

C++ für Physiker

Günter Dückeck

Johannes Elmsheuser

10.-14.03.2014

Klausur, Freitag, 14.03., Start: 12:55 Uhr

Inhalt:

Allgemeines

C/C++ Grundlagen Variables, Control, Funktionen, Arrays, I/O

Klassen und Objekte Vererbung, Operator–Overloading, Templates

Folien & Übungen im WWW ([Slides](#))

<http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckurs>

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
- Wir schicken die Liste der erfolgreichen Kursteilnehmer ans Prüfungsamt, Sie müssen aber dennoch zusätzlich im Prüfungsamt angeben welchen Kurs Sie als Schlüsselqualifikation anrechnen wollen, siehe http://www.physik.uni-muenchen.de/studium/infos_studium/faq/meldungen/index.html

Allgemeines

Themen des Kurses

- Programmiersprache C/C++
- Klassen & Objekte in C++
- Evt. noch Standard Template Library (STL)

im Prinzip Material für jeweils 1-2 Vorlesungen.

⇒ Kurs zwangsläufig oberflächlich und unvollständig.

Ziele des Kurses C++ für Physiker:

- Fähigkeit zum Erstellen kleiner Programme (z.B. Übungen–Numerik)
- Basiskenntnisse um vorhandene Programme zu verstehen, benutzen und weiterzuentwickeln
- Grundlagen zur Anwendung von Standardbibliotheken.

Aber:

- Keine umfassende Präsentation des kompletten C/C++ Sprachumfangs
- Keine Schulung objektorientiertes Design/Programmieren
- ...

Gute Ergänzung:

Objektorientiertes Programmieren in C++

- Baut auf diesem Kurs auf, d.h. setzt C/C++ Grundkenntnisse voraus
- Vertieft und erweitert Programmieren in C/C++
 - Standard Template Library
 - Abstrakte Klassen und Polymorphismus
 - Objektorientiertes Design und UML (Unified Modeling Language)
 - Grafik mit der Qt Library
 - ROOT System oder Multi-Threading

Genauer Termin unklar (SommerSemester 2014).

Ablauf

- Mo, Di, Mi, Do, Fr; jeweils 10:00 – 12:00 und 13:30 – 16:00.
- Schwerpunkt praktische Übungen: ca. 1/3 Theorie, ca. 2/3 Übungen am Rechner im CIP.
 1. **C/C++ Grundlagen** 3-4 Blöcke
 2. **Klassen und Objekte** 3 Blöcke
 3. **Templates** 1-2 Blöcke

Hinweis und Warnung:

- Kurs steigt ziemlich schnell ziemlich tief ein, Lernkurve in C/C++ ist steil
- Besonders in den ersten beiden Tagen (Schnellkurs C Basics)
- Besonders für Leute mit wenig/keiner Erfahrung im Programmieren
- Fragen, Fragen, Fragen, in Vorlesung, in Übungen, Kollegen, ...
- Nicht aufgeben, dranbleiben, Mut zur Lücke, ...
- Am wichtigsten sind praktische Übungen, eigenes Programmieren.
- Kursfolien/Webseiten zum Nacharbeiten. Unzureichend für Selbststudium ⇒ Literatur
- Zum Lernen, insbesondere als 1. Programmiersprache, sind Sprachen wie **Python, JAVA** wesentlich einfacher...

Literatur und Links

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

Lehrbücher

Thinking in C++ Bruce Eckel. Umfangreich und gut verständlich, hervorragende Einführung in objektorientiertes Programmieren, erhältlich Online (html) und als Buch.

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

Einstieg in C++ Arnold Willemer, Online und als Buch. Galileo (2004). Gut für Einsteiger, CD mit Open-source Compiler und Entwicklungsumgebung.

C++-Entwicklung mit Linux Thomas Wieland, Online und als Buch, 2004. Ausführliches, vollständiges Lehrbuch, sehr nützliche Ergänzungen und Hinweise zum Arbeiten im Linux Umfeld.

How to Think Like a Computer Scientist (C++) erhältlich als Buch und [online](#) . Gute konzeptionelle Einführung ins Programmieren. Äquivalente Versionen für Java und Python ebenfalls erhältlich !

The C++ Programming Language Bjarne Stroustrup, Addison-Wesley. Das Original vom C++ Erfinder.

Die Programmiersprache C und C++ für C-Programmierer Zwei Skripten von [RRZN](#), Uni Hannover. Übersichtlich, kompakt, preiswert; für je 3 Euro am [LRZ](#) erhältlich.

Referenzen, Ergänzungen

C++ Notes von F. Swartz Übersichtliche Referenz zu C++, für Anfänger geeignet. Gute Ergänzung zum Kurs!

C++ Annotations Gute Online Beschreibung zu C++ und STL, mit ausführlichem Indexverzeichnis.

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

Kurse im WWW

- **Programming Tutorials - C, C++, OpenGL, STL (engl.)**
- **C++ Language Tutorial (engl.)**
- **C++-Programmierung (deutsch, Wikibooks)**
- **C-Programmierung (deutsch, Wikibooks)**
- Nützlich für Physiker auch noch die **Numerical Recipes** bzw. **frei zugängliche ältere Version**.

Eine sehr schöne, für viele Systeme und Sprachen verfügbare Entwicklungsumgebung bietet das **Eclipse Projekt**:

- Von <http://www.eclipse.org/downloads/> *Eclipse IDE for C/C++ Developers (67 MB)* auswählen und installieren.
- Zusätzlich muss C++ Compiler installiert werden, am besten **MingW**:

Instruktionen nach Eclipse Start: ⇒ Help ⇒ C/C++ Development User Guide ⇒ Before you begin ⇒ **MingW**

Eine alternative **freie C++ Version für Windows** inclusive Entwicklungsumgebung ist **Dev-C++** (Version 4, 7.5 MByte).

Gute Tips zur Benutzung finden sich in den **C++ Notes von F. Swartz**.

Physik und Computing

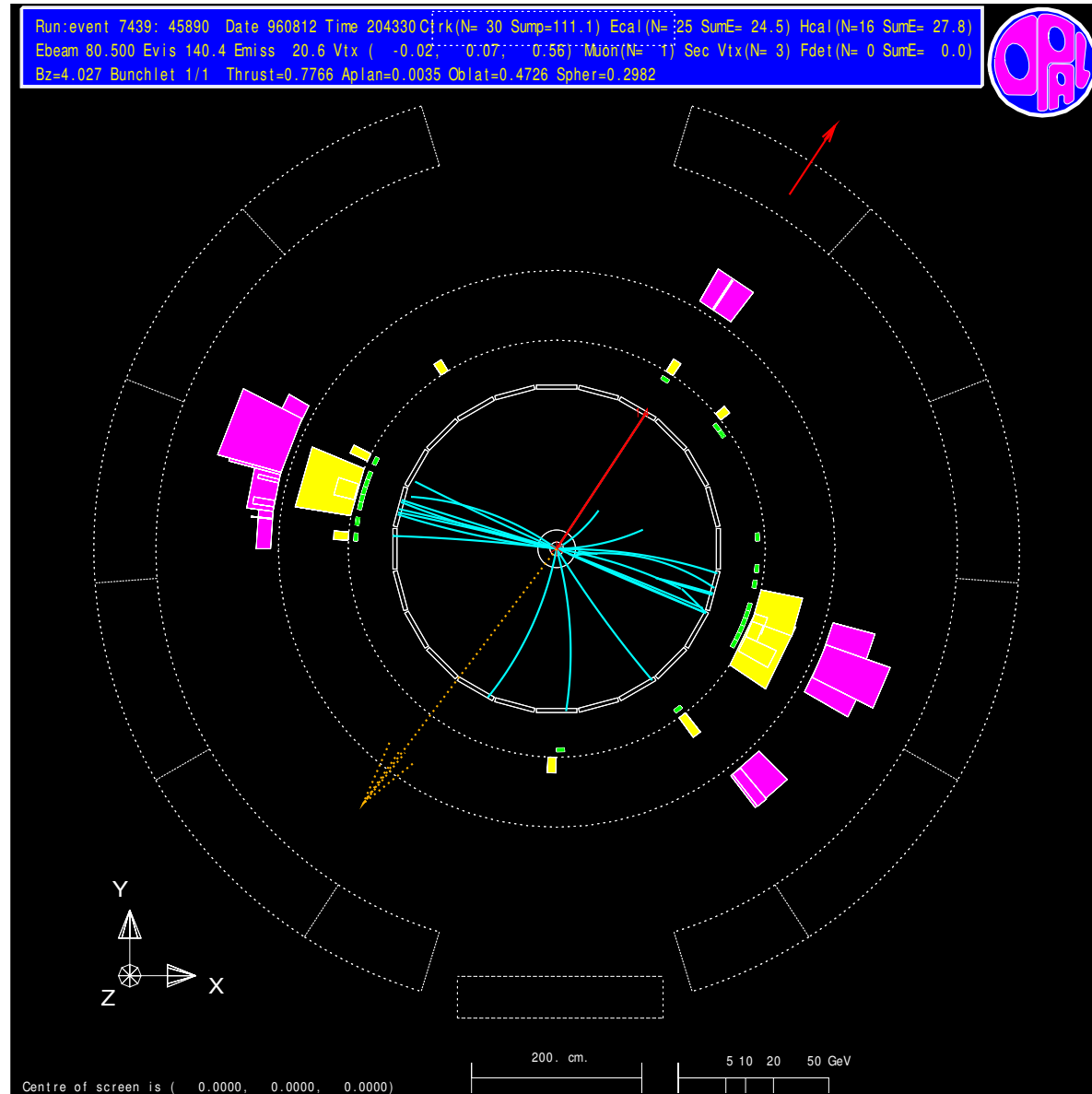
In der Physik sind Computer fast überall von zentraler Bedeutung: Design von Experimenten, Datenauslese, Auswertung und Statistik, Simulation, Theorie, Kommunikation, Recherche, ...

