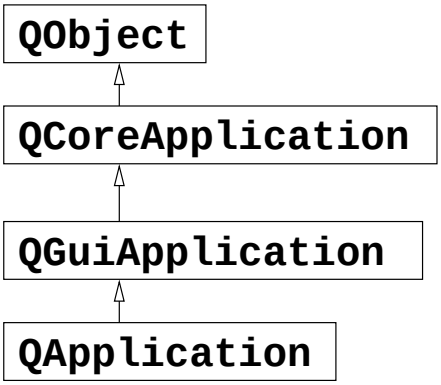
cpp-Datei:

```
#include "hellolabel.h"

HelloLabel::HelloLabel(QWidget* parent) : QLabel(parent) {
  setText("Hello World");
  setMinimumSize(300, 100);
  move(40, 20);
}

HelloLabel::~HelloLabel() {}
```

12.5.2. Hauptprogramm mit Eventloop-Start

```
#include <QApplication>
#include "hellolabel.h"

int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  HelloLabel w;
  w.show();

  return a.exec();
}
```

- Für jede Qt-Anwendung mit einem GUI existiert **genau ein** `QApplication`-Objekt.



- `qApp` ist ein Zeiger auf eine in `<QApplication>` deklarierte globale Variable, die auf diese eindeutige Instanz zeigt.
- Mit der `exec()` Methode wird die **Eventloop** gestartet, um in einer Endlos-Schleife auf den Eintritt von Ereignissen zu warten und (ggf. darauf zu reagieren).
- Diese Endlos-Schleife bricht erst ab, wenn das letzte grafische Steuerelement geschlossen wird.

12.6. Signals und Slots

- Ein **Signal** entspricht einem Ereignis z.B. `clicked()` und ist einem Sender (z.B. `QPushButtom`) zugeordnet.
- Ein **Slot** ist eine Callback-Funktion, die dem Empfänger eines Signals zugeordent ist. Um auf das Signal zu reagieren wird der Code in dieser Funktion ausgeführt.
- In Qt können allgemeiner Signale und Slots für die Kommunikation zwischen Steuerelementen eingesetzt werden.
- Signale und Slots müssen miteinander verbunden werden (s.u.).
- Der Vorteil des Signal-Slot Konzeptes besteht darin, dass Qt diese Verbindung automatisch trennt, wenn eines der kommunizierenden Objekte zerstört wird.
- Eigene Signale und Slots können nur in Klassen definiert werden, die von der Klasse `QObject` erben.
- Weiterhin muss in der Klasse am Anfang der Klassendeklaration das Makro `Q_OBJECT` gesetzt werden.
- Um das Signal-Slot Konzept zu verwenden, erweitert Qt C++ um ein sprachliches Konzept, das **Meta-Objectsystem** genannt wird.
- Schlüsselwörter für dieses System sind:

  ```
  signals, slots, Q_OBJECT,
  emit, SIGNAL, SLOT
  ```

- Dabei wird der Code zuerst mit dem **Meta Object Compiler (MOC)** übersetzt, der Dateien mit dem Präfix `moc_` erzeugt. Diese werden dann an den C++ Compiler weitergeleitet.

12.6.1. Signal mit Slot verbinden

Qt 4.8 Variante mit der statischen Methode connect():

```
QMetaObject::Connection QObject::connect(
  const QObject *sender,
  PointerToMemberFunction signal,
  const QObject *receiver,
  PointerToMemberFunction method,
  Qt::ConnectionType type = Qt::AutoConnection)
```

Bsp.:

```
connect(
  buttonQuit,
  &QPushButton::clicked,
  this,
  &QWidget::close);
```

- `signal` muss in der Klassendeklaration unter `signals:` deklariert sein.
- Die `slot` Funktion kann eine beliebige Member-Funktion sein, sie muss aber genau so viele Parameter wie das `signal` besitzen und die Parameter-Typen müssen zueinander passen bzw. müssen ineinander überführbar sein.
- Beim default ConnectionType `Qt::AutoConnection` wird das `signal` in einer Warteschlange zwischengespeichert.
- Die `connect()` Funktion liefert ein Handle zurück, über das die Verbindung wieder per Programm aufgehoben werden kann.

Signal mit Slot verbinden 2

Traditionelle Variante mit Makros und mit der statischen Methode connect():

```
QMetaObject::Connection QObject::connect(
  const QObject *sender,
  const char *signal,
  const QObject *receiver,
  const char *method,
  Qt::ConnectionType type = Qt::AutoConnection)
```

Bsp.:

```
connect(
  buttonBerechnen,
  SIGNAL(clicked()),
  this,
  SLOT(berechnen()));
```

- Die Signal-Funktion und die Slot-Funktion werden durch Methodendeklarationen angegeben.
- Diese Deklarationen müssen zwingend in die Makros `SIGNAL` bzw. `SLOT` eingepackt werden.
- Die Makros wandeln diese Deklarationen in einen `const char*` String um.
- `signal` muss in der Klassendeklaration unter `signals:` deklariert sein.
- Auch der `slot` muss in der Klassendeklaration unter `slots:` deklariert sein.
- Die `slot` Funktion darf bei dieser Variante optionale Parameter besitzen.

Bsp. MWS Gui

```
// mywidget.h
#ifndef MYWIDGET_H
#define MYWIDGET_H
#include <QWidget>
#include <QtWidgets/QLabel>
#include <QtWidgets/QLineEdit>
#include <QtWidgets/QPushButton>
class MyWidget : public QWidget {
  Q_OBJECT
 private:
  QLabel* labelPreis;
  QLabel* labelMws;
  QLineEdit* editPreis;
  QLineEdit* editMws;
  QPushButton* buttonBerechnen;
  QPushButton* buttonQuit;
 public:
  MyWidget(QWidget* parent = 0);
  ~MyWidget();

 public slots:
  void berechnen();
};
#endif  // MYWIDGET_H
```

Bsp. MWS Gui 2

```
// mwsGuiMain.cpp
#include "mywidget.h"

MyWidget::MyWidget(QWidget *parent) : QWidget(parent) {
  setWindowTitle("MWS Gui");
  resize(300, 350);
  move(100, 100);

  // Steuerelemente werden Pixelgenau abgelegt
  // dies sollte man eigentlich NIE MACHEN!!! (Warum?)
  labelPreis = new QLabel("Preis:", this);
  labelPreis->resize(100, 50);
  labelPreis->move(50, 50);

  labelMws = new QLabel("Mws:", this);
  labelMws->resize(100, 50);
  labelMws->move(50, 150);

  editPreis = new QLineEdit("", this);
  editPreis->resize(100, 50);
  editPreis->move(170, 50);

  editMws = new QLineEdit("              ", this);
  editMws->resize(100, 50);
  editMws->move(170, 150);

  buttonBerechnen = new QPushButton("Berechnen", this);
  buttonBerechnen->resize(100, 50);
  buttonBerechnen->move(50, 250);

  buttonQuit = new QPushButton("Quit", this);
  buttonQuit->resize(100, 50);
  buttonQuit->move(170, 250);
```

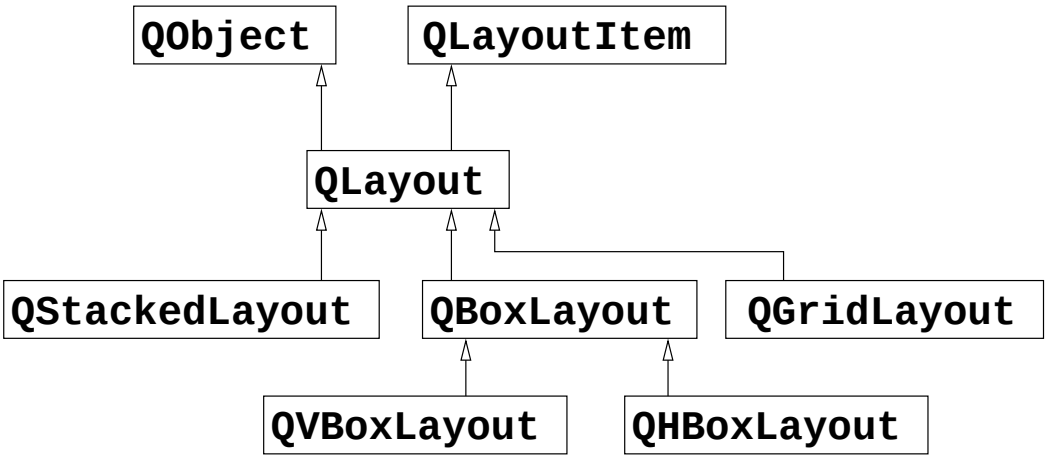
Bsp. MWS Gui 3

```
  // Signals und Slots verbinden
  connect(buttonBerechnen, SIGNAL(clicked()), this,
    SLOT(berechnen()));

  // Signals und Solts verbinden im neuen Format
  connect(buttonQuit, &QPushButton::clicked, this,
    &QWidget::close);
}

MyWidget::~MyWidget() {}
void MyWidget::berechnen() {
  double preis = editPreis->text().toDouble();
  double mws = preis * 0.19;
  editMws->setText(QString(QString::number(mws, 'f', 2)));
}
```

