Objektorientiertes Programmieren in C++ für Physiker

Günter Duckeck

29.7. – 2.8.2019

Inhalt:

Überblick

Objektorientiertes Programmieren Konzepte und Denkweise, Analyse und Design, UML Diagramme, Test Driven Development, Design Patterns

C++ Programmieren Standard Template Library, Qt–Graphics Library, GUIs, C++-11, Multi–Threaded und Parallel Programmieren

Folien & Übungen im WWW

http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckursoo

und als Slides.

Schlüsselqualifikation für Bachelor/Master

- C++ Kurs kann als Schlüsselqualifikation angerechnet werden: 3 ECTS Punkte
- Erfolgreiches Bestehen der Leistungskontrolle (= Kurztest am Ende) ist Kriterium für Punktevergabe
- Zusammen mit den Klausuraufgaben verteilen wir die Scheinformulare, die Sie ausfüllen und mit der Klausur abgeben.
- Nach Korrektur veröffentlichen wir die Liste (Matrikelnummer, bestanden/nicht bestanden) auf dieser Kursseite und leiten die Scheine ans Prüfungsamt weiter. Sie können Ihren Schein dann im Prüfungsamt abholen (Ferienzeit, spätestens Mitte September).

Kurs baut auf einführendem Kurs C++ für Physiker auf.

- Syntax, Variablen, I/O
- Pointer, Referenzen, Funktionen
- Klassen und Objekte, Vererbung

In diesem Kurs **zwei Themen/Ziele**:

1. Objektorientiertes Programmieren:
Eher konzeptioneller Background, Einführung in Denkweise, Methoden, Techniken.
I.d.R. vormittags, 10:00–12:00.

2. Fortgeschrittenes Programmieren in C++:
Eher praxis-orientiertes Programmieren, aufbauend auf erstem Kurs werden weitere wichtige Funktionen behandelt: Exceptions, STL, Qt, Graphische User Interfaces (GUI), C++-11, Threads, OpenMP, XML Prozessieren I.d.R. nachmittags, 13:30–16:00.

Nur lose Kopplung dieser beiden Bereiche ...

Kurs soll Eindruck verschaffen was modernes Programmieren in C++ beinhaltet, und Anregungen geben für kleinere eigene Projekte unter Benutzung von STL, Qt-GUIs, u.ä.

Themenliste Objektorientiertes Programmieren

Jeweils vormittags, 10:00-12:00.

Montag Klassen und Objekte in C++ (1. Kurs im Schnelldurchgang)

Dienstag OO Programmieren in C++:

- Vererbung und Composition
- Polymorphismus
- abstrakte Methoden und Klassen
- Interfaces

Einführung OO Design

Mittwoch OO Analyse und Design, UML Diagramme

Donnerstag Test Driven Development

Freitag **Design Patterns** (einführende Beispiele), Software Design Prinzipien und Fallbeispiele, ...

Themenliste Fortgeschrittenes Programmieren in C++

Jeweils nachmittags, 13:30-16:00.

Montag *Exceptions, Templates, STL (Container, Iterator, Function Objects)*
C++-11

Dienstag und Mittwoch GUI *(Graphisches User Interface)* Programmieren mit Qt in C++:

- Textfelder, Buttons, Menues
- Grafik Funktionen
- Layout
- Event-Handling

Donnerstag *Thread Programming, Parallelisieren mit OpenMP*

Freitag Prüfung, Wrap-up

Literatur und Links

C++ bietet eine enorme Funktionalität, entsprechend umfangreich sind deshalb die meisten Bücher.

C++ Primer Lippman, Lajoie, Moo Addison Wesley, 5th edition (2012). Aktuelle, vollständige Übersicht, inklusive STL, viele Beispiele.

Programming – Principles and Practice Using C++ Bjarne Stroustrup, Addison-Wesley, 2014. Vom C++ Erfinder.

Effective Modern C++ Scott Meyers, 2014, C++ Tipps für Profis vom C++ Guru, aktualisierte Version für C++11.

Exceptional C++ Herb Sutter, C++ Denksport und Puzzles vom C++ Guru.

Programming with Qt von Matthias Kalle Dalheimer, Standard-Werk zu Qt.

Grundlagen der Informatik , Helmut Balzert, Spektrum Akademischer Verlag. Sehr gute und umfassende Einführung in Informatik und objektorientiertes Programmieren. Viele Fallbeispiele in C++, Java. Reichhaltige Software auf CDs enthalten.

UML Distilled , Martin Fowler, Addison-Wesley, 2003. Kompakter Überblick zu UML Design. Viele weitere Infos auf Web page.

Design Patterns , Gamma, Helm, Johnson, Vlissides, Addison-Wesley, Referenzwerk zu Design Patterns, detaillierte Diskussion der 23 “Standard–Patterns” mit Code–Beispielen.

C++ Cookbook D.R. Stephens etal, Umfangreiches Buch mit vielen ausgearbeiteten und erklärten Beispielen zu C++ Programmierung und Verwendung in Vielzahl von Bereichen.

Coding for Fun mit C++ Arnold Willemer, unterhaltsam gemachtes Buch zum Spiele Programmieren in C++.

C++11 programmieren: 60 Techniken für guten C++11-Code Torsten T. Will, Galileo Computing. Eine praktische Anleitung für den Einsatz von C++11.

Online Links:

C++ Annotations Gute Online Beschreibung zu C++ und STL.

C/C++ Referenz Kompakte, übersichtliche Online-Referenz zu C/C++ Funktionen.

Qt Documentation Online Referenz zu Qt. Tutorials, Dokumentation, Klassen-Index ...

OpenMP Documentation Info und Dokumentation zu OpenMP, Link zu Tutorial.

UML Quick Reference

UML: One page reference card

C++ Main Features

Pros:

- Riesiger Funktionsumfang, alles möglich von kleinen hardware-drivern mit wenigen Zeilen bis komplexen S/W Projekten mit vielen Millionen Zeilen.
- Direktes Ansprechen der Hardware mit Pointern
- Volle Funktionalität für Object–Oriented und Generic Programming (Templates)
- Gute Performance
- Vielzahl von Tools und Hilfslibraries erhältlich

Cons: Zu viel Features, zu viele Freiheiten

- Code oft unleserlich und chaotisch
- Schwer zu pflegen
- Steile und lange Lernkurve für vollen Überblick

Grundlagen

- Datentypen und Operationen
- Control–Structures
- Arrays, Pointer und Referenzen
- Funktionen

siehe 1. Kurs: **http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckurs**

Kurze Wiederholung zu
Klassen, Funktionen, Objekte

C++ für Physiker ???

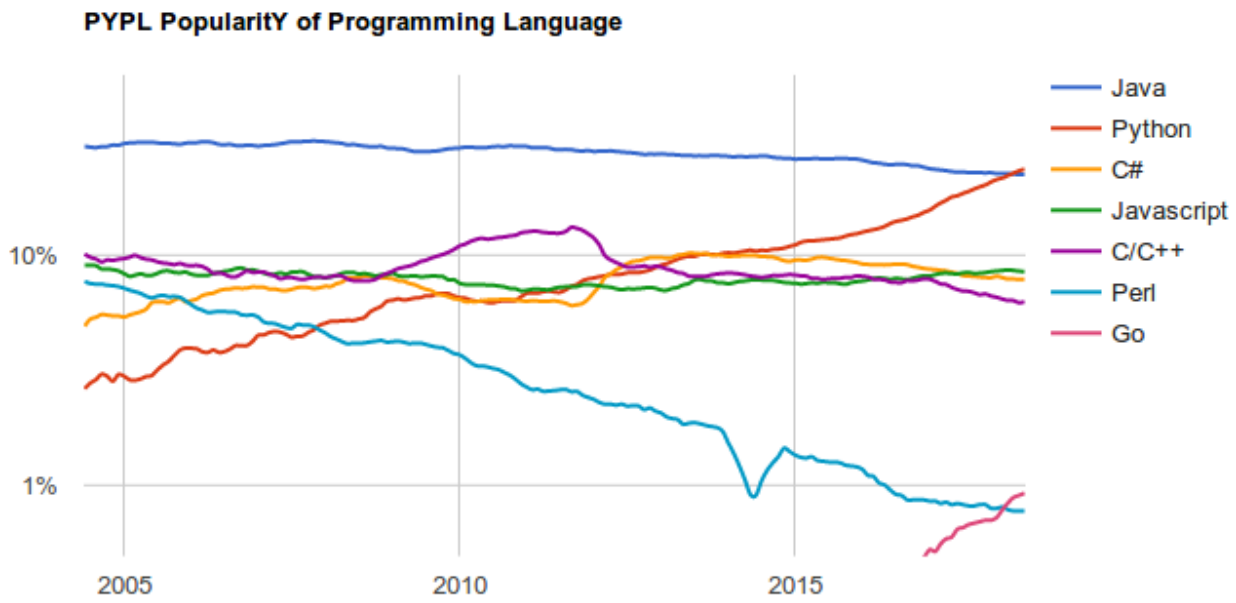
Kontroverse Diskussion ob C++ geeignete Sprache für Physiker ist:

C++ sehr komplex ... Pointer/Referenzen, Memory Management, Templates, Vererbung, virtuelle Funktionen, Konstruktoren/Destruktor, const correctness, u.v.m.

- In kleinen, überschaubaren Projekten ok, man kann sich auf's wesentliche beschränken und nicht viel Schaden anrichten
- andererseits ist Programmieren in C++ vglw. umständlich und ineffizient (cf Python, Java, Julia, ...)
- In grossen, komplexen Projekten heikel. Detaillierte Kenntnisse, professionelle Erfahrung und Schulung nötig.

Allerdings ... man hat meistens gar nicht die Wahl sondern steigt in existierendes Projekt ein mit vorgegebener Software/Programmierumgebung.

Popularität der Programmiersprachen



(Quelle: http://pypl.github.io/PYPL.html)

Klarer Trend – Python auf dem Vormarsch, C++ eher abnehmend :

⇒ *Data Science und Machine Learning*

1 Klassen und Objekte

- Warum objektorientiertes Programmieren ?
- Ein einfaches Beispiel – 3D Vektor
- Daten und Methoden
- Konstruktoren
- Operator overloading
- Übungsbeispiel BigInt
- Aufgaben

1.1 Warum objektorientiertes Programmieren ?

Intrinsische Datentypen (`int, float, double, string, ...`) sind unzureichend für die allermeisten praktischen Probleme.

Offensichtlich im Bereich Verwaltung, Wirtschaft, Handel, ...

- Studenten–Daten an der LMU
- Fahrscheine buchen bei DB
- Ebay Auktionen, ...

Praktisch immer komplexe *Datenmodelle*, d.h. zusammengesetzt aus Strings, int und float Zahlen, Arrays, Querverweise auf weitere Infos, usw.

Aber auch im naturwissenschaftlich/technischen Umfeld – mit überwiegend numerischen Informationen – sind Messdaten keine isolierten Zahlenkolonnen sondern i.d.R. komplex strukturiert und ergeben nur im Kontext mit dem Experiment *(Aufbau, Parameter, äussere Bedingungen)* einen Sinn.

- Spuren im Detektor sind 3er oder 4er Vektoren, bestehen aus Hits, gehören zu einem Sub–Detektor, ...
- Fluoreszenz–Spektrum: Wertepaare *(Frequenz, Intensität)* plus Zusatzinfo zu Probe, Apparatur, Kalibration, äussere Parameter, ...
- ...

1.2 Von primitiven Datentypen zu komplexen Datenstrukturen

Standard Datentypen alleine ungeschickt bzw. unzureichend für praktische Anwendungen, z.B. Studentendaten:

- *Name, Studiengang, Alter, Semester, Matrikel-Nummer, Noten, ...*
- Kombination aus `string, int, double` Daten.

In C++ mittels **struct** oder einfacher Form von **Klasse** möglich solche Datenstrukturen selbst zu definieren:

```
class Stud1 { // einfache Klasse fuer Studenten-Daten
public:
  string name, fach;
  int semester, alter;
  double diplomnoten[4];
    ...
};  // end of class definiton
 ...
   // Anwendung
 Stud1 sta;   // Erzeugung Variable vom typ Stud1
```

```
sta.name = "Albert Unirock";
sta.fach = "Physik";
sta.alter = 25;
...
Stud1 stb;   // Erzeugung Variable vom typ Stud1
stb.name = "Berta Bohne";
stb.fach = "Informatik";
stb.alter = 19;
...
```

- Erzeugen eines **Objektes** einer Klasse in C++ einfach durch **definieren** einer entsprechenden Variablen:
  `Stud1 sa;`
- Zugriff auf Variablen des Objektes *(member-variables)* mittels **objectname.variable**: `sa.alter = 21;`

Weiteres Beispiel: Klasse für einfachen Dreier-Vektor

```
class Dumb3Vec { // einfache Klasse fuer 3-er Vektoren
public:
  double x, y, z;
};  // end of class definition
...
// Anwendung
Dumb3Vec v; // Erzeugung Variable vom typ Dumb3Vec
Dumb3Vec w; // Erzeugung Variable vom typ Dumb3Vec
v.x = 1.0; v.y = 0.5; v.z = -0.8;
w.x = 1.5; w.y = -0.5; w.z = -2.8;
...
// Laenge ausrechnen
double len = sqrt(v.x*v.x + v.y*v.y + v.z*v.z );
Dumb3Vec u;
// u und v addieren
u.x = v.x + w.x;
u.y = v.y + w.y;
u.z = v.z + w.z;
```

...

Anstatt gängige Operationen jedesmal wieder neu zu programmieren wäre es geschickt wenn dies gleich die Klasse übernehmen könnte:

```
class Smart3Vec { // Klasse fuer 3-er Vektoren mit Methoden
public:
  double x, y, z; // member variables
  double Length() {
    return( sqrt(x*x + y*y + z*z ));
  };
  Smart3Vec Add( Smart3Vec a ) {
    Smart3Vec t;
    t.x = x + a.x;
    t.y = y + a.y;
    t.z = z + a.z;
    return(t);
  };
};  // end of class definiton
```

```
//...
// Anwendung
int main()
{
  Smart3Vec v; // Erzeugung Variable vom typ Smart3Vec
  Smart3Vec w; // Erzeugung Variable vom typ Smart3Vec
  v.x = 1.0; v.y = 0.5; v.z = -0.8;
  w.x = 1.5; w.y = -0.5; w.z = -2.8;
  //...
  // v kann seine Laenge selbst ausrechnen ...
  double len = v.Length();
  // ... und weiss auch wie es einen anderen Vektor addiert
  Smart3Vec u = v.Add(w);
  // ...
}
```

Klassen nicht nur zum Definieren beliebiger Datenstrukturen sondern auch gleich Operationen bzw. Methoden mit diesen Daten

$\Rightarrow$ **Grundkonzept für Objektorientiertes Programmieren**

1.3 Eine richtige Klasse für 3D Vektor

```
class My3Vector {
private:          // coordinates, hidden
  double x;
  double y;
  double z;
public:
  My3Vector();// The default constructor
  My3Vector(double c1, double c2, double c3);// Other constructor
  // methods:
  double Length();   //     get length of vector
  // access elements
  double X();
  double Y();
  double Z();
  My3Vector Add( My3Vector p ); // add two vectors
};
```

- Eine C++ Klasse definiert einen neuen Typ *(siehe auch struct, typedef)*. Innerhalb einer Klasse sind i.a. nicht nur Daten definiert sondern auch Methoden *(=Funktionen)*

- Syntax: `class Name { body };`
- Zugriffsmöglichkeit von aussen wird über private/protected/public Modifier kontrolliert
  - **private:** Interne Daten und Methoden, nicht von aussen sichtbar !
  - **public:** Daten und Methoden von aussen sichtbar $\Rightarrow$ Schnittstelle
- Daten sind i.a. *'versteckt'* in einer Klasse *('private')*
- *'public'* Methoden bilden die Schnittstelle
- *Konstruktor-Funktionen* zum Initialisieren

Vorteile:

- Daten geschützt, kein 'unbefugter' Zugriff von anderswo
- Methoden spezifisch für Daten entwickelt & getestet
- Leichtere 'Wartung', da Zugang nur über wohldefinierte Schnittstellen
- wiederverwendbar, vererben, erweitern

Verwendung einer Klasse:

```
#include "My3Vector.h"
int main()
{
  My3Vector a, b(1.,1.,−1.), c(0.,2.,1.); // create 3 ThreeVec objects
  a = b.Add(c); // add ThreeVec b and c, result is stored in a
  cout << a.Length() << endl;
}
```

- Die **Klasse** ist zunächst eine *abstrakte* Definition eines Datentyps mit zugehörigen Methoden
- Ein **Objekt** wird daraus wenn eine entsprechende Variable deklariert wird
- Aufruf von Methoden: `Objektname.Methodenname( ... )`

Erst muss man Methoden aber noch implementieren:

```
#include "My3Vector.h" // include declaration of class
// now implementation

My3Vector::My3Vector() { // default constructor
  x = 0.; y = 0.; z = 0.; // set coords to 0.
}
My3Vector::My3Vector( double c1, double c2, double c3 ) {
  x = c1; y = c2; z = c3; // take args for coords
}
// get length of vector
double My3Vector::Length() {
  return( sqrt( x*x + y*y +z*z ) );
}
// access elements
double My3Vector::X() { return x; }
double My3Vector::Y() { return y; }
double My3Vector::Z() { return z; }
// add
```

```
My3Vector My3Vector::Add( My3Vector p ) {
  My3Vector t;
  t.x = x + p.x;      t.y = y + p.y;      t.z = z + p.z;
  return( t );
}
```

1.4 Daten und Methoden in Klassen

Variablen und **Funktionen**, die **innerhalb** einer Klasse definiert sind, sind die sogenannten **member-variables** bzw. **member-functions**.

Falls sie von aussen zugänglich sind (public), werden sie analog zu structs mittels objectname.variable oder objectname.function() angesprochen

```
My3Vector a(1.,2.,-1.), c(0.,2.,1.);
a.x = 7; // nur wenn x public ...
a.Add(c);
```

Innerhalb einer **member-function** hat man Zugriff auf alle **member-variables**, egal ob private oder public, dabei genügt der Variablen-name:

```
double My3Vector::Length() {
return( sqrt( x*x + y*y +z*z ) ); // x,y,z sind member-variablen
```

Zugriff hat man auch auf **member-variablen** von Objekten **derselben Klasse**, die z.B. als Argument übergeben werden:

```
My3Vector My3Vector::Add( My3Vector & p ) {
   My3Vector t;     t.x = x + p.x; ...
```

Das Objekt mit dem zusammen die Methode gerufen wird ist implizit immer verfügbar, es muss nicht

als Argument angegeben werden, z.B.

```
v.Length();
```

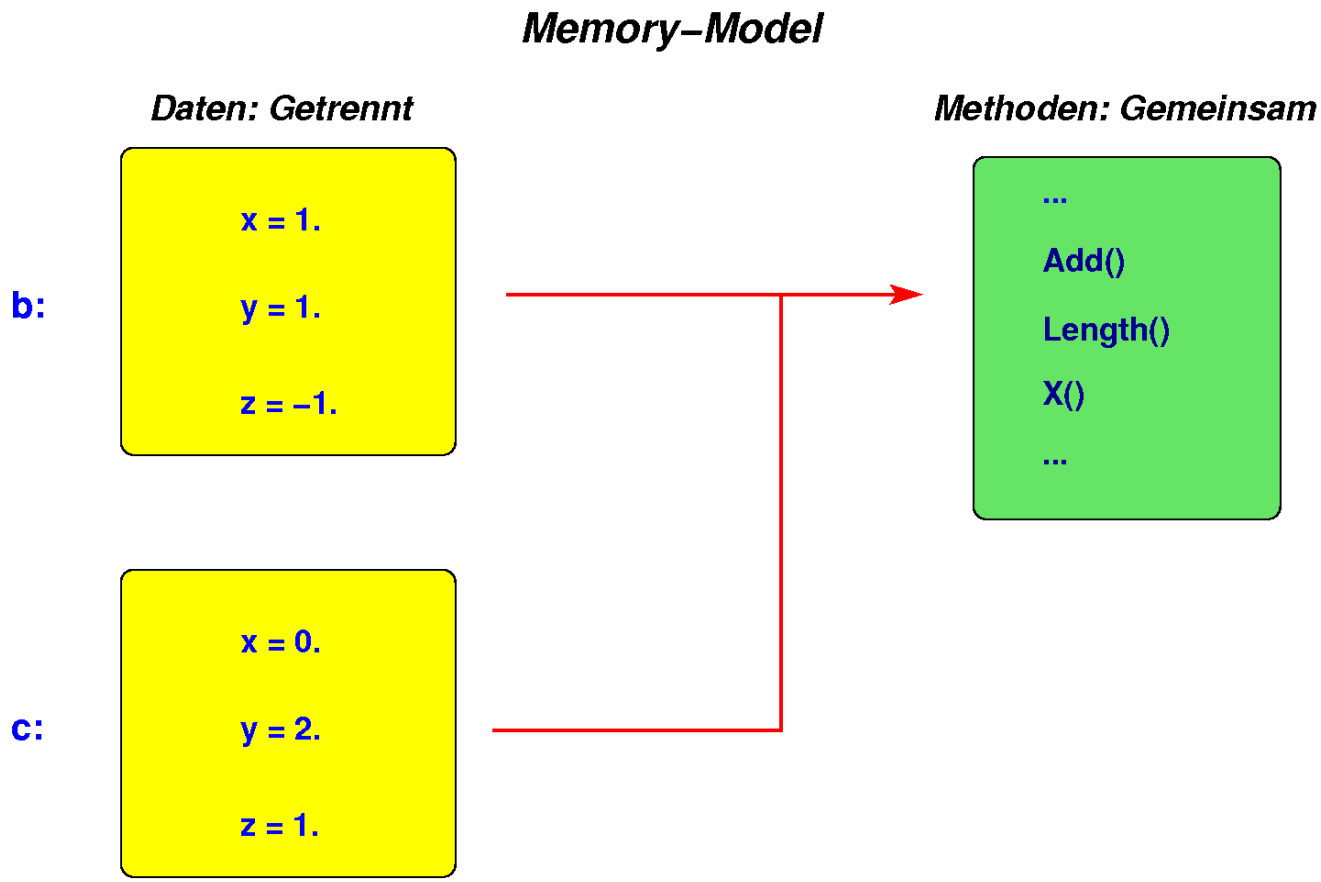
Aufruf ohne Argumente der Member-Function

```
double My3Vector::Length() { // get length of Vector
return( sqrt( x*x + y*y + z*z) );
```

`x, y, z` beziehen sich dann auf Objekt v (also `v.x, v.y, v.z`).

```
My3Vector b(1.,1.,–1.);

My3Vector c(0.,2.,1.);
```



1.5 Trennen von Deklaration und Implementierung

Übliche Praxis beim Programmieren in C++ ist es Deklarationen und Implementierung zu trennen, d.h. der C++ Code für Klassen wird aufgeteilt:

- **Klassen–Deklaration** in Header File, beinhaltet Variablen und Deklaration von Methoden (Funktions–Prototypen) z.B. *My3Vector.h:*

```
class My3Vector {
  // Variablen-deklarationen
  // Methoden-deklarationen ...  }
```

- **Methoden–Implementierung** in Source File, z.B. *My3Vector.cpp:*

```
double My3Vector::Length() { ...}
```

  Die *Syntax* ist `Klassenname::Methodenname`, wobei **::** der *scope resolution operator* ist.

  Damit wird ausgedrückt, dass es sich um eine Funktion handelt, die zu der angegebenen Klasse gehört (member–function) und nicht um eine *globale* C Funktion.

Praktische Konsequenzen

- Alle *Deklarationen* `class My3Vector { ...` wandern in eine **header Datei**, z.B. `My3Vector.h`
- Die *Implementierung* – der eigentliche Programm-code der Funktionen wird in eine separate **C–Datei** geschrieben, z.B. `My3Vector.cpp`
- Zum Compilieren von Code, der eine Klasse verwendet, genügt es das Header File, mittels `#include "My3Vector.h"` dazu zu nehmen.
- My3Vector Implementation getrennt kompilieren:
  `g++ -c My3Vector.cpp` $\Rightarrow$ *My3Vector.o*
- Dann Anwendungsprogramm kompilieren und mit *My3Vector.o* linken:
  `g++ -o t3Vector t3Vector.cpp My3Vector.o`
- **Vorteil:** Eigentliche Implementierung `(My3Vector.cpp)` muss nur **einmal** kompiliert werden *(zeitaufwendig!)*

Für unsere kleinen Testprogramme und Aufgaben nicht unbedingt nötig

Man kann auch Deklaration, Implementierung und Anwendung in ein gemeinsames File packen.

1.6 Initialisierung/Constructors

I.a. braucht eine Klasse einen constructor, das ist eine Methode mit dem Namen der Klasse, ohne weitere Typ-Angabe, und dient zum Initialisieren des Objekts.

In unserem Beispiel zwei Konstruktoren:

- **Default-constructor** ohne Argumente `My3Vector();`, wird benutzt bei Anlegen eines Objektes ohne Argumente, `My3Vector a`
- **Sonstiger Constructor** mit 3 double Argumenten,
  `My3Vector(double xv, double yv, double zv);`, wird entsprechend verwendet bei Anlegen eines Objektes wie z.B. `My3Vector b(1.,1.,-1.);`

  Konstruktoren werden implizit gerufen an der Stelle, an der im Programm ein entsprechendes Objekt erzeugt wird.

1.7 Data encapsulation – Motivation

Wichtiges Prinzip im objektorientierten Programmieren ist die *data encapsulation*:

- Datenelemente *(= Member–Variables)* einer Klasse sind als *private* deklariert
- **nur** *Member–Funktionen* der Klasse haben direkt Zugang
- von aussen (Anwendungsprogramm) nur über definierte Schnittstellen (= dedizierte Member–Functions), z.B. bei *My3Vector–Klasse* Koordinaten setzen nur via *Konstruktor* beim Erzeugen des Objektes und Koordinaten lesen via *X(), Y(), Z()* Member–Funktionen.

Nachteile (im Vergleich zum *public* Zugang):

- Mehraufwand beim Implementieren *X(), Y(), Z()* nötig
- Umständlicher in der Handhabung: My3Vector wird beim Erzeugen festgelegt *(Abhilfe: zusätzliche setX(..), setY(..), .. Methoden einführen)*
- Performance–Nachteile: Aufruf von Funktion `v.X()` dauert länger als direkter Zugriff `v.x`

Vorteile

- Zugang nur über wohldefinierte Schnittstellen
- unabhängig von spezifischer Implementierung
- grundlegendes OO Konzept: Klassen spezifiziert über **Verhalten (Was?)**, nicht die **Implementation (Wie?)**
- erlaubt flexible Änderung/Optimierung der Implementierung – **unabhängig** von Anwendungsprogrammen. 2 Beispiele zu Dreier–Vektor:
  - Flexible Wahl der internen Darstellung: *kartesische Koordinaten, Kugelkoordinaten, Zylinderkoordinaten*
    Implementierung der Methoden muss jeweils angepasst werden, aber keine Änderung bei **Deklaration/Aufruf**
  - Alternativ Erweiterung der Member–Variablen um Element `double Laenge`, wird bei Anlegen im Konstruktor berechnet $\Rightarrow$ Optimierung der rechen-intensiven Length()–Methode.

Data encapsulation ist OO-Prinzip aber kein Dogma. Data–members i.a. *private* aber in manchen Fällen *public* access durchaus vorzuziehen.

1.8 Operator overloading

In C++ können Standardoperationen, wie `=, +, *, etc` für beliebige Klassen definiert, d.h. 'überladen' werden.

Z.B. kann man in der `My3Vector` Klasse den Additionsoperator `+` definieren und dann einfach:

```
My3Vector a(1,1,0), b(1,-1,1);
My3Vector c = a + b;
My3Vector d = a.Add(b); // equivalent
```

Syntax für Deklaration/Implementierung zunächst verwirrend, aber analog zu Member-function call.

Im My3Vector–Header:

```
My3Vector operator + ( My3Vector & p );
My3Vector Add( My3Vector & p ) ;
```

Regel für binäre Operatoren:

- Objekt links vom Operator ist Objekt für das Operator wie Member-Funktion gerufen wird.
- Objekt rechts vom Operator wird als Argument an Operator Funktion üebergeben.

Implementierung:

```
My3Vector My3Vector::operator + (My3Vector & v )
{
  My3Vector tv;
  tv.x = x + v.x;
  tv.y = y + v.y;
  tv.z = z + v.z;
  return tv;
}
```

Mit Operator–Overloading ist C++ quasi **Compiler–Compiler**:
Man kann beliebige neue Datentypen erzeugen und dazu passende Operationen definieren/implementieren: *Matrix–Multiplikation, Vektor–Addition, u.v.m.*

Allerdings sorgfältiges Design wichtig:
Nicht immer ist es sinnvoll, z.B. Multiplikation `*` für `My3Vector`: `a * b ;`
Skalar– oder Kreuz–Produkt ? Nicht sinnvoll zu enscheiden $\Rightarrow$ Im Zweifelsfall lieber lassen.

C++ **Konventionen** sollen beachtet werden, z.B. `+=` Operator:

```
My3Vector a(1,1,-1), b(2,0,3):
a += b;
```

Als Operation sicher sinnvoll.

C++ Konvention ist, dass mehrere Zuweisungen möglich sind, z.B.

```
My3Vector d = a += b;
```

$\Rightarrow$ `+=` Operator muss Typ `My3Vector` zurückgeben.

Implementierung:

```
My3Vector & My3Vector::operator += (My3Vector & v ) {
  x += v.x;
  y += v.y;
  z += v.z;
  return *this;
}
```

Neues Syntax-element **this**:

“Verstecktes” Argument, ist Pointer auf das Objekt für das Funktion gerufen wird, d.h. für voriges Beispiel `a += b` gilt `My3Vector * this = &a` .

Rückgabe als Referenz verhindert unnötige Kopieraktionen.

1.9 non-member function and operators

Nicht immer kann Operator als member–function (d.h. Methode innerhalb der Klasse) implementiert werden, z.B: Operation 3–Vector $\times$ Scalar $\Leftrightarrow$ Scalar $\times$ 3–Vector

```
My3Vector a(1,1,0);
My3Vector c = a * 3.; // c = a.scale(3.)
My3Vector c = 3.  * a; // und nu ?
```

Deklaration ausserhalb der Klasse also globale Funktion:

```
My3Vector operator * ( double & c, My3Vector & v );
```

Für `My3Vector c = a * 3.;` beides möglich:

Member:

```
My3Vector operator * ( double & c ) ;
```

Non–member:

```
My3Vector operator * ( My3Vector & v, double & c );
```

Ähnliches Problem z.B für stream (`<<`) Operator: `cout << a ;`

Links von Operand steht Objekt von Typ *ostream*, rechts Typ *My3Vector*. *ostream* fester Bestandteil von C++ I/O, weiss nix von *My3Vector* und nicht vom User erweiterbar !

Einzige Alternative ist non-member Funktion:

```
// Deklaration in My3Vector.h
std::ostream & operator << ( std::ostream &, const My3Vector &);
// Implementation in My3Vector.cpp
#include <iostream>
std::ostream & operator << ( std::ostream &s, const My3Vector &v)
{
  s << "(" << v.x <<","<< v.y <<","<< v.z <<")";
  return s;
}
```

Hierbei muss Referenz auf Objekt vom Typ *ostream* zurückgegeben werden, damit Aneinanderreihen `cout << a << b << c << endl; ;` funktioniert

Aber:

Non-member functions haben **keinen** Zugriff auf `private` data members.

Ausweg:

- Geeignete (Lese-) Zugriffsfunktionen auf private data members vorsehen (z.B.: `My3Vector::X()`)

- Deklaration der Methode als **friend** in der Klasse:

```
friend std::ostream &operator << ( std::ostream &s, My3Vector &p);
```

Vergleich C++ und C

Mit den **My3Vectors** soweit jetzt diskutiert kann man sehr einfach und effizient Code für Dreier-Vektor Manipulationen schreiben, z.B. Vektoren addieren, mit Skalar multiplizieren, ...

```
My3Vector a(1,1,-1), b(2,0,3);
a += 3.*b;
```

Das *operator-overloading* macht dies besonders elegant. Aber auch wenn man darauf verzichtet (Java) ist es immer noch recht kompakt:

```
a.Increment( b.Scale(3.)  );
```

Wie würde man's ohne Klassen/Objekte in C lösen ?

Variante 1: Direkt kodieren:

```
double a[3] = { 1, 1, -1 };
double b[3] = { 2, 0, 3 };
int i;
for ( i=0; i<3; i++ ) {
  a[i] += 3. * b[i];
}
```

Variante 2: Entsprechende C Funktionen definieren/verwenden:

```
int main()
{
  double a[3] = { 1, 1, -1 };
  double b[3] = { 2, 0, 3 };
  double tmp[3];
  Scale3Vector( b, 3., tmp );
  Add3Vector( a, tmp, a );
}
void Scale3Vector( double vin[], double scale, double vout[] )
{ int i;
  for ( i=0; i<3; i++ ) {
    vout[i] = scale*vin[i];
  }
}
void Add3Vector( double v1[], double v2[], double vout[] )
{int i;
  for ( i=0; i<3; i++ ) {
    vout[i] = v1[i] + v2[i];
```

```
    }
}
```

Geht natuerlich auch in C, aber unhandlich, kryptisch, fehleranfällig, ...

1.10 Static Variablen und Methoden in Klassen

- **static Klassenvariablen** existieren nur einmal pro Klasse. D.h. jedes Objekt sieht diesselbe static Variable. Im Gegensatz dazu sind die normalen *(=automatic)* Variablen unabhängig für jedes Objekt

```
class My3Vector {
private:        // coordinates, hidden
  static int nvec;...
  double x;...
```

- **static Methoden** Diese Methoden "leben" ohne Objekt.

  Aufruf `classname::methode-name`

  Nützlich insbesondere für reine utility–Klassen ohne Daten

```
class MyMath { // define your own Math utility routines
public:
  static double sqrt( double );
  ...
double x = MyMath::sqrt(2.);
```

1.11 Const Variablen und Methoden in Klassen

Sorgfältige Verwendung **const** ist wichtig bei Klassen und Objekten *(bei bisherigen Beispielen wurde der Einfachheit halber darauf verzichtet)*.

- Objekt als Argument bei Funktionsaufruf $\Rightarrow$ bevorzugt als Referenz
  also keine Kopie, sondern "Original–Datum"
- Programmdesign: es soll klar ersichtlich sein, ob Funktion Objekt ändert oder nicht
  $\Rightarrow$ Übergabe als const wenn nicht
- Für konstante Objekte aber nur *konstante Methoden* rufbar, das sind Methoden, die die Member-Variablen nicht ändern. Dies wird explizit durch *const* am Ende der Deklaration angegeben:
  `double Length() const`
  $\Rightarrow$ Sorgfältiges Design der Klassenmethoden nötig, für *const* Objekte nur *const* Methoden rufbar!

```
// My3Vector declarations with const specifiers for arguments and methods
class My3Vector {
 private:          // coordinates, hidden
  double x;
  double y;
  double z;
 public:
  My3Vector(); // default constructor
```

```
  My3Vector(const double c1, const double c2, const double c3);// Other constructor
  // methods:
  // get length of vector
  double Length() const;
  // access elements
  double X() const;
  double Y() const;
  double Z() const;
  // add
  My3Vector Add( const My3Vector & p ) const;
  // scale with double
  void Scale( const double a ) {
    x *= a;   y *= a;   z *= a;
  };
};
```

Const Correctness – Motivation

Zitat Herb Sutter:

“Safety-incorrect riflemen are not long for this world. Nor are const- incorrect programmers, carpenters who dont have time for hard hats, and electricians who dont have time to identify the live wire. There is no excuse for ignoring the safety mechanisms provided with a product, and there is no excuse for programmers too lazy to write const-correct code.”

Objektübergabe als Referenz bei Methodenaufruf

Wenn Objekte als Argumente bei Aufruf von Funktionen oder Methoden übergeben werden empfiehlt es sich dringend die Objekt-Variable in der Funktions-Deklaration als Referenz anzugeben, also **nicht**

call-by-value

```
My3Vector Add(My3Vector p ) ;
```

sondern

```
My3Vector Add(My3Vector & p );
```

oder noch besser mit const:

```
My3Vector Add(const My3Vector & p ) const;
```

Unterschied:

- bei call-by-value wird bei Aufruf Kopie des Objektes gemacht, aufwendige Operation bei komplizierten Klassen !
- call-by-reference wesentlich effizienter (Aufwand unabhängig von Größe des Objektes).
- Allerdings **"Gefahr"** bei call-by-reference, dass Objekt verändert wird, deshalb **const** Verwendung wichtig.

1.12 Copy constructor, = Operator und Destructor

Diese drei Methoden sind essentielle Bestandteile jeder C++ Klasse. *Sie werden von C++ automatisch erzeugt falls nicht explizit definiert.*

- **copy constructor** Deklaration: `My3Vector(My3Vectcr & v);`
  und Verwendung: `My3Vector c(b);`
  `My3Vector c = b; // bei Initialisierung wird copy constructor verwendet`
- **= Operator** Deklaration: `My3Vector & operator = ( My3Vector & x)`
  und Verwendung:
  `My3Vector d;`
  `d = c;`
- **Destructor**: Deklaration `~My3Vector()` Wird implizit aufgerufen wenn Objekt gelöscht wird (out-of-scope geht).

Für bisherige Beispiele mit einfachen Member-Variablen automatische Erzeugung ausreichend:

- **Copy constructor** : Legt neues Objekt an und kopiert die Member-Variablen (`double x, y, z` bei My3Vector).
- **= Operator**: Kopiert die Member-Variablen (`double x, y, z` bei My3Vector).
- **destructor**: C++ sind Member-Variablen bekannt und räumt auf

Anders bei komplexeren Klassen:

Wenn Objekte ”Ressourcen” anlegen, z.B. *Dynamic Memory, File-Handles, Netzwerkverbindungen, ...*, muss man unbedingt selbst geeignete **Copy constructor, = Operator, Destructor** Methoden definieren, die die entsprechenden Resourcen verwalten (neu anlegen, kopieren, löschen, etc).

Die von C++ automatisch erzeugten sind **unzureichend!**.