Eine Sonderrolle spielt die Teilchenphysik:

Experimente immer am technologischen Limit bei Datenvolumen und –rate, Prozessorgeschwindigkeit

Teilchenphysik nicht nur Nutzer sondern hat viele Entwicklungen vorangebracht:

- verteiltes Rechnen
- Realtime computing
- WWW am CERN erfunden
- GRID Projekt



Computing Kenntnisse für Physiker – eine Wunschliste

- **Alltag:** Textverarbeitung (Office, Latex), Präsentationen (PPT, TeX), WWW Nutzung, E-Mail
⇒ *Selbststudium*
- **Datenanalyse/Statistik:** Tabellenkalkulation (Excel), SAS, ROOT ⇒ *(Selbst/Kurs)*
- **Mathematik/symbolische Algebra:** Maple, Mathematica, ... ⇒ *(Selbst/Kurs)*
- **Höhere Programmiersprache:** Fortran, C/C++, Java, ... ⇒ *Kurse*
- **Numerik:** Algorithmen (Fortran/C++) ⇒ *Vorlesung (Schein)*
- **Advanced concepts:** OO-Programmieren und -Design, GUI, Threads, Container ⇒ *Kurse*
- **Hardware Programmierung** ⇒ *(Kurs)*
- **Aktuelle, high-level IT Anwendungen:** Databases, XML, Skript-Sprachen, Web-Programmierung, Grid, ... ⇒ *(Python-Kurs, ???)*

Programmiersprachen und Programmieren

Natürliche Sprachen (Deutsch, Französisch, ...) entwickeln sich langsam über viele Jahrhunderte, es gibt keine klaren von vorneherein festgelegten Regeln. Im Gegensatz dazu sind **formale Sprachen** gezielt für bestimmte Aufgaben entworfen worden, z.B. in Musik, Mathematik, Chemie, oder eben die Programmiersprachen um Abläufe in der *Datenverarbeitung* auszudrücken.

Gewisse grundsätzliche Gemeinsamkeiten: *Struktur, Syntax, Begriffe*, aber formale Sprachen viel **präziser, strikter, dichter**. Keine oder kaum Ambiguitäten, wenig Redundanz, genau umrissene Bedeutung, keine Metaphern o.ä.

Konsequenz für Lesen und Verstehen:

- Informationsdichte in Programmtext wesentlich größer
- kein Lesen von Anfang bis Ende, sondern Orientierung an Struktur
- jedes Detail ist wichtig

Was ist Programmieren ?

Ein Programm ist eine Reihen von Anweisungen, die bestimmen wie eine Datenverarbeitung abläuft. Dazu sind nur wenige grundlegende Funktionen nötig, die im Prinzip allen Programmiersprachen gemeinsam sind:

- **Input:** Daten von Tastatur, File, Netzwerk, Sensor, ...
- **Output:** Daten auf Bildschirm, Datei, Drucker, Steuergerät, ...
- **Operation:** mathematischer Ausdruck, Zuweisung, ...
- **Testen und Verzweigen:** Überprüfen von Bedingungen, unterschiedliche Abläufe
- **Schleifen:** Wiederholte Ausführung bestimmter Abschnitte

C/C++: Main Features

- C ist höhere, portable Programmiersprache. Ursprünglich (70er) entwickelt als Alternative zu Assembler für hardware/systemnahe Programmierung.
- C++ ist objekt-orientierte Erweiterung, erzwingt *strong-typing* und Prototypes.
- C/C++ ist Compiler–Sprache, wie Fortran, Pascal, etc. D.h. Quellcode wird zunächst *kompiliert*, dann werden *externe libraries ge-linkt*.

Im Gegensatz zu Interpreter–Sprachen oder Skriptsprachen wie

- Basic, Java
- Perl, Python, php, Shells

Pros:

- Riesiger Funktionsumfang, alles möglich von kleinen Hardware-Treibern mit wenigen Zeilen bis komplexen S/W Projekten mit vielen Millionen Zeilen.
- Direktes Ansprechen der Hardware mit Pointern
- Volle objektorientierte Funktionalität, einschliesslich 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
- Hilfspakete (I/O, Networking, Graphik, Databases, ...) **nicht** in Standarddistribution integriert.

1 C/C++ Grundlagen

- Die ersten Schritte – Kompilieren, Programme ausführen
- C/C++ Operationen
- Grundlegende Datentypen und Definition von Variablen
- Control-statements
- Arrays, Pointer und Referenzen
- Funktionen
- Gültigkeitsbereich von Variablen
- Pointer und Referenzen, cont'd
- Sonstiges
- Programme debuggen
- Aufgaben

1.1 Die ersten Schritte

Das Standard-Minimal-Programm:

In C/C++:

```
/* Hello-world  
C Version */  
#include <stdio.h> /* pre-prozessor command */  
/* start function main */  
int main()  
{ /* begin function body */  
  printf("Hello world \n"); /* function call printf(...) */  
  return(0); /* return some value */  
} /* end function body */
```

- Preprocessor command, spezifiziert *header-file* für `printf` Definition.
- Function definition: `main()`. Funktionen brauchen Typ (`int`) und Klammern (). Der Name `main` ist speziell \equiv Hauptprogramm.
- Geschweifte Klammern `{...}` schliessen Block von Code und Deklarationen ein.
- Function call `printf(...)`. Funktion aus der C Standard Library. Nicht direkter Bestandteil der Sprache, muss mittels *header-file* deklariert werden, s.o.
- Semikolon `;` wichtig, schliesst jede Anweisung (=Statement) ab. Zeilenende spielt keine Rolle, ein Statement kann sich über beliebig viele Zeilen erstrecken.
- I.a. geben Funktionen einen Wert zurück. Hier wird einfach mit `return(0)` der `int` Wert `0` zurückgegeben.
- Kommentare: Alles zwischen `/*` und `*/` wird ignoriert, egal ob
 - Teil einer Zeile
 - ganze Zeile
 - mehrere Zeilen

In C++ ganz ähnlich:

```
// Hello-world C++ Version
#include <iostream> // pre-prozessor command
using namespace std; // declare namespace
int main() // function definition
{
    cout << "Hello world" << endl;
    return(0);
}
```

- Weitere I/O Funktionen
- Kommentare zusätzlich alles nach `//` bis Zeilenende
- namespace Deklaration bei aktuellen C++ Compilern nötig, definiert *Namensraum* der verwendeten Funktionen, falls nicht explizit angegeben. \Rightarrow Später