Hauptprogramm:

```
// mwsGuiMain.cpp
#include <QApplication>
#include "mywidget.h"

int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  MyWidget w;
  w.show();

  return a.exec();
}
```

12.7. Layouts



- Da der Benutzer die Größe eines GUIs verändern kann, ist es nicht ratsam die Größe der Steuerelemente und ihre Position (Koordinate der linken oberen Ecke) Pixel-genau anzugeben.
- Durch die Verwendung von Layouts werden die Steuerelemente automatisch relativ zueinander positioniert und in einer passenden Größe angezeigt.
- Durch das Verschachteln von Layouts können kompliziertere Layouts realisiert werden.
- **QVBoxLayout** ordnet die Steuerelemente in einer vertikalen Spalte von oben nach unten an.
- **QHBoxLayout** ordnet die Steuerelmente in einer horizontalen Zeile von links nach rechts an.
- **QGridLayout** ordnet die Steuerelemente in einer Matrix an.
- **QStackedLayout** stapelt die Steuerelemente untereinander.

Bsp. MWS Gui mit GridLayout

```
// mwsguimitlayout.cpp
#include "mwsguimitlayout.h"
#include <QGridLayout>

MWSGuiMitLayout::MWSGuiMitLayout(QWidget* parent)
  : QWidget(parent) {
  setWindowTitle("MWS Gui mit GridLayout");
  resize(300, 350);
  move(100, 100);

  // Steuerelemente werden in ein GridLayout
  // abgelegt

  labelPreis = new QLabel("Preis:", this);
  labelMws = new QLabel("Mws:", this);
  editPreis = new QLineEdit("", this);
  editMws = new QLineEdit("", this);
  buttonBerechnen = new QPushButton("Berechnen", this);
  buttonQuit = new QPushButton("Quit", this);

  QGridLayout* grid = new QGridLayout(this);
  grid->addWidget(labelPreis, 0, 0);
  grid->addWidget(editPreis, 0, 1);
  grid->addWidget(labelMws, 1, 0);
  grid->addWidget(editMws, 1, 1);
  grid->addWidget(buttonBerechnen, 2, 0);
  grid->addWidget(buttonQuit, 2, 1);
  setLayout(grid);

  // Signals und Slots verbinden
  connect(buttonBerechnen, SIGNAL(clicked()), this,
          SLOT(berechnen()));

  // Signals und Solts verbinden im neuen Format
  connect(buttonQuit, &QPushButton::clicked, this,
          &QWidget::close);
}
```

Aufgabe Clicks Zählen

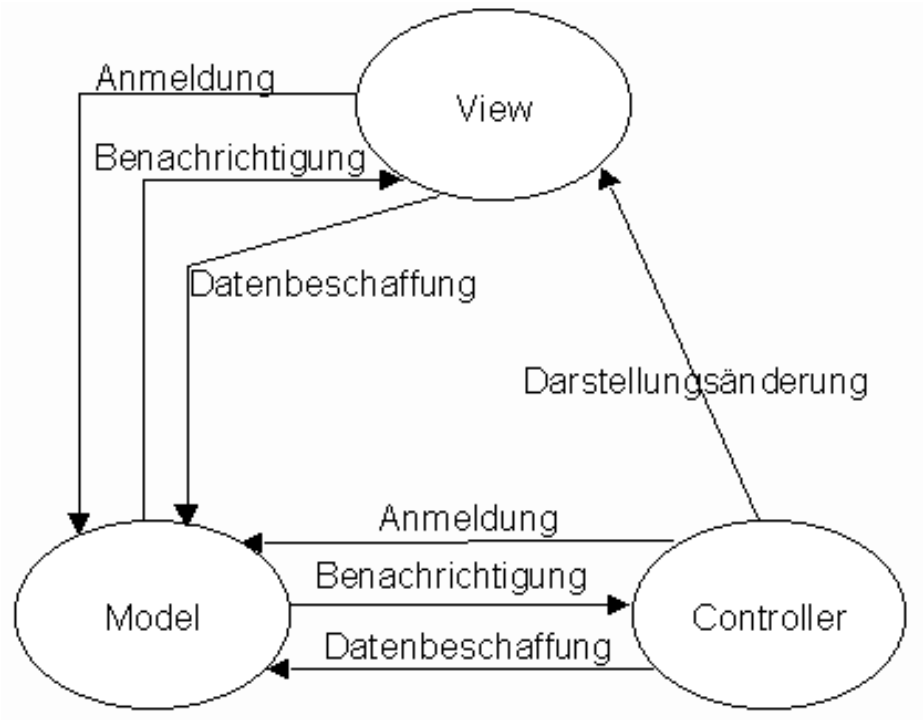
- Man erstelle das oben abgebildete GUI indem man eine eigene GUI-Klasse `MyWidget` erstellt, die von `QWidget` erbt.
- Die Steuerelemente sollen mit einem `QGridLayout` angeordnet werden.
- Um ein Steuerelement über mehrere Spalten (wie hier im Fall des Quit-Buttons) oder über meherer Zeilen auszudehnen gibt es in der Klasse `QGridLayout` die Methode

```
void QGridLayout::addWidget(
  QWidget* widget,
  int fromRow,
  int fromColumn,
  int rowSpan,
  int columnSpan )
```

  Hierbei geben die Parameter `fromRow` und `fromColumn` die Anfangsposition des abgelegten Widgets im Gridlayout an.

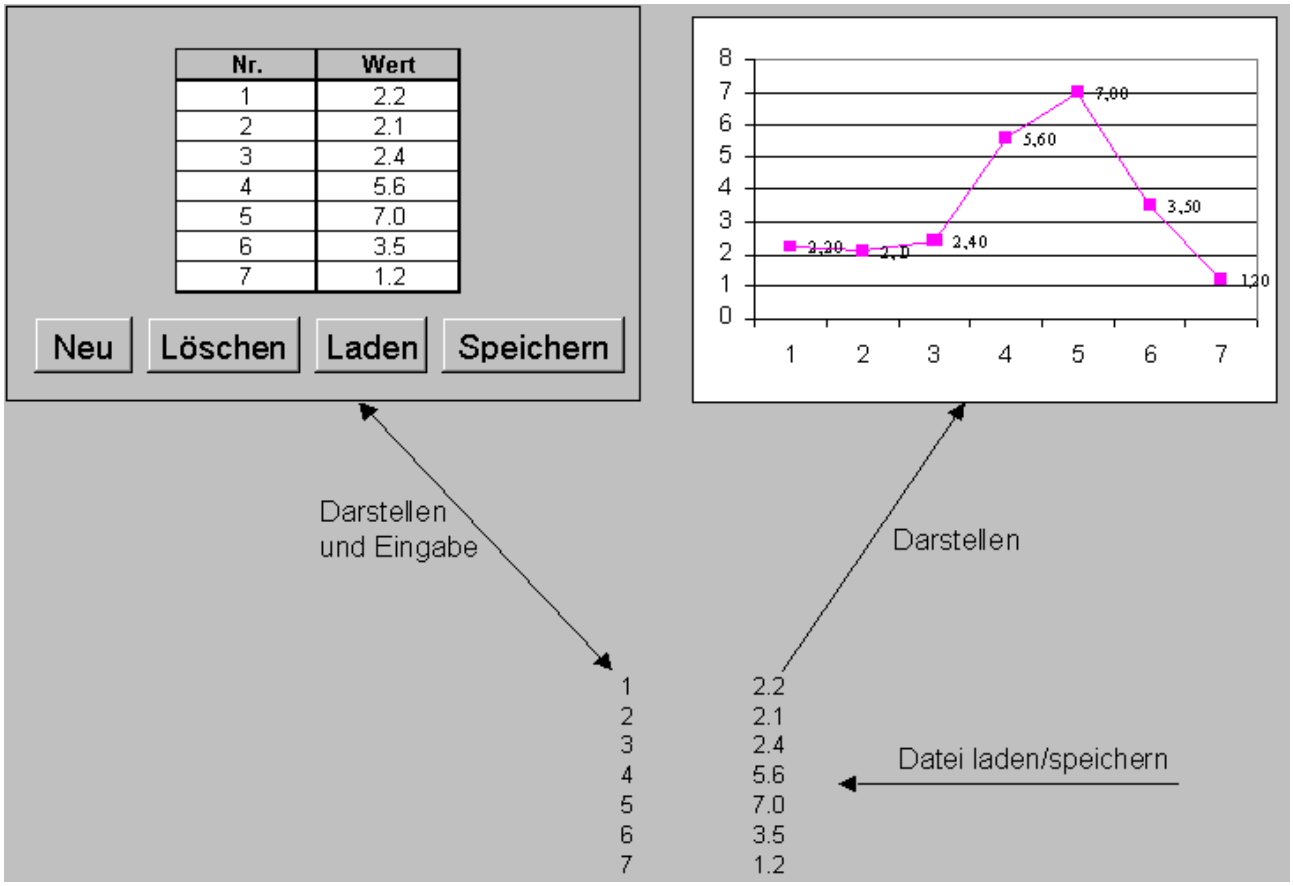
  Die Parameter `rowSpan` bzw. `columnSpan` bestimmen über wie viele Zeilen bzw. Spalten sich das Widget ersteckt.
- Die Anzahl Clicks auf den `QPushButton` sollen gezählt werden und im `QLineEdit` Textfeld angezeigt werden.
- Ein Click auf den Quit-Button soll das GUI beenden.