Beispiel **Dynamic Memory**: Konstruktor reserviert Speicher und weist ihn Member-Variablen zu:

```
int * number = new int[100];
```

- Automatisch generierte Copy constructor bzw = Operator:
  Inhalt der Pointer Variable `int * number` wird in neues Objekt kopiert, d.h. zeigt auf dieselbe Speicherstelle wie ursprüngliches Objekt.
- Destructor: Objekt und zugehöriger Pointer verschwinden, allozierter Speicher bleibt übrig $\Rightarrow$ **Memory-leak!**

Explizite Implementierung nötig:

- Copy constructor: Neuen Speicherbereich anlegen, Speicherinhalt kopieren
- = Operator: Existierenden Speicherbereich ggf freigeben, neuen Speicherbereich anlegen, Speicherinhalt kopieren.
- Destructor: Angelegten Speicherbereich freigeben

1.13 Übung: Klasse BigInt

Ziel: Klassenbibliothek für beliebig grosse Integer Zahlen

” **412375397846372189046823946179846781468578175 57**”

Mit den `BigInts` soll operiert werden können wie mit gewöhnlichen Integers, z.B:

```
int main()
 {
   BigInt a = "234578997624315";
   BigInt b = "23987489367823";
   BigInt c = a + b;
   cout << c << endl;
   return 0;
 }
```

Dazu brauchen wir:

- Funktionen zum erzeugen, löschen, kopieren, zuweisen, lesen, ausgeben
- Grundrechenarten mit Standard-operatoren: $+, -, *, /$
- Operationen mit gewöhnlichen Integern
- Dynamische Speicherverwaltung, d.h. keine Beschränkung der Größe
  $\Rightarrow$ Copy constructor und **=** Operator sowie destructor nötig

Zunächst eine einfache Version, ohne dynamic memory:

```
// BigInt.hxx      header file for class BigInt;
class BigInt
{
private:
  int number[100];              // reserve string with 100 chars
  int ndig;                     // count digits
public:
  BigInt();                     // default constructor
  BigInt(const char * s);               // standard constructor
  //     BigInt(const BigInt & x ); // copy constructor, spaeter
  //     ~ BigInt();                    // destructor, spaeter
  void print() const;
  BigInt    operator + (const BigInt & x ) const;
  BigInt    operator - (const BigInt & x ) const;
  friend std::ostream &operator << ( std::ostream &s, const BigInt &x);
};
std::ostream &operator << ( std::ostream &s, const BigInt &x);
// BigInt.cxx       implementation file for class BigInt
```

```
#include <iostream>
#include <cstring>
#include "BigInt.hxx"
using namespace std;
#define MAX(a,b) ( (a) > (b) ? ( a ) : ( b ) )
BigInt::BigInt() {       ndig = 0; }
BigInt::BigInt(const char * str)
{
  int len = strlen(str);
  ndig = 0;
  while ( len >= 0 ) {
    int c = str[len-- --];
    if ( c >= '0' && c <= '9' ) {
      number[ndig++] = c - '0';
    }
  }
}
BigInt BigInt::operator + (const BigInt & x ) const
{
  BigInt t;
```

```
  t.ndig = MAX( this->ndig, x.ndig) + 1;
  int sum = 0, carry = 0;
  for ( int i = 0; i < t.ndig; i++ ) {
    carry = sum/10;
    sum = carry;
    if ( i < this->ndig )
        sum += this->number[i];
    if ( i < x.ndig )     sum += x.number[i];
    t.number[i] = sum % 10;
  }
  if ( carry == 0 )    t.ndig --;
  return t;
}
void BigInt::print() const
  {
    for ( int i = ndig; i > 0; i-- ) {
      cout << number[i-1];
      }
    cout << endl;
}
```

1.14 BigInt Variante mit Dynamic Memory

```
class BigInt
 {
 private:
   int * number;                  // simple pointer
   int ndig;                      // count digits
  ....
 // implementation file for class BigInt
 //
 BigInt::BigInt(char * str)
 {
   int len = strlen(str);
   number = new int[len+1];
```

Jetzt müssen Copy-Constructor, Zuweisungsoperator und Destructor definiert und (sorgfältig) implementiert werden

$\Rightarrow$ ansonsten Speicherüberschreiben, Memory–leaks, etc

```
class BigInt
 {
 private:
   int * number;          // simple pointer
   int ndig;              // count digits
  ....
   BigInt(const BigInt & x ); // copy const
   ~ BigInt(); // destructor
   BigInt & operator = (const BigInt & x);
  ....
 // implementation file for class BigInt
 //
 BigInt::BigInt(char * str)
 {
   int len = strlen(str);
   number = new int[len+1];// allocate memory
 ...
 BigInt::BigInt(const BigInt & b)
 { // copy constructor
   int len = b.ndig;
```

```
  number = new int[len];// allocate memory
... // Rest wie std constructor
BigInt & BigInt::operator = (const BigInt & b);
{ // = constructor
  int len = b.ndig;
  number = new int[len];// allocate memory
... // Rest wie std constructor
   return (*this);
}
BigInt::~ BigInt()
{ // destructor
  delete[ ] number;
  number = NULL;
...
}
```

1.15 Ergänzungen

Initialisierung vs Zuweisung im Constructor

In bisherigen Beispielen wurde Member Variablen durch Zuweisung im Constructor initialisiert:

```
// Initialisierung durch assignment in constructor body
 My3Vector::My3Vector() { // default constructor
    x = 0.; y = 0.; z = 0.; // set coords to 0.
 }
 My3Vector::My3Vector( double c1, double c2, double c3 ) {
    x = c1; y = c2; z = c3; // take args for coords
 }
```

Alternativ direkte Initialisierung mit spezieller Syntax *ausserhalb* des ”Constructor-Bodys”:

```
// Direckte Initialisierung ausserhalb von constructor body
My3Vector::My3Vector() : x(0.), y(0.), z(0.)          // default constructor
{ }
My3Vector::My3Vector( double c1, double c2, double c3 ) :
```

```
x(c1), y(c2), z(c3)
{ }
// bei Vererbung einzige Moeglichkeit Basisklassen–Konstruktor zu rufen
MyLVector::MyLVector(double c0, double c1, double c2, double c3) :
    My3Vector(c1, c2, c3), t(c0) // special initialization syntax
{}
```

In ThreeVector Beispiel ziemlich egal wie man es macht,
aber manchmal direkte Initialisierung erforderlich oder vorzuziehen:

- *const* member Variablen
- komplexe Objekte zur Initialisierung
- Initialisierung der Basisklasse bei Vererbung

this Pointer

Manchmal muss man in einer Member-Funktion explizit auf das Objekt verweisen oder zugreifen können,

- Objekt als Rückgabe-Wert (z.B. = Operator)
- Mehrdeutige Member-Variablen oder Funktions-Namen

dann expliziter Zugriff mit **this**: pointer auf Objekt. **this** ist immer automatisch deklariert.

```
// gleiche Namen fuer lokale Variablen und Member-variablen
 My3Vector::My3Vector( double x, double y, double z ) {
    this->x = x;  // Expliziter Zugriff auf member variable mit this
    this->y = y;  // x alleine ist lokale Variable (Funktionsargument)
    this->z = z;
 }
```

Gültigkeitsbereich von Variablen und Objekten

In C++ unterscheidet man den *scope* und die *lifetime* von Variablen.
scope: Wo ist ein Objekt bekannt ?

- Innerhalb des Blocks (`{ ...}`) oder der Funktion, wo es definiert ist.
- Globale Objekte, ausserhalb von Funktionen definiert, sind überall bekannt.
- Auf gleichem Block level **muss** Objektname eindeutig sein; in darunterliegenden *kann* er unabhängig verwendet werden.

```
#include <iostream>
void tscope_1( );    // declaration
void tscope_2( );
int glob_a = 42;     // define a global variable
int main()
{
  double a = 3.14;
  cout << "glob_a " << glob_a << endl;
  cout << "main a " << a << endl;
  tscope_1( );
  tscope_2( );
  {
    int a = -999; // override outside definition
    cout << "main in block: a " << a << endl;
  }
  cout << "main a " << a << endl; // what happened to a ?
}
void tscope_1( )
{
  cout << "tscope_1 glob_a " << glob_a << endl; // global variable is known
```

```
}
void tscope_2( ) {
  int glob_a = -1;   // override global variable
  cout << "tscope_2 glob_a " << glob_a << endl; // unless it's overridden
}
```

Lifetime: Wie lange 'lebt' ein Objekt ?

Hängt von Art der Variablen ab: *automatic, static, dynamic*.

automatic Default, von der Definition bis zum Ende des Blocks, bzw. während des Funktionsaufrufs.

```
int a;
double arr[100];
```

static Von Anfang bis Ende des Programms mit `static` Keyword

```
static int a;
```

dynamic Dynamisches Anlegen und Löschen mit Keywords `new` und `delete` unter der Kontrolle des Programmierers.

```
cin >>n; // Eingabe fuer n zur Laufzeit
double * arr = new double[n];
ThreeVector * tv = new ThreeVector( 1.0, -0.5, 0.7);
```

Beliebig grosse Arrays/Speicherbereiche können auf diese Weise zur **Laufzeit** angelegt werden.

Oft die **einzige** sinnvolle Möglichkeit Variablen oder Objekte über Funktionsgrenzen hinaus zu verwenden.

Nach Gebrauch wieder zurückgeben mit

```
delete [] arr;
delete tv;
```

Ansonsten **Memory Leaks !**

```
for ( int i=0; i<100000; i++ ) {
  double *p = new double[200000];
  ....
} // ohne delete [] p werden hier 20 GB benoetigt ...
```

Großes Problem bei komplexen C++ Software Projekten !

```
void tc1( );
void tc2( );
int main()
{ // demonstrate automatic, static, dynamic
  for ( int i=0; i<3; i++ ) {
    tc1();
    tc2();      }
  int * ap1 = new int[10];    // neuer int array mit 10 Elem.
  ap1[8] = 99;
  cout << "ap1[8] " << ap1[8] << endl;
  int * ap2 = new int(10);    // Vorsicht, [] vs (); hier 1 int mit Wert 10
  cout << "ap2[0] " << ap2[0] << endl;
  // Loeschen nicht vergessen;
  delete [] ap1;    // Wichtig: bei new ..[] auch delete []
  delete ap2;       //     sonst nur delete
  // delete muss innerhalb des Blocks erfolgen, sonst pointer weg,
  // aber nicht der allozierte Speicher !!
}
void tc1(    ) {
  int count = 0; // automatic counter
```

```
  count ++;
  cout << "tc1 " << count << endl;
}
void tc2( ) {
  static int count = 0; // static counter
  count ++;
  cout << "tc2 " << count << endl;
}
```

1.16 Aufgaben

(Aufgaben aus erstem Kurs für ca. 1.5 Tage, bitte nach Neigung auswählen)

1. **Vektor-Klasse**
   Entwerfen Sie ausgehend von den Beispielen die `My3Vector` Klasse. Zunächst können Sie alle Teile *(Deklaration, Implementierung der Methoden und Aufruf in* `main()` *)* in ein gemeinsames source file packen. Führen Sie einige zusätzliche 'sinnvolle' Methoden ein, z.B. Länge eines Vektors, Winkel zwischen zwei Vektoren, Skalarprodukt, Vektorprodukt.

   Lösungsbeispiel: Alles zusammen T3VectorAllInclusive.cpp (html, source),

2. **Vektor-Klasse aufteilen**
   Folgen Sie jetzt der C++ Konvention das Programm in drei verschiedene Files zu trennen: die Deklaration der My3Vector-Klasse kommt in eine *Header-Datei* (z.B. `My3Vector.h`), die Implementierung der Methoden geht in ein extra source file (z.B. `My3Vector.cpp`) und schliesslich die `main()` Funktion zum Testen in z.B. `T3Vector.cpp`

   Lösungsbeispiel: Header My3Vector.h (html, source), Implementation My3Vector.cpp (html, source), Testprogramm T3Vector.cpp (html, source)
   Kompilieren: `g++ -o T3Vector T3Vector.cpp My3Vector.cpp`

3. **Operator–Overloading**
   Implementieren Sie die Addition zweier Vektoren mit dem + Operator und die Multiplikation mit

```
double
```

```
int main()
{
  double x = 3.;
  My3Vector a(1.,1.,0.);
  My3Vector b(-1.,1.,0.);
  My3Vector c = a + b;
  My3Vector d = a * x; // von rechts
  My3Vector e = x * a; // von links
   }
```

Lösungsbeispiel: Header `myvec.h`, code `myvec.cpp`, Testprogramm `tvec.cpp`.

4. **Const correct Definiton der My3Vector Klasse**

   Implementieren Sie sorgfältig die **const** Spezifikation für Argumente und Methoden in der My3Vector Klasse.

   Und zur Anschauung noch die ThreeVector Klasse aus der CLHEP Klassenbibliothek (Utility Classes for Particle Physics): Header `ThreeVector.h`, source code `ThreeVector.cc`, `ThreeVector.icc`

5. **Statistik**
   Die Klasse StatCalc (html, source) implementiert einige grundlegende Statistikfunktionen, wie Mittelwert, Standardabweichung, ...
   Ein kurzes Testprogramm ist angehängt, das Zufallszahlen in ein `StatCalc` Objekt füllt und anschliesend die Statistik–größen ausgibt.

   (a) Probieren Sie das Programm aus und Erweitern dann die `StatCalc` Klasse um Methoden zur Berechnung und Ausgabe von Minimum und Maximum.
   (c) In `semester.dat` finden Sie die Semesterzahl bis zum Physikdiplom für zufällig ausgewählte Studenten. Die ersten 100 Einträge sind von Studenten der LMU, die restlichen 100 von Studenten der TUM. Lesen Sie die Daten ein und füllen LMU bzw TUM Zahlen jeweils in ein `StatCalc` Objekt. Sind die Mittelwerte im Rahmen der Schwankungen konsistent ?
   (d) Überlegen Sie wie man das Problem aus (c) (mehrere Statistiken parallel führen) in einer prozeduralen Sprache (Fortran, C) angehen könnte, d.h. ohne Klassen und Objekte, nur mit Arrays und Funktionen.

6. **BigInts, statisch**
   Entwerfen Sie die Klasse `BigInt` zunächst statisch, so wie im Beispiel vorgegeben. Implementieren Sie den `<<` operator und wenn Sie Zeit/Lust haben auch noch Subtraktion, Multiplikation und Division.

7. **BigInts, dynamisch**

Modifizieren Sie die `BigInt` Klasse, so dass Speicher dynamisch zugewiesen wird. Jetzt müssen auch Copy-constructor, `=` operator und destructor implementiert sein.

Lösungsbeispiel: Header `BigInt.h`, code `BigInt.cpp`, Testprogramm `tBigInt.cpp`.

8. **Mondlandung**
   Ein Spieleklassiker ist die Mondlandung:
   - Raumfähre wird von Mond angezogen
   - Mit Gegen-Schub kann man Fall kontrollieren, allerdings sind Treibstoffvorräte begrenzt
   - Ziel ist möglichst weiche Landung auf Mond.

   Erstellen Sie eine C++ Klasse für Mondfähre, welche Datenelemente, welche Methoden werden benötigt?

   Lösungsbeispiel aus Buch *Coding for Fun mit C++* `mondlandung.cpp`.

2 Objektorientiertes Programmieren – Vererbung und Polymorphismus

- Vererbung
- Virtual Functions
- Polymorphismus
- Abstrakte Klassen
- Interfaces
- Aufgaben

2.1 Vererbung

In der Physik sind häufig Vierer–Vektoren *(=Lorentz–Vektor)* gefragt, d.h. Dreier–Vektoren erweitert um Zeit bzw. Energie–Komponente. Mögliche LorentzVektor Klasse:

```
class MyLVector {
private:            // coordinates, hidden
  double t;   double x;   double y;   double z;
public:
  MyLVector();// The default constructor
  MyLVector(double c0, double c1, double c2, double c3);// Other constructor
  // methods:
  double Length() const;  //     get length of vector
  double T() const;
  double X() const;
  double Y() const;
  double Z() const;
  MyLVector Add( const MyLVector & p ) const; // add two vectors
  double Angle( const MyLVector & p ) const; // angle between   two vectors
  double Mass() const;  //     get mass of 4-vector
};
```

Vierer–Vektor *kann* als Erweiterung des Dreier–Vektor angesehen werden. Viele Gemeinsamkeiten mit *Dreier–Vector* und ein paar Erweiterungen

- 3 alte, 1 neues Datenelement
- einige Methoden identisch *X(), Angle()*
- einige komplett neu (Masse zweier 4-Vectors)
- einige müssen neu implementiert werden (Add, ...)

Design Prinzip **Code-Reuse** statt cut&paste !

Also vielleicht besser *My3Vector* in *MyLVector* benutzen:

```
class MyLVector {
private:            // coordinates, hidden
  double t;
  My3Vector vec3;  // My3Vector as private member
public:
  MyLVector();// The default constructor
  MyLVector(double c0, double c1, double c2, double c3);// Other constructor
  // methods:
  double Length() const;  //     get length of vector
  double T() const;
  double X() const;
  double Y() const;
  double Z() const;
  MyLVector Add( const MyLVector & p ) const; // add two vectors
  double Angle( const MyLVector & p ) const; // angle between   two vectors
  //     ....
};
```

$\Rightarrow$ LorentzVektor enthält DreierVektor: **"has-a"** relationship *(Aggregation)*.

```
// implementation
// Constructor
MyLVector::MyLVector(double c0, double c1, double c2, double c3) :
  vec3(c1, c2, c3), t(c0) // special initialization syntax
{}
double MyLVector::Angle( const MyLVector & p ) const
{
  return( vec3.Angle(p.vec3)); // re-use code
}
double MyLVector::X() const
{
  return( vec3.X() ); // re-use code
}
```

Im Prinzip ok, allerdings viele stumpfsinnige *Mapping–Funktionen* von DreierVektor auf ViererVektor nötig.

3. Möglichkeit: **Vererbung**

ViererVektor **ist** *DreierVektor* mit ein paar Ergänzungen ....

```
#include "My3Vector.h" // My3Vector declarations
class MyLVector : public My3Vector { // inherit from My3Vector
 private:
   double t;
 public:
   MyLVector();// The default constructor
   MyLVector(double c0, double c1, double c2, double c3);// Other constructor
   // methods:
   double Mass() const; //    get mass of 4-vector
   double Mass(const MyLVector & p) const; //    get mass of two 4-vector
 };
// implementation
MyLVector::MyLVector(double c0, double c1, double c2, double c3) :
  My3Vector(c1, c2, c3), t(c0) // special initialization syntax
{}
 double MyLVector::Mass() const // Method for mass:
 {
```

```
  return( sqrt( t*t − x*x − y*y − z*z ) );
}
```

Jetzt übernimmt bzw. **erbt** *MyLVector* alle Funktionen von *My3Vector*, z.B. Winkelberechnung funktioniert sofort:

```
MyLVector c(1.000001,1.,0.,0.), d(2.,1.,1.,0.);
c.Angle(d);
```

Falls Methode geändert werden muss, z.B. `MyLVector::Add`, entsprechend neu implementieren, bei `c.Add(d)` wird dann die *MyLVector* Version benutzt.

Vererbung (**"is-a"** relationship) bedeutet alle Eigenschaften und Funktionen einer Grund-Klasse (base class) in eine weitere Klasse (derived class) zu übernehmen. Daraus ergibt sich eine enorme Erweiterung der Funktionalität. Ausgehend von vorhandenen, simplen Grundklassen können relativ leicht neue Klassen abgeleitet werden, ohne jedesmal diesselben grundlegenden Funktionen neu zu implementieren.

Allerdings: Vererbung nicht übertreiben, nicht immer sinnvoll.

Exkurs: Rechteck und Quadrat

Zunächst naheliegend *Quadrat* von *Rechteck* abzuleiten (=vererben), viele Funktionen und Eigenschaften gemeinsam. Aber grundsätzliche Probleme bei Verhalten, z.B.:

```
class Rechteck {
  // ...
  void setLength( double);
  void setWidth( double);
  // ...
  double length, width;
};
```

Was soll ein *Quadrat* mit diesen Funktionen machen ?

`setLength(..)` bei *Quadrat* ändert immer auch `width` und vice-versa. Verhalten nicht kompatibel mit *Rechteck* .

Wichtig für OO Programmieren ist **konsistentes Verhalten** von Klassen, dazu wird Vererbung eingesetzt. Es geht weniger um *bequemes* Übernehmen von Funktionalität.

Mathematisch ist Quadrat Unterklasse von Rechteck, aber nicht im Sinne von **OOP** !

Weiteres zu Vererbung:

- Spezielle Syntax zur Initialisierung der Basis–Klasse:
  ```
  MyLVector::MyLVector(double c0, double c1, double c2, double c3) :
  My3Vector(c1, c2, c3), t(c0) { ...}
  ```
  muss **vor** normalem Code-Block stehen.

- *Abgeleitete* Klasse hat keinen Zugriff auf **private** Variablen/Methoden der Basisklasse.

  Abhilfe: Weiterer Modifier **protected**, im Prinzip analog zu **private**, d.h. kein Zugriff von ausserhalb der Klasse, jedoch mit Ausnahme von abgeleiteten Klassen.

2.2 Virtual Functions und Polymorphismus

Überschreiben von Funktionen, wann wird Funktion aus Basis–Klasse und wann aus abgeleiteter Klasse gerufen ?

```
int main()
{
  My3Vector tvec(1.4, 0.7, 0.6);
  MyLVector lvec( 2.1, 1., 1., 0.5);
  My3Vector *pv;   // pointer to three-vector
  MyLVector *pl;   // pointer to lorentz-vector
  cout << tvec.Length() << endl; // Length() aus My3Vector wird benutzt
  cout << lvec.Length() << endl; // Length() aus MyLVector wird benutzt
  pv = &tvec;
  cout << pv->Length() << endl; // Length() aus My3Vector wird benutzt
  pl = &lvec;
  cout << pl->Length() << endl; // Length() aus MyLVector wird benutzt
  pv = &lvec;   // legal, Basisklassen–Pointer kann auf abgeleitetes Objekt zeigen
  cout << pv->Length() << endl; // welches Length() wird gerufen ???
}
```

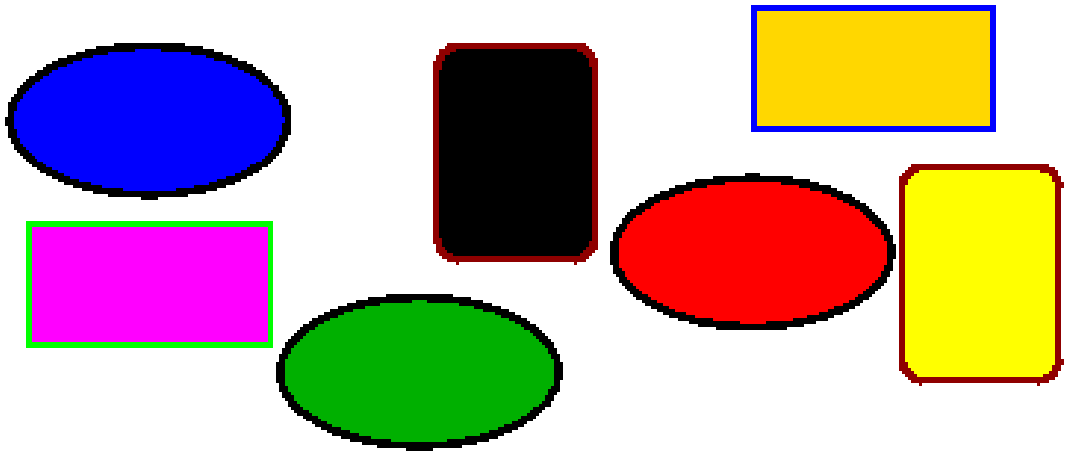
Abhängig von *Deklaration* der Funktion in **Basisklasse**:

- Nach bisherigem Beispiel wird `My3Vector::Length()` benutzt.
- Aber Deklaration mit keyword **virtual**
  `virtual double Length() const; // Laenge des 4-vector`
  bewirkt, dass `MyLVector::Length()` benutzt wird, wenn das Objekt ein `MyLVector` ist.
- Erst zur *Laufzeit* wird bei Aufruf der Funktion über eine sog. *virtual function table* die dazugehörige Funktion gesucht.

  D.h. wenn ein Pointer vom Typ `My3Vector *` für ein Objekt gerufen wird, das in Wirklichkeit ein `MyLVector` ist, dann wird auch die zugehörige Funktion `MyLVector::Length()` benutzt.

Wozu virtual Functions ?

Vererbung ist vor allem dann sehr sinnvoll wenn von einer allgemeinen Basisklasse mehrere Klassen ableiten.



Beispiel Grafik: Die grafischen Objekte Rectangle, Oval, RoundRectangle, haben bestimmte Eigenschaften und Funktionen gemeinsam, z.B. die *Farbe, Position, etc.*
$\Rightarrow$ abgeleitet von Basisklasse Shape.

Andere Funktionen, wie z.B. *zeichnen* sind spezifisch für jede Klasse und müssen spezifisch implementiert werden.

```
class Shape {
  //..
```

```
  Color color;      // Color of the shape.
  void setColor(Color newColor) {
    // Method to change the color of the shape.
    color = newColor; // change value of instance variable
  }
  virtual void Draw() { ...} // declare as virtual
  //..
};
class Rectangle : public Shape {
      void Draw() {
        drawRect . . .   // commands for drawing a rectangle
      }
      . . . // possibly, more methods and variables
}
class Oval  : public Shape {
      void Draw() {
        drawEllipse. . .
      }
}
class RoundRect : public Shape {
```

```
    void Draw() {
      drawRoundRect . . .
    }
}
```

Dann generische Verwendung als *shape* möglich:

```
int main ()
{
  vector<Shape*> vecshape; // vector with base-class pointers
  // add some concrete shapes
  vecshape.push_back( new Rectangle());
  vecshape.push_back( new Oval());
  vecshape.push_back( new Rectangle());
  vecshape.push_back( new RoundRect());
  vecshape.push_back( new Oval());
  vecshape.push_back( new Oval());
  // make the shapes all red, move and draw them
  for ( int i=0; i<vecshape.size(); i++ ){
    vecshape[i]->setColor( red ); // Shape-func
```

```
    vecshape[i]->Move( ... );        // Shape-func
    vecshape[i]->Draw( ... );        // Oval/Rectangle/.. -func
  }
  //...
}
```

⇒ Polymorphismus:

Unterschiedliche Objekte mit gemeinsamer *Basisklasse* können über diese Basisklasse angesprochen werden, d.h. es genügt ein Pointer vom Typ der Basisklasse. Funktionen, die als **virtual** deklariert sind, werden automatisch dem jeweiligen Objekt zugeordnet. Diese Zuordnung findet erst zur Laufzeit statt *(late–binding)* über die *virtual function table.*

```
// ...
void gimeSomeShapes( vector<Shape*> &vs, int nshapes )
{
    // creat concrete shapes randomly and add to vector
  for ( int i=0; i<nshapes; i++) {
    int irand = rand()*3./(RAND_MAX+1.0); // Zufallszahlen [0..3)
    switch (irand ) {
    case 0:
      vs.push_back( new Rectangle());
      break;
    case 1:
      vs.push_back( new Oval());
      break;
    case 2:
      vs.push_back( new RoundRect());
      break;
    }
  }
}
```

```
int main ()
{
  vector<Shape*> vecshape; // vector with base-class pointers
  gimeSomeShapes( vecshape, 20 ); // erzeuge shapes
  // make the shapes all red, move and draw them
  for ( int i=0; i<vecshape.size(); i++ ){
    vecshape[i]->setColor( red ); // Shape-func
    vecshape[i]->Move( ... );      // Shape-func
    // Oval/Rectangle/.. -func, richtige wird automatisch gesucht
    vecshape[i]->Draw( ... );
  }
  //...
}
```

Äusserst nützliches Konzept:

Leichte Erweiterbarkeit: man kann später weitere Klassen (*Triangle, Pentagon, ...*) zufügen ohne nutzende Funktionen oder `Shape` ändern zu müssen.

OOP: *"old code calls new code"* versus **prozedural:** *"new code calls old code"*

2.3 Pure virtual Functions und abstrakte Klassen

In der Beispiel–Klasse *Shape* wurde nichts weiter über die *Draw()* Methode gesagt
$\Rightarrow$ geht auch nicht, weil *Shape* diese Methode nicht sinnvoll implementieren kann.

Was dann ?

```
Shape::Draw() { cout <<"Shapes can't draw ";}
```

Aber besser: Deklaration als **pure virtual** Funktion in `Class Shape { ...}` mit

```
virtual Draw() = 0;
```

Die Existenz einer **pure virtual** Funktion in einer Klasse hat drastische Konsequenzen:

- Klasse wird zu einer **abstrakten** Klasse. Das bedeutet dass keine Instanzen der Klasse erzeugt werden können, es kann kein `Shape s;` angelegt werden, eben weil die Klasse eine **pure virtual** Funktion enthält.
- Jede konkrete Klasse, die von einer abstrakten Klasse erbt, **muss** die *pure virtual functions* überschreiben, sonst Fehler beim kompilieren, wenn ein Objekt angelegt werden soll.
- **Aber:** Pointer auf abstrakte Klassen sind möglich, d.h. alle grafischen Objekte (Rechteck, ..) können in Array/Vektor von Typ `Shape *` gesteckt werden.

2.4 Pure abstract Classes oder Interfaces

sind Klassen die nur **pure virtual functions** enthalten.

Das erscheint erstmal ziemlich sinnlos, wozu soll Klasse, die nichts implementiert, gut sein?

Eine **pure abstract class** macht Vorschriften, d.h. Klassen, die davon erben, sind gezwungen die vorgegebenen Methoden zu implementieren.

Bei grossen, komplexen Projekten von 1,000,000 Zeilen code ist ein Hauptproblem die Definition einheitlicher Schnittstellen zur Kommunikation der einzelnen Komponenten und die Minimierung von Abhängigkeiten. Dafür ist das Konzept der **pure abstract class** oder **interface in Java**, das einheitliches Verhalten erzwingt, extrem hilfreich.

Interfaces als Funktions–Platzhalter in der Numerik

In der Numerik tritt es häufig auf, dass man einen allgemeinen Algorithmus implementieren möchte, den man auf viele Fälle (*beliebige Funktionen*) anwenden möchte. Z.B. **Nullstellenbestimmung** (Bisektion, Gauss-Newton), **numerische Integration** (Trapez, Simpson), ...

In Fortran/C löst man sowas mit *function–pointers* das sind ziemlich unangenehme, schwer verständliche Konstrukte.

In C++/JAVA/OO lässt sich das elegant und einfach mit **interfaces** (bzw. **pure abstract class**) lösen:

```
#include <cmath>
class Func { // pure abstract class
public:
  virtual double value( double x) = 0;
};
class Nullstelle {
  // class with just one method
public:
  // finde Nullstelle von f nach Bisektionsverfahren
  // f ist Objekt das interface Func implementiert, d.h. eine Methode double value(double) hat
  double find( double x1, double x2, Func &f ) {
    double f1 = f.value(x1);
    double f2 = f.value(x2);
    double xn = x1;
    double fn = f1;
```

```
    for ( int i=0; i<1000 && fabs(fn) > 1e-6; i++ ) {  // Abbruch nach 1000 Iterationen oder bis auf 1e-6 an Nullstelle
      xn = (x1+x2)/2.;              // Neues x in Mitte
      fn = f.value(xn);             // Funktionswert dazu
      if ( fn*f2 > 0. ) {           // gleiches Vorzeichen wir f2 ?
    x2 = xn;
    f2 = fn;                  // dann ersetze x2,f2
      }
      else {
    x1 = xn;                  // andernfalls x1,f1
    f1 = fn;
      }
    }
    return( xn );
  }
};
```

Das Programm in Klasse `Nullstelle` weiss nicht's über die eigentliche Funktion die es benutzt, es kennt nur die *pure abstract class* `Func`.

Erst bei Aufruf von `ns.find(...)` in einem Anwendungsprogramm wird festgelegt welches Objekt gerufen wird:

```
#include <iostream>
#include <cmath>
```

```
#include "Nullstelle.cpp"
using namespace std;
class TFunc1 : public Func { // concrete class 1
public:
  double value( double x) { // define method
    return( cos(x) −x);
  }
};
class TFunc2 : public Func { // concrete class 2
public:
  double value( double x) { // define method
    return( exp(x) −x);
  }
};
int main() {
  double x1 = 0., x2 = 1.6; // Startintervall
  TFunc1 t1;
  TFunc2 t2;
  Nullstelle ns;
  cout << "Nullstelle = " << ns.find( x1, x2, t1 ) << endl;
```

```
  cout << "Nullstelle = " << ns.find( x1, x2, t2 ) << endl;
}
```

2.5 Ergänzungen

Virtual Destructor:

Basisklassen, die *virtual functions* enthalten, also potentiell **polymorph** benutzt, werden sollten **immer** einen **virtual destructor** deklarieren. Nur dann ist garantiert, dass auch Destructor der abgeleitetem Klasse gerufen wird $\Rightarrow$ sonst **memory leak!**

```
class Shape {
  //..
  virtual ~Shape() =0; // declare virtual destructor
  //..
};
class Rectangle : public Shape {
  TwoDPoint * points;
  Rectangle () {
    points = new TwoDPoint[4]; // allocate memory for points
  };
  ~Rectangle () {
    delete [] points;
  }
  // ..
}
```

Casting und Run-Time-Type ID:

Casting *(= Typumwandlung)* sollte möglichst wenig benutzt werden, ist aber manchmal unvermeidlich.
Im letzten Kurs wurden **C-Style** casts gezeigt:

```
int n = (int) 30.65;
```

Diese Art von casts sollte in C++ nicht mehr verwendet werden, stattdessen

- **static_cast < type > ()** entspricht i.W. dem C cast:
  `int n = static_cast <int> (30.65) ;` C++ Compiler akzeptiert **static_cast** wenn ursprünglicher Typ und neuer Typ *verwandt* sind, z.B.
  double ⇒ int , int ⇒ double, Rectangle ⇒ Shape, Shape ⇒ Rectangle , ...
  Aber nicht: *double ⇒ Rectangle, ...*
- **reinterpret_cast < type > ()** **erzingt Neu–Interpretation**, damit auch z.B. *double ⇒ Rectangle* möglich:
  `double x = 0.7; Rectangle r = reinterpret_cast <Rectangle> (x) ;`
  Sehr gefährlich, nur in Ausnahmefällen benutzen !
- **dynamic_cast < type > ()** Nützlich im Zusammenhang mit Polymorphismus: Umwandlung wird nur gemacht wenn *type* stimmt:

```
vector<Shape*> vecshape; // vector with base-class pointers
// add some concrete shapes
```

```
vecshape.push_back( new Rectangle());
vecshape.push_back( new Oval());
vecshape.push_back( new Rectangle());
vecshape.push_back( new RoundRect());
vecshape.push_back( new Oval());
vecshape.push_back( new Oval());
// make the shapes all red, move and draw them
for ( int i=0; i<vecshape.size(); i++ ){
  Rectangle * rp;
  rp = dynamic_cast<Rectangle *>( vecshape[3] );
  if ( rp ) {                                  // cast ok for i=0,2
    cout << " Can cast into rectangle" << endl;
    rp->Draw();
  }
  else {                                       // cast not ok for i=1,3,4,5
    cout << " Cannot cast into rectangle" << endl;
  }
     }
```

Damit Typ-Identifizierung zur Laufzeit möglich (**= RTTI**) !

- **Up-Cast** geht immer : Abgeleitete Klasse $\Rightarrow$ Basisklasse.
- **Down-Cast** : Basis-Klasse $\Rightarrow$ Abgeleitet Klasse. Mit **static_cast** erzwingen auch wenn falsch, besser mit **dynamic_cast** testen und ausführen.

2.6 Aufgaben

1. **Vererbung**
   Leiten Sie die Lorentz-Vektor Klasse (Vierer-Vektor $= (E, p_x, p_y, p_z) = (E, \vec{p})$) von der Dreier-Vektor Klasse ab.
   Lorentz-Vektoren sollten als zusätzliche Methoden die Masse berechnen können:

$$M = \sqrt{E^2 - \vec{p}^2}$$

   sowie die invariante Masse zweier Lorentzvektoren:

$$M_{inv} = \sqrt{(E_1 + E_2)^2 - (\vec{p_1} + \vec{p_2})^2}$$

```
int main()
 {
   MyLVector c(1.000001,1.,0.,0.);
   MyLVector d(2.,1.,1.,0.);
   MyLVector e(1.000001,-1.,0.,0.);
   cout << "angle c, e = " << c.Angle(e) << endl; // My3Vector-Methode
   cout << "mass d = " << d.Mass() << endl; // MyLVector-Methode
   cout << "mass c+e = " << c.Mass(e) << endl;
```

...

Lösungsbeispiel: Header `MyLVector.h`, Code `MyLVector.cpp`, Testprogramm `TLVector.cpp`.

2. **Vererbung und Polymorphsimus 1:**

Übernommen von http://www.lrz-muenchen.de/ẽbner/C++/Uebung/aufgaben_004.html

Das source file `particles.C` (html, src) enthält ein Grundgerüst zu einer allgemeinen Basisklasse *Particle* und davon abgeleiteter Klassen.

(a) Definieren Sie die deklarierten Konstruktoren jeder Klasse.

(b) Erstellen Sie eine Member-Funktion
`void move (double dx, double dy, double dz)` ,
die ein Particle um die angegebenen Werte in den drei Koordinaten verschiebt (also x+dx, y+dy, z+dz).
Hinweis: Überlegen Sie sich, wo diese Funktion überall deklariert/definiert werden muss, um möglichst wenig Arbeit zu haben. Zum Test löschen Sie die Kommentarzeichen in Schleife 1 in main().

(c) Wie erreicht man, dass die Schleife 2 in `main()` immer die richtige `toString()` Funktion wählt, also z.B. tatsächlich ein `Molecule` ausgibt, wenn sich hinter dem `Particle*` Pointer ein solches verbirgt?

(d) Ein Chemiker möchte natürlich immer wissen, aus welchen Elementen ein Particle besteht. Es soll also nur möglich sein, Atome und Moleküle zu erzeugen, nicht aber ein ”abstraktes” Particle. *(Sorgen Sie dafür, dass die Particle-Klasse abstrakt ist.)*

Ergänzung: In dem Beispiel wird bei der Klasse `ChargedMolecule` **multiple inheritance** ver-

wendet. Das ist ziemlich heikel, insbesondere wenn, wie in dem Fall, die Vererbung auf eine gemeinsame Basisklasse zurückgeht. Eine Diskussion dazu findet sich z.B. hier

3. **Vererbung und Polymorphsimus II:**

   (a) Laden Sie sich diese source files herunter. Schauen Sie sich Verhalten und Probleme beim Kompilieren bzw. zur Laufzeit genau an. Ändern Sie die Programme falls Compilier Fehler auftreten.