Editieren, z.B.:

```
kate Hello.C
```

oder:

```
gedit Hello.C
```

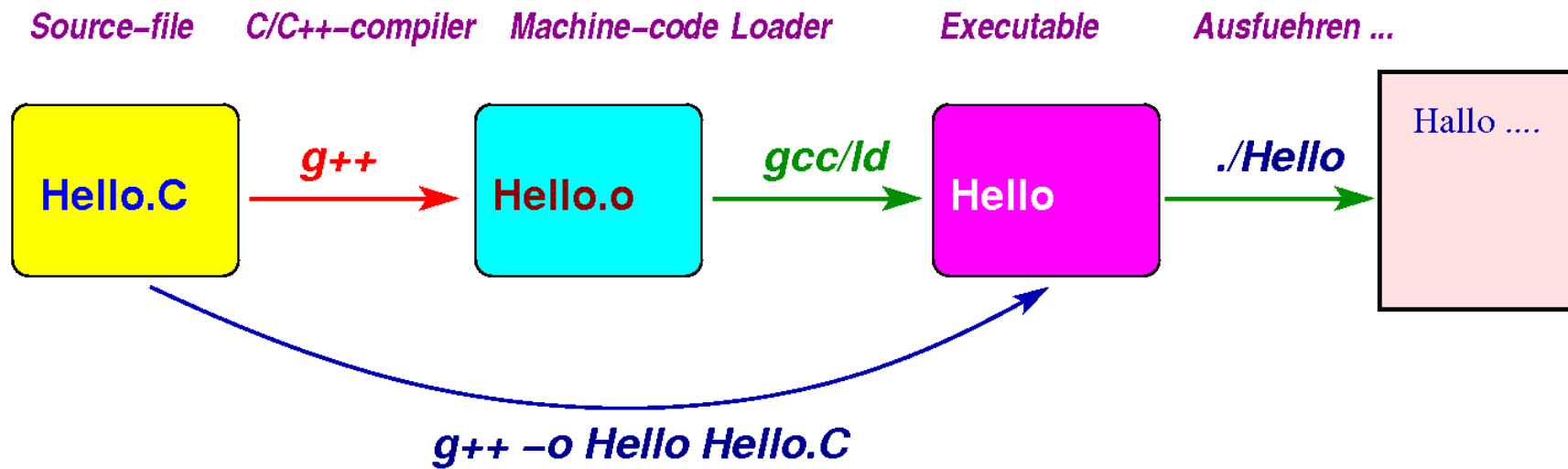
(Tip: cut & paste mit Maus aus Firefox Browser)

Kompilieren:

```
g++ -o Hello Hello.C
```

Ausführen:

```
./Hello
```



Was macht der Compiler ?

C Source

```
...  
c = a + b;  
...
```

g++ -c ...

Pseudo-Maschinen-Code

```
...  
fetch( addr_a, reg_1)  
fetch( addr_b, reg_2);  
add( reg_1, reg_2);  
put( reg_1, addr_c);
```

Was macht der Interpreter ?

Im Prinzip dasselbe, der Unterschied ist:

- Compiler übersetzt einmal alles und schreibt Ergebnis (=o-File/Executable) in neues File. Anschliessend wird Executable gestartet: \Rightarrow unabhängig von Compiler.
- Interpreter übersetzt einzelne Anweisung und führt sie sofort aus, d.h. Programm wird vom Interpreter gestartet.

Konventionen (nicht generell aber für Kurs):

- C source files Endung `.c`
- C++ source files Endung `.C`, `.cpp`, `.cxx`
- C header files Endung `.h`
- C++ header files Endung `.h`, `.hxx`

1.2 Ein kleines Programm

```
// print Fahrenheit->Celsius conversion table
#include <iostream>
using namespace std; // declare namespace
int main()
{
    int lower(0), upper(300), step = 20;           // declaration and initialization
    double fahr, celsius;
    fahr = lower;
    while ( fahr <= upper ) {                       // while loop
        celsius = (5.0/9.0) * (fahr-32.0);           // rechnen ...
        cout << fahr << " " << celsius << endl; // ausgeben ...
        fahr += step;                               // rechnen ...
    } // end-while
} // end-main
```

Kommentar:

```
// in C++,  
/* ... */ in C.
```

Ganze Zeile oder nach Statement. Kommentare werden vom Compiler ignoriert; sind aber entscheidend für Programmdokumentation,

Reichlich verwenden !

Deklaration und Initialisierung von Variablen.

In C/C++ müssen alle Variablen vor erster Verwendung deklariert werden (*strong typing!*):

```
type variable-name ; (Datentypen später)
```

loop-Statement:

```
while ( Bedingung ) { expressions }
```

- Bedingung `fahr <= upper` wird getestet
(Typ-konversion `upper: int --> double`)
- wenn erfüllt werden `expressions` ausgeführt
dann zurück zu.
- Andernfalls zum Ende der Schleife

1.3 Grundlegende Begriffe

Datentyp

In C++ sind eine Reihe von Datentypen definiert mit denen Operationen möglich sind, z.B.:

```
1247 // int = ganze Zahl
```

```
42 - 15 // Subtraktion von ganzen Zahlen
```

```
3.1415 * 2.0 // Multiplikation von Gleitkommazahlen
```

```
"Hallo Welt" // String
```

```
"Hallo " + "Muenchen" // String-addition geht
```

```
"Hallo Muenchen" * 3.1415 // falsch
```

I.a. keine Operationen mit verschiedenen Datentypen möglich.

Variable Elementares Feature einer Programmiersprache ist die Möglichkeit mit Variablen zu operieren. Eine *Variable* ist eine Art Bezeichner für eine Speicherstelle, an der ein Wert gespeichert werden kann. Für eine Variable ist ein fester Datentyp zugeordnet, vor erster Verwendung muss Variable **definiert** sein: *Datentyp Variablenname*, z.B.

```
int i; // int Variable i angelegt
i = 42 - 15; // Ergebnis einer int Operation zugewiesen
double pi = 3.1415; // Gleitkomma-Variable pi angelegt und initialisiert
double umfang, radius; // Gleitkomma-Variablen umfang und radius angelegt
radius = 2.5; // Wert fuer Radius zugewiesen
umfang = 2. * pi * radius; // Umfang berechnet nach Formel und zugewiesen
string gruss = "Hallo "; //
string name1 = "Gerd"; //
string name2 = "Angie"; //
string text1 = gruss + name1; //
string text2 = gruss + name2; //
```

Ausdruck (=expression):

ganz allgemein für beliebiges *C++ Kommando*, liefert i.d.R. Ergebnis, z.B.:

```
3 + 6
```

```
x > y
```

```
1./sqrt(2.*sigma) * exp( -(x-x0)*(x-x0)/(sigma*sigma)) // Gauss Funktion
```

```
cout << "Hello World"
```

Kommando ist in C++ definierte *Operation* oder *Funktionsaufruf*, (*keine Kommentare !*)

Zuweisung (=assignment):

Variable bekommt Wert oder Ergebnis eines *Ausdrucks* zugewiesen, z.B.:

```
a = 3 + 6
```

```
x = sqrt(2.)
```

Keine Gleichung im mathematischen Sinne sondern Zuweisung: Ausdruck *rechts* wird ausgewertet und (resultierender) Variable *links* zugewiesen

```
b = b + 1
```

Ausführbare Anweisung (=statement):

ausführbares C++ Kommando, d.h. Ausdruck durch ';' abgeschlossen

```
3 + 6;  
x = sqrt(2.);  
cout << "Hello World" ;
```

Mehrere Ausdrücke/Zuweisungen in einem Statement möglich:

```
y = ( x = sqrt(a) ) > 0 ;
```

Kontroll-Anweisung (=control-statement):

Verzweigung, Schleife

```
if ( a > b ) ... while ... for ... switch ... break ... return  
...
```

später

1.4 C++ Operationen

Arithmetische Operationen und Zuweisungen :

C++	FORTTRAN	Zweck
$x++$		Postincrement
$++x$		Preincrement
$x--$		Postdecrement
$--x$		Predecrement
$-x$		Vorzeichen
$x + y$	$X + Y$	Addition
$x - y$	$X - Y$	Subtraktion
$x * y$	$X * Y$	Multiplikation
x / y	X / Y	Division
$x \% y$	$mod(X, Y)$	Modulo
$pow(x, y)$	$X ** Y$	Exponation
$x = y$	$X = Y$	Zuweisung
$x += y$	$X = X + Y$	Zuweisung mit Änderung
$(- =, * =, / =, \% =)$		

Nützliche Kurzformen:

- `x ++` äquivalent zu `x = x + 1`
- `x += 5` äquivalent zu `x = x + 5`
- `x *= 10` äquivalent zu `x = x * 10`

Subtiler Unterschied zwischen `x++` und `++x`:

Preincrement bedeutet, zuerst Variable um 1 erhöhen, dann verwenden.