12.8. Model-View-Control (MVC) Muster



- Das **Model-View-Control (MVC) Entwurfs-Muster** dient allgemein zum Entwurf von interaktiven GUI-Programmen.
- Das MVC-Muster trennt die Datenhaltung, die Präsentation und die Steuerung voneinander:
  - Das **Model** enthält die Daten einer Anwendung und speichert ihren Zustand. Weiterhin ist es verantwortlich für alle Operationen, die auf den Daten ausgeführt werden.
  - Der **View** ist für die graphische Präsentation der Daten verantwortlich. Ein View ist immer genau mit einem Model verbunden. Andererseits kann aber ein Model von mehreren verschiedenen Views dargestellt werden.
  - Der **Controler** (Steuerung) verbindet das Model mit dem Anwender. Es nimmt Benutzereingaben entgegen und stößt die entsprecheden Maßnahmen zur Änderung von Model und View an.

Bsp. Model-View-Control Muster



- Bsp. Eingabe und Anzeige von Messwerten:
- Dem **Modell** entspricht hier eine Tabelle (Datenbank) zur Datenhaltung.
- Der **View** stellt die Daten als xy-Grafik dar.
- Der **Controler** besteht aus dem Eingabe-Formular, welches Benutzereingaben entgegen nimmt und an das Model bzw. an den View weiterleitet.
- Vorteile:
  - Die Anwendung ist leicht zu ändern und zu erweitern.
  - Die Erstellungs-Aufgabe kann auf 3 unterschiedliche Experten (Grafiker/Designer, DB-Experte, Programmierer für Buisiness-Logik im Controler) verteilt werden.

12.9. Zeichnen mit Qt

- Für das Zeichnen von einer großen Anzahl von geometrischen Formen (Linien, Punkte, Polygone, Rechtecke, Kreise etc.) auf niedriger Ebene stellt Qt die Klasse `QPainter` zur Verfügung.

- Mit `QPainter`-Objekten kann man auf einer **Zeichenfläche** zeichnen.

- Als Zeichenfläche dienen Objekte vom Typ

  `QPainterDevice`

  oder die davon direkt abgeleiteten Klassen:

  - `QPixMap`
  - `QPicture`
  - `QImage`
  - `QPrinter`
  - `QWidget`
  - `QGLPixelBuffer`
  - `QGLFrameBufferObject`

- Da insbesondere alle Steuerelemente von `QWidget` abgeleitet sind, kann auch jedes Steuerelement als Zeichenfläche verwendet werden.

- Dadurch kann man auch neue eigene Steuerelemente mit einem neu gestalteten Aussehen erstellen.

12.9.1. Vorgehensweise beim Zeichnen mit QPainter

1. Man erstellt eine eigene Klasse, die von einer **Zeichenflächen-Klasse**, z.B. `QWidget` oder einer anderen Steuerelement-Klasse erbt.

2. In dieser eigenen Klasse überschreibt man die Ereignishandler-Methode

   `paintEvent(QPaintEvent* event)`.

   Diese Methode wird immer dann aufgerufen, wenn eine Zeichenfläche (bzw. Steuerelement) neu gezeichnet werden muss, z.B. wenn sie wieder sichtbar wird oder sich die Inhalte geändert haben oder die Größe der Zeichenfläche verändert wurde.
   Im Programm kann mittels `QWidget::update()` der Ereignishandler veranlasst werden die Methode `paintEvent()` aufzurufen.

3. In der überschriebenen `paintEvent()` Methode erstellt man sich ein `QPainter`-Objekt.

4. Man nnimmt nun für dieses `QPainter`-Objekt Einstellungen vor, insbesondere für: den **Stift (QPen)**, den **Pinsel (QBrush)**, und die **Schrift (QFont)**.

5. Man kann nun mit den `draw...()`-Methoden von `QPainter` verschiedene Formen zeichnen.

12.9.2. Stift, Pinsel, Schrift

Ein `QPainter` Objekt besitzt unter anderem die folgenden Zeichengeräte:

- Den **Stift (QPen)**, der für das Zeichnen von Linien, Umrissen und Formen verwendet wird.

  Für Stifte kann z.B. die Farbe, die Stiftbreite und ein Linienstil eingestellt werden.

- Den **Pinsel (QBrush)**,
  der zum Füllen von Flächen verwendet wird.

  Für ihn kann z.B. eine Farbe und ein Füllmuster oder ein Farbverlauf eingestellt werden.

- Die **Schrift (QFont)**
  wird zum Ablegen von Text verwendet.

  Es können z.B. die Schriftfamilie und die Schriftgöße eingestellt werden.

12.9.3. Koordinatensysteme

1. Das **physische Koordinatensystem** bezeichnet das Koordinatensystem der **Zeichenfläche**:
   - Die Koordinaten werden in **Pixel-Einheiten** angegeben.
   - Der **Ursprung** befindet sich in der **linken oberen Ecke** der Zeichenfläche.
   - Die x-Achse verläuft nach rechts, aber die y-**Achse nach unten**.
   - Das letzte darstellbare Pixel in der rechten unteren Ecke hat die Koordinaten `(width()-1, height()-1)`.
   - Durch einen **Viewport** kann die Zeichenfläche weiter eingeschränkt werden.

Koordinatensysteme 2

2. Das **logische Koordinatensystem** bezeichnet das Koordinatensystem das das `QPainter`-Objekt verwendet, es wird oft auch als **Fenster (Window)** bezeichnet.

   – Defaultmäßig stimmt es mit dem pyhsikalischen Koordinatensystem der Zeichenfläche überein.

   – Um beim Programmäßigen Zeichnen von der Größe der Zeichenfläche und der Auflösung unabhängig zu sein kann man es aber umdefinieren.

   – Z.B. mit der Anweisung

   ```
   painterObject.setWindow(0, 0, 10, 10);
   ```

   hat nun die linke obere Ecke der Zeichenfläche die logischen Koordinaten $(0, 0)$, und die rechte untere Ecke der Zeichenfläche hat die logischen Koordinaten $(10, 10)$.

   – Mit der Anweisung

   ```
   painterObject.translate(0,10)
   ```

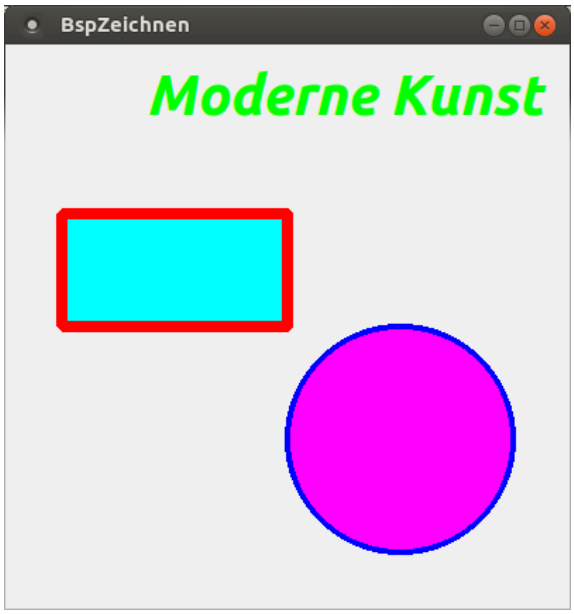
   wird der Ursprung des logischen Koordinatensystems in die linke untere Ecke verschoben.

   – Nach der Ausführung der Anweisung

   ```
   painterObject.scale(1, -1)
   ```

   verläuft die y-Achse des logischen Koordinatensystems nun nach oben.