   - `shape1.cpp`
   - `shape2.cpp`
   - `shape3.cpp`

   (b) **Static und Dynamic Casting:**

   Wie zuvor source files herunter laden und versuchen zum laufen zu bringen:

   - `shape4.cpp`
   - `shape5.cpp`
   - `shape6.cpp`

4. **Pure abstract class/Interface**

   (a) Implementieren Sie das vorgestellte Beispiel mit der Nullstellen-Suche, benutzen Sie andere Funktionen, z.B. $e^x - x^{10}$, ...

   (b) Nur die wenigsten physikalischen Fragestellungen lassen sich einfach lösen, auch bei vglw. simplen Problemen wird die analytische Lösung schnell ziemlich unhandlich. Ein klassisches Beispiel ist der **Schiefe Wurf**: Ein Ball wird mit 10 m/s geworfen, die Anfangshöhe ist 2 m, bei welchem Winkel wird die maximale Weite erreicht?

3 Objektorientiertes Programmieren – Allgemeines

Zitat: **Eine objektorientierte Programmiersprache ist weder eine notwendige noch eine hinreichende Bedingung für objektorientiertes Programmieren**

Objektorientiertes Programmieren heisst

- Denken, Modellieren, Designen, Implementieren in Objekten
- Objekte kommunizieren über definierte Interfaces direkt miteinander
- anstatt sequentiellem Aufruf aus Hauptprogramm
- OO Sprache wie Java oder C++ äusserst hilfreich für Umsetzung aber Grund–Idee auch in C oder Pascal möglich.
- Prozedurales Programmieren oder Spaghetti–Code auch in C++ oder Java möglich (*"Ich programmiere Fortran, egal in welcher Sprache"*)

3.1 UML – Unified Modelling Language

"The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a systems blueprints, including conceptual things like business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

Grady Booch, Ivar Jacobsen, Jim Rumbaugh

(OMG Unified Modelling Language Specification, Version 1.3, March 2000)

3.2 Wozu UML ?

“Standardsprache” um

- Software–Systeme darzustellen *(anders als im source-code editor)*
- Abhängigkeiten und Beziehungen auszudrücken
- Abläufe zu visualisieren
- Anforderungen zu erarbeiten
- Weitergehende Abstrahierung

Für Physiker nichts Neues:

- **Mathematik** zur abstrakten, formalen Beschreibung der Natur
- Feynman-Diagramme für Teilchenphysik–Reaktionen.

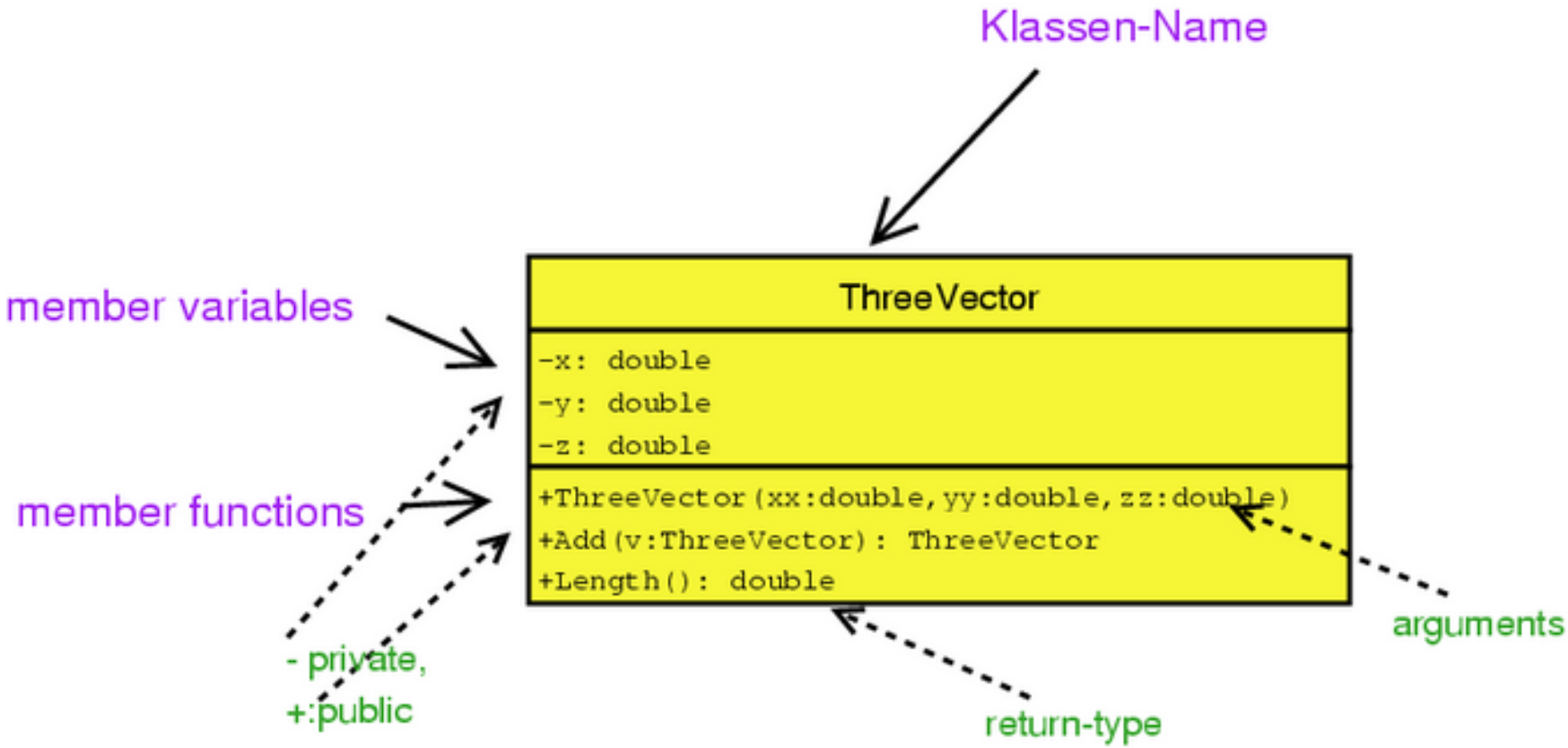
3.3 Klassen in UML

Klassen beschreiben Objekte:

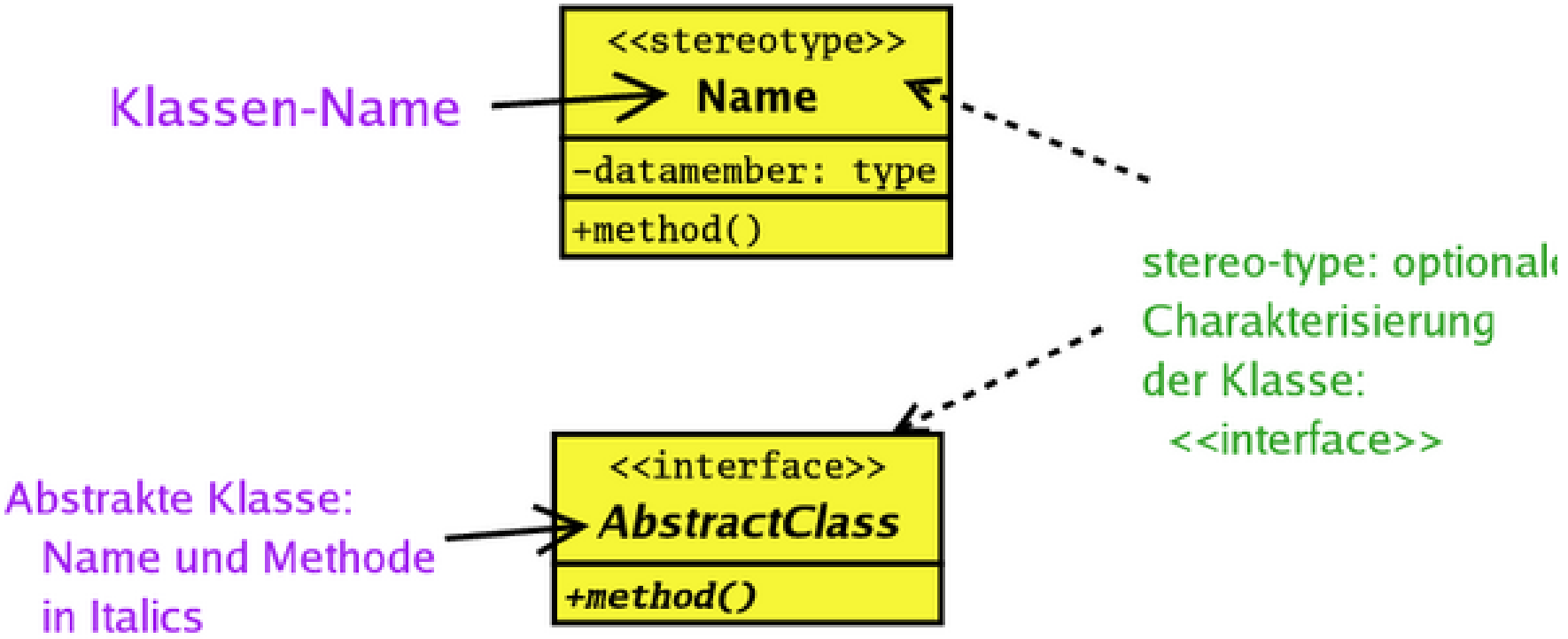
- Status: Werte der Member-Variablen
- Konstruktor: Erzeugen des Objektes
- Interface: Signatur der member functions
- Verhalten: Implementierung der Member functions

Klassen bzw. Objekte haben **Beziehungen** zu anderen Klassen/Objekten

UML Klasse

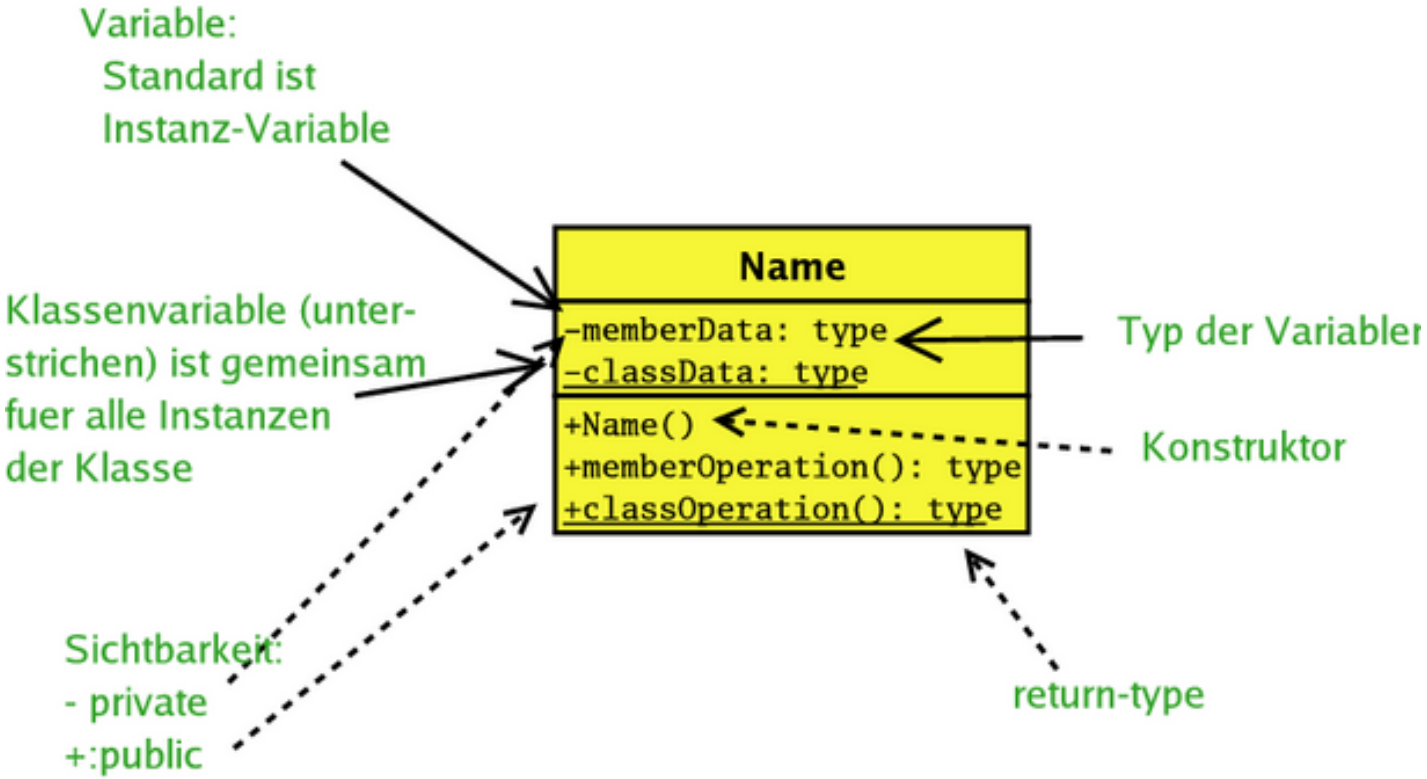

Klassen-Name
ThreeVector
member variables
-x: double
-y: double
-z: double
member functions
+ThreeVector(xx:double,yy:double,zz:double)
+Add(v:ThreeVector): ThreeVector
+Length(): double
arguments
- private,
+:public
return-type

Name der Klasse



Klassen–Attribute und –Operationen

Attribute sind die Instanz–Variablen (oder Klassen–Variablen falls *static* deklariert).



Syntax: `visibility name : type`

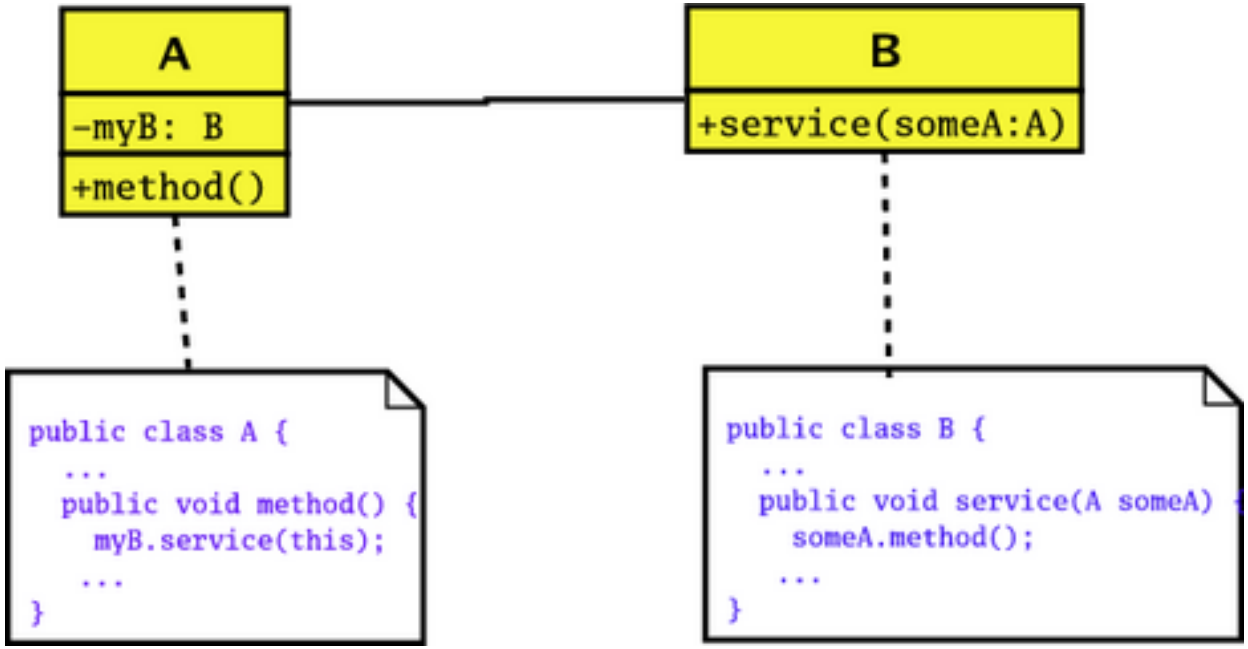
3.4 Beziehungen zwischen Klassen

Klassen bzw. Objekte können auf vielerlei Arten voneinander abhängen:

- **Association:** Nicht näher spezifizierte (i.d.R. lose) Kopplung von Klassen, nur in eine Richtung oder bi-direktional
- **Aggregation bzw. Composition:** Eine Klasse enthält ein (oder mehrere) Objekte einer anderen Klasse.
- **Inheritance**

Binary Association

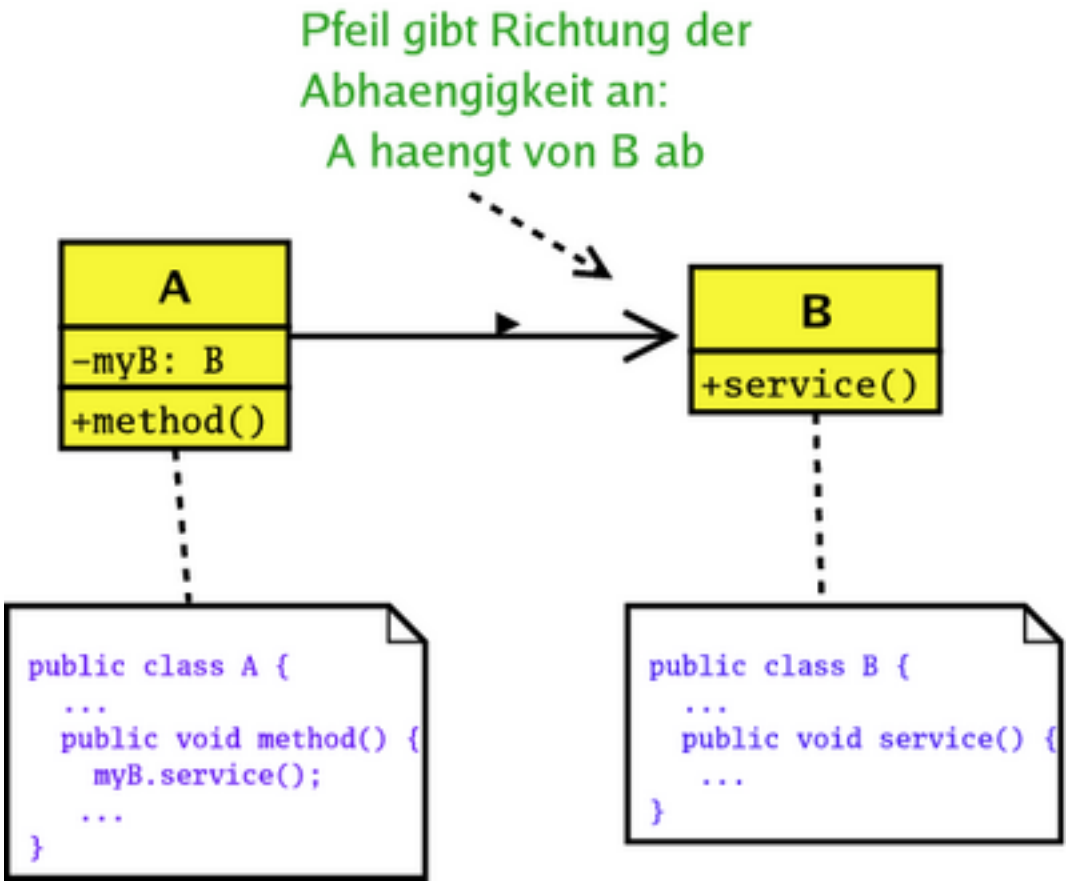
Beide Klassen kennen einander ...



... als Referenz oder Funktionsargument oder in Container oder ...

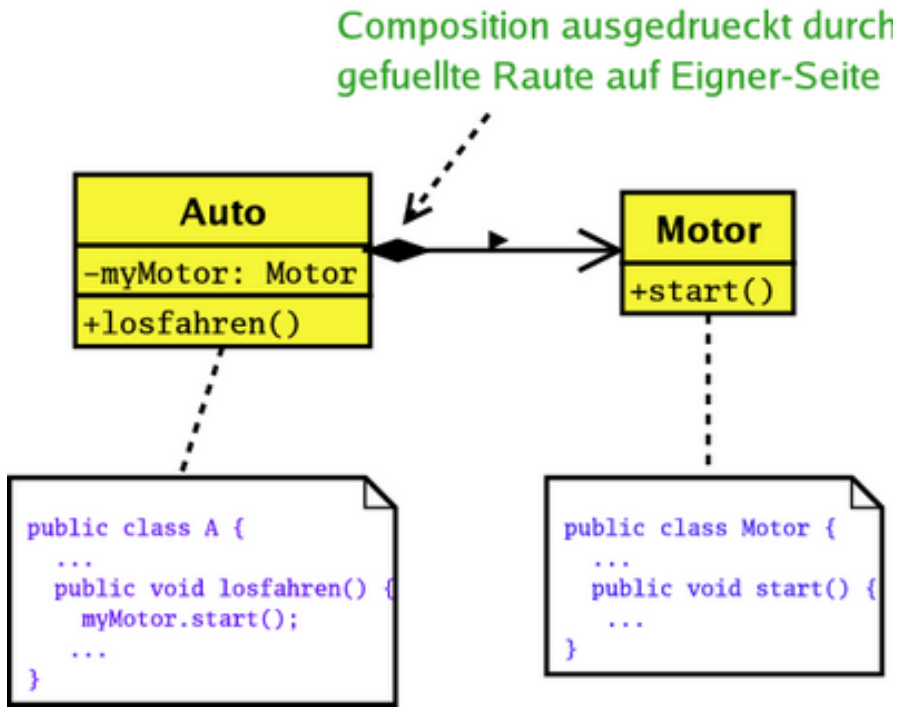
Gängiger ist die Unary Association:

A kennt **B** aber **B** weiss nichts von **A**



Aggregation/Composition:

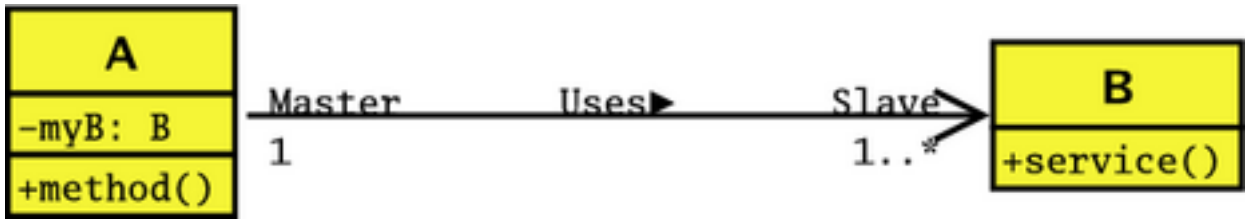
A enthält **B:** *(whole–part relation)*, z.B. Auto enthält Motor



Subtiler Unterschied zwischen Aggregation und Composition, letztere kontrolliert *lifetime* des enthaltenen Objekts $\Rightarrow$ siehe Literatur

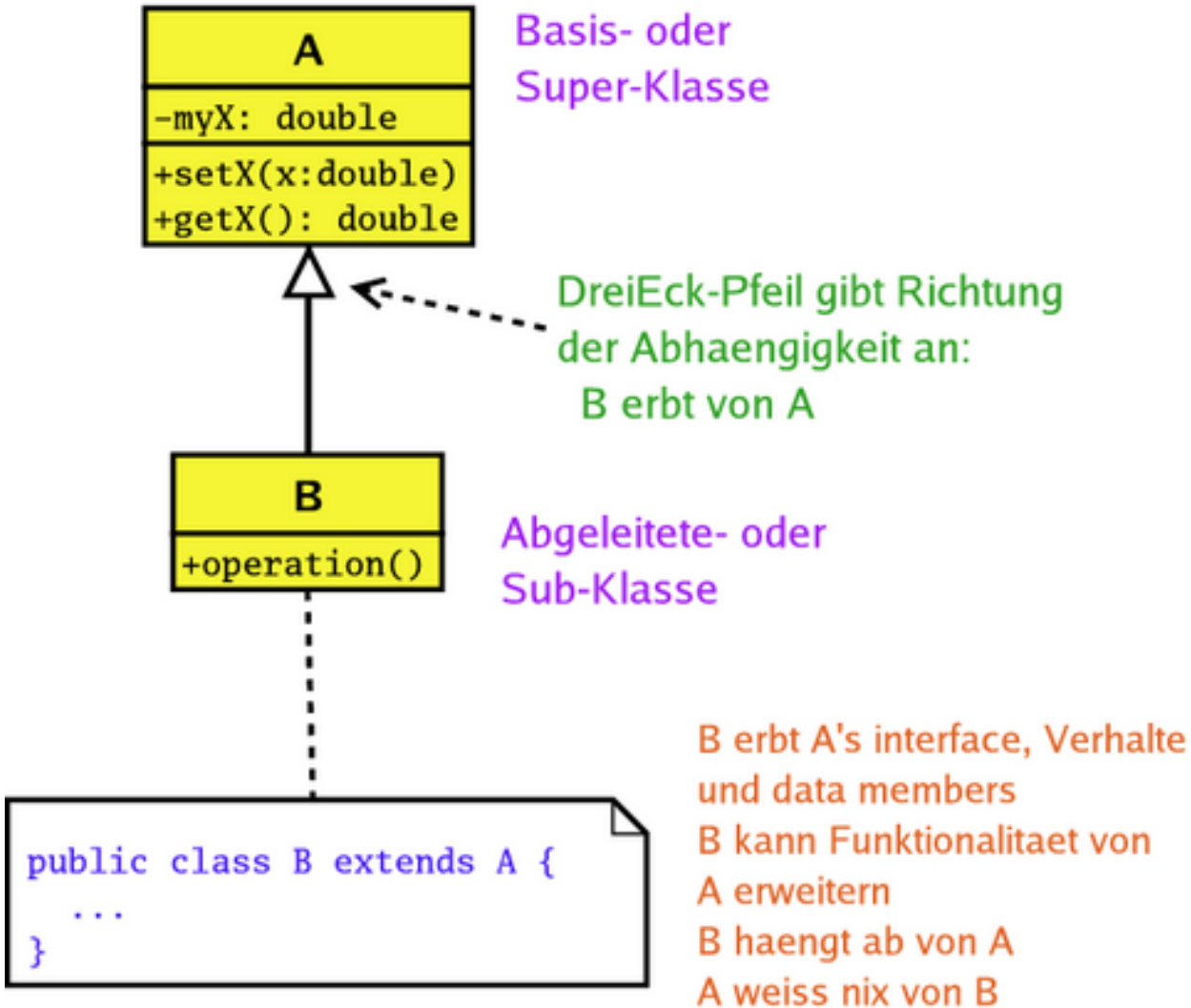
Details der Association:

- Association kann Name haben *(uses)*
- An den Enden können Namen für die **Rollen** der beiden Klassen stehen *(Master, Slave)* ...
- ... und die Multiplizität, d.h. die Zahl der beteiligten Objekte stehen



- Richtig gelesen ergibt das zusammen ein Kompakt-Drehbuch:
  One .. Master .. uses .. one or more .. Slaves

Vererbung (Inheritance):


A
-myX: double
+setX(x:double)
+getX(): double
Basis- oder
Super-Klasse
DreiEck-Pfeil gibt Richtung
der Abhaengigkeit an:
B erbt von A
B
+operation()
Abgeleitete- oder
Sub-Klasse
public class B extends A {
...
}
B erbt A's interface, Verhalte
und data members
B kann Funktionalitaet von
A erweitern
B haengt ab von A
A weiss nix von B

Zusammenfassung:

Mit UML Klassendiagrammen kann man Eigenschaften von Klassen, d.h *Name, member-variablen, interfaces und Methoden* kompakt grafisch darstellen.

Je nach Kontext kann man eine präzise Auflistung aller member-variablen und Methoden mit Argumenten ausgeben oder nur eine unvollständige Liste der relevanten Elemente.

Man kann verschiedene Arten von Beziehungen zwischen Klassen mit UML ausdrücken:

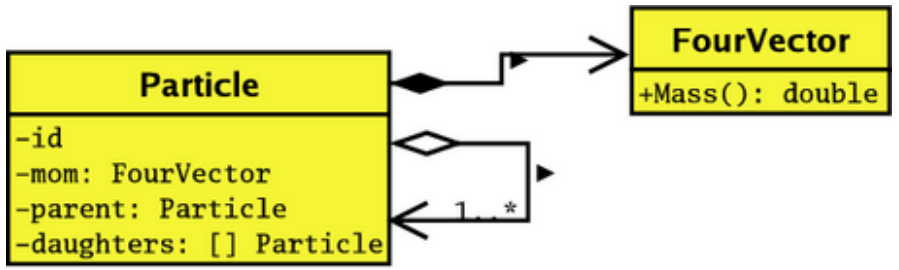
Association, aggregation, composition, inheritance

Die Verbindungen können vage Skizzen sein oder präzise Information über *Namen, Rollen, Multiplicities, ...* enthalten.

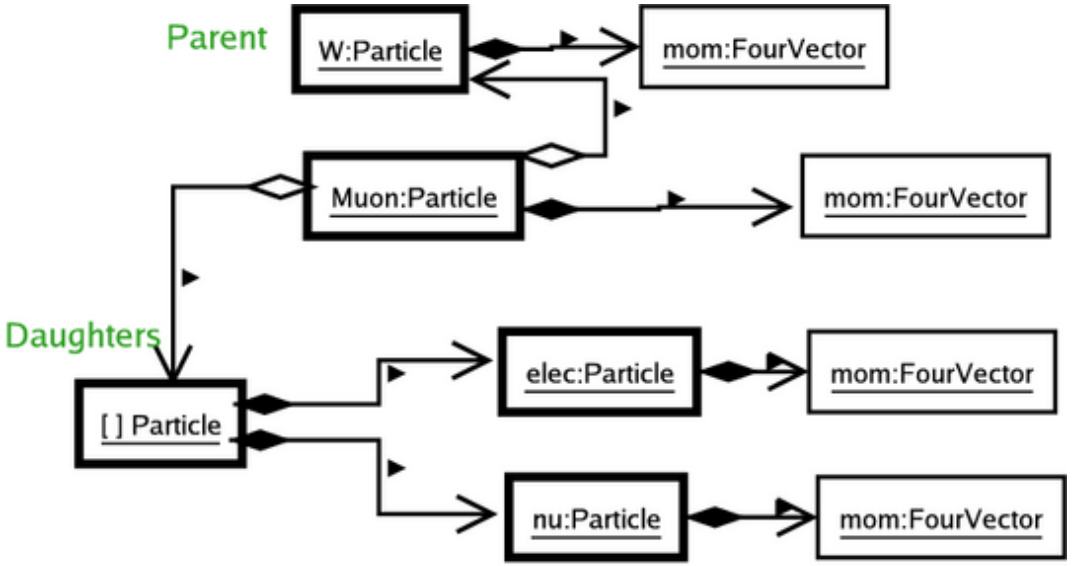
Klassendiagramme beschreiben das **statische Design** der Programm Struktur.

3.5 Objekt-Diagramme

Klassendiagramme beschreiben das **statische Design** der Programm Struktur. Fix, keine Änderung beim Programmablauf

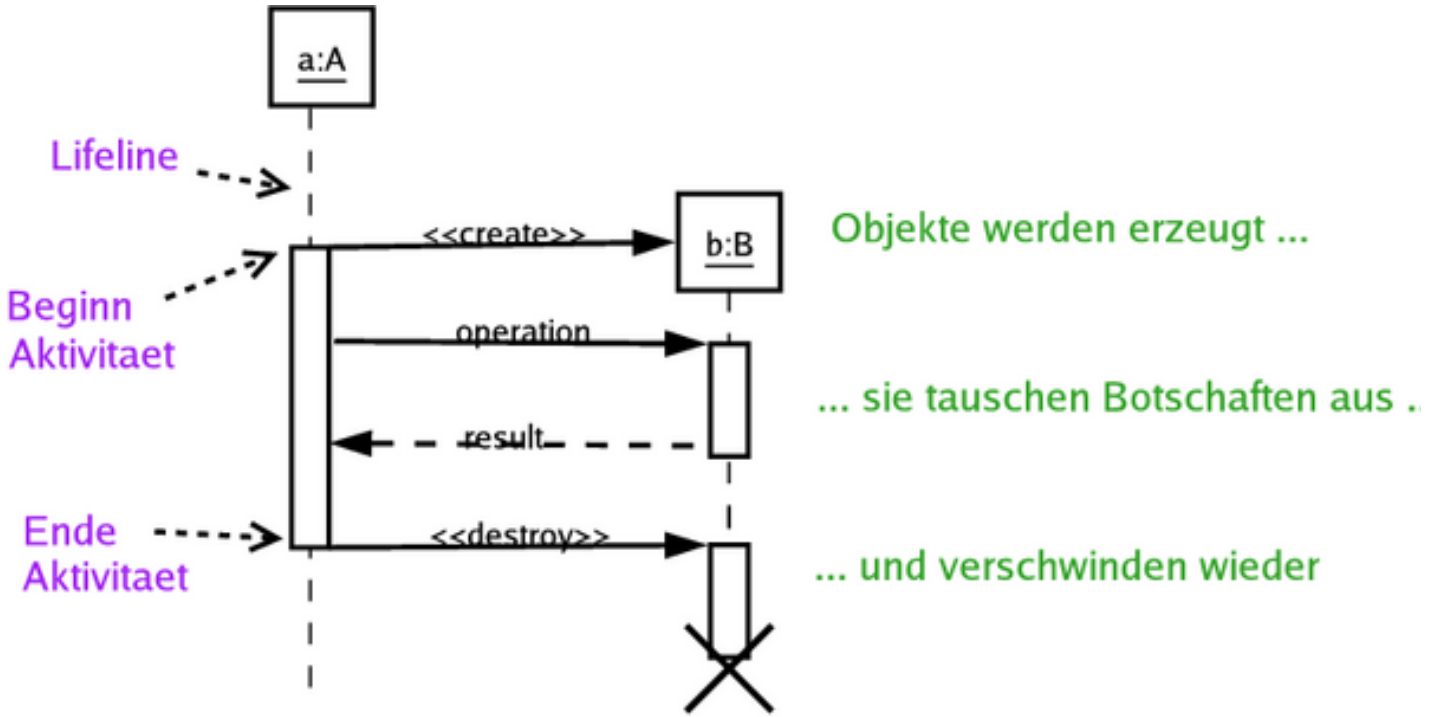


Analog **Objektdiagramme**: Beziehung zwischen den Objekten. Nur gültig für bestimmten Zeitpunkt zur Programm-Laufzeit !