Postincrement umgekehrt, zuerst verwenden, dann erhöhen.

Beispiel:

```
int n = 0;
cout << n++ << endl;
cout << n << endl;
cout << ++n << endl;
```

Aber Vorsicht, manchmal compiler-abhängige Seiteneffekte !

Mathematische Funktionen, wie `sin`, `cos`, `exp`, etc., sind in C++ nicht direkter Bestandteil der Sprache (im Gegensatz zu FORTRAN). Sie sind aber in jedem C/C++ Compiler enthalten; die Definitionen (prototypes) müssen mittels header files geladen werden, z.B.

```
#include <cmath>.
```

Logische Operationen und Vergleiche :

C++	FORTTRAN	Zweck
false oder 0	.FALSE.	falsch
true oder $\neq 0$.TRUE.	wahr
!x	.NOT.X	Negation
$x \& \& y$	X.AND.Y	AND Verknüpfung
$x y$	X.OR.Y	OR Verknüpfung
$x < y$	X.LT.Y	kleiner als
$x \leq y$	X.LE.Y	kleiner als oder gleich
$x > y$	X.GT.Y	größer als
$x \geq y$	X.GE.Y	größer als oder gleich
$x == y$	X.EQ.Y	gleich
$x \neq y$	X.NE.Y	ungleich

Vorrangregeln für Operatoren: $f = a * x * x + b * x + c$ funktioniert wie erwartet nach Mathematik (Punkt vor Strich ...). Weiteres im Prinzip eindeutig festgelegt, dennoch besser mit Klammern unmissverständlich klarmachen: $x = y > z ? (x=y) > z$ oder $x = (y > z)$

Bit Operationen:

C++	Zweck
$\sim x$	Complement
$x \& y$	bitwise AND
$x y$	bitwise OR
$x \wedge y$	bitwise XOR
$x \ll y$	left shift
$x \gg y$	right shift mit sign

```
cout << (2 | 1);    // = 3
```

```
cout << (2 & 1);    // = 0
```

```
cout << (2 & 3);    // = 2
```

```
cout << (2 << 1);   // = 4
```

Mehr Beispiele später in den Übungen.

Achtung: Viele Operatoren sind mehrfach belegt, d.h. je nach **context** haben sie unterschiedliche Bedeutung, wie hier


\ll left-shift oder Ausgabe. Auch bei anderen Mehrfach-Verwendung $*$, $\&$, \dots

1.5 Grundlegende Datentypen und Definition von Variablen

Datentypen (nur die wichtigsten):

C++	FORTTRAN	Bytes
char	CHARACTER*1	1
int	INTEGER	2/4
long		4/8
float	REAL*4	4
double	REAL*8	8
bool	LOGICAL	4

```
int i = 5;  
float y = 3.7;  
bool b = true;
```

i: 

y: 

Ganze Zahlen versus Fließkommazahlen: Computer sind Rechner ... sehr schnelle Rechner ... jedoch mit begrenztem Zahlenbereich und Rechengenauigkeit (jedenfalls mit den gängigen Compilern/Interpretern)

int i.a. 32-bit 2-komplement Form (Rechner-abhängig)

$5 \equiv 0000\dots0101 \equiv 00000005$

$-5 \equiv 1111\dots1011 \equiv \text{FFFFFFFB}$

decimal binär hex

Zahlenbereich $[-2^{31}, 2^{31} - 1] \approx \pm 2 \cdot 10^9$

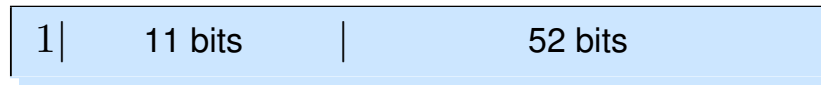
float 32-bit IEEE-Norm

1 | 8 bits | 23 bits

Sign Exponent Mantisse

Bereich: $\approx [-1.2 \cdot 10^{-38}, 3.4 \cdot 10^{+38}]$, ca. 7 Dezimalstellen

double 64-bit IEEE-Norm



Sign Exponent Mantisse

Bereich: $\approx [-2.2 \cdot 10^{-308}, 1.8 \cdot 10^{+308}]$, ca. 16 Dezimalstellen

Darstellung als Summe von 2er Potenzen:

$$(-1)^s \cdot (b_0 + b_1 2^{-1} + b_2 2^{-2} + \dots + b_{(n-1)} 2^{-(n-1)}) \cdot 2^{exponent}$$

inf und nan

Nicht immer führen Rechnungen zu einem gültigen Ergebnis, d.h. zu darstellbarer Zahl

```
double a = 0, b=1;
double c = b/a;
double d = sqrt(-1.);
cout << " a = " << a // ok
<< ", b = " << b // ok
<< ", c = " << c // inf
<< ", d = " << d // nan
...
// check mit
if ( isinf(c) ) { // Funktion deklariert in cmath
...

```

- **±inf** zeigt an, dass Ergebnis (\pm) unendlich ist, z.B. Division durch Null, $\log(0.)$, ...
- **nan** (Not A Number) zeigt an, dass Ergebnis nicht als reelle Zahl darstellbar ist, z.B. $\sqrt{-1.}$, $\log(-1.)$, ...

Variablen müssen *deklariert* sein bevor sie benutzt werden. Die Deklaration kann an jeder Stelle im Programm geschehen (C++).

```
double a = 0;
```

```
a = ...
```

```
double b;
```

```
b = 2 * a;
```

Variablen im Detail

Leicht Verwirrung möglich was genau mit *Variable* gemeint ist, präzise Verwendung der Begriffe wichtig ! Ein Ausdruck der Form

```
double x = 2.7;
```

x:

double 2.7

definiert eine Variable.

Diese Variable umfasst **vier** Dinge:

- **Identifier** (= Name) der Variable (x)
- **Typ** der Variable (double).
- **Wert** der Variable (2.7)
- **Adresse** der Variable, d.h. wo sie sich im Speicher befindet.

Mit Deklaration einer Variablen (double x) wird ein dem **Typ** entsprechender Speicherbereich reserviert, der **Identifier** erlaubt den Zugriff auf den **Wert** der an dieser Stelle gespeichert ist bzw. wird. Die **Adresse** der Variable ist in C++ über Pointer zugänglich ⇒ später

Konstanten und “Literals”

Man kann jederzeit im Programm Zahlenwerte oder Strings direkt hinschreiben, das sind die sog. *Literals*.

```
cout <<"9 * 9 = " <<9*9 <<endl;
```

Für simple Zwecke ok, aber bei komplexeren Grössen und/oder häufigem Gebrauch besser als *Konstante* definieren:

```
const double C_LIGHT = 2.99792458e8;  
const double M_PROTON = 1.6726215813e-27;
```

Das Keyword **const** bewirkt dass “Variable” bei Initialisierung festgelegt wird und später nicht mehr geändert werden kann.

Konvention: *Grossbuchstaben* für Konstanten verwenden !