Bsp. Zeichnen

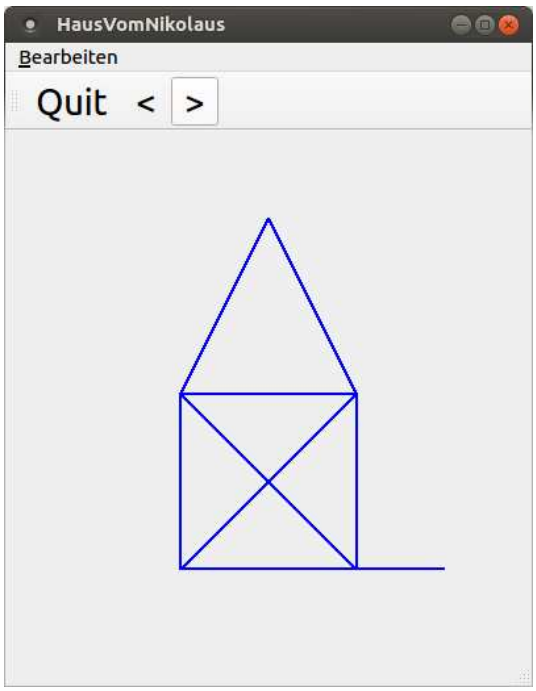



```
// zeichenflaeche.cpp
#ifndef ZEICHENFLAECHE_H
#define ZEICHENFLAECHE_H

#include <QWidget>

class Zeichenflaeche : public QWidget {
  Q_OBJECT

 public:
  Zeichenflaeche(QWidget* parent = 0);
  ~Zeichenflaeche();
  void paintEvent(QPaintEvent* event);
};

#endif  // ZEICHENFLAECHE_H
```

Bsp. Zeichnen 2

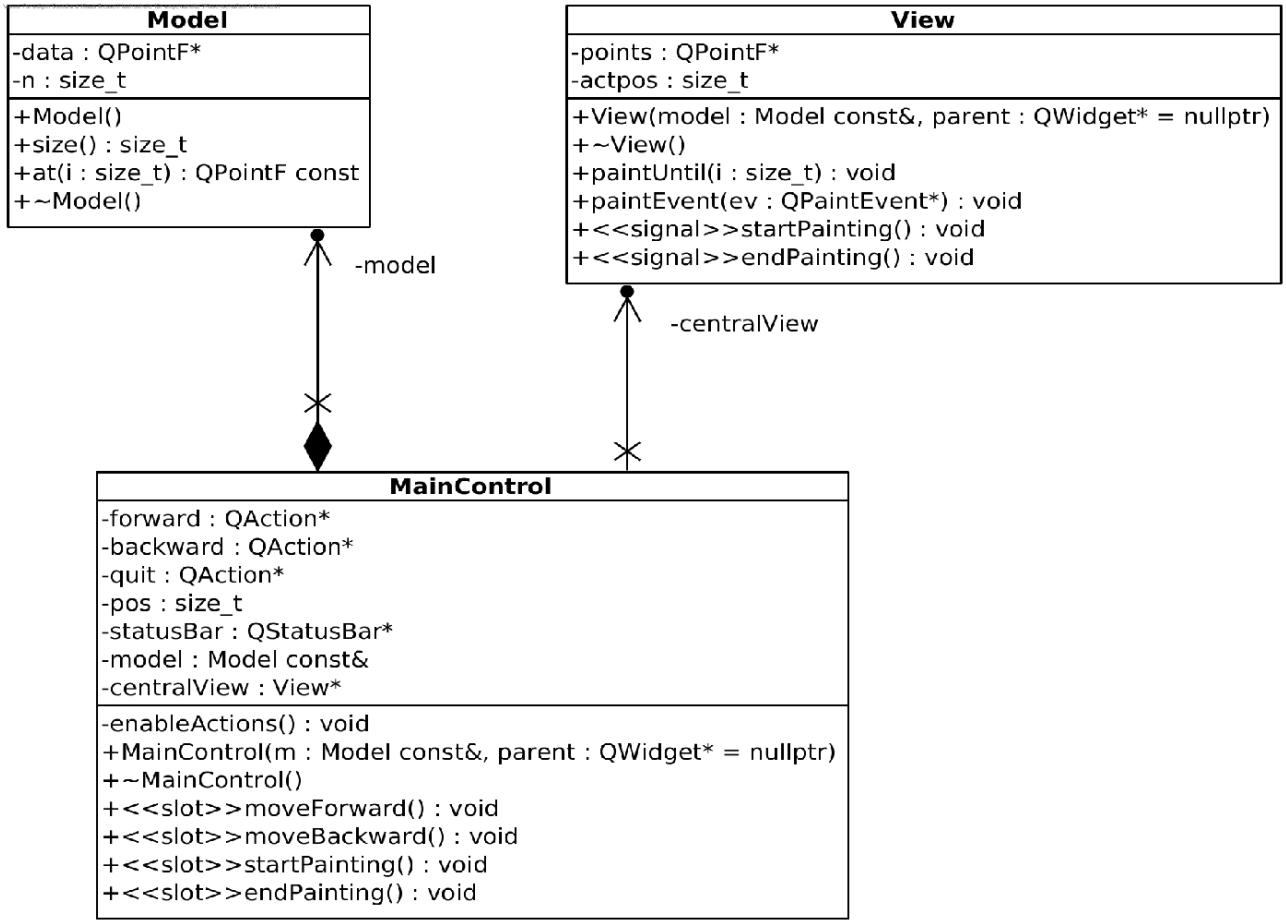
```
// zeichenflaeche.cpp
#include "zeichenflaeche.h"
#include <QBrush>
#include <QFont>
#include <QPainter>
#include <QPen>
Zeichenflaeche::Zeichenflaeche(QWidget* parent)
  : QWidget(parent) {
  resize(400, 400);
  move(100, 100);
}
Zeichenflaeche::~Zeichenflaeche() {}
void Zeichenflaeche::paintEvent(QPaintEvent* event) {
  QWidget::paintEvent(event);
  QPainter painter(this);

  QFont font = painter.font();
  font.setItalic(true);
  font.setBold(true);
  font.setPointSize(30);
  painter.setFont(font);
  painter.setPen(QPen(Qt::green, 2));
  painter.drawText(100, 50, tr("Moderne Kunst"));

  // logisches Koordinatensystem setzen
  painter.setWindow(0, 0, 10, 10);
  painter.translate(0, 10);
  painter.scale(1, -1);

  painter.setPen(QPen(Qt::red, 0.2));
  painter.setBrush(QBrush(Qt::cyan, Qt::SolidPattern));
  painter.drawRect(1, 5, 4, 2);

  painter.setPen(QPen(Qt::blue, 0.1));
  painter.setBrush(QBrush(Qt::magenta, Qt::SolidPattern));
  painter.drawEllipse(5, 1, 4, 4);
}
```

Aufgabe Haus vom Nikolaus



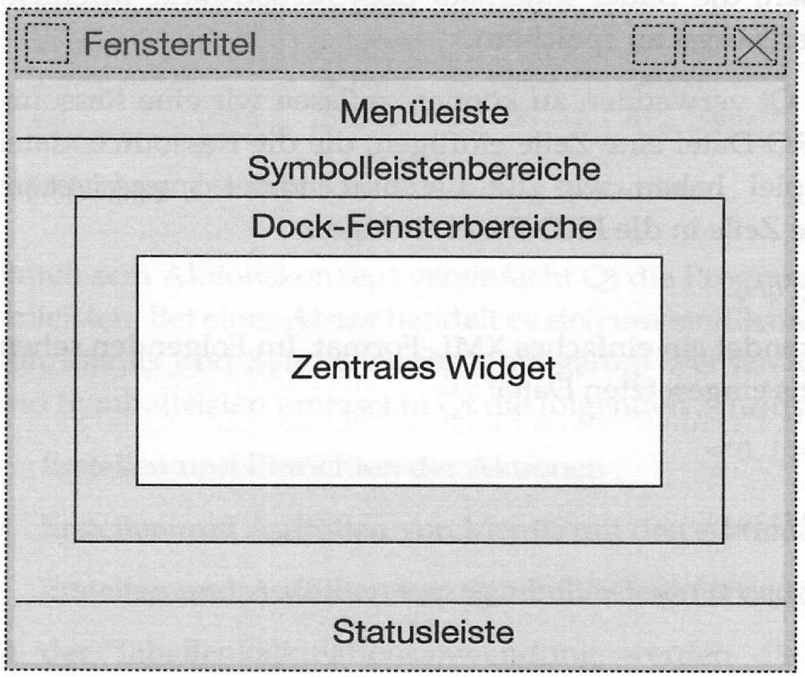
- In der bereitgestellten Klasse `Model` sind die Daten für das Zeichnen des Hauses vom Nikolaus gespeichert.
- Genauer: es werden die Koordinaten der Punkte gespeichert, die durch einen Linienzug (Polygonzug) zu verbinden sind.
- Die Angabe der Koordinaten bezieht sich auf ein logisches Koordinatensystem für den Bereich $[0,\ 6] \times [0,\ 6]$ mit Ursprung in der linken unteren Ecke und einer y-Achse die nach oben verläuft.
- Wie aus der Kinderzeit bekannt ist es das Ziel, das Haus in einem Linienzug aus genau acht Strecken zu zeichnen, ohne eine Strecke zweimal zu durchlaufen. Begleitet wird das Zeichnen mit dem simultan gesprochenen Reim aus acht Silben: *Das ist das Haus vom Ni-ko-laus*.
- Man erstelle eine Klasse `View` die von der Klasse `QWidget` abgeleitet ist und unter der Verwendung der Daten in der Klasse `Model` das Haus vom Nikolaus zeichnet.