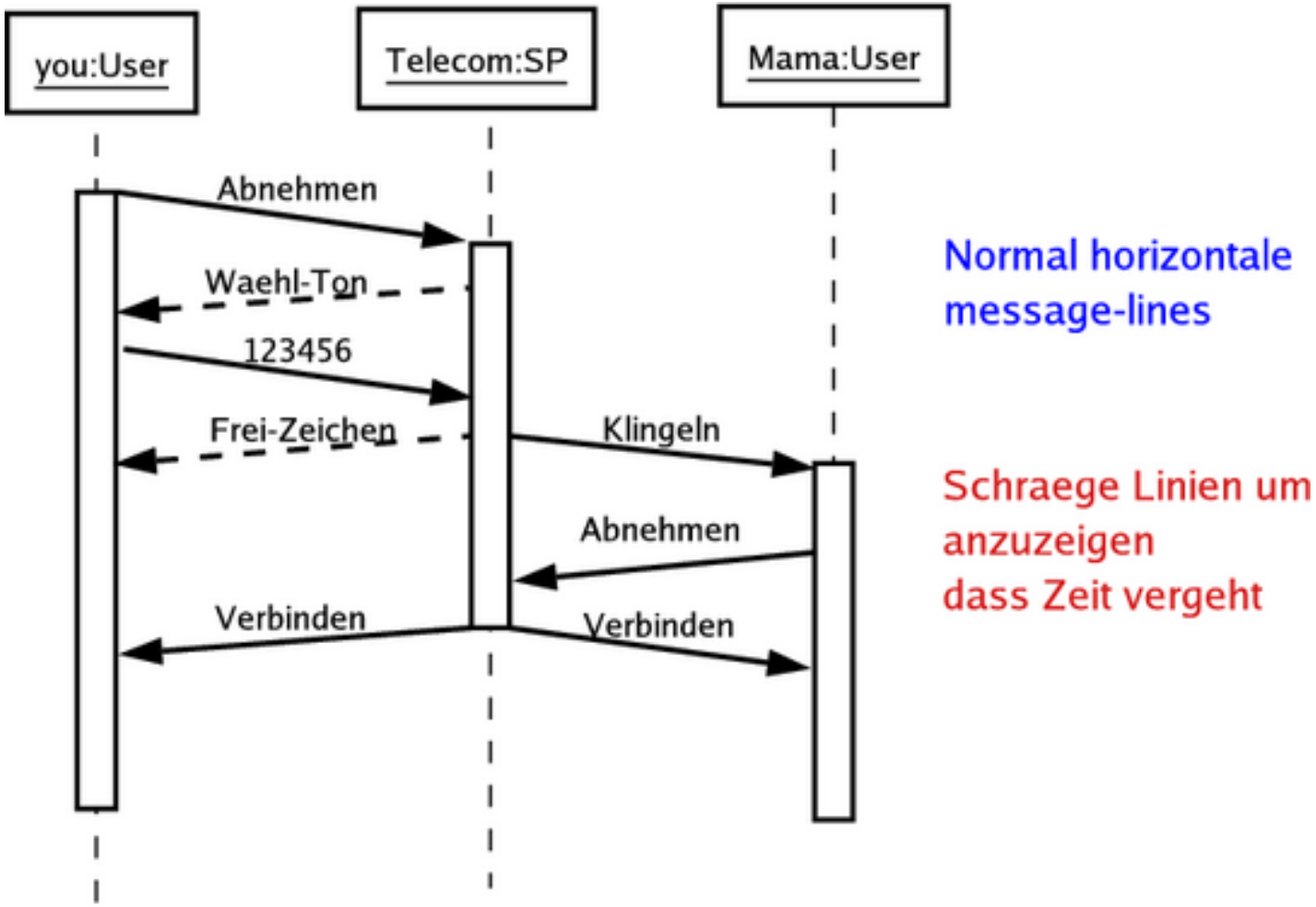


3.6 Sequenz-Diagramme

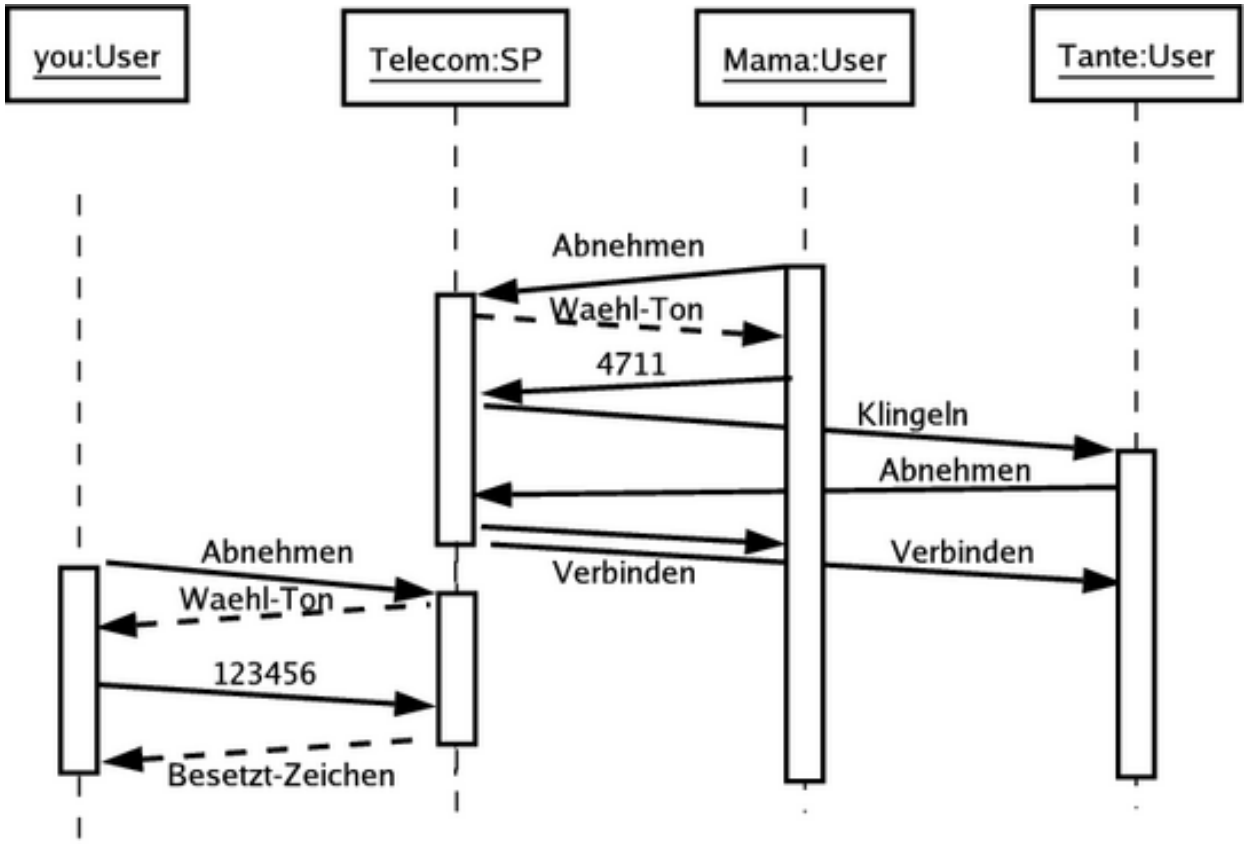
Neben der Klassen bzw. Objektstruktur und den Beziehungen zwischen ihnen ist die **Dynamik**, d.h. der Ablauf des Programms in einem vorgegebenen Szenario wichtig für das Design



Konkretes Beispiel – Mama anrufen



Meistens ist besetzt...



3.7 UML Summary

- **Class–Diagrams** zeigen die statischen Eigenschaften von Klassen und die (möglichen) Verbindungen zwischen ihnen.
- **Object–Diagrams** zeigen die konkreten Eigenschaften und Verbindungen für Objekte als **Momentaufnahme** für ein bestimmtes Szenario.
- **Sequence-Diagrams** zeigen den zeitlichen Ablauf, die Kommunikation der beteiligten Objekte für ein bestimmtes Szenario.

Das ist noch nicht alles: Darüberhinaus gibt's *use-case–Diagrams, Collaboration–Diagrams, Package–Diagram, ...*
siehe z.B.: UML Quick Reference

Für grosse OO–Projekte mit professionellen Teams spielt **formales UML Design** wichtige Rolle: $30 - 60\,\%$ des Aufwands.

- Umfassende Schulung nötig
- Mächtige kommerzielle Tools für UML Design: *Rational Rose, Together, ...*
  - Design in allen Details in UML Diagrammen $\Rightarrow$ automatische Generierung von Source–Code daraus
  - analog **re-engineering**: Tools analysieren existierenden Code und erstellen UML Diagramme

Aber auch im kleineren Massstab *(Projekte in Physik)* sehr nützlich:

- UML Diagramme als einfache Skizzen per Hand oder mit **Umbrello** Programm *(Linux-tool)*
- Gute Basis für Diskussionen *(cf Source-Code)* und zur Dokumentation
- Präzises Erfassen der Anforderungen am besten über Diskussion von **use–cases** $\Rightarrow$ **Sequence-Diagrams**

3.8 Software Engineering

Art und Weise des Programmierens extrem abhängig von Grösse des Projekts *(beachte: Faustregel sind 10 Zeilen Programmcode pro Tag !)*

O(50-500) Zeilen Kurse, Mini–Projekte. I.d.R. ohne formalen Design–Prozess lösbar: kurz Nachdenken, dann Programmieren. Programm von Einzelperson voll überschaubar. Kommentierter, strukturierter Code auch von anderen weiterzubenutzen *(...aber meist wird's komplett neu gemacht)*.

O(500-10k) Zeilen Von Einzelperson oder Klein–Gruppe zu bewältigen. Weiterentwicklung und Weitergabe kritisch ohne formales Design.

O(100k-1M) Zeilen Entwickler–Team nötig. Aufteilung in Unterprokjekte. Ausführliches Design entscheidend, u.U. Hauptteil der Zeit.

$>$**1M Zeilen** scheitern häufig ...

Aufwand wächst stark nicht-linear mit Grösse des Projektes !

3.9 Probleme bei grossen Software Projekten

Software ist **starr**, d.h.

- Abhängigkeiten schwer zu überschauen
- Unvorhersehbare Nebeneffekte treten auf bei jeder Änderung im Ablauf oder Austausch von Modulen
- Aufwand für Modifikationen kaum abzuschätzen
- Management Vorgabe *"Don't touch a working system"*

Software ist **fragil**, d.h.

- Kleine Änderungen haben grosse Nebenwirkungen
- Viele, schwer zu überschauende Stellen müssen angepasst werden
- Gefahr einiges zu übersehen $\Rightarrow$ Bug
- Wenn $P(bug|change) \approx 1$ kann das System nicht mehr weiterentwickelt werden.

Software ist nicht wiederverwendbar, d.h.

- Code zur Lösung eines Problems schon vorhanden
  - Enthält aber Abhängigkeiten auf viele andere, für Problem irrelevante Dinge
  - einige kleinere Anpassungen nötig
- 2 Möglichkeiten:
  - Man übernimmt Verantwortung für den code und die nötigen Anpassungen
  - Man macht unabhängige *cut& paste* Kopie des benötigten codes.
- Idealerweise möchte man den code aber einfach nur benutzen und nicht Verantwortung für Maintenance übernehmen.

Dependency Management

- In einem grossen Software Projekt sind eine Vielzahl von Abhängigkeiten zwischen den einzelnen Komponenten unvermeidbar
- Entscheidend ist die Kontrolle der Abhängigkeiten der einzelnen Komponenten
  - OO Methoden um Abhängigkeiten zu überwachen, z.B. UML Diagramme
  - OO “Tricks” um Abhängigkeiten zu minimieren *(abstrakte Klassen, Interfaces)*

3.10 Software Engineering Modelle

Vielfalt von Methoden im Einsatz, kein bestimmtes Verfahren konnte sich allgemein durchsetzen.

Code&Fix Einfach drauflos, nimm den Stier bei den Hörnern, Trial and Error. Nicht wirklich ein Modell, aber de-facto Standard in vielen Bereichen.

- Vorteil: Sehr flexible, schnelle Entwicklung
- Nachteil: Schwer zu pflegen und weiterzuentwickeln über längere Perioden

Ok, für kleine, überschaubare Projekte (sowohl Code-Umfang (1 kloc) als auch beteiligte Personen (1-3) als auch Lebensdauer (Monate–Jahr)

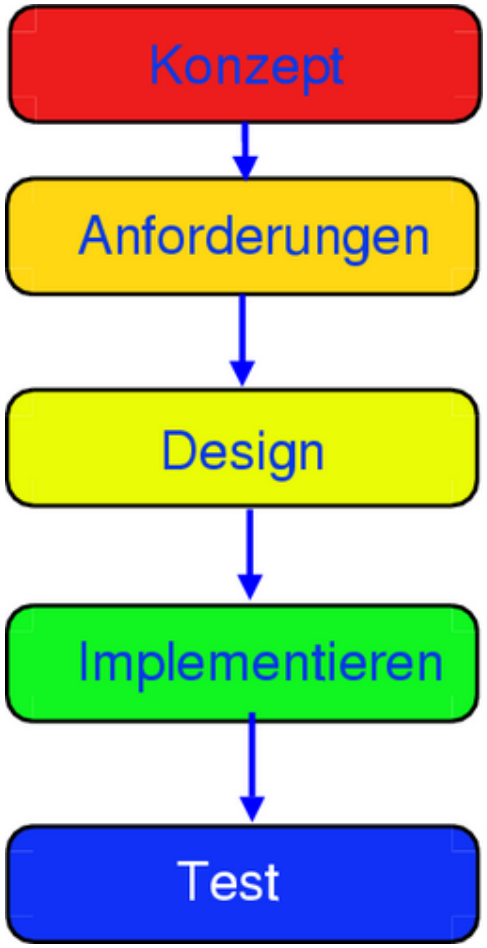
Wasserfall Modell lineare Design – Implementierung Sequenz

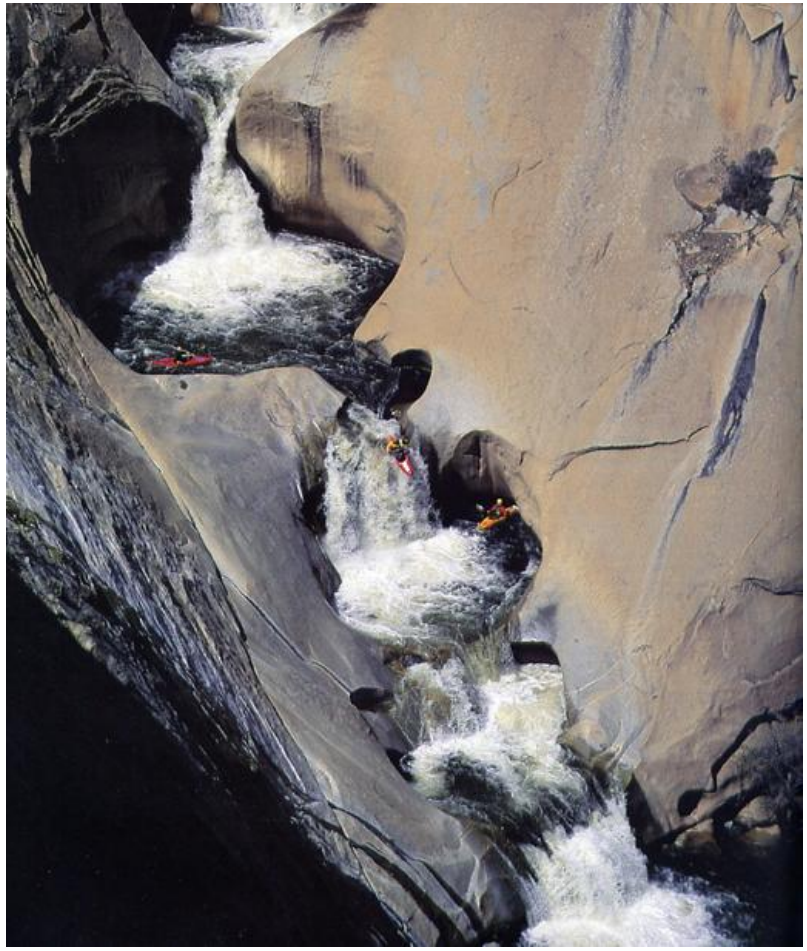
RUP Rational Unified Process, Design mit UML, iterativ, d.h. Zyklen von Design – Implementierung. "Schwergewichtiges Modell", viel Aufwand und Erfahrung nötig, lange Design-Phasen

Agile Methoden XP, RAD, Scrum, ... Konträrer Ansatz, kurze Zyklen, schnelle Implementierung, Änderungen nicht Ausnahme sondern Regel, inkrementelles Umsetzen der Spezifikationen nach Prioritäten, lauffähige Zwischenprodukte.
Erst detaillierte Test-suite definieren, dann Erstellen des eigentlichen Programms. Test-Suite integraler Bestandteil der Software, wird bei jeder Änderung ausgeführt. $\Rightarrow$ *Sehr nützlicher Ansatz auch für kleine Projekte und Teams ! (Nightly builds)*

noch viele weitere Varianten, Alternativen ...

Wasserfall Modell





Einzelner Durchlauf !

Wasserfall Modell entwickelt von US Militär in den 60ern für grosse Programmierprojekte, analog zu Vorgehen in klassischen Ingenieur–Disziplinen.

Entscheidende Voraussetzungen:

- Detailliertes Verständnis aller Probleme im Vorhinein
- Stabile Umgebung, keine neuen Anforderungen während der Nutzung

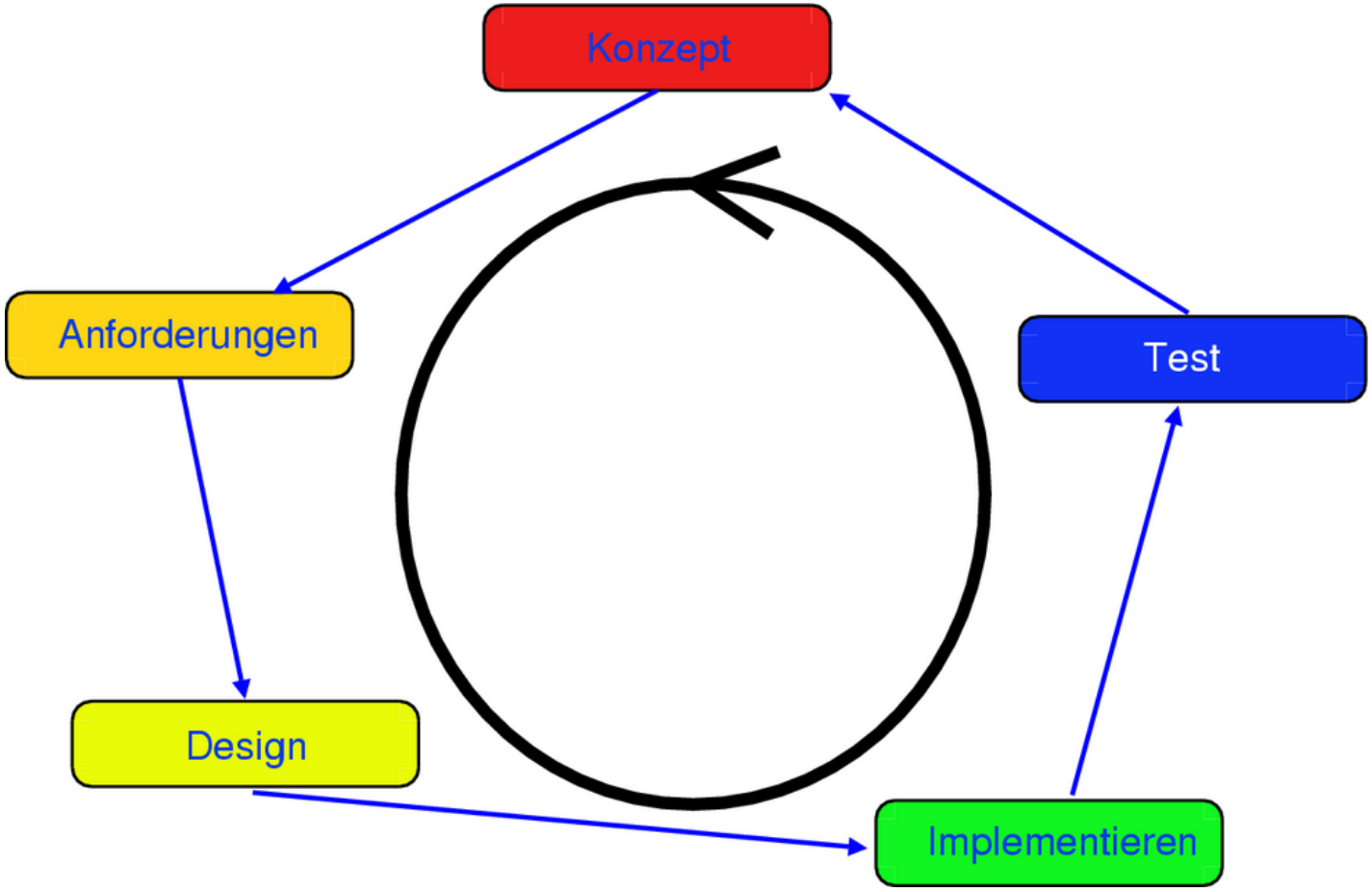
Änderungen später sind sehr schwierig, genauso Wiederverwendung und Erweiterung.

Software-Entwicklung $\neq$ Brückenbau :

- Nutzeranforderungen ändern sich
- Umgebung, Tools, Services ändern sich
- ...

UML Modell

Einerseits:



Iterativ, zyklisches Durchlaufen !

UML Modell cont.

und andererseits:

Objektorientiertes Programmieren

- Objekte als **Black Boxes**: *Komplexität* innen versteckt, einfache Bedienung von aussen.
  - *interface* zur Kommunikation
  - *Zustand:* Objekt enthält Daten
  - *Verhalten:* Methoden und Algorithmen
- Höhere Stufen der Abstraktion

XP – Extreme Programming

Beinhaltet 2 wesentliche Konzepte

- Tests sind zentraler Bestandteil der Entwicklung: **“write tests first”** . D.h. bevor Klassen und Funktionen konkret implementiert werden, entwickelt man erst die Tests, die diese Klassen und Funktionen erfüllen müssen. $\Rightarrow$ Tests als Detailspezifikation.

  Diese Tests sind Teil des Entwicklungspakets und werden bei jeder Änderung und Neu–Übersetzung ausgeführt *(nightly builds)* , dadurch werden Fehler und Probleme bei Weiterentwicklung schnell erkannt.

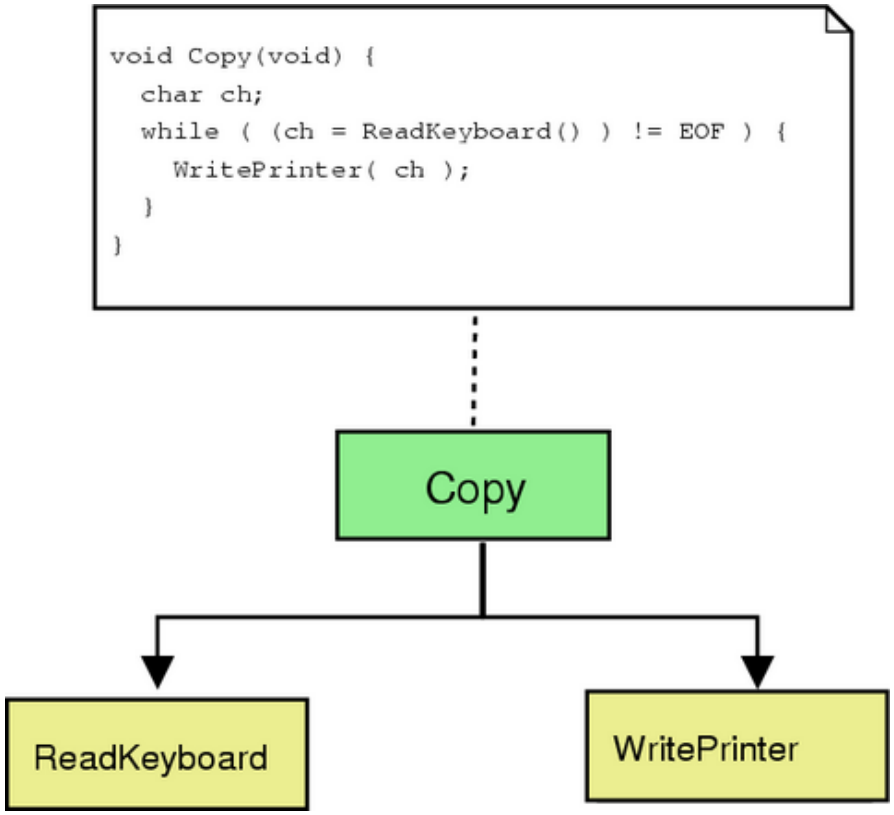
- **Pair programming:** Entwicklung nicht als Einzelkämpfer sondern als 2er Team: Einer schreibt den Code, der andere denkt darüber nach, entwirft Tests, behält die grossen Ziele im Auge, etc. Die Rollen werden abgewechselt.

Zumindest das Einbeziehen von Tests in die Entwicklung lässt sich auch bei kleinen Projekten ohne weiteres anwenden, erfordert allerdings entsprechende Disziplin.

3.11 Code&Fix – ein Beispiel

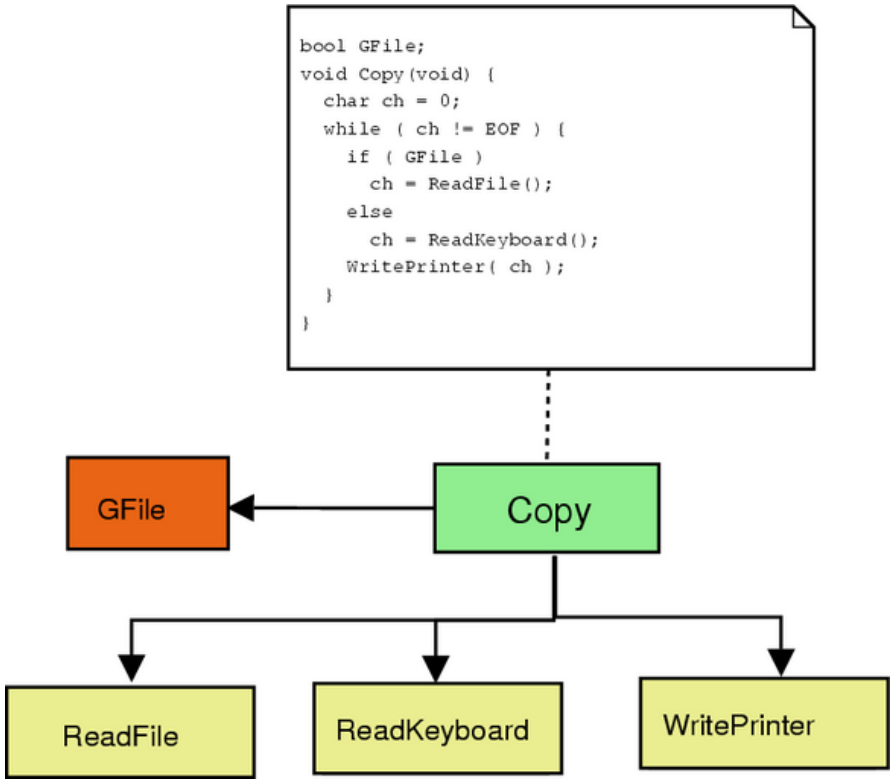
Eine simple *C-Funktion* `Copy()`, liest vom Keyboard und schreibt auf einen Printer.

Copy – Version 1



Copy – Version 2

Es wäre schön auch von Files lesen zu können. **Aber:** Es soll **backward–compatible** sein, vorhandene Applikationen müssen nichts ändern.



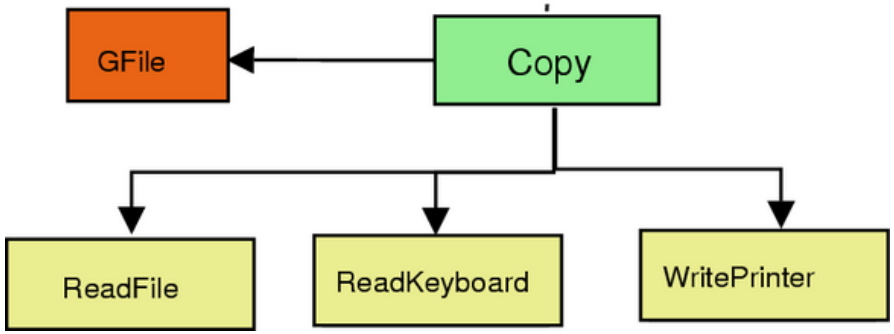
Globale Variable als Flag

Copy – Version 3

Oh je, da hat sich ein Fehler eingeschlichen: Es soll kein `EOF` ausgegeben werden !

Bug-fix:

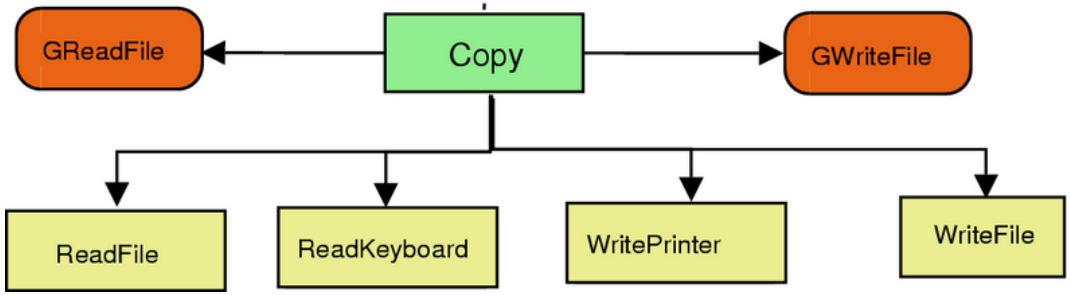
```
bool GFile;
void Copy(void) {
  char ch = 0;
  while ( 1 ) {
    if ( GFile )
      ch = ReadFile();
    else
      ch = ReadKeyboard();
    if ( ch == EOF ) break;
    WritePrinter( ch );
  }
}
```



Copy – Version 4

Ausgabe in Files wäre ja auch noch ganz nett. Natürlich **backward–compatible**, also noch eine globale Variable !

```
bool GReadFile;
bool GWriteFile;
void Copy(void) {
  char ch = 0;
  while ( 1 ) {
    if ( GReadFile ) ch = ReadFile();
    else ch = ReadKeyboard();
    if ( ch == EOF ) break;
    if ( GWriteFile ) WriteFile(ch);
    else WritePrinter( ch );
  }
}
```



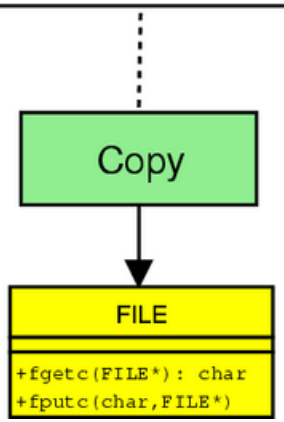
Die Funktion wird immer grösser und komplexer, die Verwendung komplizierter ...

Copy – Version 5

Zeit für ein ordentliches Re-design. Eine erfahrene C Programmiererin zeigt uns wie man's richtig macht:

```
#include <stdio.h>

void Copy(File *in, File *out) {
  char ch ;
  while ( (ch = fgetc( in )) != EOF ) {
    fputc( ch, out );
  }
}
```



OO-like in C: `FILE` wie Klasse für generische Byte–Streams.

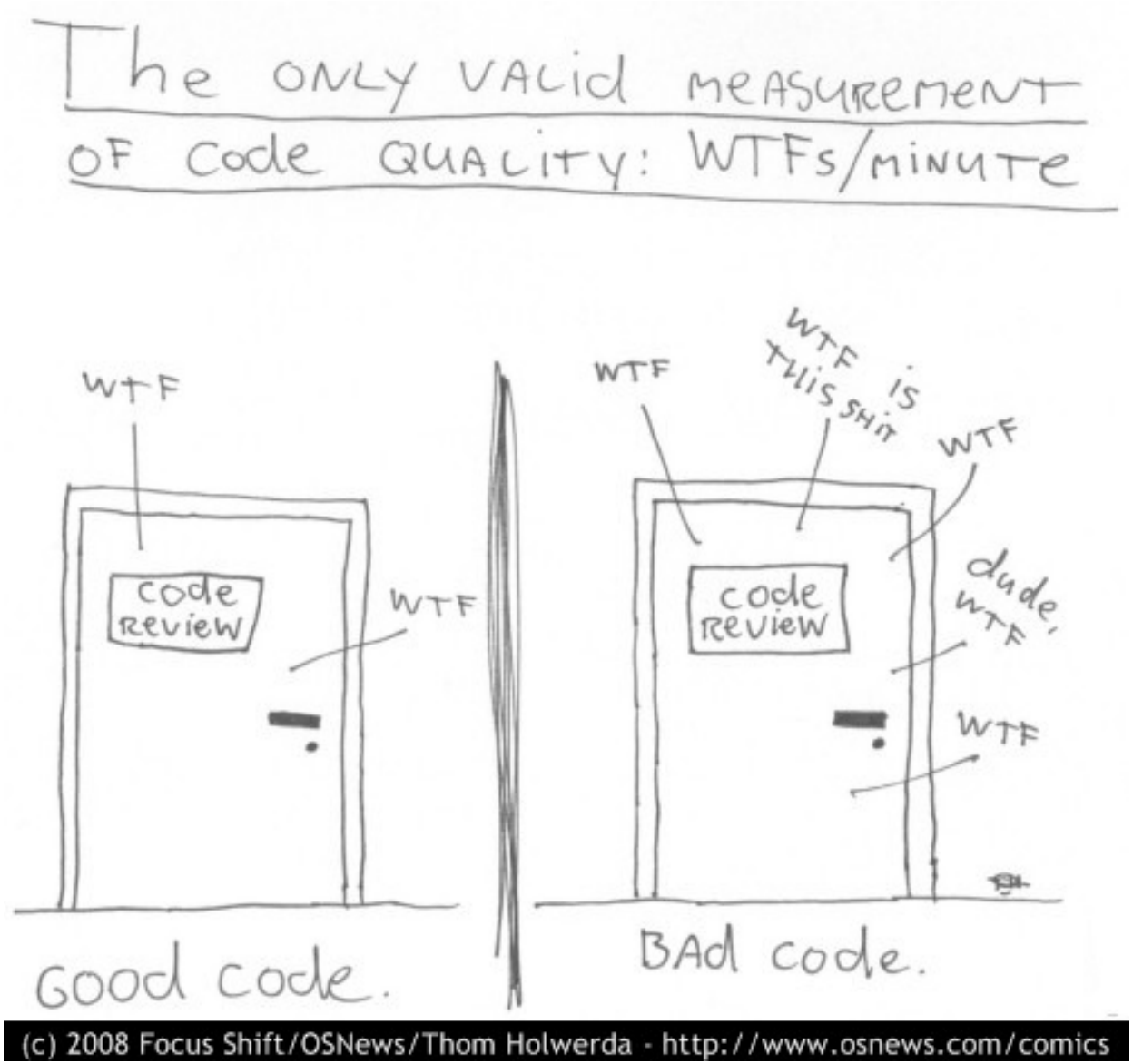
ABER: Alle Stellen in denen *Copy()* verwendet wird müssen entsprechend angepasst werden !

Typisches Beispiel wie Code altert und quasi *vergammelt*.

- einfache, klare Funktion zu Beginn
- mehr und mehr Features werden dazugepackt
- backward–compatibility erzwingt obskure Konstruktionen

OO-Analyse und -Design:

- Flexible, leicht erweiterbare Systeme
- Klar-definierte *responsibility* einer Klasse/Funktion.


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

3.12 Aufgaben

1. **umbrello Programm**
   Unter Linux gibt es das Open–Source Programm umbrello zum Zeichnen von UML–Diagrammen.
   (Applications ⇒ Programming ⇒ Umbrello)

   Sie können damit Diagramme für neue Klasen erstellen und über den Source–Code Generator das Programmgerüst ausgeben lassen.

   Oder man kann existierenden Source–Code importieren und UML Klassendiagramme dazu erzeugen. Probieren Sie es aus für unsere Beispiele zu *ThreeVector* und *LorentzVector*

2. **UML Diagramm für Interface-Beispiel**
   Erstellen Sie das UML Klassendiagramm für das Beispiel zu den **interfaces**, d.h. die Klassen *Nullstelle* und das *TestNull* main Programm sowie das Interface *Func*. (Mit umbrello oder Papier/Tafel).

3. **Fahrkartenautomat**
   Analysieren Sie die Funktion eines **DB–Fahrkartenautomates**, d.h entwerfen Sie typische *use-cases*, definieren Sie *Komponenten* und *Operationen*, erstellen Sie Sequenz–Diagramme. (Am besten als Teamarbeit von 2-3 Personen).

4 Software Testen

Warum?

Grundsätzlich

- Nachweis, dass Software macht was sie soll
- Fehler und Probleme finden bevor echte Anwendung *(= Production-use)* stattfindet

Rasante Zunahme von Systemen und Abläufen, die durch Software kontrolliert werden, Fehler können gravierende Folgen haben. Sicherstellen dass Programme wie vorgesehen funktionieren und das Ausmerzen von Bugs ist der wichtigste Aspekt beim Erstellen von **Production–Use** Programmen.

Eine interessante Auflistung von zum Teil kuriosen zum Teil aber auch fatalen Fehlen in Software findet sich z.B. in **Collection of Software Bugs**

Man kann mit Tests zwar nie nachweisen, dass ein Programm keine Fehler hat, aber zumindest das Auftreten von Fehlern stark reduzieren....

Apple goto-fail bug

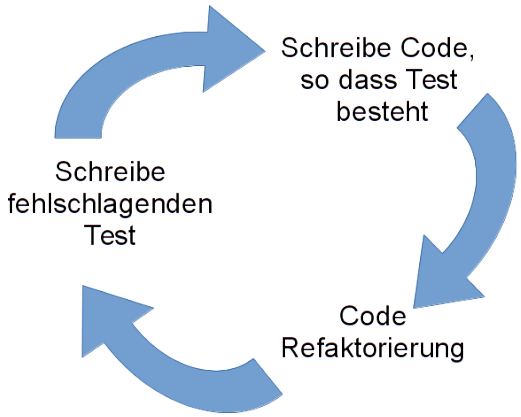
Aktuelles (2014) Beispiel für sicherheitsrelevanten Bug in Apple IOS 7 *(C-Programmcode)*.

```
// ...
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
err = sslRawVerify(...);
//...
```

Entscheidender Check (`sslRawVerify()`) wird u.U. nicht ausgeführt, ermöglicht Angriffe auf verschlüsselte (TLS/SSL) Verbindungen.

4.1 Test-Driven Development (TDD)

- Tests integraler Bestandteil der Entwicklung
- Zunächst wird Test geschrieben
- Erst danach der eigentliche Programmcode implementiert
- Iteratives Vorgehen ⇒ nächster Test ⇒ weiterer Code
- Tests sind Teil des Programmpakets und werden regelmässig wiederholt ⇒ **Nightly-Tests**



Die Programmentwicklung ist somit ein iterativer Prozess, der einen zwingt, über das Programmverhalten **vor** der Implementierung nachzudenken. Die Tests sollen möglichst vollständig verschiedene Fälle und Verzweigungen im Programm überprüfen.

Test-Ebenen

- Unit-Tests: Tests von einzelnen Komponenten (Funktionen, Klassen)
- Integration-Tests: Testen das Zusammenspiel verschiedener Komponenten (Klassen, Pakete, ...)
- System-Tests: Tests des ganzen Systems
- ...

Grundsätzlich kann man Tests natürlich als ganz normalen Anwendungs- oder Beispielprogramme implementieren, die die jeweiligen Klassen bzw. Funktionen exemplarisch verwenden und Ausgabe machen ob das erwartete Ergebnis eingetroffen ist, siehe z.B. `ThreeVector Class` und Test-Programme dazu.

Allerdings empfiehlt sich für systematisches Vorgehen die Verwendung von *Test Frameworks*, die viele Features bieten verschiedene Testabläufe zu erstellen.

Wir benutzen im Weiteren **Google Test** zum Erstellen von Unit-Tests.

4.2 Google–Test für Funktionen

```
#include <cmath>
#include <iostream>
#include "gtest/gtest.h" // googletest includes
int add( int a, int b)
{
  return ( a + b );
}
int main(int argc, char* argv[])
{
  testing::InitGoogleTest(&argc, argv); // initialize Google Test Framework
  return RUN_ALL_TESTS();
}
TEST(Addition, CanAddTwoNumbers_1) // define test ( name, sub-name )
{
  ASSERT_EQ( 3, add(1, 1) ); // assert stops further execution if failure
  ASSERT_EQ( 5, add(2, 2) );
}
TEST(Addition, CanAddTwoNumbers_2)
```

```
{
  EXPECT_TRUE(add(2, 1) == 4); // expect continues furher execution if failure
  EXPECT_EQ( 5, add(2, 2) );
}
```

Vorgehen:

- In `main()` Funktion Google-Test initialisieren:
  `testing::InitGoogleTest(...)`
  und anschliessend Tests ausführen
  `RUN_ALL_TESTS()`
- Einzelne Tests definieren
  `TEST( name, subname )`
- Darin Google-Test Funktionen aufrufen, z.B.
  `ASSERT_EQ() , EXPECT_EQ()`
- Programm kompilieren *(Vorsicht: Google-Test include/link Pfade!)*
- Programm ausführen:
  `./ut_tfunc # alle Tests werden ausgefuehrt`

```
./ut_tfunc --gtest_filter='Addition.CanAddTwoNumbers_2' # bestimmte
Tests auswaehlen
./ut_tfunc --gtest_list_tests # definierte Tests anzeigen
./ut_tfunc --help # ...
```

4.3 Google–Test für Klassen

Weiteres Beispiel für Tests der `My3Vector` Klasse

```
#include <cmath>
#include <iostream>
#include "gtest/gtest.h" // googletest includes
#include "My3Vector.h"
int main(int argc, char* argv[])
{
  testing::InitGoogleTest(&argc, argv); // initialize Google Test Framework
  return RUN_ALL_TESTS();
}
TEST(My3Vector, Basics) // define test ( name, sub-name )
{
  My3Vector a;
  EXPECT_EQ( 0, a.Length() );
  My3Vector b(3., 4., 0.);
  EXPECT_EQ( 5, b.Length() );
}
TEST(My3Vector, SkalarProdukt) // define test ( name, sub-name )
{
```

```
  My3Vector a(1.,1.,1.);
  ASSERT_EQ( 3., a.Dot(a));
  My3Vector b(1.,-1.,0.);
  ASSERT_EQ( 0., a.Dot(b));
}
```

Neben dem einfachen `TEST()` Makro kann man in Google Test auch *Test Fixtures* `TEST_F()` verwenden. Das erlaubt komplexeren Setup und Initialisierung einer Testumgebung, siehe Test Fixtures.