Ansonsten gilt für Variablen-Namen (Identifier):

- 1. Zeichen Buchstabe
- Dann beliebig viele Buchstaben, Ziffern oder UnderScore (_), **case-sensitiv**
`MasseDesNeutrons`, `mass_of_neutron`, `X127612a8313`
- Variablen-Name soll möglichst Aufschluss über Inhalt/Verwendung geben

Liste der reservierten Wörter in C++

Dürfen nicht als Namen für Variablen, Funktionen, Klassen verwendet werden!

```
asm auto bitand bitor bool break case catch char class compl const  
const_cast continue default delete do double dynamic_cast else enum explicit export  
extern false float for friend goto if inline int long mutable namespace new not not_eq  
operator or or_eq private protected public register  
reinterpret_cast return short signed sizeof static static_cast struct switch template  
this throw true try typedef typeid typename union unsigned using virtual void volatile  
wchar_t while xor xor_eq  
define else endif if ifdef ifndef include undef
```

Hinzu kommen noch die Namen gängiger Klassen und Methoden wie

`printf cout sin exp ...`

Wird zwar syntaktisch von C++ akzeptiert, führt aber leicht ins Chaos ...

In **C** kein eigener Datentyp für Zeichenketten (character strings), sondern gebildet als **array** (s.u.) von `char` Variablen:

`char s[] = "Hallo" ⇒ 'H' | 'a' | 'l' | 'l' | 'o' | '\0'`

Länge = # Buchstaben + 1, `\0` wird angehängt zum Kennzeichnen des String-Endes.

Wichtig: Unterschied zwischen single und double quotes:

`'a'` = single character

`"a"` = character string mit 2 Elementen: `'a' | '\0'`

In In C++ mittlerweile `string` Datentyp (als Klasse) definiert

```
#include <string>

..

string s1("Hello "); // s1 deklariert und initialisiert
string s2; // s2 deklariert
s2 = s1 + "world "; // zusammenketten mit '+'
                // => s2 = "Hello world"

cout << s2 << endl; // Ausgabe ...
cout << "Name ? ";
cin >> s2;           // Einlesen von Tastatur
cout << s1 << s2 << endl;
```

- dynamisches Anlegen/Erweitern bei Verwendung des Strings
- “Intuitive” Operationen `=`, `+`, `+=`, `[]` definiert
- I/O direkt in strings mit `>>`, `<<`

In C/C++ implizite Typumwandlung von int nach float/double und float nach double
aber Vorsicht, im Zweifelsfall besser explizit (**casts**):

```
int i = 3, k = 1;  
double a = i;  
double b = k/i; // b = 0  
double c = double(k)/double(i); // cast
```

Zuweisung meist ok, aber Vorsicht bei Rechen-Operationen:

int und float Arithmetik **völlig** verschieden !

C/C++ bietet darüberhinaus noch komplexere Datentypen an; die **structures**, die aus beliebigen anderen Typen zusammengesetzt sind:

```
struct person {  
    char *name;  
    int   age;  
    ...  
}  
struct person gd;  
gd.name = "Guenter Duckeck";  
gd.age  = 0x29;
```

Diese Funktionalität erlauben auch Klassen, daher selten verwendet in C++ (⇒ später).

typedef

damit kann ein Alias zu einem Standard-Datentyp oder einem selbstdefinierten Typ gesetzt werden:

```
typedef int Length;
```

```
....
```

```
Length len, maxlen;
```

Kann Verständlichkeit des Variablengebrauchs erleichtern, oft aber auch komplex auf echten Datentyp zurückzuschliessen.

Häufig benutzt in C/C++

1.6 Control-statements

”Eigentliches Programmieren” braucht Kontrollstrukturen:

Bedingungen testen und ggf. ausführen,

Zähl-Schleifen, Endlosschleifen, etc.

if-else:

```
if ( expr ) { statements }
```

- true: statements werden ausgeführt
- false: statements werden übersprungen

```
if ( expr ) { statements_1 } else { statements_2 }
```

- true: statements_1 werden ausgeführt
- false: statements_2 werden ausgeführt
- statements_2 kann ein weiterer **if-block** sein, nur **if** und **else** sind keywords.

```
...
if ( a < b ) {    // einfaches if, auch ohne {}
    a = b;         // wenn nur ein Befehl folgt
}                 // (gefaehrlich!)
...
if ( a < b ) {
    a = b;
}
else {
    b = a;
}
...
if ( a < b ) {
    a = b;
}
else if ( a > b ) {
    b = a;
}
else if ( a == b ) {
    ...
}
```

```
    }  
    else { // wenn alles andre fehlschlaegt ..  
cout << "What else could I test ?" << endl;  
    }
```

for-loops:

```
for ( expr1; expr2; expr3 ) { statements }
```

- `expr1` wird einmal am Anfang ausgeführt
- `expr2` wird jedesmal ausgeführt
 - `true`: `statements` werden ausgeführt, danach `expr3`
 - `false`: Schleife ist beendet

```
for ( int i = 0; i < MAX; i++ ) { // Standard
    a[i] = i*i;                  // Abzaehlschleife
    sum += i;
}

for ( int i = 0; i < N; i++ ) { // Verschachtelte Schleifen
    for ( int j = i; j < M; j++ ) { //
        a[i][j] = i*j;
    }
}

char *text = "Donaudampfschiffahrtsgesellschafts"
            "kapitaenskajuetenblumenvase";

int count = 0;
for (char *p = text; *p; p++ ) {
    count ++;
}

cout << count << " Buchstaben in " << text << endl;
```

while-loops:

```
while ( expr2 ) { statements }
```

analog zu `for (; expr;)`

`statements` werden ausgeführt wenn `expr2` true.

```
do { statements } while ( expr2 ) ;
```

Unterschied: `statements` werden mindestens einmal ausgeführt.

...

```
string buffer, line;
```

```
while ( cin >> line )    { // read from stdin until EOF
    buffer += line;      // append to buffer
}
```

....

```
int count = 0;
```

```
do { // read from stdin until EOF
    count ++; // zaehlen
} while ( cin >> line );
```


break/continue :

Innerhalb des statements Blocks von `for` bzw. `while` Schleifen kann man mit

- `break` die Schleife beenden oder mit
- `continue` zur nächsten Iteration springen.

```
for ( ; true; ) { // Endlosschleife
    ...
    if ( ... ) continue;    // Springt ans Ende der Schleife
    if ( ... ) break;       // Springt aus der Schleife raus
    ...
} // end of loop
```

In C/C++ gibt es auch noch die `goto` Anweisung, mit der man an beliebige Stellen im Programm springen kann.

Verwendung möglichst vermeiden (\Rightarrow *Spaghetti-Code*)

switch/case :

Variante von `if/else` ist die `switch/case` Kontrollstruktur:

```
char response;
cin >> response;
switch(response) {
    case 'a' : cout << "you choose 'a'" << endl;
               break;
    case 'b' : cout << "you choose 'b'" << endl;
               break;
    case 'c' : cout << "you choose 'c'" << endl;
               break;
    case 'x' :
    case 'q' : cout << "exiting/quitting menu" << endl;
               quit = true;
               break;
    default  : cout << "Please use a,b,c or q!"
               << endl;
}
```

`break` nötig um `switch` Block zu beenden, ansonsten werden Anweisungen nach weiteren `case` ausgeführt.

`default`: wird immer ausgeführt (soweit vorhanden und nicht vorher mit `break` beendet).

1.7 Arrays, Pointer und Referenzen

Arrays:

Arrays sind Kollektionen von Objekten derselben Art, die aufeinanderfolgend im Speicher angelegt werden.

Deklaration `Type name[Laenge]`

Zugriff auf einzelne Elemente kann direkt mittels des Positions-Index zugegriffen werden: `name[index]`

Vorsicht: C++ zählt von 0 bis N-1: *1. Element = Index '0'*

```
int n[20]; // Deklaration von int array mit 20 Elementen
```

```
// direkter Zugriff auf einzelne Elemente mittels [index]
```

```
// aber Vorsicht, C/C++ zaehlt von 0 bis N-1
```

```
n[0] = ...; //1. Element
```

```
...
```

```
n[19] = ...; // Letztes Element
```

```
// Auch direkte Initialisierung moeglich,
```

```
// dann keine Dimensionierung noetig
```

```
double x[] = { 2.7, 8.1, 9.0, 27.4, 1.45e8, -0.886e-50 };
```

```
//
```

```
// Ausgabe 1. und letztes Element
```

```
cout << x[0] << " " << x[5] << endl;
```

*In modernem C++ wird weitgehend auf die Verwendung von Standard Arrays verzichtet und stattdessen der **STL Vector** (`vector<Type>`) verwendet. Dieser ist wesentlich flexibler in Bezug auf dynamisches Erzeugen, Erweitern, Initialisieren, etc. Siehe *letztes Kapitel*.*

Pointer:

Ein Pointer ist eine Variable, die eine Speicheradresse enthält.