Aufgabe Haus vom Nikolaus 2



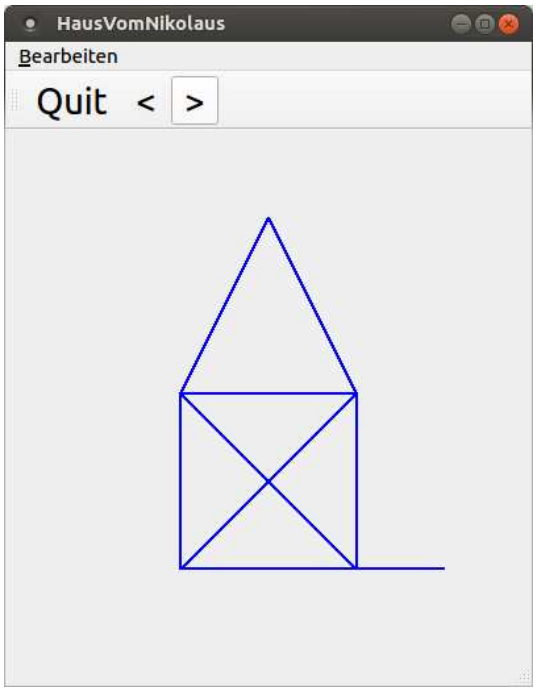
- Die Methode `paintUntil(size_t i)` zeichnet das Haus vom Nikolaus bis einschließlich zur Linie `i`.
- In dieser Methode wird vor dem Zeichnen mittels `emit startPainting();` das Signal `startPaintig` ausgelöst.
- Ist die Methode mit dem Zeichnen fertig, löst sie mittels `emit endPainting();` das Signal `endPainting` aus.

12.10. Qt Hauptfenster



- Ein Hauptfenster wird in Qt durch die Klasse `QMainWindow` dargestellt.
- In einer **Menüleiste** (`QMenuBar`) können Aufklappmenüs (`QMenu`) abgelegt werden. Jede `QMenu` Instanz enthält `QAction` Elemente.
- Sämtliche Aktionen (`QAction`) können auch für einen schnellen Zugriff in der **Symbolleiste (Werkzeugleiste, Toolbar)** (`QToolBar`) abgelegt werden.
- Im **Dock-Fensterbereich** können frei plazierbare Dialoge abgelegt werden.
- Das **zentrale Widget** ist das wichtigste Element des Hauptfensters. Es ist die hauptsächliche Interaktionsfläche mit dem Benutzer.
- In der **Statusleiste** (`QStatusBar`) können dem Benutzer kurze flüchtige Meldungen angezeigt werden.

Aufgabe Haus vom Nikolaus 3



- Man füge der bereits erstellten Anwendung für das Zeichnen des Hauses vom Nikolaus eine Klasse `MainControl` hinzu, die von der Klasse `QMainWindow` abgeleitet ist und die die Rolle des Controls übernimmt.
- In der Symbolleiste sollen die folgenden Aktionen (`QAction`) vorhanden sein:
  - `forward` beschriftet mit dem Symbol >. Bein Click() auf diese Aktion wird das Haus um eine Linie vorwärts gezeichnet.
  - `backward` beschriftet mit dem Symbol <. Bein Click() auf diese Aktion wird das Haus um eine Linie rückwärts gezeichnet.
  - `quit` beschriftet mit `Quit` um die Anwendung zu beenden.
- Man versuche auch diese Aktionen in der Menue-Leiste abzulegen und für Quit einen Hot-Key zu vergeben.
- Man versuche auch in der Statusleiste Informationen an den Benutzer auszugeben.

Index

#ifndef, 167
#include, 57
++, 214
->, 177
..., 296
:, 232, 233
::, 58
<<, 63, 209, 349
<stdexcept>, 306
>>, 64, 351
NULL, 135
QAction, 403
QApplication, 383
QBoxLayout, 390
QBrush, 397
QFont, 397
QGridLayout, 390
QHBoxLayout, 390
QLabel, 382
QLayout, 390
QLayoutItem, 390
QMainWindow, 403
QMenu, 403
QMenuBar, 403
QObject, 378
QPainter, 395
QPainterDevice, 395
QPen, 397
QStackedLayout, 390
QStatusBar, 403
QToolBar, 403
QVBoxLayout, 390
QWidget, 382
Q_OBJECT, 382, 384
SIGNAL, 384
SLOT, 384
[], 210
[](){}, 361
accumulate(), 365
adjustfield, 67
algorithm, 354
auto, 128
bad(), 350
bad_alloc, 96, 305
bad_cast, 245, 305
bad_exception, 305
bad_typeid, 305
badbit, 307, 350
basefield, 67
begin(), 132, 199
bool, 127
boolalpha, 67, 74
c++filt, 321
catch, 296
cerr, 61
cin, 61
class, 162, 163
clear(), 64, 350
clog, 61
close(), 349, 351
connect(), 385
const, 80, 85, 86, 89, 172
const_cast, 247
constexpr, 77, 92
copy(), 132
count(), 132, 355
count_if(), 355
cout, 61
cstdio, 66
dec, 67, 74
default, 188, 189
delete, 98, 190
delete[], 100
deque, 131
domain_error, 305
draw(), 396
dynamic_cast, 245
emit, 384
end(), 132, 199
endl, 63, 74
ends, 74
eof(), 350
eofbit, 307, 350
exception, 295, 306
exceptions(), 308
exec(), 383
extern, 119
fail(), 350
failbit, 307, 350
false, 127
filter, 365
final, 234

find(), 132, 355
fixed, 67, 74
flags(), 68
floatfield, 67
flush, 74
flush(), 349
fold, 365
for, 130
for_each(), 132, 362
forward_list, 131
free(), 103
friend, 205
fstream, 62
functional, 357
get(), 352
getline(), 352
good(), 350
goodbit, 307, 350
hex, 67, 74
ifstream, 62, 351
ignore(), 64
initializer_list, 195
inline, 124, 170
internal, 67, 74
invalid_argument, 305
ios, 62
ios::fmtflags, 67
ios_base::failure, 305, 307
iostate, 307
iostream, 57, 62
is_open(), 349, 351
istream, 62
left, 67, 74
length_error, 305
list, 131
logic_error, 305
make_unique, 178
malloc(), 103
map, 131, 365
multimap, 131
multiset, 131
mutable, 364
new, 96
noboolalpha, 74
noexcept, 299
noshowbase, 74
noshowpoint, 74
noshowpos, 74
noskipws, 74
nouppercase, 74
nullptr, 98, 135
nullptr_t, 135
oct, 67, 74
ofstream, 62, 348
open(), 349, 351
operator, 203
ostream, 62
out_of_range, 305
overflow_error, 305
override, 257
paintEvent(), 396
priority_queue, 131
private, 163, 232, 233
protected, 163, 232, 233
public, 163, 232, 233
put(), 349
qApp, 383
queue, 131
range_error, 305
rdbuf(), 349
rdstate(), 350
read(), 352
reduce, 365
reinterpret_cast, 248
remove_if(), 365
resetiosflags(), 73
right, 67, 74
runtime_error, 305
scale(), 399
scientific, 67, 74
search(), 132
seekp(), 349
set, 131
setWindow(), 399
setbase(), 73
setf(), 68
setiosflages(), 73
setprecision(), 73
setstate(), 350
setw(), 73
show(), 383
showbase, 67, 74
showpoint, 67, 74
showpos, 67, 74
signals, 384
size(), 199
skipws, 67, 74
slots, 384
sort(), 354
stack, 131
static, 167, 173, 273
static_cast, 193, 244
std, 57, 340
std::nothrow, 96
std::nullptr_t, 320
stdio.h, 66

strstream, 62
struct, 166
sync(), 64
tellp(), 349
template<>, 317, 319
terminate(), 294
this, 168, 172
throw, 295, 298
throw(), 299
transform(), 365
translate(), 399
true, 127
try, 296
type_info, 252
typeid(), 252
typename, 317, 323
underflow_error, 305
unique_ptr, 178
unitbuf, 67
unsetf(), 69
update(), 396
uppercase, 67, 74
using, 58
usingnamespace std, 57
vector, 131
virtual, 256
void*, 248
volatile, 129
what(), 306
write(), 349
ws, 74
{}, 94
*/, 56
/*, 56
//, 56
byte, 45