Wie schon diskutiert sollten Tests nicht nur als nachträgliche Ergänzung verwendet werden sondern im Sinne von **TDD** von vorneherein in die Entwicklung integriert sein. Die Tests werden zusammen mit der eigentlichen Implementierung erstellt (oder sogar vorab).

4.4 Aufgaben

- **Beispiele ausprobieren und erweitern**
  Probieren Sie die vorgestellten Beispiele aus, provozieren sie Fehler bei den Tests, verwenden Sie verschiedene Optionen für Google Test Aufruf.

  Hinweis: Zum Kompilieren und Linken der Programme brauchen Sie die Google Test Headers und Libs, die sind aber am CIP nicht systemweit installiert, so dass man explizit Pfade dafür angeben muss. In solchen Fällen benutzt man am besten ein passendes Makefile .

- **Primzahlen** Schreiben Sie einen Test für den Algorithmus zur Primzahlenberechnung aus dem ersten C++-Kurs: prime_simple.C

5 Design–Patterns

Worum geht es ?

- Design–Patterns beziehen sich auf **Software Architektur und Design**.
- Sie beinhalten die statischen und dynamischen Strukturen und Collaborations von erfolgreichen und bewährten Lösungen.
- Es geht **nicht** um effiziente Algorithmen, smarte Datenstrukturen, o.ä.
- Design–Patterns stellen **Schlüsseltechnologie** für OO Design dar, um Wiederverwendung, Erweiterbarkeit, Modularität zu erreichen.
- Sie beziehen sich auf bestimmten Problembereich. Patterns sind *problem/solution* Paar in diesem Zusammenhang
- Neue Stufe der Abstraktion: routinierte OO Programmierer sind vertraut mit den Standard Patterns
  $\Rightarrow$ Diskussion auf dieser Ebene.

Analogie Schach ...

Zunächst lernt man wesentliche Regeln und Anforderungen
(Namen der Figuren, erlaubte Züge, Schachbrett, ...)

Als nächstes die Grundprinzipien
(Wert der Figuren, strategische Bedeutung der Positionen, ...)

Um aber Schachmeister zu werden muss man intensiv die Meisterpartien studieren. Darin finden sich Muster von Stellungen, Situationen, die muss man verstehen, abspeichern, wiederverwenden. Im Schach gibt's Hunderte solcher *Patterns*

... zu Software–Design

Zunächst lernt man wesentliche Regeln und Anforderungen
(Datentypen, Syntax, Kontrollstrukturen, Funktionen, ...)

Als nächstes die Grundprinzipien
(Structured programming, modular programming, object–oriented programming, ...)

Um aber Design–Meister zu werden muss man intensiv die Lösungen anderer Meister studieren. Darin finden sich Muster von Problemen, Situationen, die muss man verstehen, abspeichern, wiederverwenden.

In Basis/Standard-Werk zu Design–Patterns werden **23 Patterns** diskutiert.

Aber noch viele weitere in nachfolgender Literatur !

Ausführliche Diskussion auch nur der 23 Standard–Patterns im Rahmen dieses Kurses nicht sinnvoll. Aus Zeitgründen sowieso, aber v.a. wg. Anspruch:

- Die meisten Patterns sind komplex. Meist ist es schon schwierig das Problemfeld zu erfassen und nachzuvollziehen.
- Unterschiede zwischen Patterns/Problemfeld z.T. sehr subtil
- I.a. solide Erfahrung erforderlich in OO Programmieren und damit verbundener Probleme

Dennoch: Es gibt einige *leicht zu verstehende und nachvollziehbare* Patterns, drei davon werden wir diskutieren:
Template, Singleton, Observer

5.1 Einschub – Accumulate "Pattern"

Zum Einführen erstmal noch eine Art simpler Pattern aus dem Bereich Algorithmen (**kein** Design Pattern !):

Bei vielen Programmierproblemen geht es um's Aufsummieren bzw. Akkumulieren von Sequenzen oder Arrays, z.B. *Summe der Quadratzahlen von 1–N, Berechnung der Fakultät, Bestimmung des Mittelwerts, etc*

```
int sum = 0;
for ( int i=1; i<=N; i++ ) {
  sum += i*i;
}
//
int fak = 1;
for ( int i=2; i<=N; i++ ) {
  fak *= i;
}
//
double sum = 0;
for ( int i=0; i<N; i++ ) {
```

```
  sum += a[i];
}
double mean = sum/N; // Mittelwert
//..
```

Oder am besten entsprechenden STL Algorithmus `accumulate` verwenden

$\Rightarrow$ höhere Abstraktion

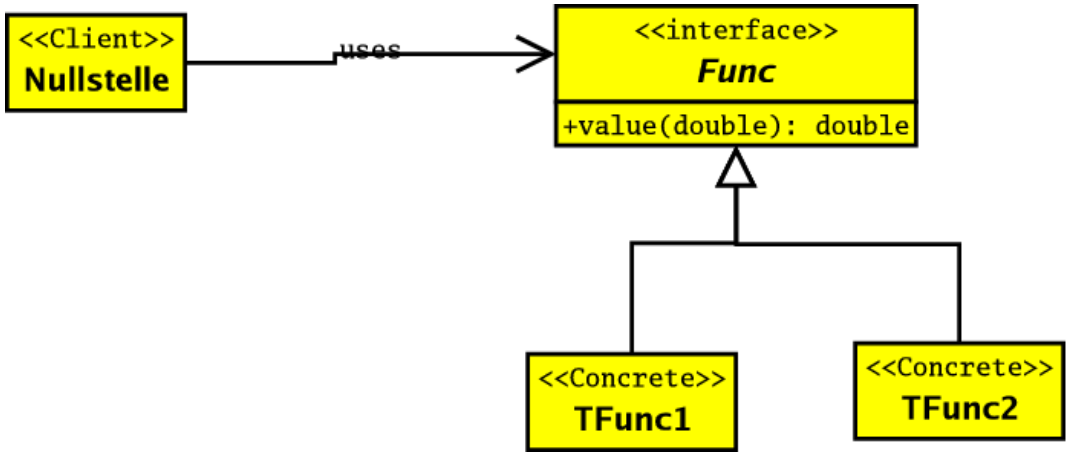
Allgemeines Muster für Vielzahl von Fragestellungen: *Initialisieren, Akkumulieren in Schleife, ..*

5.2 Template Pattern

In einfacher Form schon diskutiert, es geht um:

- **Abstrakte Klasse** oder **Interface** gibt Methode(n) vor
- Client Klasse sieht/kennt nur dieses **Interface**
- Konkrete Klassen implementieren Interface

Prinzip schon verwendet und diskutiert bei Beispiel mit *class Nullstelle* und *interface Func*



Einfaches aber wichtiges Pattern: Trennung der Abhängigkeiten , Client kennt nur Interface, nicht die konkreten Klassen $\Rightarrow$ leichte Erweiterbarkeit

5.3 Singleton Pattern

Für bestimmte Probleme ist es nützlich eine Klasse zu haben, von der genau eine einzige Instanz erzeugt werden kann.

- Diese Instanz kann dann an beliebigen, verschiedenen Stellen angefordert und verwendet werden.
- Das **Singleton** stellt sicher dass es sich immer um dasselbe Objekt handelt

Anwendung, z.B.

- Singulärer, konsistenter Zähler
- Konfigurations- oder Options-Handling
- Singulärer Zugriff auf Resourcen

```
// Sequence Klasse
class Sequence {
private:
  static Sequence * instance;
  int counter;
  Sequence() { // private constructor
    counter = 0;
  };
public:
  static Sequence * getInstance() {
    if(instance==0)    { // Lazy instantiation
      instance = new Sequence();
    }
    return instance;
  };
  int getNext() {
    return ++counter;
  };
};
Sequence * Sequence::instance = 0; // initialization
```

Singleton Pattern ist i.W. Programmier-Trick um Singularität der Instanz zu erreichen:

- Konstrukor ist **private**, also kein Erzeugen ausserhalb der Klasse mit `new` möglich
- **static** Methode `getInstance()`, wird über Klasse gerufen: `Sequence.getInstance()`.
- Bei erstem Aufruf erzeugt `getInstance()` einmal die **Sequence** Instanz.

```
#include <iostream>
using namespace std;
int main()
{
  Sequence * sq1 = Sequence::getInstance();
  cout << "SQ1 " << sq1->getNext() << endl;
  cout << "SQ1 " << sq1->getNext() << endl;
  Sequence * sq2 = Sequence::getInstance();
  cout << "SQ2 " << sq2->getNext() << endl;
  cout << "SQ2 " << sq2->getNext() << endl;
  cout << "SQ1 " << sq1->getNext() << endl;
    }
```

5.4 Observer Pattern

Observer gibt eine Lösung vor für gängiges und wichtiges Problem:

- Ein **Daten–Objekt** wird von verschiedenen anderen Objekten verwendet.
- Jede Änderung muss diesen Objekten mitgeteilt werden, damit sie entsprechend darauf reagieren können.

Praktisches Beispiel: Spreadsheet/Tabellenkalkulation

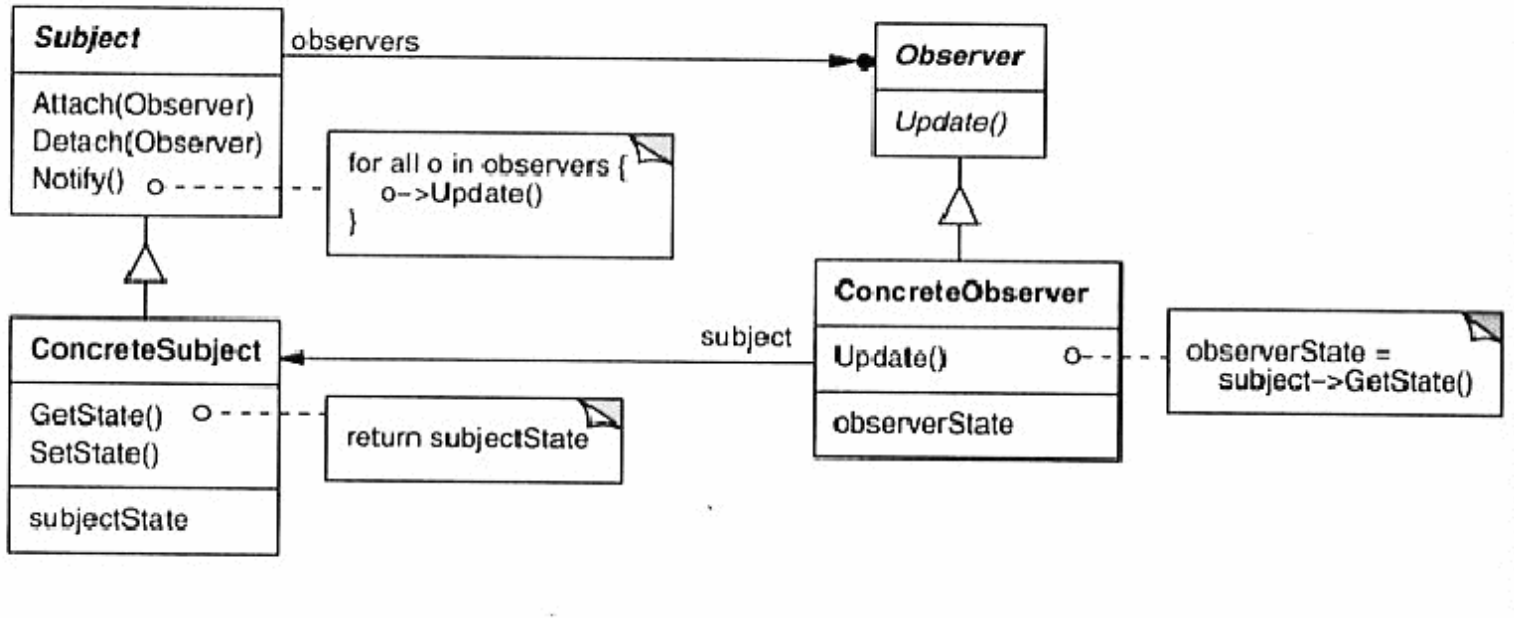
- Basisdaten sind numerische Werte in File/Datenbank
- Unterschiedliche Darstellungs-Objekte: **Tabelle, Balkendiagramm, Tortendiagramm**
- Bei Änderung der Basisdaten müssen alle diese Objekte entsprechend reagieren

Auch **Signal–Slot Mechanismus** behandelt dieses Problem. Qt Signal–Slot mögliche Lösung, allerdings *Erweiterung von C++* nötig.

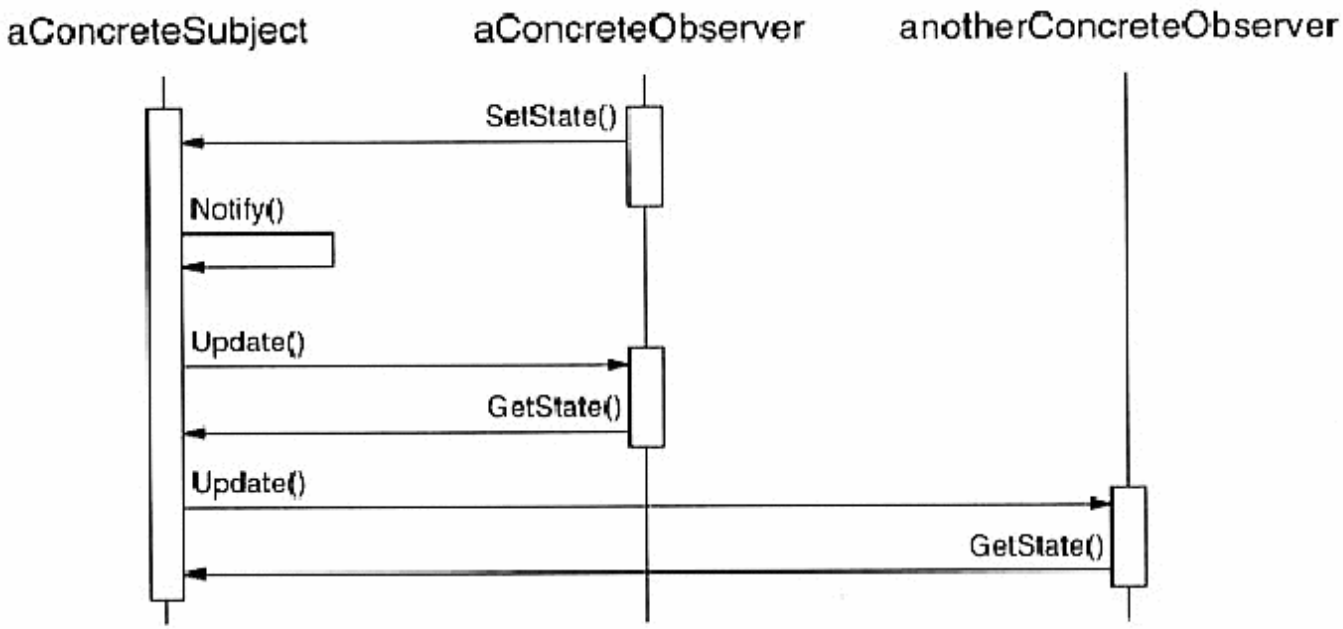
Das **Observer Pattern** gibt vor, wie solche Abhängigkeiten zwischen verschiedenen Klassen/Objekten implementiert werden:

- Es gibt ein **subject** und **observer**
- Die Zahl der **observer** ist beliebig
- Alle **observer** werden benachrichtigt, wenn sich das **subject** ändert.
- Die **observer** fragen daraufhin den neuen Zustand von **subject** ab.

Observer – Class Diagram



Observer – Sequenz Diagram



Observer in C++

- Basis–Klasse **Subject** (=Observable)
- Methoden *attach(), setChanged(), notify()* implementiert
- Observers werden in *list* gespeichert
- Interface **Observer** mit Methode
  `void update(Subject *)`

```
class Subject;
// Observer: just an interface
class Observer {
public:
      Observer(){};
      virtual ~Observer(){};
      virtual void update(Subject* ) = 0;
};
class Subject {
public:
  Subject();
  virtual ~Subject(){};
  virtual void attach(Observer*);
  virtual void detach(Observer*);
  virtual void notify();
```

```
  virtual void setChanged();
private:
  list<Observer*> observers; // list to hold observers
  bool changed;
};
void Subject::setChanged() {
  changed = true ;
}
Subject::Subject() : changed(false) {}

void Subject::attach (Observer* o) {
  observers.push_back( o);
}
void Subject::detach (Observer* o) {
  observers.remove(o);
}
void Subject::notify () {
  // notify observers if changed flag is set
  if ( changed ) {
    list<Observer*>::iterator i = observers.begin();
    for ( ; i!= observers.end(); i++ ) {
      (*i)->update(this);
    }
    changed = false;
  }
}
```

Beispiel: Temperatur-Sensor Simulation

- Simulierter Temperatur–Sensor als Subject
  - erbt von Subject
  - weiss nix von den Observern
  - ruft *Subject::setChanged()* wenn sich was ändert
  - Rest macht *Basisklasse Subject*
- Temperatur–Display als Observer
  - implementiert die Observer Methode *update()*
  - fragt dann den Temperatur–Sensor via *getReading()* nach aktuellem Wert

```
#include <list>
#include <iterator>
#include <iostream>
using namespace std;
// real classes using the Observer
class Sensor : public Subject { // simulates temperature sensor
private:
  int temp;
public:
  Sensor();
```

```
  void takeReading();
  int getReading();
};
Sensor::Sensor() {
  temp = 20; // set some start value
}
void Sensor::takeReading() {
  double d;  // randomly change temperature in 50% of times
  d = rand()*1./(RAND_MAX+1.0);
  if(d>0.75) {
    temp++;
    setChanged();
  } else if (d<0.25) {
    temp--;
    setChanged();
  }
  cout << " [Temp: " << temp << "] ";
}
int Sensor::getReading() {
  return temp;
}
class Display : public Observer {
public:
  virtual void update(Subject * o);
};
void Display::update(Subject * o) {
  Sensor * s = dynamic_cast< Sensor *>(o);
```

```
  if ( s ) {
    cout << "New Temp: " << s->getReading();
  }
}
int main()
{
  Sensor *sensor = new Sensor();   // subject
  Display *display = new Display(); // Observer
  // register observer with observable class
  sensor->attach(display); // Simulate measuring temp over time
  for(int i=0; i < 20; i++) {
    sensor->takeReading();
    sensor->notify();
    cout << endl;
  }
  return 0;
}
```

Im Beispiel wird *notify()* von aussen (=main()) angestossen. Kann aber genauso direkt vom *Subject Sensor* gemacht werden.

5.5 Aufgaben

1. **Singleton**
   Gibt es Alternativen zum Singleton, d.h. kann ich dieselbe Funktionalität anders erreichen ?

2. **Observer**
   Testen Sie das vorgestellte Beispiel mit dem C++ Subject/Observer. Fügen Sie weitere *Observer* dazu.

6 Objektorientiertes Design – Prinzipien

Es gibt eine Reihe wichtiger Grundprinzipien, die man beim Design von Objektorientierten Programmen beachten sollte.

Zentraler Punkt: **Abhängigkeiten minimieren**

Object-oriented programmings most important benefit is that it lets us reduce software complexity by managing code interdependencies.
(Herb Sutter, Exceptional C++)

In der Literatur werden die wichtigen Designprinzipien mit mehr oder weniger griffigen Akronymen aufgeführt ... ein kurzer Überblick:

KISS *Keep It Simple, Stupid*

Allgemeines Grundprinzip, Programmieren, insbesondere in C++, ist schon schwierig genug.

- Abhängigkeiten minimieren
- Programmstruktur simpel halten
- Standard Komponenten verwenden
- ...

DRY *Don't Repeat Yourself*

Komponenten sollten eindeutig und klar erkennbar genau einmal vorhanden sein. Keine mehrfachen, leicht unterschiedlichen Variationen

Gegensatz = WET – Write Everything Twice

SOLID *Akronym aus Akronymen*

SRP *Single Responsibilty Principle* Klasse soll genau eine wohl-definierte Aufgabe haben.

OCP *Open-Closed Principle* Offen für Erweiterung, geschlossen für Veränderung

LSP *Liskov Substitution Principle* Wenn Programm mit Objekt einer Basisklasse funktionert sollte es auch genauso mit Objekt einer davon abgeleiteten Klasse funktionieren *(aber nicht umgekehrt)*

ISP *Interface Segregation Principle* Besser klein-teilige Interface Klassen als große General-Purpose Interface Klassen

DIP *Dependency Inversion Principle* Software Komponenten sollten von Interfaces abhängen, nicht von konkreten Implemntierungen.

SRP – Single Responsibilty Principle

Klasse soll genau eine wohl-definierte Aufgabe haben.

Beispiel wie man's **nicht** machen soll:

```
class Circle {
private:
  Position p;
  double radius;
  Color c;
  FillArea f;
public:
  double getArea();
  double getCirumference();
  double move(Position np);
  // ...
  void setColor(Color nc);
  void setFillArea(FillArea fa);
  void draw();
  //..
}
```

Klasse `Circle` ist zuständig sowohl für geometrische Daten und Methoden (Position, Radius, Fläche) als auch für die Darstellung (Color, draw(), ...)

⇒ **getrennte Klassen**

Siehe auch Beispiel `copy` Funktion.

LSP – Liskov Substitution Principle

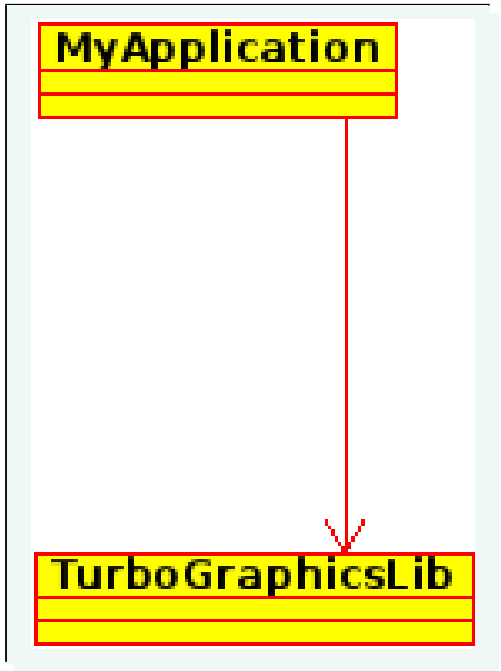
Verhalten von Objekten aus abgeleiteten Klassen muss konsistent mit Verhalten von Objekt aus Basisklasse sein, d.h. überall wo Basisklassenobjekt verwendet wird kann genauso auch abgeleitetes Objekt verwendet werden.

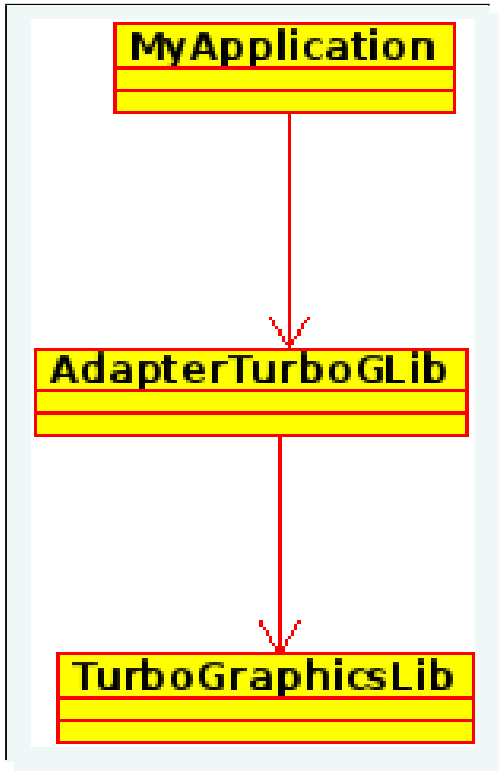
Siehe Beispiel Rechteck vs Quadrat

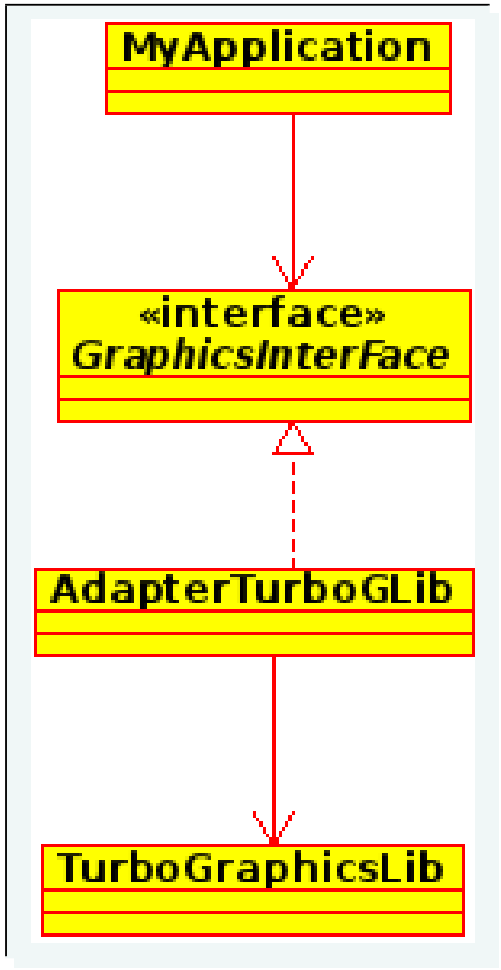
DIP – Dependency Inversion Principle

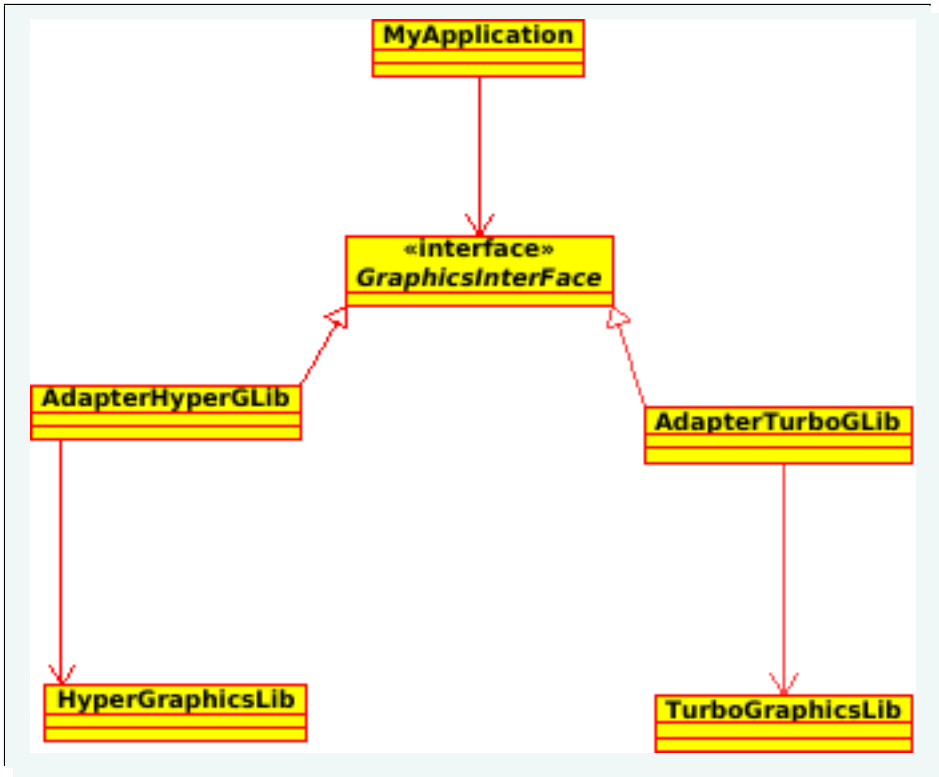
Wieder **Minimierung von Abhängigkeiten**:
Anwendungscode soll möglichst vermeiden direkt konkrete Klassen/Funktionen von externen Tools zu rufen, sonst extreme Abhängigkeit zwischen Anwendung und Tools-Lib.

1. Adapterklasse/funktion dazwischen schalten $\Rightarrow$ Abhängigkeiten in dieser Klasse konzentriert
2. Weiteren Interface-Layer $\Rightarrow$ Anwendungscode kennt/nutzt nur Interface, Adapterklasse implementiert Interface. Damit keine direkte Abhängigkeit mehr zwischen Anwendungscode und Tools-Lib.
3. Damit vglw. einfach auf alternative Tools-Lib umzustellen, nur ensprechende Adapterklasse nötig.


MyApplication
TurboGraphicsLib


MyApplication
AdapterTurboGLib
TurboGraphicsLib


MyApplication
«interface»
GraphicsInterFace
AdapterTurboGLib
TurboGraphicsLib


MyApplication
«interface»
GraphicsInterFace
AdapterHyperGLib
AdapterTurboGLib
HyperGraphicsLib
TurboGraphicsLib

7 Exceptions, Templates und STL

- Fehlerbehandlung und Exceptions
- Beispiele für Templates
- STL (Standard Template Library)
- Container
- Iteratoren
- Algorithmen
- Aufgaben

7.1 Exceptions

Fehlerbehandlung in C++

In allen *richtigen* Programmen spielt die Behandlung von Fehlern eine zentrale Rolle.

Programme sollen **robust** sein, d.h. sie sollen falsche Eingaben, extreme Datenwerte oder generell unerwartete Situationen vernünftig abfangen. Das Programm soll weder ausser Kontrolle in einem undefinierten Zustand enden noch bei jeder Kleinigkeit abgebrochen werden.

Beispiel:

```
double fak( int n )
{ // Funktion zur Berechnung der Fakultaet
  double t = 1.;
  for ( int i = 2; i <= n; i++ ) {
    t *= i;
  }
  return(t);
}
```

Funktion völlig korrekt, verlässt sich aber darauf dass Funktion *richtig* verwendet wird, d.h. dass Eingabewerte bestimmte Bedingungen erfüllen (`n >= 0, n < NMAX`). **Unzureichend** für allgemeine Anwendungen, keine **production quality**, Überprüfung und ggf Aktion nötig.

Klassisch gibt es 2 Methoden:
Die **sanfte** mittels Rückgabewert und `if` Abfragen:

```
double fak( int n )
{ // Funktion zur Berechnung der Fakultaet
  if ( n < 0 ) {
    cout << "Error in fak: Argument < 0 " << n << endl;
    return(-1.);
  }
  if ( n > NMAX ) {
    cout << "Error in fak: Argument > NMAX " << n<< endl;
    return(-2.);
  }
  double t = 1.;
  for ( int i = 2; i <= n; i++ ) {
    t *= i;
```

```
  }
  return(t);
}
```

Nachteile:

- u.U. viele Abfragen nötig $\Rightarrow$ unleserliche Programme
- schwer alle Möglichkeiten abzudecken
- check kann nicht erzwungen werden, z.B. in C
  `int fd = open("file.dat",0)` ist ein return Wert (`fd = -1`) Zeichen für Probleme beim Öffnen des files, aber das **muss** nicht behandelt, sondern kann einfach ignoriert werden.
- Rückgabewerte die Problem anzeigen nicht allgemein festzulegen, spezifisch für jede Funktion, manchmal gar nicht möglich, z.B. `log(-4.)`
- generell schlechtes Programmdesign: Rückgabewert soll eine Aufgabe erfüllen und nicht abhängig vom spezifischem Kontext mehrere Rollen spielen.

2. Alternative, die **brutale** mittels `exit()` call bei Fehler. Definiertes Verhalten, allerdings keine Fehlerbehandlung im laufenden Programm mehr möglich.

```
double fak( int n )
{ // Funktion zur Berechnung der Fakultaet
  if ( n < 0 ) {
    cout << "Error in fak: Argument < 0 " << n << endl;
    exit(1);
  }
  if ( n > NMAX ) {
    cout << "Error in fak: Argument > NMAX " << n<< endl;
    exit(2);
  }
  double t = 1.;
  for ( int i = 2; i <= n; i++ ) {
    t *= i;
  }
  return(t);
}
```

Beispiel File-Öffnen, sinnvoll je nach Situation:

- Neuer Versuch mit anderem Namen oder anderem Pfad: `int fd = open("/home/gduckeck/file.dat",0)`
- Netzwerkprobleme, lieber nach 5 min nochmal probieren: `sleep(300); int fd = open("file.dat",0)`

Wesentlich flexibler und eleganter in C++ mit **exceptions** (=Ausnahmen).

- Bei Fehler wird an der Stelle an der das Problem auftritt eine exception ausgelöst: `throw( "Trouble opening file" )`
- Im rufenden Progamm kann die **exception** behandelt werden mittels `try { ... }` und `catch (..) { ... }` Blöcken
- Andernfalls wird das Programm abgebrochen (abort())

```
#include <vector> // vector headers
#include <iostream>
#include <cmath>
#include <stdexcept>
using namespace std;
double root(double A, double B, double C)
{
  // Returns the larger of the two roots of
  // the quadratic equation A*x*x + B*x + C = 0.
  // (Throws an exception if A == 0 or B*B-4*A*C < 0.)
  if (A == 0) {
    throw ("A can't be zero.");
  }
  else {
    double disc = B*B - 4*A*C;
    if (disc < 0)
```

```
        throw ("Discriminant < zero.");
      return  (-B + sqrt(disc)) / (2*A);
    }
}
int main()
{
  //   cout << root( 0., 1., 1.) << endl;
  try {
      cout << root( 0., 1., 1.) << endl;
  }
  catch (char const *message ) {
    cout << "root failed: " << message << endl; // print error

  }
  try {
      cout << root( 1., 2., 1.5) << endl;
  }
  catch (char const *message ) {
    cout << "root failed: " << message << endl; // print error

  }
}
```

Ähnlich wie Funktionsaufruf: `throw( Type a )` bewirkt Verzweigung nach passendem `catch ( Type x)`.

- Allerdings wird nicht wie bei Funktionsaufruf in Funktion nach **unten** verzweigt und anschliessend zurück.
- Genau das Gegenteil passiert: es wird solange die rufenden Funktionen nach **oben** *(im sog. call-stack)* zurückgegangen bis `try` mit passendem `catch()` gefunden wird.
- Falls nicht bricht Programm ab.

if Abfragen vs **Exceptions** ist klassisch–philosophisches IT–Problem, siehe z.B. **LBYL** vs **EAFP**.

Leider ist das exception–Konzept erst nachträglich in C++ aufgenommen worden

- nicht wirklich in Sprache integriert
- nur wenige Standardfunktionen nutzen exceptions
- wird aber zunehmend verwendet in modernen C++ Bibliotheken

Beispiel mit STL Vektoren:

```
#include <vector> // vector headers
#include <iostream>
#include <stdexcept>
using namespace std;
int main()
{
   vector< double > v(10,1.); // <double> vector, 10 elements filled with 1
   cout << v[5] << endl; // ok, exists
   cout << v[15] << endl; // doesn't exist but program continues
   cout << v.at(5) << endl; // ok
   cout << v.at(15) << endl; // doesn't exist, throws exception, abort
   try {
     cout << v.at(15) << endl; // doesn't exist, throws exception, catched
   }
   catch (out_of_range e) {   // pre-defined logical exception class is thrown
     cerr << "Out of range exception " << e.what() << endl; // print error
     exit(1);     // stop program or take other action
```

```
    }
}
```

Anders in JAVA oder Python, exceptions sind integraler Bestandteil der Sprache: *I/O, array bounds, mathematische Operationen, ...*

7.2 Template Funktionen

Oft kann man den gleichen Algorithmus für mehrere Klassen gebrauchen.

- MAX (wenn $>$ für die Klasse definiert)
- x^2 und x^n (wenn $*$ für die Klasse definiert)
- Daten Verwaltung (Listen, einfügen, sortieren, etc.)

```
#include <iostream>
#include <string>
using namespace std;
template < class T>        // declare template function
T Max(const T &x,const T &y)
{
  return (x < y ? y:x);
}
int main()
{
  double a=3.2,b=4.2;
  int c=4,d=7;
```

```
  string s="Hallo",t="Hello";
  cout << "Max(a,b) :" << Max(a,b) << endl;
  cout << "Max(c,d) :" << Max(c,d) << endl;
  cout << "Max(s,t) :" << Max(s,t) << endl;
};
```

Der Pre-Compiler erstellt die Funktionen

```
double Max(const double &x,const double &y)
{
  return (x < y ? y:x);
}
int Max(const int &x,const int &y)
{
  return (x < y ? y:x);
}
string Max(const string &x,const string &y)
{
  return (x < y ? y:x);
}
```

7.3 Template Klassen

Analog für Klassen Definitionen:

```
#include <iostream>
template < class T1, class T2>
class pair {
public:
  T1 first;
  T2 second;
  pair(const T1& x,const T2&y): first(x),second(y){}
};
int main()
{
  pair <double,int> p1(3.14,7);
  pair <int,double> p2(9,4.4);
  cout << "p1:" << p1.first << " " << p1.second << endl;
  cout << "p2:" << p2.first << " " << p2.second << endl;
};
```

Der Pre-Compiler erstellt:

```
class pair_di {
public:
  double first;
  int second;
  pair(const double & x,const int &y): first(x),second(y){}
};
class pair_id {
public:
  int first;
.
  pair_di p1(3.14,7);
  pair_id p2(9,4.4);
..
```

Anmerkung:

In obigem Beispiel wurde eine besonders kompakte Variante von Verzweigungen in **C/C++** verwendet:

```
// Kurzform
 r = (x < y ? y:x);
// entspricht
if ( x < y ) {
   r = y;
 }
else {
   r = x;
}
```

Generic Programming

Templates ermöglichen das sog. *Generische Programmieren*

- Ausgehend von konkretem Algorithmus reduziert man den Code auf die minimalen semantischen Anforderungen.
- Konkrete Datentypen werden möglichst durch Template Typen ersetzt.
- Dadurch maximale Wiederverwendbarkeit.