Deklaration `Type * name`, d.h. auch hier Typ-Angabe erforderlich, Unterschied zu *normaler* Variablen-Deklaration ist `*` vor dem Identifier.

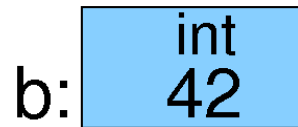
Zuweisung `name = & var`, Address-Zuweisung mittels Adress Operator `&`.

Verwendung (1) Wert (=Inhalt der Pointer Variablen) ist Adresse

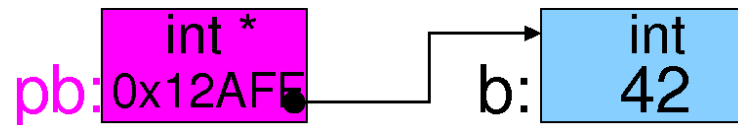
Verwendung (2) **De-Referenzieren**, d.h. Auslesen des Wertes an der Adresse mittels `*` vor Namen.

```
int b = 42;
int *pb;    // pb deklariert als pointer auf int
pb = &b;    // & ist Adress operator, liefert Adresse von b
cout << b << endl;
cout << pb << endl; // Kryptische Adresse
cout << *pb << endl; // De-referenzieren: *pb == b
double x = 7.256;
double *px = &x; // px deklariert als pointer auf double
px = &b; // illegal pointer auf double kann nicht Adresse von int nehmen
```

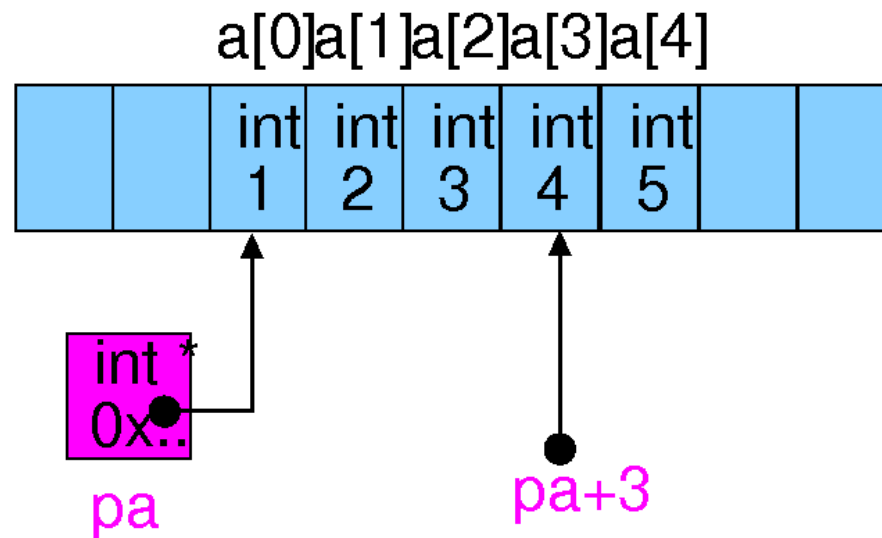
Einfache Variable Name ist "Label" mit dem **Inhalt** der Variablen angesprochen wird:



Pointer Auch Variable, aber Inhalt ist **Adresse** und **Typ** einer anderen Variablen:



... oder ein **Array**:



Pointer-Arithmetik

Für pointer ist Addition und Subtraktion definiert

- `pa+3` zeigt auf Speicherstelle `3*int` weiter oben als `pa` ($= 3 \cdot 4 = 12$ Bytes).
- `px - 5` zeigt auf Speicherstelle `5*double` weiter unten als `px` ($= 5 \cdot 8 = 40$ Bytes).
- Analog decrement/increment: Nach `pa--; ... px++;`
zeigt `pa` auf die um 4 Byte kleiner Adresse und `px` auf die um 8 Byte höhere Adresse.

Pointer und Arrays sind eng verknüpft, die Syntax kann man wechselweise verwenden:

```
int a[5] = { 1, 2, 3, 4, 5}; // direct array initialisation !
int *pa = a;
cout << pa[2] << endl;    // array-syntax for pointer
cout << *(a+2) << endl;   // pointer-syntax for arrays
pa += 2;
cout << *pa << endl;     // .. 3
a += 2;                  // illegal, no pointer arithmetics for arrays
```

Vorsicht, bei Definition eines Pointers wird nur der Pointer selbst angelegt, nicht das Objekt worauf er zeigt:

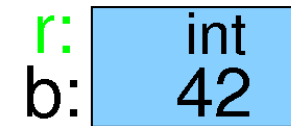
```
int c = 55;
int d = 66;
int *pc;
*pc = d;    // illegal, pointer zeigt ins jenseits: → core dump
pc = &d;    // ok, nun zeigt er auf d
*pc = c;    // ok, Wert bei d wird ueberschrieben
```

Nicht-initialisierte Pointer gefährlich:

- *core dump* bei Zugriff auf illegalen Bereich in Unix ⇒ Programm-Abbruch aber klare Diagnose.
- Überschreiben von anderen Datenbereichen des eigenen Programms ⇒ schwer zu findende, nicht reproduzierbare Laufzeit-Fehler.
- Überschreiben von Datenbereichen anderer Programme bzw. des Betriebssystems. Nicht möglich unter Unix/Linux und Windows NT/XP, aber grosses Problem für Windows 95/98/...

Referenzen

Referenzen (**neu in C++**) sind Aliase – ein anderer Name für dasselbe Objekt:



```
int b = 42;
int &r = b;
int a[100];
int & start_of_a = a[0];
int & end_of_a = a[99];
a[0] = 9;
start_of_a = 9;           // same effect
// mit pointern
int * start = a;
int * end    = a+99;
*start = 9;
```

Referenzen **müssen** initialisiert werden, d.h. bei Deklaration muss Zuweisung einer anderen Variablen erfolgen. Die Referenz verweist immer auf diese Variable. Referenzen v.a. nützlich bei Funktionsaufruf und -rückgabe (s.u.).

Vorsicht: Verwendung von *, & bei Deklaration bzw Zuweisung:

```
int arr[] = {1,2,4,8} ;  
int & ref = arr[2];  
int * pa;  
int * pb = arr; // Pointer wird gesetzt  
pa = arr; // Pointer wird gesetzt  
pa = & arr[2]; // Adress-Operator  
*pa = 99; // Wert wird gesetzt: arr[2] = 99
```

Bei Variablen-Deklaration:

- **Type &x** \Rightarrow Referenz
- **Type *x** \Rightarrow Pointer

Bei sonstiger Verwendung:

- **&x** \Rightarrow Speicheradresse der Variable,
- ***x** \Rightarrow De-Referenzierung (=Wert an der Adresse wird benutzt)

Matrizen, höherdimensionale Arrays

Arrays gibt es auch mit mehr Dimensionen:

- Matrix mit 3 x 4 `int` Elementen: `int A[3][4];`
- Zugriff via `A[0][0]` (1. Element) bis `A[2][3]` (letztes Element).
- Anlegen auch mit direkter Initialisierung

```
int a[3][4] =      { { 1, 0, 12, -1 }, { 7, -3, 2, 5 }, { -5, -2, 2, 9 } };  
for ( int i = 0; i < 3; i++ ) { // Verschachtelte Schleifen  
    for ( int j = 0; j < 4; j++ ) { //  
        cout << setw(6) << i << setw(6) << j << setw(6) << a[i][j] << endl;  
    }  
}
```

Analog Erweiterung auf höhere Dimensionen > 2:

```
int B[5][2][3][4]; ( 5*2*3*4 = 120 Elemente)
```

Pointer – Wozu?

Pointer sind für eine Reihe von Aufgaben unverzichtbar:

- Direktes Ansprechen von Ein-/Ausgabe-Ports oder sonstigen Peripherie-Geräten.
- Arbeiten mit dynamisch angelegtem Speicher oder Datenstrukturen (⇒ später)
- Verknüpfen von Datenstrukturen.