Abbau von Objekten, 238
ABC, 259
ablaufgesteuerte Programmierung, 371
Ableitung, 157
Ableitung unterbinden, 234
Abstammungsliste, 232
Abstrac Base Class, 259
abstract factory, 271
Abstrakte Klasse
    UML, 155
abstrakte Klasse, 259
abstrakter Deklarator, 106
Abstraktion, 146
Activity Diagram, 22
adapter, 271
Äquivalentes Funktionsobjekt, 363
Äquivalenz, 220
Aggregation
    UML, 159
agiles Vorgehensmodell, 13
Akteur, 19
aktivieren
    Methoden, 189
Aktivitätsdiagramm, 22
Algorithmen
    STL, 132
Algorithmus
    Soundex, 53
Algotithmen
    STL, 354
Allzweck-Funktionswrapper, 357
Anforderungen
    funktionale, 7
    nicht funktionale, 7
Anti-Pattern, 272
Anwendungsfalldiagramm, 19
Anwendungsprogrammierung, 46
API, 36
arc42, 13
Architektur-Brezel, 11
Ariane 5, 290
Arithmetik
    mit gemischtem Typ, 211
ArrayBool, 314
Artefakt-Repository, 29
Assoziation
    UML, 158
Assoziativer Container, 131
Attribut, 147
Attribute, 162
    in Klasse, 167
Aufbau von Objekten, 238
Aufgabe
    Klasse Vec, 227
Aufruf-Operator, 208
Aufruf-Overhead, 124, 170
Aufteilung
    des Hauptspeichers, 95
Ausdruck
    konstant, 92
    Lambda, 361
    vollständig geklammert, 337
Ausgabe
    C-Stil, 66
    formatiert, 67, 73
Ausgabe-Operator, 63
    Überladung, 209
Auslieferungsphase, 11
Ausnahmebehandlung, 288

Auswurf
    einer Exception, 293, 295
Automat, 21
automatischer Speicher, 95

B 737, 289
Basisinitialisierer, 237
Basisklasse, 151, 228
Baum, 280
Baumstruktur, 278
BDD, 35
Behavior Driven Development, 35
Benchmark, 51
Bereichsbasierte Schleife, 130
Besitz
    in Qt, 379
Big-Five, 222
Big-Three, 216
Binden
    dynamisch, 256
    spätes, 256
Bindung
    dynamisch, 152
    späte, 152
    statisch, 152
bitweise Kopie, 179
Block-Kommentar, 56
BPM, 16
brace initialization, 94
Branch, 28
bridge, 271
Bubble-Sort, 322
Bugtracking-Systeme, 26
Build-Management, 29
builder, 271

C++11, 45
C++14, 45
C++17, 45
C++99, 45
call by value, 114
call-back, 372
call-back Funktion, 107
capture list, 361
CASE Tools, 25
Cast, 193
    Const, 247
    Down, 245
    dynamic, 245
    Reinterpret, 248
    static, 244
    Typinformation, 249
    Up, 245
Cast-Operator, 243
catch by const reference, 295
cdecl
    Tool, 106
chain of responsibility, 271
Clean Code, 36
Code-Faktorisierung, 150
Code-Wiederholung, 151
command, 271
communication diagram, 23
component diagram, 18
Composite, 278
composite, 271
const correctness, 172
const Methoden, 172
Container, 131
Contravariation, 262
Control, 393
Copy-Konstruktor, 179, 216
Copy-Semantik, 216
Copy-Zuweisungs-Operator, 179, 208, 216
Covariation, 262
cpp-Datei, 121
cpplint, 31
CppUnit, 34
Cucumber, 35

Dateiverarbeitung
    Streams, 345
Daten-Member, 162
    in Klasse, 167
Datenkapselung, 149
Datensegment, 95
deaktivieren
    Methoden, 190
decorator, 271
deduzieren Typ, 318
Default Parameter, 122
default Parameter
    Konstruktor, 194
Default-Konstruktor, 184, 194
    Notwendigkeit, 188
Definition, 120
Definitionsphase, 9
Deklaration, 119
    komplex, 106
Deklarator
    abstrakter, 106
delegierender Konstruktor, 191
deployment diagram, 18
design by contract, 261
design pattern, 271
Destruktor, 197, 216

Vererbung, 238
virtuell, 264
Dock-Fensterbereich, 403
Down-Cast, 244, 245
Doxygen, 30
Dreier-Regel, 216, 219
Verletzung, 217
Dummy Parameter, 214
Durchführbarkeitsstudie, 8
dynamic cast
Typinformation, 249
dynamische Bindung, 152
dynamische Polymorphie, 241
dynamische Quellcode Analyse, 32
dynamischer Speicher, 95
dynamisches Binden, 256
dynamisches Instanzieren, 177

E-Cube-Routing, 136
Echtzeitmodellierung, 16
einfach verkettete Liste, 199
Einfach-Vererbung, 231
Eingabe
C-Stil, 66
Eingabe-Operator, 64
Ellipse, 296
Elternklasse, 228
endlicher Automat, 21
Entwurfsmuster, 271
Ereignis, 372
Ereignisorientierte Programmierung, 372
Ereignisschleife, 372
Ereignisse, 372
Erfassung
explizite, 362
implizite, 362
Erfassungsliste, 361
Ergebnisrückgabe
cpplint konform, 118
von Funktionen, 112
Erzeugungsmuster, 271
Event, 372
event loop, 372
Eventloop-Start, 383
Events, 372
Exception, 293
Auswurf, 295
erneut auswerfen, 298
Maske, 307
Missbrauch zur Ablaufsteuerung, 312
Spezifikation, 299
Exception-Handler, 293
explizite Instanzierung, 323
explizite Template Instanzierung, 318
Extreme Programming, 12

F-16, 289
factory method, 271
fascade, 271
Fehler
Laufzeit, 288
logischer, 288
Fehler-Bits, 350
Fehler-Code, 292
Fehler-Nummer, 292
Fehlerausgabe, 61
Fehlerbehandlung
klassisch, 292
Fehlerhäufigkeit, 291
Fehlerklassen
der Standard-Bibliothek, 305
Fehlertoleranz, 302
Feld
partiell gefüllt, 131
Fenster, 399
Fibonacci Generator, 359
FIFO, 131
finally, 302
First-Class-Funktion, 356
flache Kopie, 179
Flag, 67
flyweight, 271
formatierte Ausgabe, 67, 73
Formulardatei, 375
freie Funktion
Überladung, 203
freie Operatorfunktion, 204
friend Konzept, 205
Funktion
call-back, 372
Ergebnisrückgabe, 112
First-Class, 356
höherer Ordnung, 356
inline, 124, 170
Signatur, 125
überladen, 125
überschreiben, 125
funktionale Anforderungen, 7
Funktionale Programmierung, 356
funktionales Programmierparadigma, 46
Funktions-Objekt, 208
Funktions-Pointer, 357
Funktions-Template
Spezialisierung, 319
Funktions-Templates, 317
Funktionsobjekt, 358

Äquivalentes, 363
Funktionswrapper, 357
Funktionszeiger, 107
Funktor, 208, 358

Gang of Four, 40, 271
Garbage Collection, 98
gebundene Polymorphie, 241
Geheimnisprinzip, 149
Generalisierung, 228
UML, 157
generatives Programmieren, 316
Generator, 358
generische Klasse, 323
generisches Programmieren, 316
generisches Programmierparadigma, 46
Geschäftsprozessmodellierung, 16
Git, 28
GoF, 40, 271
Google Test, 34
GOTO, 141
Graphical User Interface, 370
Gray-Code, 136
GTK, 373
GUI, 370
GUI-Anwendung, 372
GUI-Toolkits, 373

Halteproblem, 33
Hamming-Distanz, 136
Handle, 385
Hauptprogramm
Start der Eventloop, 383
Hauptspeicher
Aufteilung, 95
Haus der Objektorientierung, 154
Haus vom Nikolaus, 402
Header-Datei, 121
Header-Dateien
Inkludieren, 57
Standard-Bibliothek, 340
Heap, 95
Hello World Programm, 56
heob, 32
Hierarchie
Vererbung, 151
Higher Order Function, 356
Hyper-Cube, 136