7.4 STL

Standard Bibliothek mit Template Klassen und Funktionen um Daten zu verwalten.

Wichtige Komponenten:

- Container: Enthält die Daten (z.b. Array, Liste etc.)
- Iterator: Eine Art *Pointer*, ermöglicht Zugriff auf Daten im Container.
- Algorithmen: angewand auf die Daten in den Containern, z.B. suchen, sortieren, etc.
- ...

Grundlegende Problematik:

- `N` Datentypen
- `M` Container
- `K` Algorithmen

$\Rightarrow$ `N*M*K` Klassen & Funktionen nötig

Abhilfe durch STL:

- Templates $\Rightarrow$ `N = 1`
- Generische Algorithmen & Iteratoren `M*K` $\Rightarrow$ `M+K`

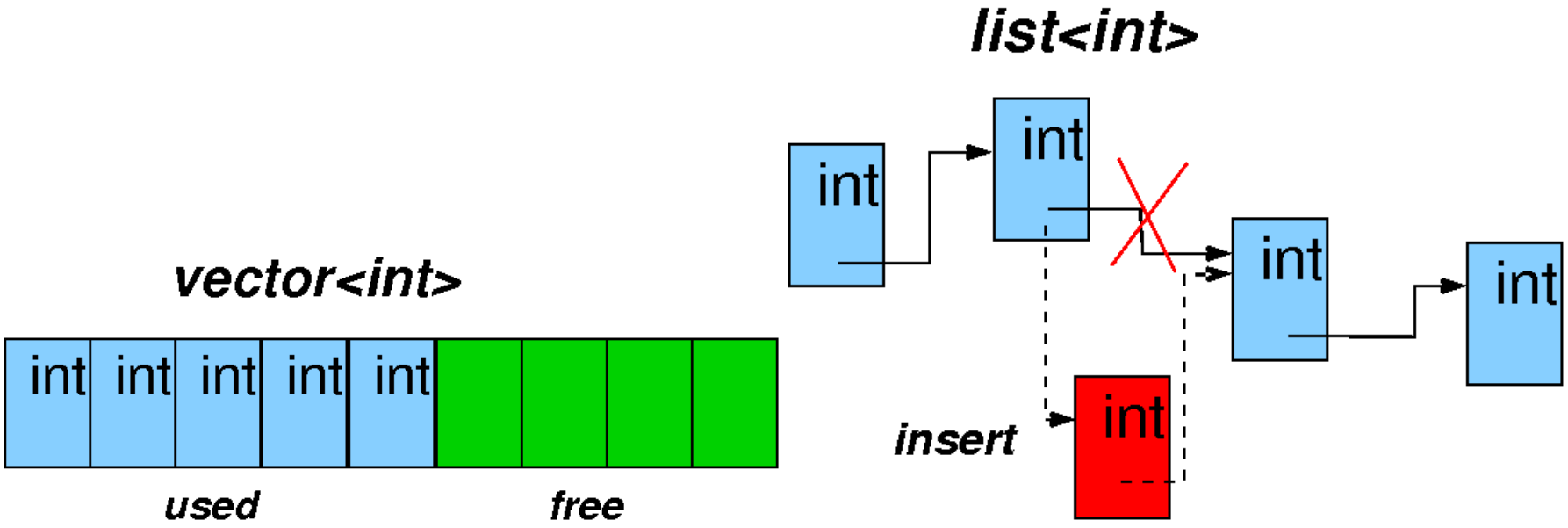
Wir müssen uns auf einführende Beispiele beschränken ...

7.5 Container

Zwei grundlegende Arten von Containern:

- **Collections:** Ansammlung von beliebigen Objekten. Beispiele sind:
  - `T a[n]`, der normale Array, besitzt random access, d.h. alle Elemente werden gleichschnell angesprochen. Die Länge wird beim Anlegen festgelegt, danach nicht mehr änderbar.
  - `vector<T>`, random access, variable Länge, schnelles anhängen am Ende.
  - `deque<T>`, random access, variable Länge, schnelles anhängen am Anfang und Ende.
  - `list<T>`, schnelles Einfügen an jeder Stelle. Aber Zugriffszeit hängt von der Länge ab.
  - `set<T>`, eine Art sortierte Liste, jeder Wert kommt nur einmal vor.
- **Map oder assoziativer Array**, *key–value* Paare werden zusammen abgespeichert. Zugriff erfolgt über *key*.

Im Zweifelsfall `vector<T>` nehmen ...


list<int>
int
int
int
int
vector<int>
int
int
int
int
int
used
free
insert
int

Wichtige Zugriffsfunktionen:

- `push_back(T &x)` Anhängen am Ende
- `pop_back()` Löschen am Ende
- `push_front(T &x), pop_front()` analog
- `insert(iterator position, T &x)` Einfügen an bestimmter Position
- `erase(iterator anfang, iterator ende)` Bereich löschen
- `operator []` Random access auf Elemente (Lesen & Schreiben), nicht bei `list<T>`

Standard–Iteratoren:

- `begin()` Anfang
- `end()` Ende

Und noch die Grösse:

`size()`

```
#include <vector> // vector headers
#include <iostream>
using namespace std;
int main()
{
    vector<double> v; // empty <double> vector
    double x;
    while ( cin >> x ) { // read input until EOF
      v.push_back(x); // append in vector
    }
    // classical loop, mit Index
    for (int i = 0; i < v.size(); i++ ) {
      cout << v[i];
    }
    // container loop mit Iterator
    for ( vector<double>::iterator vp = v.begin(); vp != v.end(); vp++ )
      {
        cout << *vp ; // de-referencing as for pointers
      }
    // am einfachsten mit C++-11 auto und foreach loop
```

```
    for ( auto vx : v )
      {
        cout << vx ; // vx enthaelt jeweils direkt Element
      }
}
```

Wozu Iteratoren?

Zugriff auf einzelne Elemente direkt mit Index (`v[i]`) funktioniert und ist leicht verständlich, wozu also den Umstand mit **Iteratoren** ?

- Index geht nicht für alle Containerarten, Iteratoren schon, z.B. für *list*.
- Flexibilität: Wenn man von vorneherein Iteratoren benutzt, kann man später leicht den Container ändern, ohne sonst das Programm ändern zu müssen.
- Algorithmen und Container member functions benutzen Iteratoren

7.6 Algorithmen

find :

```
#include <iostream>
#include <cstring>      // C std string ops
#include <algorithm>  // STL algorithms, find ...
int main()
{
  char* s = "C++ is the better C";
  int len = strlen(s);
  char * where = find(&s[0],&s[len],'e');
  cout << *(where) << *(where+1) << endl;
};
```

- `& s[0], & s[len]` sind Pointer auf den Anfang und Ende des Bereiches den `find` bearbeiten soll
- Der Rückgabewert von `find` ist ein Pointer (Iterator)
- Der Algorithmus von `find` ist eine Template Funktion, die für beliebige Datentypen verwendet

werden kann, eingebaute Typen als auch Klassen.

Einzige Voraussetzung ist dass der `==` Operator für die Klasse definiert ist.

sort :

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;
int main()
{
  const char* s = "C++ is the better C";
  int len = strlen(s);
  vector<char> vec1(&s[0],&s[len]); // vector of chars created and filled
  cout << vec1.size() << endl;
  sort( vec1.begin(), vec1.end() );
  cout << vec1.data() << endl; // .data() returns char array
}
```

- Der Constructor erzeugt einen vector der mit dem Array von `& s[0]` bis `& s[len]` gefüllt wird.

- `vec1.begin(),vec1.end()` sind Iteratoren, die auf den Beginn bzw. das Ende des Vectors Zeigen.
- Der Algorithmus von `sort` ist eine Template Funktion, die für beliebige Datentypen verwendet werden kann, nur der `<` Operator muss definiert sein.
- Für Iteratoren sind die Operatoren so definiert, dass man sie wie Pointer benutzen kann.

Viele Varianten:

- `partial_sort` nur Teil des Containers wird sortiert, z.B. wenn man nur die 3 kleinsten Elemente will
- `nth_element` macht nur Teil-Sortierung dass Element an n-ter Stelle geliefert wird.
- `stable_sort` bei gleichen Elementen bleibt bisherige Reihenfolge erhalten. Sehr nützlich wenn man mehrfach sortiert nach verschiedenen Eigenschaften, z.B. Studentendaten: Erst Studienfach, dann Semesterzahl, dann Alter, ...

accumulate : Aufsummieren

```
#include <numeric>
  ...
  int* ia = { 1, 4, 9, 16, 25 };
  vector<int> vi(ia, ia+5); // vector mit ints
  int sum = accumulate( vi.begin(), vi.end() , 0);
  ...
```

reverse : Reihenfolge umdrehen

```
#include <numeric>
  ...
  vector<char> vec1(&s[0],&s[len]); // vector mit chars
  reverse( vec1.begin(), vec1.end() );
  ...
```

random_shuffle : in zufällige Reihenfolge bringen

```
#include <numeric>
  ...
  int* ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
  vector<int> vi(ia, ia+5); // vector mit ints
  random_shuffle( vi.begin(), vi.end() ); // in zufaellige Reihenfolge bringen
  ...
```

copy : Kopieren zwischen Containern

```
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
  ...
  vector<char> vec1(&s[0],&s[len]); // vector mit chars
  list<char> list1; // liste mit chars
  copy( vec1.begin(), vec1.end(),      back_inserter(list1) ));
  ...
```

`back_inserter()` ist template–iterator function analog zu `push_back()`

Iterator/copy Mechanismus sehr flexibel:

```
istream_iterator<double> inp(cin);
istream_iterator<double> eof;
vector<double> vec2;
copy( inp, eof, back_inserter(vec2) );
```

Liest Daten von Standard-Eingabe und packt sie in Vektor (beliebig viele, bis zum end-of-file)

analog die Ausgabe

```
#include <vector>
#include <iterator>
ostream_iterator<double> out(cout, "\n" );
copy( vec2.begin(), vec2.end(), out );
```

partition : Elemente in Container finden die bestimmte Bedingung erfüllen:

```
// partition algorithm example
// from http://www.cplusplus.com
//
#include <iostream>     // std::cout
#include <algorithm>    // std::partition
#include <vector>       // std::vector
#include <iterator>
using namespace std;
bool IsOdd (int i) { return (i%2)==1; }
int main () {
  vector<int> vec;
  // set some values:
  for (int i=1; i<10; ++i) vec.push_back(i); // 1 2 3 4 5 6 7 8 9
  vector<int>::iterator bound;
  bound = partition (vec.begin(), vec.end(), IsOdd);
  //bound = stable_partition (vec.begin(), vec.end(), IsOdd);
  // print out content:
  cout << "odd elements:";
```

```
  ostream_iterator<int> out(cout, " " );
  copy( vec.begin(), bound, out );
  cout << endl;
  cout << "even elements:";
  copy( bound, vec.end(), out );
  cout << endl;
  return 0;
}
```

Variante **stable_partition** : Ändert nicht die bisherige Reihenfolge.

Und noch viele weitere Algorithmen

siehe

CppReference

oder

105 STL algorithms ...

Wann immer möglich STL–Algorithmen verwenden statt eigene Schleifen zu programmieren

$\Rightarrow$ **higher level of abstraction**

7.7 Arbeiten mit vector<>

Der vector Container ist der gebräuchlichste STL Container. Er kann ähnlich wie der normale Array verwendet werden, und sollte in fast allen Fällen stattdessen benutzt werden.

- Initialisierung:
  - Leeren Vektor anlegen:
    `vector <double> vd;`
    Grösse ist 0. Noch **kein** Zugriff mit `vd[..]` möglich !
    Wird verwendet wenn anschliessend mit `vd.push_back(..)` oder `copy` gefüllt wird.
    Optional kann auch nachträglich die Grösse gesetzt werden, z.B. `vd.resize(100)`
    Dann Zugriff mit z.B. `vd[17]` ok.
  - Grösse angeben: `vector <double> vd(50);`
    vector wird mit 50 Elementen angelegt. `vd.push_back(..)` fügt ans Ende ($51, 52, ..$) weitere Elemente an.
  - Grösse und Wert für alle Elemente angeben:
    `vector <double> vd(50, double-wert);` s.o, zusätzlich werden Elemente mit übergebenem Wert intialisiert.
- Viele praktische und sehr flexible Algorithmen verfügbar
  `copy, acccumulate, find, sort, ...`

z.B. in 3 Zeilen *genom.txt* in `vector<char>` einlesen:

```
ifstream inf("genom.txt");
istream_iterator<char> inp(inf);
vector<char> vec2;
copy( inp, istream_iterator<char>(), back_inserter(vec2) );
```

- Vektoren kann man verschachteln:

  ```
  vector< vector< double > > vvd;
  ```

  Damit deklariert man einen vector der weitere vectors vom Typ double enthält. *Zunächst sind aber alle diese vectors leer!*

  Damit kann man z.B. einen 2-dim Arrray oder Matrix emulieren

  ```
  vector< vector< double > > vvd(4, vector<double>(6));
  ```

  legt eine Matrix mit 4 Zeilen und 6 Spalten an. Zugriff auf die Elemente geht mit z.B. `vvd[3][2]`

- Gängige Fehler mit Vektoren:

  – Leer angelegt `vector<int> vi;` aber Zugriff mit `[..]` Operator bevor etwas eingefügt wurde oder Grösse gesetzt wurde.

  Insbesondere bei verschachtelten Vektoren:

  ```
  vector< vector< int > > vvi(10);
  ```

  Legt nur den *ausseren* Vektor an, d.h. `vvi[4]` existiert, ist aber leerer Vektor vom Typ int.

– Zugriff ausserhalb des angelegten Bereiches. Beim Zugriff auf Elemente mittels `[..]` oder *iterators* wird **nicht** überprüft ob man auf gültigen Bereich zugreift.

  Am besten immer `size()` oder `begin()` bzw. `end()` mittesten:

```
for(vector <char> ::iterator p=vec1.begin();
    (p !=vec1.begin()+30)&&(p !=vec1.end()); p++) {
  cout << *p;
};

for(int i=0; (i<30)&&(i<vec1.size()); i++) {
  cout << vec1[i];
};
```

7.8 Function Objects

Ein neues Element sind die *function objects*, sie sind wichtig für die generischen STL–Algorithmen.

Z.B. der `sort` Algorithmus: Beim Aufruf mit 2 Argumenten (Anfang und Ende des Container–Bereichs) wird implizit der `<` Operator für die jeweilige Klasse benutzt, d.h. Sortieren in aufsteigender Reihenfolge.

Das umgekehrte, d.h. sortieren in absteigender Reihenfolge kann man erreichen mit:

```
vector<string> vecstr(...); ...  ;
sort(vecstr.begin(), vecstr.end(), greater<std::string>());
```

`greater<std::string>()` ist ein *Konstruktor*:
Es wird ein Objekt der `greater<>()` template Klasse für `string` angelegt.

In dieser Klassse ist der `()` Operator definiert, ungefähr so:

```
bool operator () (const string & s1; const string & s2 ) {
    return( s1 > s2 ); }
```

Solche Function Objects kann man auch selbst definieren, hier ein Beispiel für die Dreier-Vektoren:

```
#include "myvec.hxx"
class Comp3VecX { // "<" Vergleich fuer DreierVektor x-Komponente
public:
  bool operator () (const My3Vector & v1, const My3Vector & v2) {
    return( v1.X() < v2.X() );
  }
};
int main () {
  vector<My3Vector> vvec; // empty <My3Vector> vector
  //.. fill it
  sort(vvec.begin(), vvec.end()); // sortieren mit "<" Operator
  // sortieren mit function object, nur x-Komponente
  sort(vvec.begin(), vvec.end(), Comp3VecX());
}
```

Comp3VecX ist eine Art Hilfs–Klasse für My3Vector, einziges Member ist der `()` Operator.

7.9 Maps

Eine andere Art von Container sind die *Maps*.

Bei normalen Arrays oder Vectors läuft der Zugang über eine Index-Nummer, d.h. assoziiert Index mit dem Objekt

Maps dagegen speichern Paare von Werten, den **key** und den **value**.

Zugriff auf die Elemente erfolgt dann über den **key**, man nennt maps auch assoziativer Array.

Beispiel Wörterbuch:

```
#include <map>
int main()
{
  map< string, string > engdeut;
  engdeut["hello"]    = "Hallo";
  engdeut["world"]    = "Welt";
  engdeut["computer"] = "Rechner";
  engdeut["physics"] = "Physik";
  engdeut["physicist"] = "Physiker";
  engdeut["physician"] = "Arzt";
```

```
}
```

Verwendung:

- Initialisierung: `map< key-type, value-type> name`
  key und **value** können beliebige Typen/Klassen sein, wobei für **key** die Operatoren `"=="` und `"<"` definiert sein müssen.
- Zugriff: Direkt mit Angabe eines *keys* als *array-index*, z.B. `engdeut["physics"]`.

  Oder sequentiell durchlaufen mit Iteratoren:

```
for ( map<string,string>::iterator it = engdeut.begin();
  it != engdeut.end(); ++it ) {
  cout << setw(20) << it->first <<  // first => key
    setw(20) << it->second << endl; // second => value
    }
```

- Test ob Element schon angelegt ist:
  `if ( engdeut.count("Bremsstrahlung") > 0 ) ....`
  `map::count(key)` liefert 0 oder 1 zurück

Eine Map ist eine Art Liste von Paaren. Diese Liste ist sortiert nach dem *key*. Deshalb:

- Operatoren `"=="` und `"<"` müssen für *key* definiert sein.
- Jeder *key* kann nur einmal vorkommen. Bei erneuter Zuweisung wird existierender Wert überschrieben:
  `engdeut["computer"] = "Gombuder"; // fraenkische Version`
- Zugriff viel langsamer und umständlicher als normaler Array/vector über numerischen Index. Bei grossen Maps bzw. zeit-kritischen Anwendungen besser *Hash-maps* verwenden.

7.10 Aufgaben

1. **Quadratische Gleichung**
   Erstellen Sie ein Programm zur Lösung der quadratischen Gleichung

   $$A\,x^2 + B\,x + C = 0$$

   basierend auf obigem Beispiel `root`
   - Die Koeffizienten `A,B,C` von standard-input lesen
   - **try–catch** im rufenden Programm um auf die exceptions *sinnvoll* zu reagieren.

2. **Idiotensichere Fakultät**
   Modizfizieren Sie ihre Funktionen zur Berechnung der Fakultät so, dass kein Überlauf mehr auftreten kann, d.h. lösen Sie eine exception aus
   `throw ("Number too large")`
   wenn der Wert zu gross wird.

   Hinweis: Mit `LONG_MAX` *oder* `DBL_MAX` *(definiert in limits.h bzw. float.h) kann man die Grenzen der Zahlenbereiche abfragen. Jetzt muss man nur noch überlegen wie man's programmiert* **ohne** *dass zunächst die Grenze überschritten wird.*

3. **Template**
   Falls Sie Templates zum erstenmal sehen sollten Sie erstmal mit Funktionen üben:

Schreiben Sie Template Funktionen um x^2, x^n, `Min()`, `Abs()` zu berechnen.

4. **Zahlen sortieren**
   In der Datei `numbers.dat` finden Sie eine Liste mit 100 Fließkommazahlen.
   - Lesen Sie die Zahlen ein und speichern Sie sie in einem vector.
     (a) Zunächst ”klassisch”, d.h. mit einer **for** oder **while** loop über den Container.
     (b) Dann ohne jegliche Loop, mit den STL **Iterator/copy** Methoden.
   - Sortieren Sie anschliessend einmal in aufsteigender und dann in abfalleder Reihenfolge.

   Ein Beispiel zum Einlesen aus Datei ist in iovec.cpp .

5. **Poker simulieren**
   Mit *vector* und dem *random_shuffle* Algorithmus kann man leicht Spiele simulieren, und damit die Wahrscheinlichkeit für bestimmte Kombinationen abschätzen (ohne sich in den Feinheiten der Kombinatorik zu verirren). Simulieren Sie z.B. das Pokerspiel, was ist die Wahrscheinlichkeit ein Full-House auf die Hand zu bekommen ?

   Lösungsbeispiel: (html, source),

6. **Function Object zum Dreier-Vektor sortieren**
   Container und Algorithmen funktionieren auch für eigene Klassen/Objekte. Testen Sie das speichern in vector und sortieren mit zufällig erzeugten Dreier-Vektoren:

```
#include <vector> // vector headers
#include <cstdlib>
#include "myvec.hxx"
using namespace std;
int main()
{
  vector<My3Vector> vvec; // empty <My3Vector> vector
  for ( int i = 0; i<200; i++ ) {
    // create random three-vectors
    My3Vector p( rand()*10./(RAND_MAX+1.0),
          rand()*10./(RAND_MAX+1.0),
          rand()*10./(RAND_MAX+1.0) );
    // add My3Vector at the end of the vector
    vvec.push_back( p );
  }
  // sort My3Vectors
  // ...
      // output
```

Erstmal das direkte sortieren ausprobieren, dann Function Object benutzen und nach x oder y Komponente sortieren.

(My3Vector Header `myvec.hxx` und code `myvec.cxx`.)

7. **Vorwahl–Map**

   In der Datei `vorwahl.txt` stehen alle Vorwahlen und zugehörige Orte in Deutschland. Lesen Sie die ein, speichern Sie's in einer Map und machen damit ein kleines Programm das zu einer gegebenen Vorwahl den Ort ausgibt, und umgekehrt.

   Ein Beispiel zum Zeilen-weisen Einlesen aus Datei ist in vorwahl.cpp .

8. **Genom Projekt**

   Eine DNA Sequenz kann als Array von N Char Werten dargestellt werden (N sehr gross). Das Problem ist nun wiederkehrende Strukturen zu finden, d.h. Patterns der Länge M, wobei M fix und klein ist. In der Datei genom.txt finden Sie einen Abschnitt einer solchen DNA Sequenz. Überlegen Sie Algorithmen um signifikant häufige Patterns für vorgegebene Länge M zu finden.

   Lösungsbeispiel: (html, source),

   Human Genome Project: html

8 C++11

8.1 C++11 – Kurzer Überblick

C++ wird kontinuierlich weiterentwickelt und typisch alle 3-4 Jahre wird ein neuer Standard veröffentlicht. Mit C++11 (2011 veröffentlicht) wurden besonders viele und weitreichende Änderungen eingeführt, die gängige Programmierprobleme betreffen bzw. vereinfachen.

Vor der Veröffentlichung war C++11 unter dem Namen C++0x bekannt. Es existiert noch zahlreiche Literatur, die den alten Namen verwendet, d.h. C++11 und C++0x sollten als Synonyme betrachtet werden. Der Standard selbst ist nur ein langes Dokument mit Spezifikationen und es ist Aufgabe verschiedener Gruppen oder Firmen entsprechende Compiler zu implementieren, die diesem Standard folgen.

Ab der GCC Version 4.8 werden fast alle Features von C++11 unterstützt, siehe (GCC C++11 Status). Details zur Unterstützung der verschiedenen C++ Versionen in der GCC Standard-Bibliothek finden sich in (libstdc++ Status). Alternativ bietet auch der populäre Open-Source Compiler Clang volle C++11-Unterstützung sowohl in der Hauptfunktionalität (Clang C++11 Status) als auch in der eigenen Version der Standard-Bibliothek (libc++ Status).

Momentan ist im CIP-Pool-Cluster (unter Linux Ubuntu 16.0.4) die GCC Version 5.4 verfügbar, diese Version unterstützt alle hier besprochenen Beispiele.

Es sollte aber erwähnt werden, dass oft einige spezielle Features verbleiben, die von den Compilern nicht unterstützt werden, z.B. unterstützt GCC immer noch nicht den vollen Umfang der alten C++98 oder C++03 Standards.

Insgesamt wurden in vielen Bereichen Änderungen gegenüber dem bisherigen Standard gemacht, die zum einen die Funktionalität erweitern aber zum andern auch die Bedienung vereinfachen oder robuster machen.

Im folgenden wird ein kleiner Teil dieser Änderungen kurz vorgestellt, Schwerpunkt ist Thread Programmierung in C++11.

Die Entwicklung geht natürlich weiter, mittlerweile gibt es die Standards **C++14** und **C++17**, die jeweils etliche weitere Änderungen und Erweiterungen beinhalten, allerdings eher für Experten ...

auto

Neues key-word erlaubt vereinfachte Deklaration von Variablen, wenn sich die Typ–Defintion aus dem Kontext eindeutig ableiten lässt, z.B.:

```
// C++ 11
  map< string, string > engdeut;
  for ( auto it = engdeut.begin(); it != engdeut.end(); ++it ) { ....

// C++ 03
  map< string, string > engdeut;
  for ( map<string,string>::iterator it = engdeut.begin(); it != engdeut.end(); ++it ) { ..
```

for-each

Vereinfachte Schleife über Container mit direktem Zugriff auf jedes Element (for-each Loop in anderen Sprachen), z.B.

```
// C++ 11
  vector<double> vec1; ...
  for( auto x :  vec1 ) {
    cout << x;
};

// C++ 03
  vector<double> vec1; ...
  for(int i=0; i<vec1.size(); i++) {
    cout << vec1[i];
};
```

tuple

Container um Objekte verschiedenen Typs und beliebiger Anzahl zu gruppieren (wie *pair*, aber auch für $>= 2$ Objekte):

```
// C++ 11
tuple<int,float,string> t(1,2.f,"text");
int x = get<0>(t);
float y = get<1>(t);
string z = get<2>(t);
```

lambda Funktionen

sind ”anonyme” Funktionen, sehr praktisch als Argument bei Aufruf von anderen Funktionen, z.B. STL Algorithmen die Funktion-Objekt bzw. Functor als Argument erwarten:

```
vector <double> vec2; ....
// lambda funktion als 3.  Argument, sortiert vector nach Betrag
sort(vec2.begin(), vec2.end(),  [](double x, double y) { return abs(x)<abs(y);}  );
```

Verschiedene Optionen auf momentane Umgebung in lambda Funktionen zuzgreifen:

- [] kein Zugriff auf Variablen
- [=] Variablen werden kopiert
- [&] Variablen Zugriff per Referenz
- u.a.

Speicher–Management – Smart Pointer

Smart Pointer mit memory management (*shared_ptr*)

Für bestimmte Probleme braucht man dynamisch angelegte Arrays oder Objekte, z.B. wenn Funktion Objekt zurückgeben soll.

Bisher in C++ kein automatisches Memory Management im Standard–Sprachumfang, Programmierer muss sorgfältig darauf achten alle mit `new` angelegten Speicherbereiche auch mit `delete` wieder zurückzugeben, sonst **memory leaks**.

Großes Problem dabei sind auch *dangling pointers*, d.h. Speicherbereich wurde mit `delete` zurückgegeben, es gibt aber noch pointer, die darauf zeigen und evt. verwendet werden.

Mit C++–11 und `shared_ptr` automatisches Memory Management und Referenz-Zähler.

```
//....
// C++-03
My3Vector * vp = new My3Vector(1.0, 2.5, -0.8 ); // dynamisch angelegter 3-Vektor
cout << vp->Length() ;
//...
My3Vector * vp2 = vp;
delete vp; // zurueckgeben
```

```
// vp2 ?? -> dangling pointer
//....
// ---------------------------------------
// C++-11
#include <memory>
//...
shared_ptr<My3Vector> vp(new My3Vector(1.0, 2.5, -0.8 )); // dynamisch angelegter 3-Vektor mit shared_ptr
cout << vp->Length() ; // wie std pointer zu verwenden
shared_ptr<My3Vector> vp2(vp);
//
// kein delete noetig, shared_ptr hat Referenz-count und raeumt automatisch auf wenn kein Verweis mehr aktiv
//
```

`shared_ptr` kann fast universell für Memory Management verwendet werden und erlaubt damit ähnlich flexible und einfache Nutzung wie in Java oder Python. Es gibt daneben auch noch `unique_ptr` und `weak_ptr`, für spezielle Memory Management Probleme.

C++11 weitere Features und Kompilieren

Es gibt darüberhinaus noch viele weitere nützliche Features, z.B.

- Threading
- Vereinfachte Initialisierung von Member Variablen und Container Elementen
- ...

In den meisten aktuellen Compilern ist die C++11 Funktionalität noch nicht standardmässig aktiviert, sondern muss über Optionen explizit eingeschaltet werden.

Für die Version von gcc am CIP pool geht das mittels Option *-std=c++11* , d.h. z.B.

```
g++ -std=c++11 -o myprog myprog.cpp
```

9 Einführung in Qt

- Qt ist etablierte Standard–Bibliothek für *Grafik* und *GUI*
- An sich kommerzielles System, aber Hersteller Trolltech (mittlerweile Nokia–Tochter) bietet auch leistungsfähige freie Version an.
- Qt erlaubt weitgehend *plattformunabhängige* Programmierung, verfügbar auf vielen Unix Systemen und für Windows.
- **KDE**, die Linux Standard–Benutzeroberfläche basiert auf Qt
- Qt inzwischen weit mehr als nur GUI Bibliothek, bietet viele Erweiterungen für C++ (Threading, Networking, DB access, u.v.m.). C++ mit Qt bietet vergleichbare Allround–Funktionalität wie JAVA-SDK.

Beispiele im Kurs für aktuelle Qt Version 4.5!

Was ist **GUI** *(= Graphical User Interface = Graphische Benutzer Schnittstelle)* ?

Jedes moderne, interaktive Programm (Editor, Browser, Spreadsheet, ...) macht Interaktion mit User über

- Buttons
- Menues
- Textfelder
- Grafik
- Slider
- Scrollbars
- ...

für Kontrolle, I/O, Visualisierung, usw.

9.1 Qt Komponenten

Vielzahl von Qt Komponenten für *Buttons, Labels, Menues, Textfelder, Zeichenfläche, Scoll–Bars, ...*

Hier nur kurzer Überblick. Qt Komponenten relativ leicht anzuwenden, wenn Prinzip von Benutzung klar ist.

Zum Anfang erst mal eine Art *Hello world* in Qt:

```
// Ein einfaches Qt-Programm, das nur einen beschrifteten Knopf zeigt.
#include <QApplication>
#include <QPushButton>
int main(int argc,char **argv)
{
  QApplication app(argc,argv); // Dieses Objekt braucht man fr jedes GUI-Programm.
  QPushButton *hello = new QPushButton("Hello QT-world!",0); // Ein Knopf (noch ohne Funktion)
  hello->resize(100,30);          // Gre von Hand bestimmen
  hello->show();                  // Sichtbar-machen des Haupt-Widget
  return app.exec();              // Hier beginnt die Event-Schleife
}
```

Jedes Qt Programm beinhaltet folgendes Grundgerüst:

- Ein Objekt der Klasse QApplication. Das ist eine Art *main–Objekt* für Qt, in das alle weiteren Qt GUI Komponenten eingebettet sind.

  `QApplication app(argc, argv);` (Argumente wie main).

- Eine oder mehrere GUI Komponenten werden erzeugt, z.B. ein Knopf (Button), und im Haupt-Widget sichtbar gemacht.

```
QPushButton *hello = new QPushButton("Hello I'm Qt!",0);
hello->show();
```

- Die QApplication wird gestartet:

```
app.exec();
```

Nochmal (fast) dasselbe Programm aber mit separater Klasse für die GUI Komponente, die von *QWidget* ableitet

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
class MyWidget : public QWidget  // erbt von QWidget
{
public:
  MyWidget(void);
};
MyWidget::MyWidget(void)
{
  QPushButton *hello = new QPushButton("Hello Qt-World!",this);
  hello->setGeometry(20,20,100,30);   // Position von Button
}
int main(int argc,char **argv)
{
  QApplication app(argc,argv);
  MyWidget *mywidget = new MyWidget();
```

```
  mywidget->resize(140,70);
  mywidget->show();
  return app.exec();
}
```

Standard–Vorgehen für Qt Programme:

Qt–Komponenten in separater Klasse zusammenbasteln. `main()` erzeugt nur noch zugehöriges Objekt, verknüpft es mit *QApplication* und startet diese.

9.2 Kompilieren, Linken und Ausführen von Qt Anwendungen

Qt in der Regel auf Linux Systemen verfügbar, aber nicht mehr Teil der **g++** Standard–Libs.

Beim Kompilieren und Linken muss deshalb Pfad für Header Dateien und Libraries explizit angegeben werden.

Am einfachsten zu handhaben wenn man *Qt-spezifische* Tools/Makefiles verwendet.

Rezept:

- Eigenes Verzeichnis für jedes eigenständige Qt Programm
- Quell-Dateien erstellen mit Standard-Editor
  I.d.R. gibt es zu jeder Qt–Klasse bzw. -Komponente eine entsprechende *Header* Datei:
  ```
  #include <QPushButton>
  ```
- Befehlssequenz erstellt zuerst Projekt, dann Makefile und anschliessend wird kompiliert
  ```
  qmake -project
  qmake
  make
  ```
- **qmake ...** Aufrufe nur einmal nötig, anschliessend genügt Aufruf von **make**

9.3 Qt–Komponenten und Layout

Es gibt Vielzahl von Qt Komponenten. Komponenten können gemeinsam in das Widget gepackt werden:

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
class MyWidget : public QWidget
{
public:
  MyWidget(void);
};
MyWidget::MyWidget(void)
{
  QLabel* qlab = new QLabel(this);  // Label
  qlab->setText( "first line\nsecond line" );
  qlab->setGeometry(20,20,100,30);
```

```
    QLineEdit* ledit = new QLineEdit(this);      // Text-Feld
    ledit->setText( "One editable line ..." );
    ledit->setGeometry(120,20,200,30);
    QPushButton *qpb = new QPushButton("Hello world!",this); // Knopf
    qpb->setGeometry(320,20,400,30);
}
// main ....
```

- *QLabel* ist ein *passives* Textfeld, in das man einen Text-String reinschreiben kann, das aber keine Aktionen auslöst.
- *QPushButton* ist dagegen ein *aktives* Textfeld. Klicken mit Maustaste kann Aktion auslösen ($\Rightarrow$ später)
- *QLineEdit* ist 1-zeiliges Textfeld, das editiert werden kann.

Hier absolute Positionierung mit `setGeometry(xlow, ylow, xhigh, yhigh)` Angabe der Punkte links oben und rechts unten.

Einfacher und flexibler mit Layout–Manager:

```
// 3 Knoepfe mit einem Layoutmanager
// 3 Knoepfe mit einem Layoutmanager
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QLayout>
class MyWidget : public QWidget
{
public:
  MyWidget();
};
MyWidget::MyWidget()
{
  // Knoepfe erzeugen
  QPushButton* b1 = new QPushButton("Button 1", this);
  b1->setMinimumSize(b1->sizeHint()); // minimal Groesse festlegen
  QPushButton* b2 = new QPushButton("Button 2", this);
  b2->setMinimumSize(b2->sizeHint());
```

```
  QPushButton* b3 = new QPushButton("Button 3", this);
  b3->setMinimumSize(b3->sizeHint());
  // Statt QHBoxLayout knnte natrlich auch ein
  // QVBoxLayout fr vertikales Layout erzeugt werden.
  QHBoxLayout* layout = new QHBoxLayout(this);// Layoutmanager fr horizontales Layout
  layout->addWidget(b1); // Knoepfe hinzufuegen.
  layout->addWidget(b2);
  layout->addWidget(b3);
  layout->activate();        // Layoutmanager aktivieren
}
//....
```

Text–Editor Fenster

QTextEdit ergibt Text–Editor inclusive automatischen Scrollbars.

```
#include <QApplication>
#include <QWidget>
#include <QTextEdit>
#include <QResizeEvent>
class MyWidget : public QWidget
{
private:
  QTextEdit* qtxt;
public:
  MyWidget(void);
};
MyWidget::MyWidget(void)
{
  qtxt = new QTextEdit(this);     // TextArea
  qtxt->setText( " first line\n second line\n more lines..." );
  qtxt->setGeometry(1,1,280,280);
}
```

```
// wird aufgerufen, wenn die Groesse des Fensters veraendert wird
void MyWidget::resizeEvent( QResizeEvent* event )
{
  qtxt->resize( event->size() ); // neue Groesse
}
//....
```

Painter – Fenster zum Zeichnen

```
#include <QApplication>
#include <QPainter>
#include <QWidget>
class QPaintEvent; //forward declaration
class MyWidget : public QWidget
{
public:
  MyWidget();
protected:
  virtual void paintEvent(QPaintEvent*);
};
MyWidget::MyWidget()
{
  setBackgroundRole(QPalette::Base);
  setAutoFillBackground(true);
}
void MyWidget::paintEvent(QPaintEvent *)
{
  QBrush b1( Qt::blue );
  QBrush b2( Qt::green, Qt::Dense6Pattern );        // green 12% fill
  QBrush b3( Qt::NoBrush );                         // void brush
  QBrush b4( Qt::CrossPattern );                    // black cross pattern
  QPainter *paint = new QPainter();
  paint->begin(this);
  paint->setPen( Qt::red );
```

```
  paint->setBrush( b1 );
  paint->drawRect( 10, 10, 200, 100 );
  paint->setBrush( b2 );
  paint->drawRoundRect( 10, 150, 200, 100, 20, 20 );
  paint->setBrush( b3 );
  paint->drawEllipse( 250, 10, 200, 100 );
  paint->setBrush( b4 );
  paint->drawPie( 250, 150, 200, 100, 45*16, 90*16 );//      paint->translate(20.0,40.0);
  paint->end();
}
int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    MyWidget *wid = new MyWidget();
    wid->show();
    return app.exec();
}
```

QPainter als Widget für grafische Elemente:

- Methoden zum Zeichnen von *Rechtecke, Ellipsen, Linien, Polygone, ...*
- *setPen(..)* für Linien–Farbe, *setBrush* für Fill–FarbeFarbe

Es gibt noch viele weiter nützliche Komponenten: *QSlider, QFileDialog, QDateTime, QTable, QMenu, QMenuBar, ...*

9.4 Reagieren auf Maus–Bewegung

Die zentrale Komponente für ein User–Interface ist die Möglichkeit auf Ereignisse reagieren zu können, d.h. *Knopfdruck, Mausbewegung, Texteingabe, etc* mit einer **Aktion** zu verknüpfen.

Erstmal Reaktion auf Mausbewegung im Main-Widget:

- *QWidget* hat Methoden *mousePressEvent()* und *mouseMoveEvent()*
- *myWidget*, das von *QWidget* erbt, kann diese überschreiben.
- Methoden bekommen Pointer auf *QMouseEvent* Objekt als Argument übergeben.
- *QMouseEvent* hat Methode *pos()*. Liefert *QPoint* Objekt, das ist einfach 2D Koordinate mit Methoden *x()* und *y()*.