Allerdings ist die Verwendung heikel, kleine Programmier-Fehler können zu bizarren, schwer zu findenden Laufzeit-Fehlern führen.

In C werden Pointer sehr häufig verwendet für Arbeiten mit *character strings* (*char **), für Parameterübergabe bei Funktionsaufrufen (⇒ später), u.v.m.

In C++ deutlich reduziert, an vielen Stellen kann man mit *Referenzen* arbeiten, STL container anstelle dynamischer Arrays verwenden, ...

1.8 Funktionen

Die klassische Art (*modulares Programmieren*) ein großes, komplexes Programm überschaubar zu gestalten ist die Aufteilung der Funktionalität in kleinere, eigenständige Untereinheiten für bestimmte Aufgaben, d.h. Einteilung in **Funktionen**.

Einfaches Beispiel – Quadrat- und Kubikfunktion:

1 Argument wird an Funktionen `quadrat(..)`, `kubik(..)` übergeben, Ergebnis (`double` Wert) wird zurückgegeben.

```
double quadrat( double x ) // Berechne Quadrat
{
    double r = x*x;
    return r;
}
double kubik( double x ) // Berechne Kubik
{
    double r = x * quadrat(x); // verwende quadrat funktion
    return r;
}
int main()
```

```
{  
  double a1 = 2.5;  
  double a2 = quadrat(a1); // verwende Funktion  
  double a3 = kubik(a1);  
  // Argument kann auch komplexer Ausdruck sein ...  
  double b = quadrat( kubik(a3) * sin(0.8) + 0.99 );  
}
```

Weiteres Beispiel – Gravitationskraft:

3 Argumente werden an Funktion übergeben, Ergebnis (double Wert) wird zurückgegeben.

```
double GravForce( double mass1, double mass2, double distance) //      Kopf der Funktion
{
    const double GRAV_CONST = 6.673 e-11; // Gravitationskonstante       $m^3 / (kg \ s^2)$ 
    double f = GRAV_CONST * mass1 * mass2 / (distance * distance);
    return(f);
}
int main()
{
    double mSonne = 1.9889e30, mErde = 5.974e24, dErdeSonne = 1.49597e11, rErde = 6.378e6;
    // Aufruf von Funktion GravForce
    double gErdeSonne = GravForce( mSonne, mErde, dErdeSonne ); // Kraft Sonne-Erde
    double gPerson = GravForce( mErde, 80., rErde ); // Gewichtskraft 80 kg auf Erde
    ...
}
```

Syntax von Funktionsaufrufen:

```
Type name( Type arg1, ...) { statements }
```

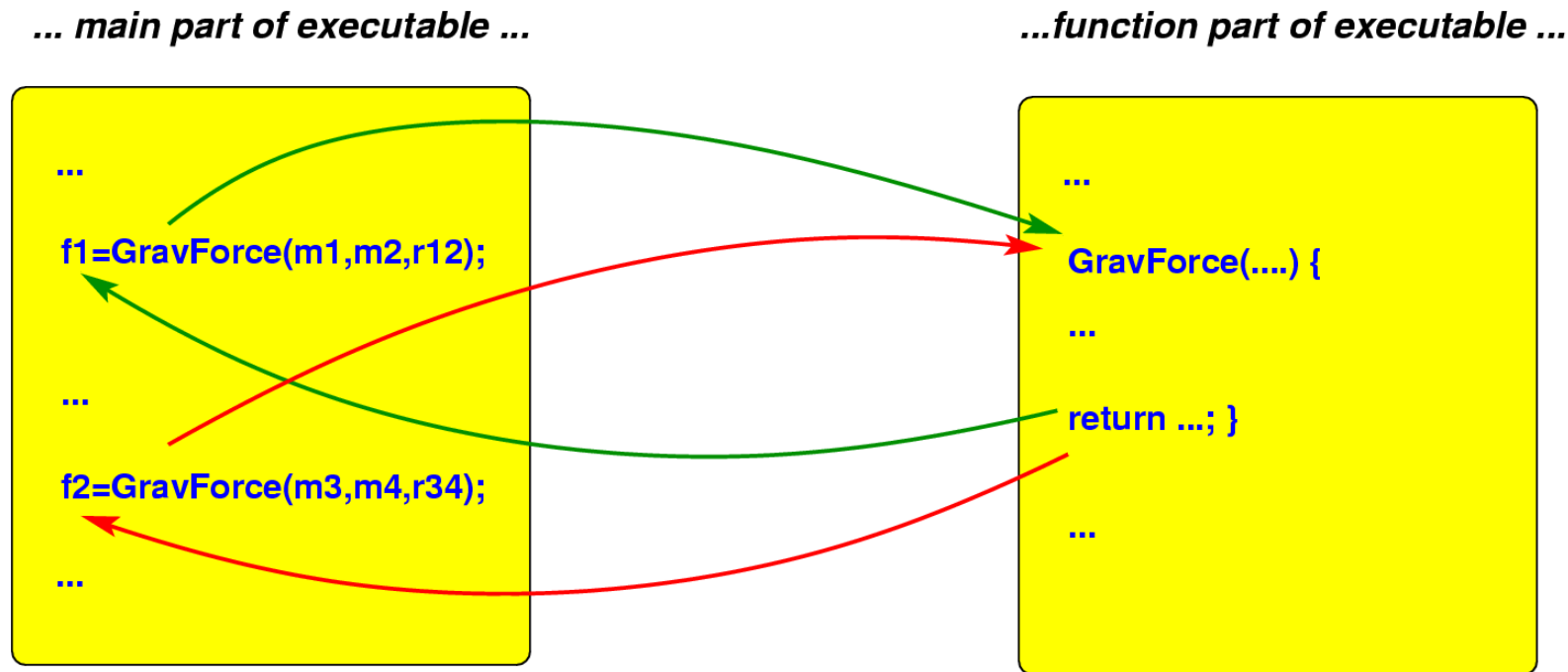
- `Type` return Typ der Funktion, das kann jeder in C++ definierte Datentyp sein.
Kann auch `void` sein (=leer), wenn nix zurueckkommt.
- `name` Name der Funktion
- `arg` Der Funktion können beim Aufruf Argumente übergeben werden.
- Funktionsaufruf bewirkt:
 - Bei Programmlauf wird zu der entsprechenden Funktion gesprungen
 - Die *formalen* Argumente (`mass1, mass2, ...`) erhalten die beim Aufruf übergebenen *aktuellen* Werte der Argumente (`mSonne, mErde, ...`), d.h.

```
double mass1 = mSonne, double mass2 = mErde, double distance = dErdeSonne
```

bzw.

```
double mass1 = mErde, double mass2 = 80., double distance = rErde
```
 - Am Ende der Funktion wird an die rufende Stelle zurückgesprungen und ggf. der Wert zurückgegeben:

```
gPerson = 783.;
```

Programm-Design

- Aus Sicht der Anwendungsprogrammiererin ist die Funktion eine *Black Box*.
- Es interessiert nur, was die Funktion leisten soll und wie die *Schnittstelle* aussieht, d.h. Typ der Funktion, Typ und Zahl der Argumente.
- Details der *Implementierung* sind irrelevant.