Identität, 147
IEEE 1417, 13
Impedanzberechnung, 279
Implementierungs-Datei, 121
Implementierungsvererbung, 261
implizite Instanzierung, 323
implizite Template Instanzierung, 318
include guard, 167
Include-Wächter, 167
Indizierungs-Operator
Überladung, 210
Infix-Form, 337
Information Hiding, 149
Inheritance, 151
inheritance, 228
Initialisierung
Klassenvariablen, 173
vereinheitlichte, 94
Initialisierungslisten Konstruktor, 195
Inkludieren von Header Dateien, 57
Inklusions-Modell, 325
Inkrement-Operator
Überladung, 214
inline Funktion, 170
inline Funktionen, 124
inline Methode, 170
Instanz, 148, 162
Instanzieren, 177
Instanzierung
explizit, 323
implizit, 323
integral promotion, 127
interaction overview diagram, 23
Interaktionsübersichtsdiagramm, 23
Interface
UML, 157
interface, 261
UML, 155
interpreter, 271
Invarianten, 150
Invariants, 150
Invariation, 262
is-a Beziehung, 228, 232
ISO, 46
Iterator, 132
iterator, 271

Jenkins, 30
JSON, 36
JUnit, 34

Kardinalität
der Vererbung, 231
KDE, 373
Kindklasse, 228
Klasse, 146, 148, 162
abstrakt, 259
abstrakte UML, 155

Attribute, 167
Daten-Member, 167
generisch, 323
konkrete, 259
Member-Funktionen, 168
Methoden, 168
Modifikator, 163
polymorphe, 256
Schnittstelle, 261
Vererbung, 228
Klasse Vec, 227
Klassen Methoden
UML, 155
Klassen Variablen
UML, 155
Klassen-Templates, 323
Klassendiagramm, 17
statische Member, 155
Klassenhierarchien, 151, 228
Klassenname
Template, 324
Klassenvariable, 167
Klassenvariablen
Initialisierung, 173
Klassifizierung, 146
klassische Fehlerbehandlung, 292
Kodierrichtlienien, 31
Kommentare, 56
Kommunikationsdiagramm, 23
Komplement, 197
komplexe Deklarationen, 106
Komponenten
Standard-Bibliothek, 341
Komponentendiagramm, 18
Komposition
UML, 159
Kompositionsstrukturdiagramm, 18
konkrete Klasse, 259
Konsistenz
der Daten, 150
Konsolen-Anwendung, 371
konstanter Ausdruck, 92
Konstruktionsphase, 10
Konstruktor, 181
Cast, 243
Copy, 216
Default, 184
default Parameter, 194
delegierender, 191
Initialisierungslisten, 195
konvertierender, 192
mit Memberinitialisierungsliste, 186
Move, 222
nicht konvertierender, 193
Sequenz, 195
Standard, 184
Vererbung, 237
Verschiebe, 222
Konstruktor-Schreibweise
Cast, 243
Kontrakt, 149
konvertierender Konstruktor, 192
Operator-Überladung, 213
Konvertierung
verengende, 94
Koordinatensystem
in Qt, 398
logisches, 399
Kopie
flache, 179

L-Value, 79, 84
L-Wert, 84
Lambda Ausdruck
Typ der Rückgabe, 364
Lambda-Ausdruck, 361
Lastenheft, 8
Laufzeit-Typinformationen, 249
Laufzeitfehler, 288
Layouts, 390
Lebenslinie, 20
Lesen einer Datei
blockweise, 352
unformatiert, 352
LIFO, 131
Linting, 31
Liskovsches Substitutionsprinzip, 232, 242
Liste, 131
einfach verkettete, 199
logische Fehler, 288
logisches Koordinatensystem, 399

MAJOR
Versionsnummer, 29
Makro Variante
Signal und Slot, 386
Manipulator, 63
Manipulatoren, 73
Mariner 1, 289
MDA, 16
mediator, 271
Mehrfach-Vererbung, 231
Member
der abgeleiteten Klasse, 235
member initializer list, 186
Member-Funktion

Überladung, 203
Member-Funktionen, 162
in Klasse, 168
Memberinitialisierungsliste, 186
memento, 271
Memory-leak, 98
Memory-Leaks, 32
Menge, 131
Menueleiste, 403
message, 152
Meta Object Compiler, 384
Meta-Objektsystem, 384
Metaklasse, 18
Metaprogrammierung, 45
Methode, 147, 162
aktivieren, 189
deaktivieren, 190
inline, 170
override, 257
Redefinition, 257
rein virtuelle, 259
virtuell, 241, 256
Methoden
const, 172
in Klasse, 168
Methoden-Templates, 332
Methodentabelle
virtuelle, 257
MFC, 373
Microsoft Foundation Classes, 373
MINOR
Versionsnummer, 29
MOC, 384
Model, 393
Model Driven Architecture, 16
Model-View-Control Muster, 393
Modifikator
in Klasse, 163
Move, 222
Move-Konstruktor, 222
Move-Semantik, 222
Move-Zuweisungs-Operator, 208
Multiplizität
UML, 158
Mutex, 182
MVC Muster, 393

Nachbedingungen, 150
Nachricht, 152
Name
voll qualifizierter, 58
Name mangling, 321
Namensbereich, 58
Namensraum, 57, 58, 162
Namenssuche
Vererbung, 235
namespace, 57, 58, 162
Nassi-Shneiderman-Diagramm, 141
New York Stock Exchange, 290
nicht funktionale Anforderungen, 7
nicht konvertierender Konstruktor, 193
Nicht-Typ-Parameter, 320
no-elide-constructors, 180
Notwenigkeit
Standard-Konstruktor, 188
Nutzungsfälle, 9

Oberklasse, 228
Object Constraint Language, 150
object diagram, 18
Object Management Group, 15
Objekt, 142, 148
der Computer Welt, 147
Abbau bei Vererbung, 238
Aufbau bei Vererbung, 238
Computer-Welt, 147
Erzeugung, 177
this, 168
Objektbäume, 379
Objektdiagramm, 18
Objektorientierte Analyse, 144
objektorientierte Modellierung, 143
Objektorientierte Programmierung, 144
Objektorientiertes Design, 144
objektorientiertes Programmierparadigma, 46
Objektorientierung, 142, 153
Haus der, 154
Observer, 282
observer, 271
OCL, 150
OMG, 15
One Definition Rule, 120
OOA, 144, 160
OOD, 144, 160
OOP, 144, 154
Operator
Ausgabe, 63
Cast, 243
Eingabe, 64
Überladung, 201
Operatorfunktion, 203
als Member-Funktion, 207
freie, 204
overloading, 125
overriding, 125

package diagram, 18

Pair-Programming, 12
Paketdiagramm, 18
Parameter
    default, 122
    Referenz, 81
    Typ, 317
ParameterÜbergabe, 78
Pascal-Dreieck, 105
PATH
    Versionsnummer, 29
Persistenz, 345
Pflichtenheft, 8, 147
Pinsel, 397
Planungsphase, 8
Pointer
    smart, 178
policy object, 358
polymorphe Klasse, 256
polymorpher Typ, 256
Polymorphie, 152, 241
Postconditions, 150
Postfix-Form, 214, 337
Prädikat, 355
Präfix-Form, 214
Preconditions, 150
private
    UML, 156
Profildiagramm, 18
Programmgröße, 291
Programmier-Paradigma, 141
Programmieren
    generativ, 316
    generisch, 316
Programmierfehler, 288
Programmierparadigma
    funktionales, 46
    generisches, 46
    objektorientiert, 46
    prozedurales, 46
Programmiersprache C++, 43
Programmierung
    ablaufgesteuert, 371
    ereignisorientiert, 372
    Funktional, 356
    Schnittstelle, 261
Programmsegment, 95
protected
    UML, 156
prototype, 271
proxy, 271
Prozedurale Programmierung, 141
prozedurales Programmierparadigma, 46
public
    UML, 156
public Vererbung, 232
Puffer, 346
Pull-Modell, 283
Punkt-Notation, 177
pure virtual method, 259
Push-Modell, 283
PYPL-Index, 49

qmake
    Flag setzen, 180
Qt, 370
    Assistent, 374
    Besitz, 379
    Bestandteile, 374
    Core, 376
    Designer, 375
    Dock-Fensterbereich, 403
    eigene GUI Klasse, 382
    Erstellung eines GUIs, 380
    Essentials, 376
    Gui, 376
    Hauptfenster, 403
    Klassenhierarchie, 378
    Koordinatensystem, 398
    Layouts, 390
    Linguist, 375
    Menueleiste, 403
    Meta-Objectsystem, 384
    Multimedia, 376
    Networks, 376
    Pinsel, 397
    Quick, 377
    Schrift, 397
    Speicherverwaltung, 379
    SQL, 376
    Statusleiste, 403
    Stift, 397
    Symbolleiste, 403
    Test, 376
    Toolbar, 403
    Viewport, 398
    Werkzeugleiste, 403
    Widgets, 376
    Zeichnen, 395
Qt Creator, 380

R-Value, 84
R-Wert, 84
R-Werte Referenz, 222
RAII, 98, 182, 197
    anstelle von finally, 302
RAII-Konzept, 347