```
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QMouseEvent>
#include <iostream>
using namespace std;
class myWidget : public QWidget
{
public:
  myWidget(){};
```

```
protected:
  virtual void mousePressEvent( QMouseEvent* );
  virtual void mouseMoveEvent( QMouseEvent* );
};
// reagiert auf Maus-Klick
void myWidget::mousePressEvent( QMouseEvent* event )
{
  int xpos = (event->pos()).x();
  int ypos = (event->pos()).y();
  cout << xpos << " " << ypos << endl;
}
// reagiert auf Mausbewegung, waehrend eine Taste gedrueckt ist.
void myWidget::mouseMoveEvent( QMouseEvent* event )
{
  int xpos = (event->pos()).x();
  int ypos = (event->pos()).y();
  cout << xpos << " " << ypos << endl;
}
int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    myWidget *wid = new myWidget();
    wid->resize(300,300);
    wid->show();
    return app.exec();
}
```

Kann leicht auf Bewegung von Grafik–Objekten angewendet werden:

```
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QPaintEvent>
#include <QMouseEvent>
class myWidget : public QWidget
{
public:
  myWidget();
protected:
  virtual void paintEvent(QPaintEvent*);
  virtual void mousePressEvent( QMouseEvent* );
  virtual void mouseMoveEvent( QMouseEvent* );
private:
  int xpos, ypos, radius; // circle parameters
};
myWidget::myWidget()
{
  xpos = 50; ypos = 50; radius = 40;
  setBackgroundRole(QPalette::Base);
  setAutoFillBackground(true);
}
void myWidget::paintEvent(QPaintEvent *) // called by update()
{
  QBrush b1( Qt::blue );    // blue filler
```

```
  QPainter *paint = new QPainter();
  paint->begin(this);
  paint->setPen( Qt::red );
  paint->setBrush( b1 );
  paint->drawEllipse( xpos, ypos, radius, radius );
  paint->end();
}
// reagiert auf Maus-Klick
void myWidget::mousePressEvent( QMouseEvent* event )
{
  xpos = (event->pos()).x();
  ypos = (event->pos()).y();
  update();
}
// reagiert auf Mausbewegung, waehrend eine Taste gedrueckt ist.
void myWidget::mouseMoveEvent( QMouseEvent* event )
{
  xpos = (event->pos()).x();
  ypos = (event->pos()).y();
  update();
}
int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    myWidget *wid = new myWidget();
    wid->resize(300,300);
    wid->show();
```

```
    return app.exec();
}
```

Jetzt wird in member-Variablen *xpos, ypos* die Position eines Kreises abgespeichert. Mausklick oder Mausbewegung ändert diese Variablen entsprechend.

Zeichen–Funktion in überschriebener QWidget–Methode *paintEvent()*, die bei Änderung des Fensters oder bei Mausbewegung via *update()* gerufen wird.

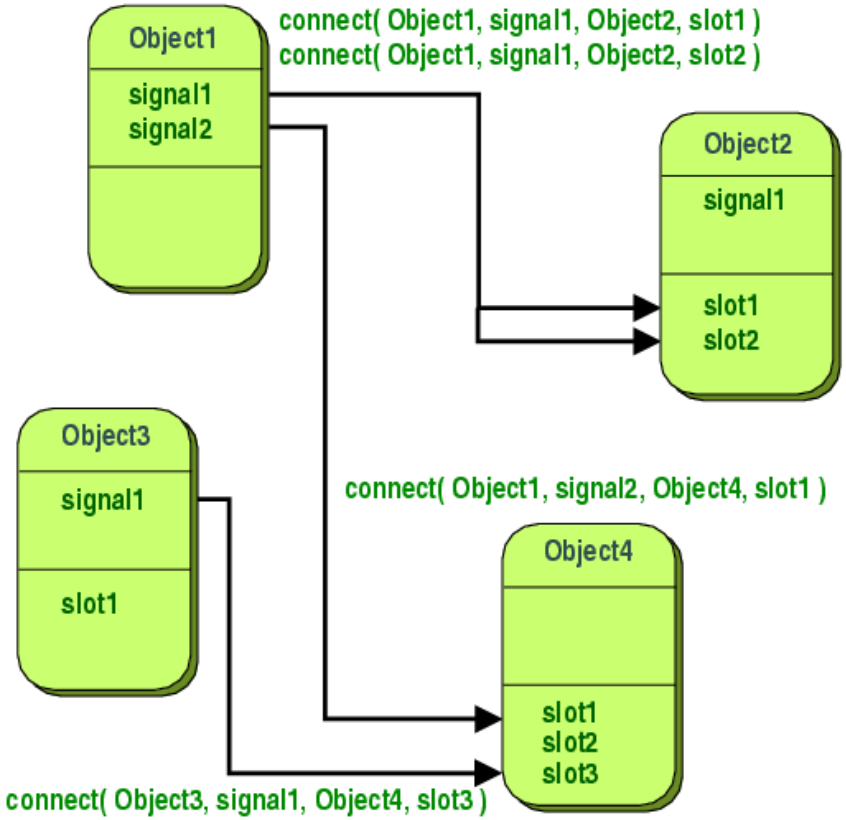
Qt–Anwendungsprogramme beruhen i.a. auf Polymorphismus

Beispiel der Maus–Aktion:

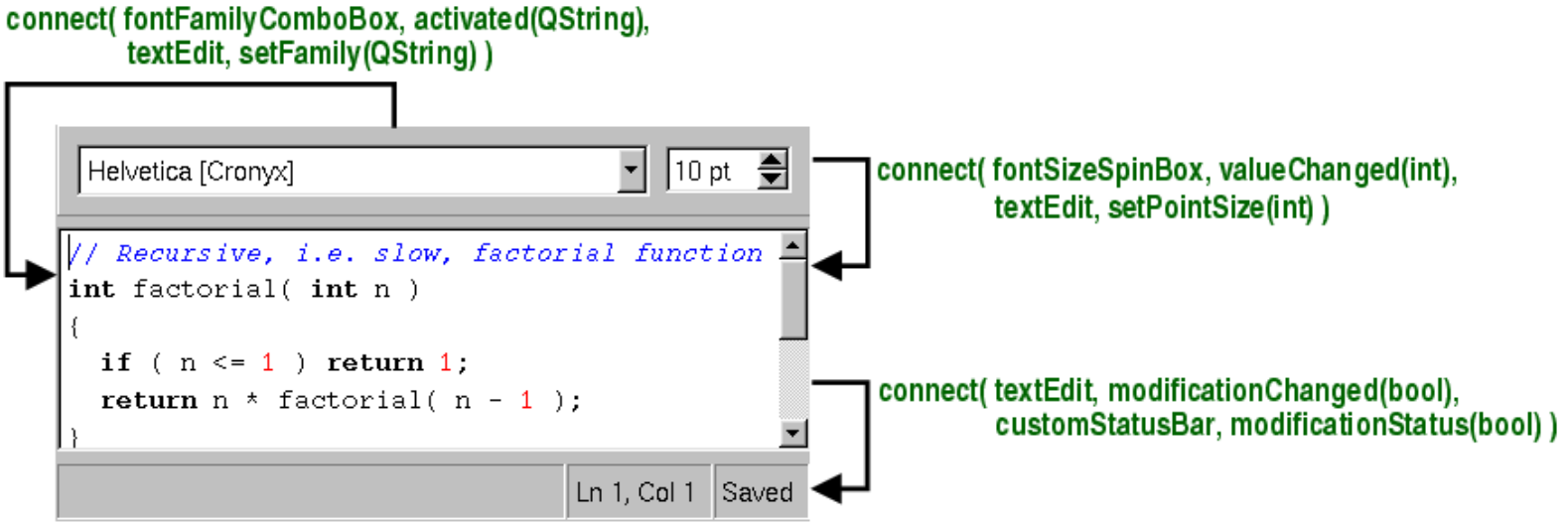
- **QApplication** organisiert die Qt Aktionen, kennt aber nur Qt–interne Klassen, wie z.B. **QWidget**.
- Eigene Klasse **MyWidget** leitet von **QWidget** ab.
- Vor Start wird **MyWidget** als ”Haupt-Widget” an **QApplication** übergeben:
  QApplication weiss nix von **MyWidget** , es kennt nur **QWidget** und die dazugehörigen Methoden.
- Nach Start und Maus–AKtion ruft **QApplication** für das übergebene **QWidget–Objekt** die Methode *mousePressEvent()* auf.
- **Falls** diese Methode *mousePressEvent()* in der Klasse **MyWidget** vorhanden ist, wird diese benutzt $\Rightarrow$ **Polymorphismus**
- Andernfalls wird die entsprechende *mousePressEvent()* Methode der Basisklasse **QWidget** gerufen (die nichts weiter macht = dummy)

9.5 Kommunikation zwischen Objekten – Signal & Slots

Zentraler Teil eines GUI ist die Reaktion auf Ereignisse: Maus-Klick oder -Bewegung, Text-Eingabe, Menueauswahl, etc. Dies ist in der Regel der komplexeste Teil bei der Erstellung eines GUIs. Qt benutzt dafür den sog. *Signal & Slots* Mechanismus.



- **Signale** werden ausgesandt wenn ein bestimmtes Ereignis auftritt. Die verschiedenen Qt Widgets haben vor–definierte Signale, man kann aber auch via Vererbung eigene definieren.
- **slot** ist eine Funktion, die gerufen wird als Reaktion auf ein bestimmtes Signal. Auch hier gibt es vordefinierte Slots, und man kann eigene hinzufügen.
- Mit **connect(...)** wird eine Verbindung hergestellt zwischen *Signal* und *Slot*.



Signal & Slots ist eine Qt spezifische C++ Erweiterung.

- Zur Verwendung muss man von der Qt Basis–Klasse *QObject* ableiten (Bei Qt Klassen ist das implizit immer erfüllt).
- Ausserdem sind spezielle Präprozessor–Anweisungen und Schritte nötig.

```
// Beispiel zum SIGNAL-SLOT-Mechanismus
//
#include<QObject>
#include<iostream>
using namespace std;
// Eine Klasse, die Signals und Slots besitzt.
class Foo : public QObject
{
  Q_OBJECT;
public:
  Foo(); // Konstruktor
  int   value() const { return val; }
public slots:
 // Der Wert von "val" wird gendert.
 void setValue( int );
signals:
 // Das Signal soll ausgesandt werden, wenn "val" gendert wird.
 void valueChanged( int );
private:
```

```
 int    val;
};
Foo::Foo()
{
  val = 1;
}
void Foo::setValue( int v )
{
  // val wird nur neu gesetzt, wenn tatschlich ein anderer Wert bergeben wird.
  if ( v != val ) {
    val = v;
    emit valueChanged(v);
  }
}
#include"sigslot.moc"
int main()
{
  // Zwei Foo-Objekte
  Foo *a = new Foo();
  Foo *b = new Foo();
  // Das Signal des einen wird mit dem Slot des anderen Objekts verbunden.
  QObject::connect(a, SIGNAL(valueChanged(int)), b, SLOT(setValue(int)));
  b->setValue( 11 ); // b.val hat den Wert 11
  cout << "\nb:" << b->value() << "\n" ;
  a->setValue( 79 ); // a.val hat den Wert 79 und durch die obige Verknpfung
                     // hat auch b.val den Wert 79
  cout << "\na:" << a->value() << "\n" ;
```

```
  cout << "b:" << b->value() << "\n\n" ;
  return 0;
}
```

- *Slot* ist Member Funktion einer Klasse.
- Dagegen ist *Signal* eine *emit* Anweisung innerhalb einer Member Funktion.
- Daten–Typ der bei *Signal* übergeben wird muss mit Argumenttyp der *Slot* Member Funktion übereinstimmen.
- Mit *connect(..)* wird eine Verbindung zwischen Signal und Slot erzeugt:

  ```
  connect(Sender-obj, SIGNAL(signal-name(type)), receiv-obj,
  SLOT(member-func(type)));
  ```

- Ein *Signal* kann mit keinem, einem oder vielen *Slots* verbunden sein.
- Ein *Slot* kann mit keinem, einem oder vielen *Signalen* verbunden sein.

Einfaches Beispiel mit Knopf und vordefinierten Signal & Slots

```
// Knopf mit Funktion: wenn man draufdrueckt beendet sich das Programm.
// Signal und Slot beide vor-definiert
#include <QApplication>
#include <QPushButton>
#include <QWidget>
class MyWidget : public QWidget
{
public:
  MyWidget(void);
private:
  QPushButton *ende;
};
MyWidget::MyWidget(void)
{
  ende = new QPushButton("Raus hier!",this);
  ende->setGeometry(10,10,100,30);
  // qApp ist ein globaler Zeiger auf QApplication
  QObject::connect(ende,SIGNAL(clicked()),qApp,SLOT(quit()));
}
int main(int argc,char** argv)
{
  QApplication app(argc,argv);
  QWidget* mywidget = new MyWidget();
  mywidget->resize(120,50);
  mywidget->show();
```

```
  return app.exec();
}
```

Beispiel mit Knopf und eigenem Slot

```
// Knopf mit    Funktion: jetzt selbst-definierte Aktion.
//
// erst mit moc drueber: $QTDIR/bin/moc -o drueckmich.moc drueckmich.C
#include <QApplication>
#include <QPushButton>
#include <QObject>
class MyWidget : public QWidget
{
  Q_OBJECT;
public:
  MyWidget(void);
public slots:
  void autsch(void);
private:
  QPushButton *ende;
};
MyWidget::MyWidget(void)
{
  ende = new QPushButton("Drueck mich!",this);
  ende->setGeometry(10,10,200,40);
```

```
  // qApp ist ein globaler Zeiger auf QApplication
  QObject::connect(ende,SIGNAL(clicked()),this,SLOT(autsch()));
}
void MyWidget::autsch() {
  ende->setText("Aua, nicht so grob !");
}
#include "drueckmich.moc"
int main(int argc,char** argv)
{
  QApplication app(argc,argv);
  QWidget* mywidget = new MyWidget();
  mywidget->resize(120,50);
  //    app.setMainWidget(mywidget);
  mywidget->show();
  return app.exec();
}
```

Einfacher Taschenrechner

```
// addInt.C
#include <QApplication>
#include <QLineEdit>
#include <QLabel>
#include <QPushButton>
#include <QLayout>
#include <QString>
class MyWidget : public QWidget
{
  Q_OBJECT;
public:
  MyWidget(void);
public slots:
  void addint(void);
private:
  QPushButton *add;
  QLineEdit* num1 ;
  QLineEdit* num2 ;
  QLineEdit* num3 ;
};
MyWidget::MyWidget(void)
{
  num1 = new QLineEdit(this); // Text-Feld
  num1->setText( "0" );
  num2 = new QLineEdit(this); // Text-Feld
```

```
  num2->setText( "0" );
  num3 = new QLineEdit(this);  // Text-Feld
  num3->setText( "0" );
  QPushButton *qpb = new QPushButton("Add",this); // Knopf
  QLabel *qlab = new QLabel("Result",this); // label
  QVBoxLayout* layout = new QVBoxLayout(this);
  layout->addWidget(num1);
  layout->addWidget(qpb);
  layout->addWidget(num2);
  layout->addWidget(qlab);
  layout->addWidget(num3);
  //
  QObject::connect(qpb,SIGNAL(clicked()),this,SLOT(addint())); // button => addint
  QObject::connect(num1,SIGNAL(returnPressed()),this,SLOT(addint())); // textfield => addint
  QObject::connect(num2,SIGNAL(returnPressed()),this,SLOT(addint()));
}
void MyWidget::addint() { // add 2 integers
  int v1 = (num1->text()).toInt(); // QString ==> int
  int v2 = (num2->text()).toInt();
  int v3 = v1 + v2;
  QString qs;
  qs.setNum(v3);   // int ==> QString
  num3->setText(qs);
}
#include "addInt.moc"
int main(int argc,char** argv)
{
```

```
  QApplication app(argc,argv);
  QWidget* mywidget = new MyWidget();
  mywidget->resize(120,130);
  mywidget->show();
  return app.exec();
}
```

Pulldown-Menues Und hier noch ein etwas laengeres Beispiel zu Pulldown-Menues.

```
// moveCircle3.C
//
#include <QApplication>
#include <QMenuBar>
#include <QMessageBox>
#include <QPainter>
#include <QPixmap>
#include <QWidget>
#include <QResizeEvent>
#include <QMouseEvent>
#include <QPaintEvent>
#include <QScrollArea>
#include <QColorDialog>
#include <iostream>
#include <cmath>
class myDraw : public QWidget // Zeichen–Widget
{
  Q_OBJECT;

public:
  myDraw();
public slots:
  void setColor( QColor );
  QColor getColor() ;
protected:
```

```
  virtual void paintEvent(QPaintEvent*);
  virtual void mousePressEvent( QMouseEvent* );
  virtual void mouseMoveEvent( QMouseEvent* );
private:
  int xpos, ypos, radius; // circle parameters
  QColor mycol;
};
/// Separate Klasse fuer Haupt-Widget
class myWindow : public QWidget
{
  Q_OBJECT
public:
  myWindow();
private slots:
  void slotAbout();
  void slotAboutQt();
  void slotColorMenu( );
  void slotColorRed( );
  void slotColorYellow( );
  void slotColorBlue( );
signals:
  void colorChanged( QColor );
protected:
  virtual void resizeEvent( QResizeEvent* );
private:
  QMenuBar *menubar;
  QMenu *filemenu;
```

```
  QMenu *colormenu;
  QMenu *helpmenu;
  QScrollArea *scrollarea;
  myDraw *mydr;
};
#include "moveCircle3.moc"
// Implementations
myDraw::myDraw()
{
  xpos = 50; ypos = 50; radius = 40;
  mycol = Qt::yellow;
  setBackgroundRole(QPalette::Base);
  setAutoFillBackground(true);
}
void myDraw::paintEvent(QPaintEvent* )
{
  QBrush b1( mycol );
  QPainter *paint = new QPainter();
  paint->begin( this);
  paint->setPen( Qt::red );
  paint->setBrush( b1 );
  paint->drawEllipse( xpos, ypos, radius, radius );
  paint->end();
}
// reagiert auf Maus-Klick
void myDraw::mousePressEvent( QMouseEvent* event )
{
```

```
  xpos = (event->pos()).x();
  ypos = (event->pos()).y();
  update();
}
// reagiert auf Mausbewegung, waehrend eine Taste gedrueckt ist.
void myDraw::mouseMoveEvent( QMouseEvent* event )
{
  xpos = (event->pos()).x();
  ypos = (event->pos()).y();
  update();
}
void myDraw::setColor( QColor new_color )
{
  mycol = new_color;
  update();
}
QColor myDraw::getColor( )
{
  return( mycol );
}
myWindow::myWindow()
{
  // setup menues
  // first three popup-menues
  filemenu = new QMenu("File");
  filemenu->addAction( "&Quit", qApp, SLOT( quit() ) );
```

```
colormenu = new QMenu("Color");
colormenu->addAction( "Red", this, SLOT( slotColorRed( )));
colormenu->addAction( "Yellow", this, SLOT( slotColorYellow( )));
colormenu->addAction( "Blue", this, SLOT( slotColorBlue( )));
colormenu->addAction( "Set color ...", this, SLOT( slotColorMenu( )));
helpmenu = new QMenu("Help");
helpmenu->addAction( "&About MoveCircle", this, SLOT( slotAbout() ) );
helpmenu->addAction( "About &Qt", this, SLOT( slotAboutQt() ) );
// create menubar in widget and add the 3 popup-menues
menubar = new QMenuBar( this );
menubar->addMenu(filemenu );
menubar->addMenu( colormenu );
//    menubar->insertSeparator();
menubar->addMenu(helpmenu );
// Hier wird ein ScollArea definiert
scrollarea = new QScrollArea( this );
scrollarea->setGeometry( 0, menubar->height(), width(), height() - menubar->height() );

// Die Malflaeche wird in ausreichender Groesse erzeugt.
mydr = new myDraw();
mydr->setGeometry( 0, 0, 1000, 1000 );
// und in die ScrollArea eingefgt.
scrollarea->setWidget( mydr );
// Verknuepfung des Signals colorChanged() mit dem Slot setColor()
// der Malflche zum Setzen der Farbe
QObject::connect( this, SIGNAL( colorChanged( QColor ) ),
                  mydr, SLOT( setColor( QColor ) ) );
```

```
}
// Bei einer Vernderung der Gre muss ScrollArea angepasst werden.
void myWindow::resizeEvent( QResizeEvent* )
{
  scrollarea->setGeometry( 0, menubar->height(), width(), height() - menubar->height() );
}
void myWindow::slotAbout()
{
  QMessageBox::information( this, "About QtTriangle 2",
              "This is the MoveCircle application\n"
              "No Copyright by Nobody\n");
}
void myWindow::slotAboutQt()
{
  QMessageBox::aboutQt( this, "About Qt" );
}
// dieser Slot setzt die Farbe nicht direkt, da die zugehrige
// Datenstruktur nicht zur selben Klasse gehrt.
// Statt dessen wird ein Signal ausgesandt.
void myWindow::slotColorMenu( )
{
  QColor col = QColorDialog::getColor(mydr->getColor(), this);
  if (!col.isValid())
    return;
  colorChanged(col);
}
void myWindow::slotColorRed( ) { colorChanged(Qt::red); }
```

```
void myWindow::slotColorYellow(    ) {   colorChanged(Qt::yellow); }
void myWindow::slotColorBlue(    ) {   colorChanged(Qt::blue); }
int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    myWindow *wid = new myWindow();
    wid->resize(300,300);
    wid->show();
    return app.exec();
}
```

- Getrennte Klassen für Haupt-Widget *(myWindow)* und Zeichen-Widget *(myDraw)*
- Farbe ändern über doppelte Signal-Slot Verbindung:
  1. Von *QMenu colormenu* Signal zu SLOT *myWindow::slotColorMenu()* Member-Funktion
  2. *myWindow::slotColorMenu()* schickt *colorChanged(color)* Signal zu SLOT *myDraw::setColor(QColor)* Member-Funktion

9.6 Grafik Animationen mit Qt

An sich sind Animationen Parade–Anwendung für Multithreading:

- Jedes Objekt das bewegt werden soll läuft als unabhängiger Thread
- Endlosschleife in *run()* Methode, unterbrochen durch *sleep()* Aufrufe

Allerdings in Qt keine direkten GUI Aufrufe aus extra threads, kein Signal–Slot zwischen threads. Nur indirektes handling, ziemlich umständlich.

Ähnliche Funktionalität bequemer mit **QTimer** möglich, funktioniert wie Küchenwecker:

- *QTimer Objekt* anlegen:
  ```
  QTimer *internalTimer = new QTimer( );
  ```
- *QTimer Objekt–Signal* verknüpfen mit eigenem Objekt und entsprechender Funktion:
  ```
  connect(internalTimer, SIGNAL(timeout()), this, SLOT(timeout()));
  ```
- Timer starten mit Angabe von Intervallen in *ms*
  ```
  internalTimer->start( 20 );
  ```
- Funktion *timeout()* des eigenen Objektes wird in diesen Intervallen gerufen, übernimmt Bewegung und Neu–Zeichnen

Längeres, ausgearbeitetes Beispiel: Ball fliegt im Zeichen–Widget herum und prallt an Wänden ab.

(Hauptaufwand ist Algorithmus für Ballbewegung):

```
// to compile do:
#include <qapplication.h>
#include <qpainter.h>
#include <qpixmap.h>
#include <qwidget.h>
#include <iostream>
#include <cmath>
#include <qtimer.h>
#include <cstdlib>
#include <vector> // vector headers
#include <iostream>
using namespace std;
class Ball {
private:
  QPoint ballpos;
  QPoint ballveloc;
  int ballradius;
public:
  Ball();
  Ball(const QPoint &);
  virtual void travel(double time, int w, int h);
  void paint(QPainter &painter) const;
};
Ball::Ball()
{
```

```
  ballpos.setX(200);
  ballpos.setY(200);
  ballveloc.setX( rand()%20−10);
  ballveloc.setY( rand()%20−10);
  ballradius = 10;
}
Ball::Ball(const QPoint & p) : ballpos(p)
{
  ballveloc.setX( rand()%20−10);
  ballveloc.setY( rand()%20−10);
  ballradius = 10;
}
void Ball::paint(QPainter &painter) const
{
  painter.drawEllipse( ballpos.x()−ballradius,
                       ballpos.y()−ballradius, ballradius, ballradius );
}
void Ball::travel(double time, int w, int h) {
  // Move the ball for the specified number of time units.
  // The ball is restricted to the specified rectangle.
  // Note:    The ball won't move at all if the width or height
  // of the rectangle is smaller than the ball's diameter.

  /* Don't do anything if the rectangle is too small. */
  int xmax = w;
  int ymax = h;
  int xmin = 0;
```

```
int ymin = 0;
int x = ballpos.x();
int y = ballpos.y();
int radius = ballradius;
if (xmax - xmin < 2*radius || ymax - ymin < 2*radius)
  return;

/* First, if the ball has gotten outside its rectangle, move it
   back.  (This will only happen if the rectagnle was changed
   by calling the setLimits() method or if the position of
   the ball was changed by calling the setLocation() method.)
*/

if (x-radius < xmin)
  x = xmin + radius;
else if (x+radius > xmax)
  x = xmax - radius;
if (y - radius < ymin)
  y = ymin + radius;
else if (y + radius > ymax)
  y = ymax - radius;

/* Compute the new position, possibly outside the rectangle. */

double dx = ballveloc.x();
double dy = ballveloc.y();
double newx = x + dx*time;
```

```
double newy = y + dy*time;

    /* If the new position lies beyond one of the sides of the rectangle,
       "reflect" the new point through the side of the rectangle, so it
       lies within the rectangle. */

if (newy < ymin + radius) {
  newy = 2*(ymin+radius) - newy;
  dy = fabs(dy);
}
else if (newy > ymax - radius) {
  newy = 2*(ymax-radius) - newy;
  dy = -fabs(dy);
}
if (newx < xmin + radius) {
  newx = 2*(xmin+radius) - newx;
  dx = fabs(dx);
}
else if (newx > xmax - radius) {
  newx = 2*(xmax-radius) - newx;
  dx = -fabs(dx);
}
ballveloc.setX((int)dx);
ballveloc.setY((int)dy);
/* We have the new values for x and y. */
ballpos.setX((int)newx);
ballpos.setY((int)newy);
```

```
} // end travel()
// Das Hauptwidget
class BallAnim : public QWidget
{
    Q_OBJECT
public:
  BallAnim();
protected:
  virtual void paintEvent( QPaintEvent* );
public slots:
virtual void timeout(     ); // slot for timer
private:
  Ball* vb;
  QColor bgcol;
  QPixmap buffer;
};
// Konstruktor
BallAnim::BallAnim()
{
  setBackgroundRole(QPalette::Base);
  setAutoFillBackground(true);
  vb = new Ball();
  // create timer
  QTimer *internalTimer = new QTimer( this ); // create internal timer
  connect( internalTimer, SIGNAL(timeout()), this, SLOT(timeout()) ); // connect timer signal
  internalTimer->start( 20 );                 // emit signal every 20 ms
}
```

```
//
void BallAnim::timeout()      // action method invoked by timer
{
  int w = width();
  int h = height();
  vb->travel( 1., w, h ); // move the ball
  update();
}
// wird aufgerufen, wenn sich das Fenster neu zeichnen soll
void BallAnim::paintEvent( QPaintEvent* )
{
  QPainter bufferpainter;
  bufferpainter.begin( this );
  bufferpainter.setBrush( Qt::red );
  bufferpainter.setPen( Qt::red );
  vb->paint( bufferpainter );
  bufferpainter.end();
}
#include "BallAnim1.moc"
int main( int argc, char **argv )
{
  QApplication myapp( argc, argv );
  BallAnim* mywidget = new BallAnim();
  mywidget->setGeometry( 50, 50, 400, 400 );
    mywidget->show();
  return myapp.exec();
}
```

9.7 Aufgaben

1. **QT–Komponenten**
   Implementieren und studieren Sie die vorgestellten Beispiele zu
   - Labels
   - Buttons
   - LineEdit
   - TextEditor
2. **QT–Grafik**
   Analog für *QPainter*.
   - Untersuchen Sie die verschiedenen Zeichenfunktionen (*Ellipse, Rectangle, Polygon, ...*)
   - Variieren Sie Linien- bzw. Füll-farben/muster.
3. **Maus-Events**
   Implementieren und variieren Sie die Beispiele zu den Maus–Events.

   Erweitern Sie das moveCircle Beispiel, so dass Klick mit rechter Maus Änderung des Radius macht statt Kreis zu bewegen.
4. **Projekt: Demo zur Dreieck–Geometrie**
   Eine erstaunliche geometrische Eigenschaft von Dreiecken besteht darin, dass die Mit-

telsenkrechten sich immer in einem Punkt treffen; das gilt für beliebige Dreiecke. Mit Qt ist es *relativ* einfach eine entsprechende graphische Demo zu programmieren. Benötigt wird:

- Zeichnen eines Dreiecks durch Angabe von drei 2D Koordinaten
  `drawPolygon ( qpa );`
  wobei `QPolygon qpa(3);` Array mit 3 Elementen = Ecken des Dreiecks.
- Reaktion auf Maus:
  - *MouseClick* soll nächstliegende Ecke an Maus-Position verschieben
  - *MouseDrag* soll Dreieck-Ecke mitziehen
- Zeichnen der Mittelsenkrechten für das Dreieck
  $\Rightarrow$ etwas kniffliger Algorithmus, später ...

5. **Signal & Slots**
   Implementieren und studieren Sie die vorgestellten Beispiele zu Signal & Slots.

6. **Taschenrechner für Big-Integers**
   Verwenden Sie im vorgestellten Taschenrechnerbeispiel die *BigInt* Klasse anstatt der normalen *Integers*. (BigInt Header `BigInt.hxx` und code `BigInt.cxx`.)

7. **Farbmenues**
   Erweitern Sie die Menues im Beispiel *moveCircle3* so dass nicht nur Hintergrund, sondern separat Randfarbe, Füllfarbe und Hintergrund eingestellt werden können. Am besten eine weitere *QMenu* Ebene einführen.

8. **Animationen**
   (a) Implementieren Sie das BallAnim1 Beispiel. Variieren Sie Timer-Intervall und Ball-Geschwindigkeit. Was sind *gute* Werte für ruhige Bilder ?
   (b) Erweitern Sie die Animation so, dass bei jedem Mausklick ins Fenster ein weiterer Ball gestartet wird (Hinweis: Bälle in `vector< Ball* >` abspeichern)
   (c) In `PlanetAnim.C` finden Sie eine simple Animation zur Bewegung von Erde um Sonne bzw. Mond um Erde. Fügen Sie weitere Planeten hinzu.

10 Thread Progamming in C++11

Multi–Threading bzw. *Concurrency* wird in C++ direkt erst ab C++11 unterstützt. Davor war es aber in gängigen Zusatz–Libraries enthalten (Boost, Qt, ...) verfügbar. Im wesentlichen bedeutet Multi–Threading, dass aus einem C++ Programm heraus mehrere, "parallel laufende" Prozesse (=*Threads*) gestartet werden können.

Historie:

- In frühen Computern wurde ein Programm vom Anfang bis Ende ausgeführt, erst dann wurde das nächste Programm gestartet $\Rightarrow$ *Batch–Processing*
- Modernere Betriebssysteme unterstützen *Multi–Tasking*, d.h. über einen Time–sharing Mechanismus wird die CPU-time auf die verschiedenen Prozesse verteilt. Auf Single-CPU Rechner läuft natürlich jeweils nur ein Prozess aber nach typisch einigen Millisekunden ist der nächste an der Reihe $\Rightarrow$ Basis für *interaktive Multi–User Systeme*

  Die einzelnen Prozesse laufen aber völlig getrennt, jedes Programm hat seinen eigenen Speicherbereich, seine eigenen Variablen, usw.
- *Multi–Threading* geht darüber hinaus: Es laufen *gleichzeitig* mehrere Versionen *eines* Programms, die alle auf den *gleichen* Speicherbereich zugreifen.
  $\Rightarrow$ Heikel, mögliche Konflikte beim Zugriff auf Speicherbereiche (Lesen, Schreiben, Löschen).

Qt–Programme mit GUI sind automatisch multi–threaded:

- Thread für *main()* und was daraus gerufen wird, Thread für z.B. Zeichnen der Graphic–Pane

Neuer Trend im Programmieren: Concurrency = Multithreading

Bis vor kurzem war Multi-Threading auf spezielle Anwendungen beschränkt (GUIs, Steuerung, Server-Prozesse) und etwas für Experten.

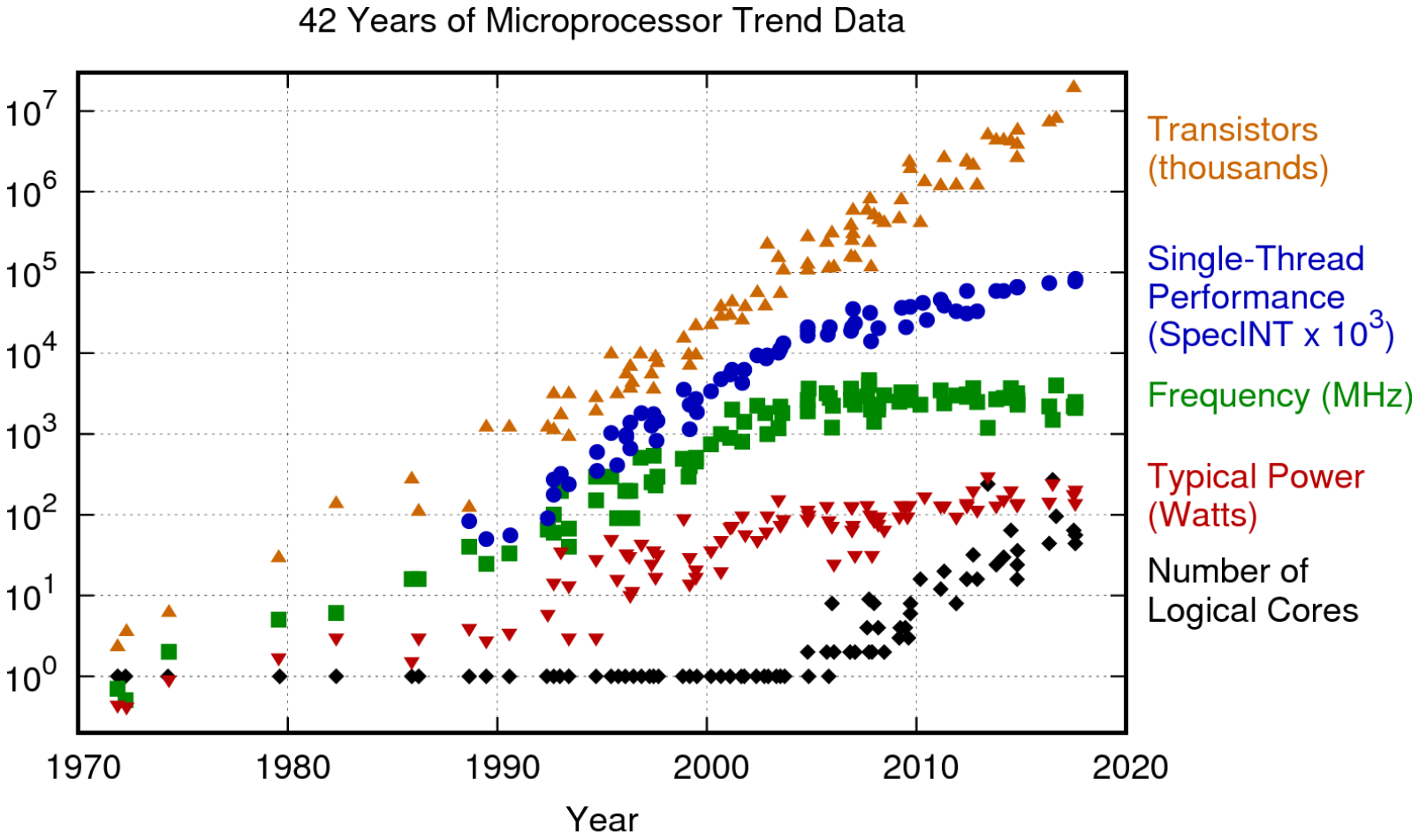
Allerdings vor einigen Jahren Trendwende bei Hardware Entwicklung:

- Leistungssteigerung der Prozessoren durch steigende Taktraten und optimierte Prozessorabläufe ziemlich ausgereizt, jedenfalls stark verlangsamt.
- Stattdessen Einführung von **Multi-Core** CPUs, d.h. mehrere Prozessorkerne auf einem Chip. Z.Z. 4, 8, 12, 20 oder mehr Cores pro CPU gängig bei Server Prozessoren, noch deutlich mehr bei Systemen mit Ko-Prozessoren (GPUs, MIC/Phi)
- Performance in Zukunft v.a. durch mehr Prozessorkerne.
- **Vectorisation** bzw **SIMD** Processing weiteres wichtiges Feature zur Steigerung der Performance

Verlangt zwingend **multi-threaded** Programme um Leistungssteigerungen auszunutzen.

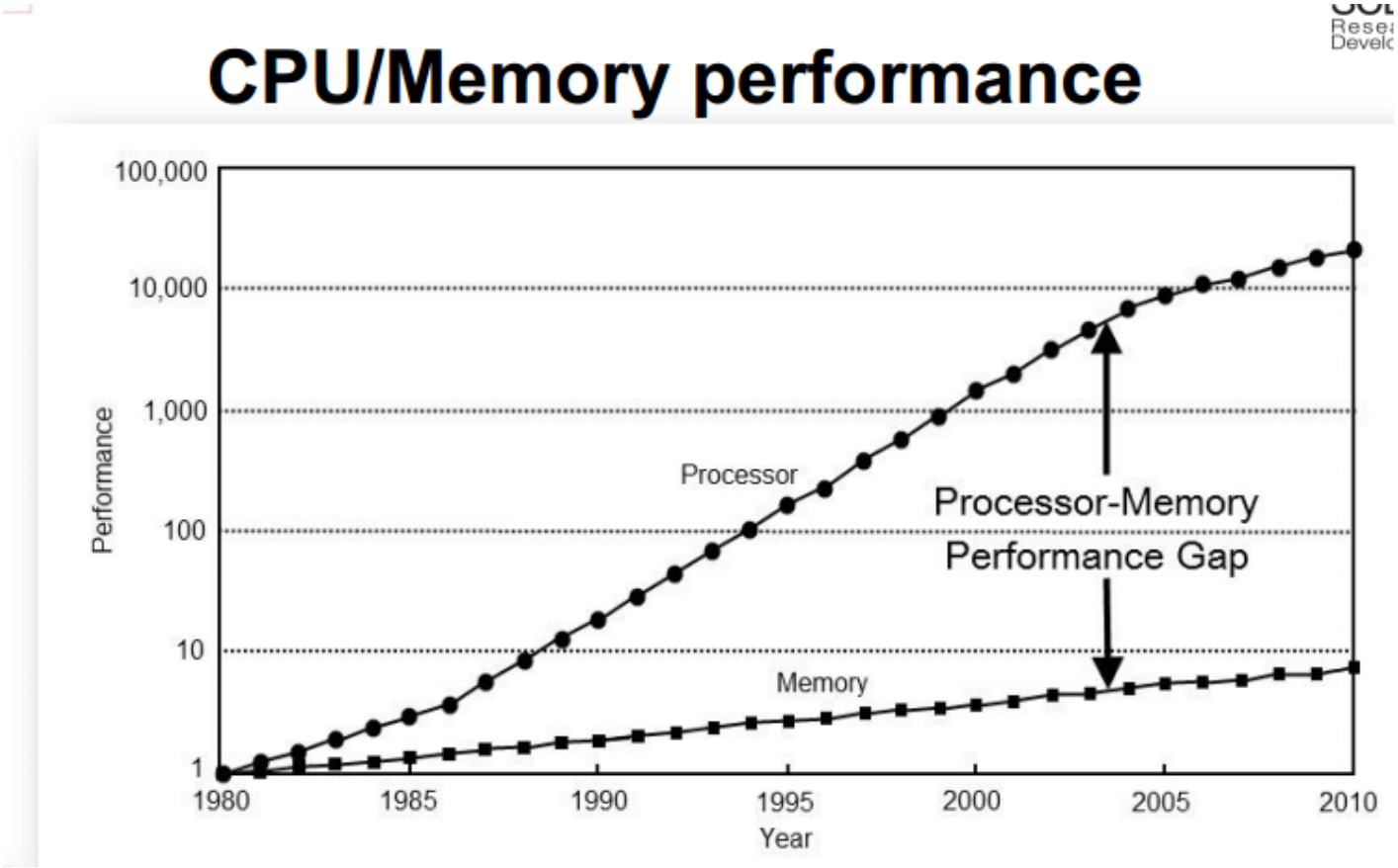
Siehe auch Artikel von Herb Sutter: **The Free Lunch Is Over**

Historie Transistor/CPU Entwicklung



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

CPU vs Memory Speed



10.1 Threads

Threads zu erzeugen ist einfach:

```
// Compile with:    g++ -std=c++11 -pthread -o threadex1 threadex1.cpp
#include <iostream>
#include <thread>
using namespace std;
void func(int x) {
    cout << "Innerhalb des threads" << x << endl;
}
int main() {
    thread th(func, 100);
    th.join();
    cout << "Ausserhalb des threads" << endl;
    return 0;
}
```

Dieser code sollte mit folgendem Kommando kompiliert werden:

```
g++ -std=c++11 -pthread -o threadex1 threadex1.cpp
```

Mit der Option `-std=c++11` wird angegeben, daß der Compiler C++11 verwenden werden soll. Es muss außerdem `-pthread` angegeben werden, da die verwendete GCC-Version sog. pthreads

(POSIX threads) als backend zum Ausführen verwendet.

Falls Sie `eclipse` zur Entwicklung verwenden, muss der aktuellen Version im CIP-Pool-Cluster C++11 auch bekannt gemacht werden. Die Konfiguration hierfür ist auf folgenden Seiten beschrieben: (link1) und (link2).

10.2 Race conditions

Folgendes Beispiel demonstriert eine sog. *"race condition"*, die bei parallel aufgeführten Prozessen, die auf gleiche Ressourcen zugreifen, auftreten können. Nehmen wir an, daß die Operation `x*x` sehr zeitaufwendig sei und wir möchten die Summe der Quadrate bis zu einer bestimmten Zahl berechnen. Es ist somit sinnvoll, die hypothetisch rechenintensive Berechnung von `x*x` für jede Zahl zu parallelisieren und auf mehrere threads zu verteilen. Dies kann z.B. mit folgendem Programm erfolgen:

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;
int accum = 0;
void square(int x) {
    accum += x * x;
}
int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
```

```
    return 0;
}
```

Es wird die Summe aller Quadrate bis einschließlich 20 gebildet. Dabei wird beim Iterieren bis 20 in jedem Iterationsschritt ein neuer thread gestartet, dem der aktuelle Schleifenzähler als Parameter übergeben wird. Anschließend wird `join` für alle threads aufgerufen, der die Ausführung des Programms blockiert und wartet, bis alle threads beendet sind. Anderenfalls wären die threads eventuell noch nicht beendet bevor das Endergebnis `accum` auf dem Bildschirm ausgegeben wird. Man sollte immer `join()` für die threads verwenden bevor ein Programm in `main` beendet wird.

Das Beispielprogramm verwendet eine kompakte C++11-Syntax für die Iteration über eine `vector` Klasse, die sehr ähnlich der Syntax in Java ist. Zusätzlich wird das Schlüsselwort `auto` anstelle der Datentypendeklaration `thread` verwendet. Diese kann immer verwendet werden, falls der Compiler eindeutig den korrekten Typen aus dem Kontext erkennen kann. Es wurde außerdem `&` verwendet, um die Referenz und nicht die Kopie des Objektes zu erhalten, da `join` das Objekt verändert.

Beim Ausführen des Programms sollte zufällig das korrekte Resultat von *2870* angezeigt werden. Beim mehrfachen Ausführen des Programms treten aber evt. unterschiedlichen Resultate auf - beim 50fachen Ausführen von `threadex2` auf der bash Kommandozeile treten Inkonsistenzen auf:
`for i in {1..50}; do ./threadex2; done`
Deutlicher wird dies beim 1000fachen Wiederholen, wobei die Häufigkeit der Resultate auf der bash

Kommandozeile gezählt werden:

```
for i in {1..1000}; do ./threadex2; done | sort | uniq -c
```

Es tritt eine sog. *race condition* auf. Bei jedem Prozessieren von `accum += x * x;` wird der aktuelle Wert von `accum` gelesen und der neue Wert gesetzt. Dies ist aber keine sog. `atomic` (d.h. unteilbare) Operation. Dies wird deutlicher, wenn man die Methode `square` entsprechend umschreibt:

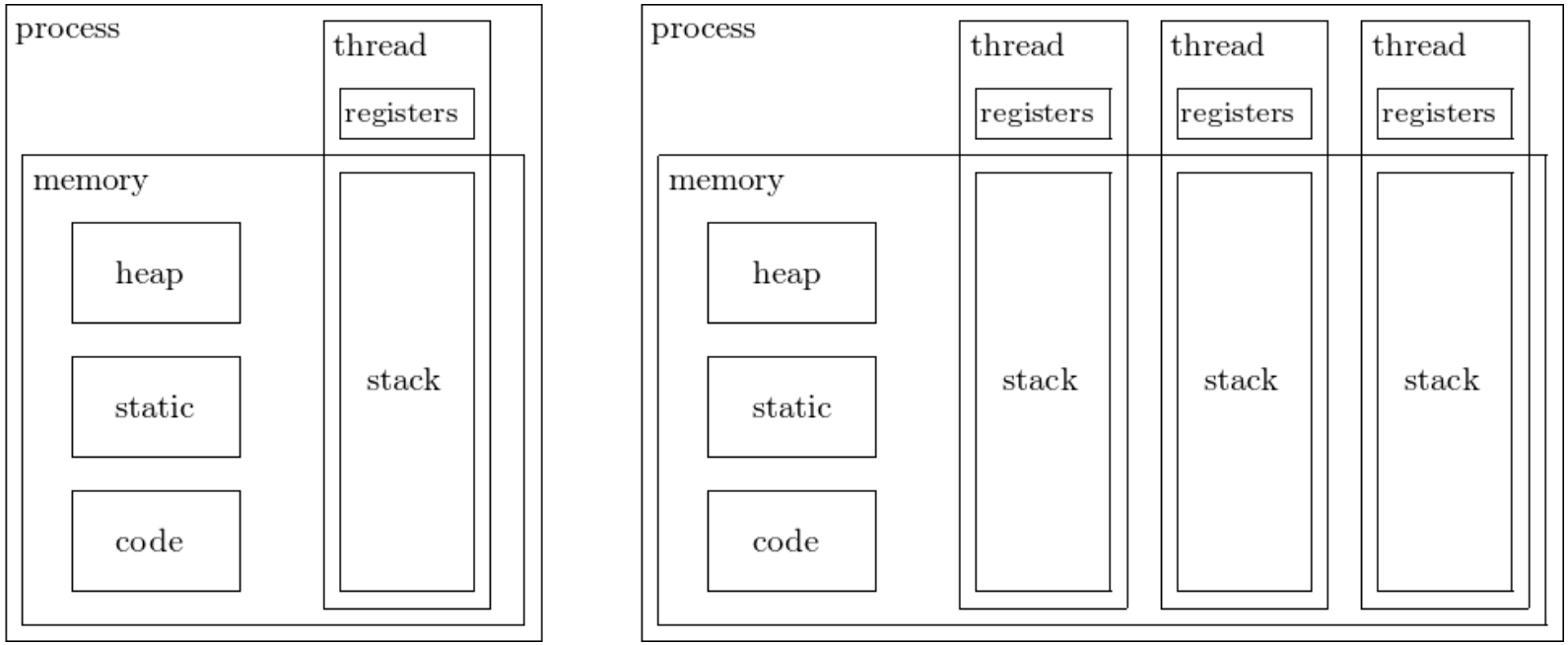
```
int temp = accum;
temp += x * x;
accum = temp;
```

Angenommen die ersten beiden threads laufen gleichzeitig, dann erhält man folgenden Ablauf:

```
// Thread 1                 // Thread 2
int temp1 = accum;          int temp2 = accum;          // temp1 = temp2 = 0
                            temp2 += 2 * 2;             // temp2 = 4
temp1 += 1 * 1;                                         // temp1 = 1
accum = temp1;                                          // accum = 1
                            accum = temp2;              // accum = 4
```

Das Ergebnis von `accum` ist *4* anstelle des korrekten *5*.

Memory-Layout – Single Thread vs Multi Thread



10.3 Mutex

Ein sog. `mutex` (mutal exclusion) erlaubt es, Programmblöcke einzuschliessen, die nur in einem thread gleichzeitig ausgeführt werden dürfen. Vorheriges Beispiel sollte also folgendermaßen erweitert werden:

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;
int accum = 0;
mutex accum_mutex;
void square(int x) {
    int temp = x * x;
    accum_mutex.lock();
    accum += temp;
    accum_mutex.unlock();
}
int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
```

```
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

Die vorher aufgetretene *"race condition"* ist somit behoben. Der erste thread erhält den sog. *lock* nach dem Aufruf von `lock()`. Währenddessen warten alle anderen threads, die `lock()` aufgerufen haben, bis der lock des mutex aufgehoben ist. Im Programmbeispiel ist es wichtig, die zusätzliche Variable `temp` einzuführen, das die Berechnung von `x*x` außerhalb des lock-unlock-Block stehen sollte, da anderenfalls der lock blockiert wird, während die zeitintensiven Berechnungen ausgeführt werden.

Das Vermeiden von *race conditions* bzw. das Verwenden von *locks* ist ziemlich heikel. Insbesondere bei Verwendung von mehreren *locks* kann man leicht in eine *dead-lock* Situation kommen.

10.4 Atomic

C++11 bietet für das im vorhergehenden Abschnitt beschriebene “race condition” Problem eine noch elegantere Lösung mit dem sog. `atomic` container:

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>
using namespace std;
atomic<int> accum(0);
void square(int x) {
  accum += x * x;
}
int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

Die Variable `temp` muss in diesem Fall nicht einführt werden, da die Operation `x*x` vorher ausgewertet wird, bevor das Ergebnis zu `accum` weitergegeben wird, d.h. es ist außerhalb des atomic Ereignisses.

10.5 Tasks

Eine noch höhere Stufe der Abstraktion vermeidet das Konzept von threads und verwendet anstatt den Begriff `tasks`. Betrachten wir folgendes Beispiel:

```
#include <iostream>
#include <future>
#include <chrono>
using namespace std;
int square(int x) {
    return x * x;
}
int main() {
    auto a = async(&square, 10);
    int v = a.get();
    cout << "The thread returned " << v << endl;
    return 0;
}
```

Der `async` Befehl verwendet ein Objektpaar, welche `promise` und `future` genannt werden. Der promise macht eine “Versprechung” irgendwann einen Wert zurückzugeben. Der future ist mit dem promise verbunden und kann zu jeder Zeit versuchen, den versprochenen Wert mit der Methode `get()` zu erhalten. Wenn der promise noch nicht erfüllt ist, wird der future auf diesen warten, bis der Wert

erhältlich ist. `async` verbirgt das meiste hiervon außer dem Rückgabewert, der in diesem Fall ein Objekt vom Typ `future<int>` ist. Auch hier kann `auto` verwendet werden.

Ein weiteres Beispiel zu tasks und deren Ausführung wird in den Übungsaufgaben besprochen, Dort wird auch auf die Verwendung von `this_thread` eingegangen.

10.6 Condition Variables

Bei der Verwendung von threads ist es manchmal sinnvoll, daß ein thread auf einen anderen thread wartet, bis dieser mit dem Prozessieren einer Aufgabe beendet ist, d.h. daß Signale zwischen den threads ausgetauscht werden sollten. Dies kann im Prinzip mit der Verwendung von `mutexes` geschehen, ist aber etwas umständlich. Oder eine globale boolean Variable `notified` könnte z.B. auf `true` gesetzt werden, wenn ein Signal gesendet werden soll. Der andere thread kann in einer Schleife diese Variable notified überprüfen und bricht die Schleife ab und fährt fort, falls `notified` auf `true` gesetzt wurde. Allerdings verbraucht der wartende thread aufgrund der Schleife unnötige CPU-Zeit. Mit einem zusätzlichen `sleep_for` innerhalb der Schleife könnte die CPU die meiste Zeit in den Leerlauf gesetzt werden.

Eine elegantere Methode ist allerdings einen Aufruf von `wait` hinzuzufügen, der auf eine *condition variable* innerhalb der Schleife wartet:

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
#include <chrono>
#include <queue>
using namespace std;
condition_variable cond_var;
```

```
mutex m;
int main() {
    int value = 100;
    bool notified = false;
    thread reporter([&]() {
        unique_lock<mutex> lock(m);
        while (!notified) {
            cond_var.wait(lock);
        }
        cout << "The value is " << value << endl;
    });
    thread assigner([&]() {
        value = 20;
        notified = true;
        cond_var.notify_one();
    });
    reporter.join();
    assigner.join();
    return 0;
}
```

Das Beispielprogramm verwendet die neue C++11-Syntax, die es erlaubt eine thread Funktion direkt als anonyme Funktion zu definieren. Diese anonyme Funktion wird im lokalen scope definiert und kann damit auf die Variablen `value` und `notified` zugreifen.

Ohne *condition variable* würde das Programm in den meisten Fällen *100* ausgeben.

Mit *condition variable* wartet der thread `reporter` auf den thread `assigner`, der den Wert *20* setzt. Im thread `assigner` wird `notified` auf `true` gesetzt und das Signal über die conditions Variable `cond_var` gesendet. Im thread `reporter` wird eine Schleife solange ausgeführt solange `notified` den Wert `false` hat und in jeder Iteration wird auf das Signal gewartet.

Die Verwendung von `notified` in einer Schleife im thread `reporter` ist eigentlich “doppelt gemoppelt”, und nur zusätzliche Absicherung falls `notfiy_one` anderswo gesetzt wird.

10.7 Thread–Pool

Neben den Problemen mit shared-memory Zugriff und Race–Conditions ist ein weiteres Problem die sinnvolle Regulierung der Anzahl der aktiven Threads.

- Freie CPU cores möglichst gut ausnutzen, d.h. mit aktiven Threads bestücken.
- Zu viele Threads sind schlecht: viele Context–Switches, cold caches, etc., kosten Zeit und bremsen den Gesamt–Durchsatz
- Memory und IO limits beachten

Was genau ”sinnvoll” ist hängt von verschiedenen Parametern ab:

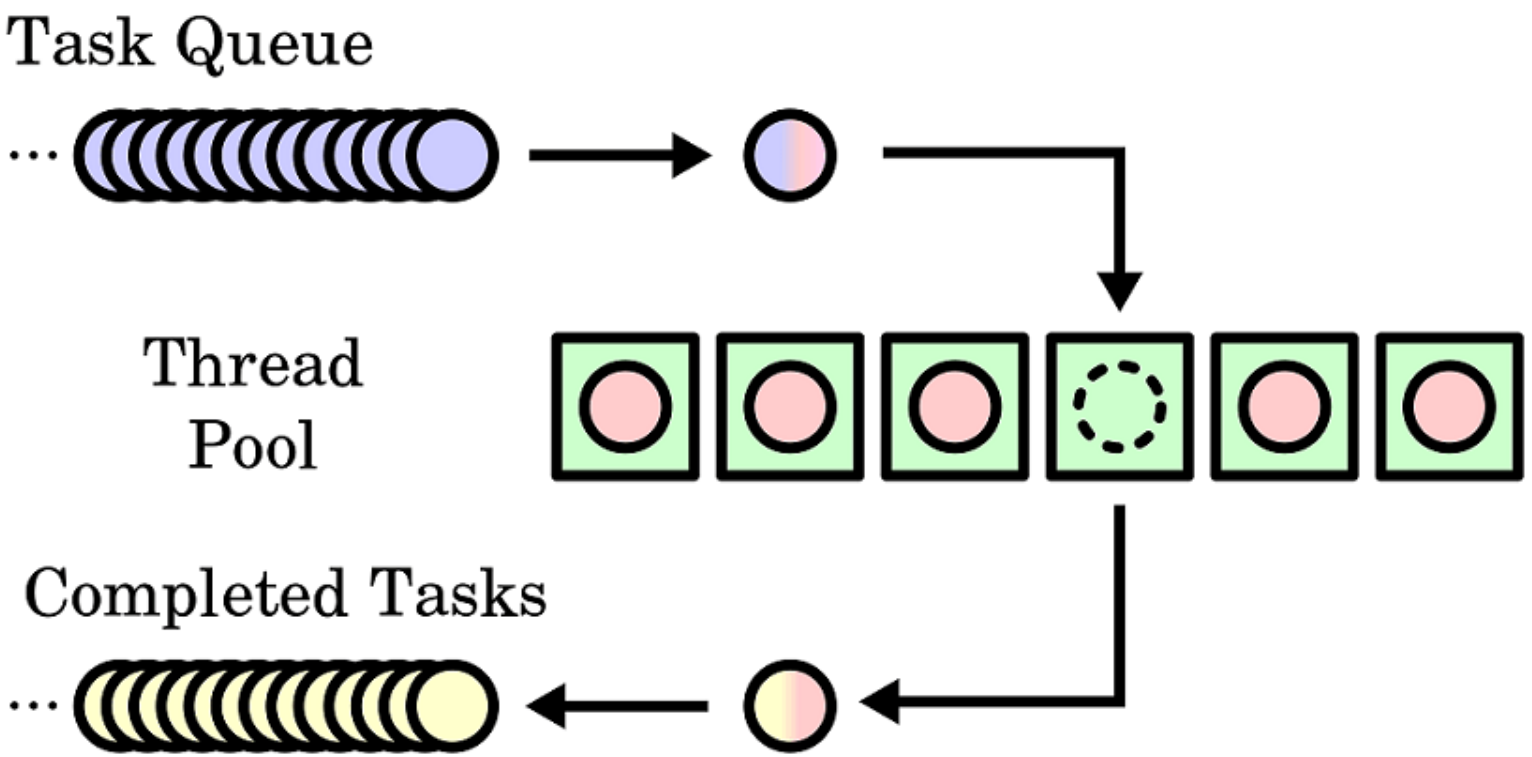
- Rechnerkonfiguration: Zahl CPUs/Cores, Memory, etc
- Konkurierende Prozesse: Hat man System exklusiv oder ist es Teil eines Batch Clusters auf dem man nur eine bestimmte Zahl CPU cores gebucht hat
- Resourcenbedarf der Threads
- ...

Das Anpassen der Zahl der Threads an die jeweilige Rechnerumgebung ist mühsam, zeitaufwendig und fehleranfällig.

Besser ist andere Organisation bzw Re-Strukturierung des Ablaufs mit **Thread-Pools**:

- Parallel ausführbares Programm wird in Unter-Aufgaben sinnvoller Größe aufgeteilt, die über *TaskManager* abgerufen werden können
- Ein Thread–Pool wird aufgesetzt, d.h. eine konfigurierbare Zahl von Threads wird gestartet.
- Jeder Thread holt sich vom TaskManager die nächste anstehende Aufgabe und führt sie aus ...
- ... und nach Abschluss holt er sich die nächste Aufgabe
- ... solange bis alle Aufgaben erledigt sind.

Damit ist Aufteilung des Problems in Unter-Aufgaben und Zahl der laufenden Threads entkoppelt und man kann das Programm flexibel in unterschiedlichen Umgebungen ausführen. **Aber:** Erhöhter Programmieraufwand (Beispiel in Aufgaben).



((Source))

Es wäre wünschenswert wenn C++11 `future` bzw `async` die Thread–Pool Funktionalität leisten würden. In der Praxis aber auf aktuellen Linux/gcc Umgebungen nicht implementiert, nur 2 Betriebs-Modi:

- *default:* Kein paralleler Thread, serielle Ausführung getriggert durch `get` Aufruf
- `launch::async` flag: Jeder `async` Aufruf started neuen thread

10.8 OpenMP – Alternative für parallele Programme

Wenn es hautptsächlich darum geht Programmabschnitte parallel auszuführen, z.B. *for-loops*, dann bietet **OpenMP** eine interessante und vergleichsweise einfache Alternative zur expliziten Verwendung von Threads.

OpenMP ist weit verbreiteter Standard für Parallelisierung und unterstützt verschiedene Sprachen (C, C++, Fortran)

OpenMP wird gesteuert über C++ Pre-Prozessoranweisungen und speziellen Link–Optionen:

```
// compile: g++ -fopenmp -o helloOMP helloOMP.C
//
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
    #pragma omp parallel
    // Bereich in {} wird in multiple threads parallel ausgefuehrt
    {
```

```
        int ID = omp_get_thread_num();
          cout << " hello" << ID;
          cout << " world " << ID << endl;
    } // Ende paralleler Bereich
}
```

OpenMP macht ein sinnvolles **Thread-Pool** Management auf den Systemen/Umgebung im CIP, d.h. abhängig von Zahl der vorhandenen CPU-Cores wird eine entsprechende Zahl von Threads gestartet, die gleichzeitig laufen.

- Anweisung `# pragma omp parallel for` vor *for-loop* sorgt für parallele Ausführung der Schleife, Zahl der Threads entspricht per Default Zahl der Prozessorkerne.
- Anweisung `# pragma omp critical` schützt nächste Zeile vor paralleler Ausführung, entspricht in etwa dem `lock/unlock` mit `mutex`.

Intel Threading Building Blocks – TBB

sind ein weiterer weit-verbreiteter Standard für multi–threaded Programmieren. TBB erlaubt wie OpenMP einfache parallele for–loops, aber flexibler und moderner mit vielen weiteren Features *(parallel-reduce, Thread-safe Container, ...)*.

Ein schöner Überblick ist in diesem Kurs zu **Super Computing**.

10.9 Aufgaben

1. **C++11 Thread Variablenübergabe**
   Im besprochenen Beispielprogramm zu threads wird die Variable `accum` als globale Variable definiert, was als schlechter Programmierstil gilt. Besser ist es, diese Variable als Parameter zu übergeben. Ändern Sie das Beispielprogramm, indem Sie einen Parameter `int& accum` zu `square` hinzufügen. Dies sollte eine Referenz sein, da `accum` sich ändern lassen sollte. Allerdings kann der Aufruf nicht über `thread(&square, accum, i)` erfolgen, da dies eine Kopie von `accum` erzeugt und `square` mit dieser Kopie aufruft. Verwenden Sie anstatt `ref()`, d.h. `thread(&square, ref(accum), i)`.
   Lösungsbeispiel: thread_lsg1.cpp (nicht thread safe), thread_lsg1b.cpp (thread safe)

2. **C++11 tasks**
   Verwenden Sie anstatt threads den `async`-Befehl im beschriebenen Programm zur parallelen Berechnung der Summe der Quadarate bis 20. Fügen Sie hierzu die `future<int>` Objekte zu einem `vector<future<int>>` hinzu. Am Ende des Programms iterieren Sie über alle futures und erhalten den jeweiligen Rückgabewert und fügen diesen zur Gesamtsumme hinzu.
   Lösungsbeispiel: thread_lsg2.cpp

3. **C++11 tasks und this_thread**
   Implementieren Sie das C++11 task Beispielprogramm, wie in der vorhergehenden Aufgabe

beschrieben. Um festzustellen, ob das Programm tatsächlich parallel ausgeführt wird, versuchen Sie folgendes:

- Fügen Sie eine Bildschirmausgabe der Variable `x` in die Methode `square` hinzu und führen das neue Programm aus. Die Bildschirmausgabe zeigt eine deterministische Ausgabe und keine besonderen Anzeichen für paralleles Ausführen.
- Fügen Sie als nächstes eine `sleep`-Aufruf zur Methode `square` hinzu, um eine CPU-intensive Berechnung zu simulieren:
  ```
  this_thread::sleep_for(chrono::milliseconds(100));
  ```
  Führen Sie das Programm wieder aus und bestimmen Sie die Ausführzeit. Die tasks werden hintereinander ausgeführt und laufen nicht parallel.
- Fügen Sie eine Bildschirmausgabe von `this_thread::get_id()` zur Methode `square` und zu `main` hinzu. Auch die Ausführung von main wird als thread aufgefasst - was ist die entsprechende Bildschirmausgabe ?
- Die Funktion `async` kann Methoden *asynchron* oder *verzögert* ausführen. Eine verzögerte Ausführung bedeutet in unserem Beispiel, daß die Methode `square` erst aufgerufen wird, wenn `get()` aufgerufen wird. Zusätzlich wird die Methode im gleichen thread (also in diesem Fall `main()`) ausgeführt. Das Programm bzw. der Compiler sollte aber eine intelligente auf Geschwindigkeit optimierte Entscheidung über den Ausführmodus treffen. GCC verwendet aber als Standardeinstellung den verzögerten Modus. Eine parallele bzw. asynchrone Ausführung

kann erzwungen werden mit:

`async(launch::async, &square, ...)`

Fügen Sie diesen code zum Beispielprogramm hinzu und führen Sie diesen wieder aus und messen die Ausführgeschwindigkeit - welche Geschwindigkeitverbesserung erhalten Sie ?

Lösungsbeispiel: thread_lsg3.cpp

4. **Erzeuger-Konsumenten Problem**

   Das Erzeuger-Konsumenten (producer-consumer) Problem ist in einfacher Form ein thread, der Waren erzeugt, und ein weiterer thread, der diese Waren konsumiert. Dabei soll der consumer thread beim Warten eine conditions Variable verwenden. `goods.push(i)` und `good.pop()` sollen sich gegenseitig ausschließen, so daß die Daten nicht beschädigt werden. Die Variablen `c++` und `c--` sollen dabei als Kontrolle dienen, da am Ende sich alles zu *0* addieren sollten.

   Führen Sie folgendes Beispiel-Programm aus: `tconsumer.cpp`.
   Welche Ergebnisse erhalten Sie bei mehrfachem Ausführen ?

   Versuchen Sie das Programm zunächst mit lock-unlock Blöcken zu reparieren und führen das Programm 1000-10000 mal aus, um das Ergebnis zu verifizieren. Versuchen Sie dann, conditions Variablen zu verwenden.

   Lösungsbeispiel: thread_lsg4.cpp

5. **Aufzug Simulation**

   Nehmen Sie ein mehrstöckiges Haus an, in dem mehrere Personen arbeiten. Die Personen wech-

seln gelegentlich die Etagen und benutzen dazu einen 1–Personen Fahrstuhl (also noch kleiner als in Schelling 4 ...). Jede Person soll in einem eigenen Thread simuliert werden, die Arbeit zwischen der Aufzugbenutzung wird durch *sleep(some-random-time)* simuliert. Aufzugfahren ist eine separate Klasse/Methode, die Fahrzeit wird wiederum mit *sleep()* proportional zur Distanz simuliert. Mittels *Mutex lock()/unlock()* kann sichergestellt werden, dass nur eine Person (=Thread) den Aufzug verwendet. Ausgabe der Aktionen auf *stdout*.

Studieren Sie den Einfluss von Zahl der Stockwerke, Zahl der Beschäftigten, u.a., auf den Durchsatz.

Einfaches Beispielprogramm: AufzugSimCpp11.cpp (mit C++11 threads) bzw. AufzugSim.cpp (mit QThreads).

6. **(Projekt) Primzahl-Finder Multi–Threaded**
   Ein einfaches Programm zum Finden von Primzahlen prime_simple.C soll so abgeändert bzw. erweitert werden, so daß die Suche verteilt und unabhängig auf verschiedenen Zahlenbereichen durchgeführt werden kann.
   - Anschließend soll die Primzahlsuche in den verschiedenen Zahlenbereichen auf verschiedene Threads verteilt werden.
   - Lösungsbeispiel:

     – (a) Direktes Starten der Threads/Tasks ('Push-Model'): TaskThreadPrime.cpp
     – (b) Thread-Pool/Task-Manager, die Threads holen sich Tasks ('Pull-Model'): TaskThreadPrime_TMan.cpp

7. **OpenMP Beispiel zur Berechnung von Pi**
   PiOMP.cpp ist ein kleines Programm das Pi über Zufallszahlen berechnet und OpenMP zur Parallelisierung verwendet.

   Versuchen Sie es nachzuvollziehen.

   Wozu braucht man `# pragma omp critical`?

   Warum benutzt man `drand48_r` und `srand48_r( ...)` ?

   Modifizieren Sie das Programm und verwenden statt **OpenMP** C++11 `tasks` bzw `async`.

11 Glossar

11.1 C/C++ Basics

Argument Ein Wert, der an eine Funktion übergeben wird. (siehe auch Parameter).

Array (Feld) von Variablen eines bestimmten Typs, z.B. `int a[10]`.

Ausdruck (expression) Eine Folge von Operatoren und Operanden, die ein einzelnes Ergebnis liefert.

Ausführbare Anweisung Ausdruck gefolgt von einem Semikolon.

Ausführbares Programm (Executable) Ein Programm, das unter Steuerung eines Betriebssystemes oder einer ähnlichen Laufzeitumgebung arbeitet.

Block Eine Folge (Sequenz) von Definitionen, Deklarationen und Anweisungen, die in geschweiften Klammern {} eingeschlossen werden.

Compiler (Übersetzungsprogramm). Ein Programm zur Umwandlung von Quellcode (z.B. C-Programm) in Maschinencode.

Datentyp Definition der Daten, z.B. int, char, float.

Definition 1. Die Definition einer Variablen ist eine Deklaration, bei der Speicherplatz zugeordnet wird, z.B.: `int a; char c; ThreeVector v`

2. Die Definition einer Funktion oder Klassen–Methode ist die Implementierung der Funktion/Methode; im Unterschied zur *Deklaration* umfasst dies den eigentlichen Programm-code der Funktion.

Deklaration Ein Konstrukt, der einem Variablen-, Klassen- oder Funktionsbezeichner Attribute zuordnet. Es wird kein Speicherplatz reserviert. Typisches Beispiel: Funktion- oder Klassendeklaration in Header-File; damit wird der Name, die Parameter und Typ des Rückgabewertes einer Funktion festlegt. z.B.: `double square( double x);`

Escape Sequenz Steuerzeichen, die aus einer Kombination von Back-slash (\) und einem Buchstaben oder einer Zahl bestehen, z.B. $\backslash n$ für "newline".

Floating-point Number (Gleitkommazahl) Eine Zahl mit Dezimalpunkt und Exponent.

Funktion Eigenständiger Programm–Block mit *Identifier*. Kann von andren Stellen im Programm oder in anderen Source-Files gerufen werden. I.a. Übergabe von Argumenten und Rückgabe von Werten. Z.B.

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

Header Files enthalten Deklarationen von Funktionen oder Klassen. Um externe Funktionen oder Klassen verwenden zu können, muss zuvor das entsprechende header file eingebunden werden, z.B.

`#include <cmath>` um `sin(), tan(), sqrt()` verwenden zu können.

Identifier (Bezeichner) Ein Name der verwendet wird, um auf abgespeicherte Daten wie Konstanten,

Variablen or Funktionen zuzugreifen.

Integer (ganze Zahl) Eine Zahl ohne Nachkommastellen.

Keyword (Schlüsselwort) Ein Wort, das in C++ eine festgelegte Bedeutung hat, z.B. `if, while, int, ...`; darf nicht für andere Zwecke verwendet werden.

Konstante Eine Größe, die einen festen Wert repräsentiert, der nicht verändert werden kann.

Library (Bibliothek) Ein File, das vorcompilierte Funktionen enthält. Diese können zu einem Objekt-File hinzu *gelinkt* werden, um daraus ein ausführbares Programm zu machen.

Library Funktion (Bibliotheksfunktion) Eine Funktion, die als Objektfile in einer Library gespeichert ist.

main Funktion: per Konvention in C/C++ ist die **erste** Funktion, die immer beim Start eines C/C++ Programms gerufen wird, eine Funktion mit Signatur `int main( int argc, char** argv)`

Literal Buchstaben, Ziffern und Zeichenketten, die nicht interpretiert, und deshalb als Konstanten und nicht als Bezeichner verwendet werden.

Object-Code Anweisungen in Maschinensprache, erzeugt der Compiler aus dem Quell-Code.

Operand Variable oder Ausdruck neben einem Operator, z.B.: `z = a + b;` a und b sind beide Operanden des + Operators, z und (a+b) sind Operanden des = Operators

Operator Symbol für Standard-Operationen, `+-*/=<>...`

Pointer (Zeiger) Eine Variable, die eine Adresse (Verweis auf Stelle im Speicher) enthält.

Parameter Ein Wert, der von einer Funktion übernommen wird.

Preprocessor (Präprozessor) Schritt vor dem eigentlichen Kompilieren: bestimmte Direktiven im Quellcode (`#include ...`, `#define ...`, `#if ...`) werden erkannt und ausgewertet.

Quellcode Ein Textfile, das C++ Anweisungen enthält, die compiliert werden können.

Scope (Sichtbarkeitsbereich)

String (Zeichenkette) Als Konstante einfach durch *double-quotes* (”Some String”) definiert, in C ein Array von Buchstaben, mit dem Null Zeichen ($\backslash 0$) begrenzt sind, in C++ eigener Datentyp *(string Klasse)*.

Syntax error (Syntaxfehler) Ein Fehler im Quellcode, der dazu führt, dass der Compiler das Programm nicht übersetzt.

Variable Bezeichner (und Speicherplatz) für eine Größe mit bestimmten Eigenschaften (Datentyp) deren Wert sich ändern kann, während das Programm läuft.

Zuweisung (Assignment) Variable bekommt Wert oder Ergebnis eines *Ausdrucks* zugewiesen, z.B.:
`a = 3 + 6`

11.2 C/C++ Klassen und Objekte

Constructor Spezielle Member–Funktion, die implizit gerufen wird, wenn ein Objekt einer Klasse angelegt wird; dient also v.a. zur Initialisierung. Name des Constructors ist Name der Klasse, kein Typ, kein Rückgabewert.

Destructor Spezielle Member–Funktion, die implizit gerufen wird, wenn ein Objekt einer Klasse gelöscht wird (out-of-scope geht). Name ist ~ plus Name der Klasse. Explizite Implementierung i.d.R. nur dann nötig falls das Objekt *dynamischen Speicher* anlegt.

Klasse (class) Grundbaustein für objekt-orientiertes Programmieren. Ermöglicht Definition eines maßgeschneiderten Datentyps, beinhaltet Member–Variablen und Member–Funktionen.

Member–Function Funktion, die innerhalb einer Klasse definiert, aber nur zusammen mit einem konkreten Objekt der Klasse aufgerufen werden kann: `Objekt-name.Member--Function(...)`

Member–Variable Variable, die innerhalb einer Klasse deklariert ist. Für jedes Objekt der Klasse wird ein unabhängiger Satz von Member–Variables angelegt. Aufruf: `Objekt-name.Variable`

Object (Objekt) *Instanz* einer Klasse. Wird angelegt bei *Definition* einer entsprechenden Variablen: `Class-name Var-name`

Operator overloading C++ Operatoren (z.B. `+-*/=<>...`) können analog zu Funktionen definiert

werden für beliebige neue Datentypen.

private Zugriffs–Bezeichner: Nur Funktionen, die zur Klasse gehören, dürfen auf *private* members (Funktionen, Variablen) zugreifen

public Zugriffs–Bezeichner: Zugriff auf Member–Variables und Member–Functions von überall her.

this C++ Schlüsselwort. Innerhalb einer member-function ist *this* pointer auf das Objekt für das die member-function gerufen wurde.

Vererbung (inheritance) Mechanismus wie eine abgeleitete Klasse alle Member–Variables und Member–Functions einer Basis–Klasse übernehmen kann.