Rational Unified Process, 12
RCS, 28
Redefinition, 235
Redifinition
    Methode, 257
Refactoring, 10
Referenz, 79
    als Parameter, 81
    auf Array, 80
    auf Feld, 80
    auf Konstante, 80
    auf R-Wert, 222
    Schreibschutz, 89
    Vergleich zu Zeiger, 83
Referenz-Semantik, 112
Regel
    Dreier, 216
Reihenfolge
    Memberinitialisierung, 187
rein virtuelle Methode, 259
Repository, 28
Resource Acquisition Is Initialization, 183
Ressourcenbelegung ist Initialisierung, 98, 182, 197
Return Value Optimization, 180
Reverse Engineering, 27
Richtlinienobjekt, 358
RT, 16
RTTI, 249
Rule of Fife, 222
Rule of Three, 216, 219
    Verletzung, 217
Run-Time Type Information, 249
RUP, 12
RVO, 180

Schlange, 131
Schleife
    bereichsbasiert, 130
Schnittstelle, 149, 165, 261
    UML, 155, 157
Schnittstellen-Programmierung, 261
Schnittstellenvererbung, 261
Schreibschutz, 85
    für Referenzen, 89
    für Zeiger, 86
Schrift, 397
Scope-Operator, 58
Scrum, 12
SDD, 35
Selbstzuweisung, 219
Semaphore, 182
Sequence Diagram, 20
Sequenzdiagramm, 20
Sequenzkonstruktor, 94, 195
Serialisierung, 36
Signal und Slot
    Makro-Variante, 386
    neue Variante, 385
Signal verbinden mit Slot, 385
Signals und Slots, 384
Signatur, 125
SIMULA, 43
Simula, 141
Singleton
    klassische Version, 273
    Scott Meyers, 276
singleton, 271
Slicing, 113, 242
Slot, 384
SMALLTALK, 43
Smaltalk, 141
Smart Pointer, 178
Sortieren, 354
Soundex Algorithmus, 53
späte Bindung, 152
spätes Binden, 256
Specification Driven Development, 35
Speicher
    automatischer, 95
    dynamischer, 95
Speicherverwaltung
    in C, 103
Spezialisierung, 151, 228
    UML, 157
Spezialisierung Funktions-Template, 319
Spiralenmodell, 12
Splitting, 22
Stack, 95
    Template, 336
Stack Unwinding, 183, 300
Stack-Abwichlung, 183
Stack-Unwinding, 293
Standard Template Library, 341
Standard-Ausgabestrom, 61
Standard-Bibliothek, 338
    Design-Aspekte, 339
    Header-Dateien, 340
    Komponenten, 341
    working drafts, 338
Standard-Eingabestrom, 61
Standard-Konstruktor, 184, 194
    Notwendigkeit, 188
Stape, 131
state, 271
state diagram, 21

statische Bindung, 152
statische Member, 173
    UML Klassendiagramm, 155
statische Polymorphie, 241
statische Quellcode Analyse, 31
statisches Instanzieren, 177
Statusleiste, 403
Stereotyp
    UML, 155
Steuerelemente, 370
Stift, 397
STL, 131, 341
    Algorithmen, 354
    Sortieren, 354
    Suche, 355
strategy, 271
Stream, 61
Streams
    Dateiverarbeitung, 345
Strich-Männchen, 19
Stroustrup, 38, 44
Strukturdiagramm, 17
Strukturen, 166
Strukturmuster, 271
Subclass, 151
Subklasse, 228
Substitutionsprinzip
    Liskovsches, 242
Subversion, 28
Suche
    STL, 355
Superclass, 151
SVN, 28
swim-lane, 20
Symbolleiste, 403
Synchronisation, 22
Systemprogrammierung, 46

TCL, 373
TDD, 34
Technischer Report 1, 45
Temperatur-Sensor, 284
Template
    Aufteilung Quellcode, 325
    Contravariation, 262
    Covariation, 262
    Invariation, 262
    Klassenname, 324
    Nicht-Typ-Parameter, 320
    Polymorphie, 241
    Stack, 336
    Vererbung, 230, 262
Template Instanzierung
    explizit, 318, 323
    implizit, 318
template method, 271
Template-Instanzierung
    implizit, 323
Templates, 316
    Funktions, 317
    Klassen, 323
    Methoden, 332
Test Driven Development, 34
Test-Quadranten, 33
Therac-25, 290
Threads, 45
Throw by value, 295
tiefe Kopie, 216, 219
timing-diagramm, 23
TIOBE-Index, 48
Tk, 373
Toolbar, 403
Toolkit
    GUI, 373
Toolkits, 373
Trapezregel, 109
Typ
    polymorpher, 256
Typ-Parameter, 317
Typinformation
    typeid, 252
Typinformationen
    Laufzeit, 249
Typplatzhalter auto, 128
Typprüefung, 245

Überdeckung, 235
Überdeckungsgrad, 35
Überladen
    von Funktionen, 125
Überladung
    Ausgabe-Operator, 209
    freie Funktion, 203, 204
    Indizierungs-Operator, 210
    Inkrement-Operator, 214
    Member-Funktion, 203, 207
Überladung von Operatoren, 201
Überschreiben, 151, 235
    von Funktionen, 125
Uhr
    Klasse, 198
UI-Compiler, 375
UMK
    Komponentendiagramm, 18
UML, 15
    Objektdiagramm, 18

Aggregation, 159
Aktivitätsdiagramm, 22
Anwendungsfalldiagramm, 19
Assiziation, 158
Interaktionsübersichtsdiagramm, 23
Interface, 157
interface, 155
Klassendiagramm, 17
Kommunikationsdiagramm, 23
Komposition, 159
Kompositionsstrukturdiagramm, 18
Multiplizität, 158
Paketdiagramm, 18
private, 156
Profildiagramm, 18
protected, 156
public, 156
Schnittstelle, 155, 157
Sequenzdiagramm, 20
Spezifikation, 16
Stereotyp, 155
Strukturdiagramm, 17
Vererbung, 157
Verhaltensdiagramm, 19
Verteilungsdiagramm, 18
Zeitverlaufsdiagramm, 23
Zugriffsrechte, 156
Zusicherung, 155
Zustandsdiagramm, 21
UML Klassendiagramm, 155
UMlet, 160
Unabhängigkeit, 220
unformatiertes Lesen einer Datei, 352
ungebundene Polymorphie, 241
Unified Modeling Language, 15
Unterklasse, 151, 228
Unwinding
Stack, 183
Up-Cast, 245
use case diagram, 19
Use Cases, 9

V-Modell, 12
valgrind, 32
Vec, 227
vereinheitlichte Initialisierung, 94
verengende Konvertierung, 94
Vererbung, 151, 228
Abbau von Objekten, 238
Aufbau von Objekten, 238
C++, 232
Contravariation, 262
Covariation, 262
Destruktor, 238
einfach, 231
Implementierungs, 261
Invariation, 262
Konstruktor, 237
mehrfach, 231
Member der abgeleiteten Klasse, 235
Namenssuche, 235
public, 232
Redefinition, 235
Schnittstelle, 261
Slicing, 242
Template, 262
Überschreiben, 235
Überdeckung, 235
UML, 157
unterbinden, 234
virtuelle Methode, 241, 256
virtueller Destruktor, 264
Zugriff auf Basisklassen-Member, 233
Verhaltensdiagramm, 19
Verhaltensmuster, 271
verkettete Liste, 199
Verkettung
Ausgabe-Operator, 209
Verletzung
Dreier-Regel, 217
Rule of Three, 217
Verschiebe-Konstruktor, 222
Verschieben, 222
Verschiebesemantik, 222
Versions-Nummer, 29
Versionsverwaltung, 28
Verteilungsdiagramm, 18
Vielgestaltig, 152
Vielgestaltigkeit, 241
View, 393
Viewport, 398
virtuelle Destruktoren, 264
virtuelle Methode, 241
virtuelle Methoden, 256
virtuelle Methodentabelle, 257
visitor, 271
VMT, 257
voll qualifizierter Name, 58
vollständig geklammerter Ausdruck, 337
Vorbedingungen, 150
Vorgehensmodell
agile, 13

Wasserfallmodell, 12
Werkzeugleiste, 403
Wert-Semantik, 112, 216

wertorientierte Programmierung, 220
Widget, 370
Wiederverwendbarkeit, 151
Window, 399
working drafts
    Standard-Bibliothek, 338
wxWidgets, 373

XP, 12

Y2K, 290

Zeichnen mit Qt, 395
Zeiger
    `const`, 86
    auf Funktion, 107
    Schreibschutz, 86
    Vergleich zu Referenz, 83
Zeitverlaufsdiagramm, 23
Zugriff
    auf Basisklassen-Member, 233
Zugriffs-Operator, 208
Zugriffskontrolle
    in Klasse, 163
Zusicherung
    UML, 155
Zustandsdiagramm, 21