Funktionen brauchen keinen Typ und auch keine Argumente ...

```
void newline( ) // Funktion gibt nur neue Zeile aus
{
    // keine Parameter-uebergabe
    cout << endl; // keine Rueckgabe
}
int main()
{
    newline();
}
```

Argumentübergabe bei Funktionsaufruf:

- Bei direkter Übergabe wird eine lokale Kopie gemacht, Änderungen in der Funktion haben keine Auswirkung im rufenden Programm (call by value)

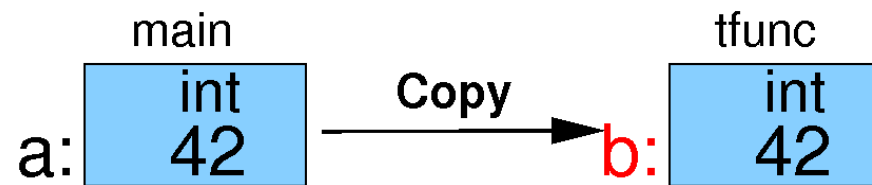
```
int a = 42;
```

```
tfunc(a);
```

```
...
```

```
void tfunc(int b)
```

```
{...}
```



- Für 'bleibende' Änderungen entweder Adressen-Pointer Übergabe

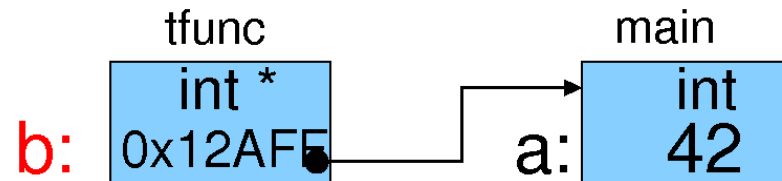
```
int a = 42;
```

```
tfunc(& a);
```

```
...
```

```
void tfunc(int *b)
```

```
{...}
```



- oder Deklaration als Referenz (C++) in der Funktion

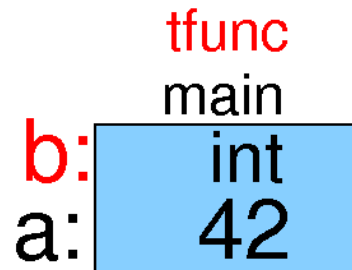
```
int a = 42;
```

```
tfunc(a);
```

```
...
```

```
void tfunc(int & b)
```

```
{...}
```



Hinweis: In fast allen Fällen würde ich letzteres Verfahren empfehlen, d.h. Deklaration der Variablen in Funktion als Referenz: \Rightarrow keine komplexe Pointer-Syntax, gute Performance. Gab's aber noch nicht in C, d.h. alte Programme nur *call-by-value* oder *call-by-address*.

```
#include <typeinfo>
#include <iostream>
#include <cmath>
using namespace std;
void tfunc1( double x);
void tfunc2( double *x);
void tfunc3( double & x);
int main()
{
    double a = 3.14;
    cout << "main start " << a << endl;
    tfunc1( a );
    cout << "main tfunc1 " << a << endl;
    tfunc2( &a );
    cout << "main tfunc2 " << a << endl;
    tfunc3( a );
    cout << "main tfunc3 " << a << endl;
}

void tfunc1( double x )
```

```
{ // x ist Kopie von a
  x *=2;
  cout << "tfunc1 " << x << endl;
}
void tfunc2( double * x )
{ // x ist pointer auf a
  *x *= 2;
  cout << "tfunc2 " << *x << endl;
}
void tfunc3( double & x )
{ // x ist identisch a
  x *= 2;
  cout << "tfunc3 " << x << endl;
}
```

“Strong Typing” und Funktionendeklaration

C++ ist eine sog. strong-typing Sprache, d.h.

- alle *Variablen* müssen vor Benutzung deklariert sein.
- und genauso müssen alle *Funktionen* mit zugehörigen Argument-Typen vor Verwendung bekannt sein.

Ansonsten bekommt man Fehlermeldungen vom Compiler.

Um Funktionen “bekanntzumachen” gibt es drei Möglichkeiten:

- Komplette Funktion steht im selben File und zwar **vor** erster Verwendung !
- Deklaration der Funktion (*=Prototype*) ist vorher angegeben:

```
double GravForce( double m1, double m2, double d);
```

Implementierung (= *Definition*) der Funktion, kann an anderer Stelle, auch in externem File, erfolgen.

- Oder äquivalent: Deklarationen werden in den header Files gesammelt und mittels `#include "MyFile.h"` eingebunden \Rightarrow Standardverfahren in C++:

Deklaration von Funktionen und Klassen in Header-File,

Implementation in separatem Source-File.

Weiteres zu Funktionen

- **Signatur:** Funktion wird erst eindeutig mit der entsprechenden Argumentliste, C++ sucht abhängig vom Aufruf die passende.
- C++ sehr flexibel:
 - Mehrere Funktionen gleichen Namens möglich.
 - Unterscheidung durch **Signatur**; abhängig von Argumenten wird passende Funktion gesucht.
 - Default Werte für Argumente im Prototyp, dann Aufruf ohne Argumente möglich.
 - Falscher Argument-typ wird entsprechend umgewandelt, soweit möglich.

```

// demo function overloading
// Deklaration Funktions Prototypen
void tfunc1( double x = 999); // deklariert tfunc1() und tfunc1( double )
void tfunc2( int x = -1); // deklariert tfunc2() und tfunc2( int )
void tfunc2( double x);
// void tfunc2( float x = 0.1); // illegal, tfunc2() schon deklariert
int main()
{
    int a = 5;
    int *b;
    char *c = "ABC";
    b = &a;
    tfunc1( 1. );
    tfunc1( 1 ); // int wird umgewandelt
    tfunc1( a );
    tfunc1( ); // default wird genommen
    tfunc1( b ); // illegal, mit pointern geht die Umwandlung nicht
    tfunc1( *c ); // Seltsam, aber char sind einfach Zahlen == tfunc(65)
    tfunc2( 1 ); // int version wird benutzt
    tfunc2( 1. ); // double version ""
    tfunc2( ); // default darf nur einmal deklariert sein,
                // in dem Fall also die int version.
}
// Definition/Implementierung der Funktionen
void tfunc1( double x )
{
    cout << "tfunc1 " << x << endl;
}

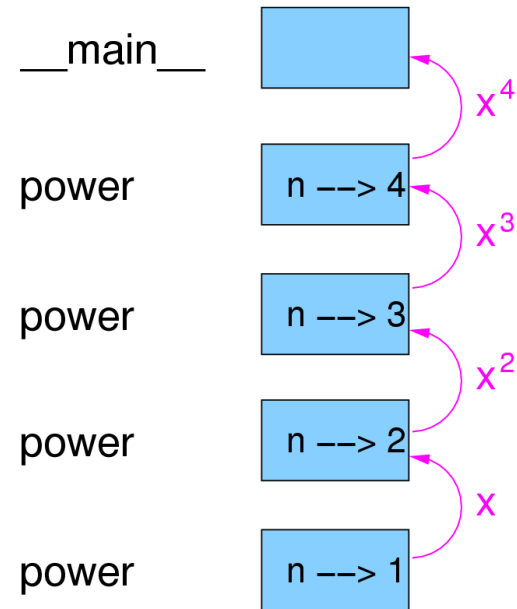
```

```
}  
void tfunc2( int x )  
{  
    cout << "tfunc2 int " << x << endl;  
}  
void tfunc2( double x )  
{  
    cout << "tfunc2 double " << x << endl;  
}
```

Rekursive Funktionsaufrufe (Funktion ruft sich selbst) sind möglich:

```
double power( double x, int n )
{
    if ( n > 1 ) {
        return( x * power(x, n-1) ); // rekursiver Aufruf
    }
    else {
        return( x );
    }
}
```

```
double y = power( x, 4 );
```



1.9 Gültigkeitsbereich von Variablen

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];
```

Bei Arrays muss Grösse beim **Erzeugen** bekannt sein, d.h. folgendes ok:

```
const int n = 200; // n festgelegt beim Compilen  
double arr[n]; // ok
```

Aber das ist problematisch:

```
cin >>n; // Eingabe fuer n zur Laufzeit  
double arr[n]; //
```

Früher Compiler-Fehler, möglich in aktuellen C++ Compilern, allerdings Grösse durch System-Limits eingeschränkt.

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 Programmiers.

```
cin >>n; // Eingabe fuer n zur Laufzeit
double * arr = new double[n];
```

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

Nach Gebrauch wieder zurückgeben mit

```
delete [] arr;
```

Ansonsten **Memory Leaks !**

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

Das Hauptproblem bei grossen 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;  
}
```

1.10 Pointer und Referenzen, cont'd

Ein paar pointer Spielereien:

```
#include <iostream>
int main ()
{
    int array[] = { 0, 10, 20, 30, 40, 50 };
    int val, *p;
    p = array;
    val = *p;
    val = *p++;
    val = *++p;
    val = (*p)++;
    val = *(p++);
    val = ++*p;
    val = ++(*p);
    // welchen Wert hat val jeweils ?
    // wohin zeigt p ?
    // wie sieht array[] am Ende aus ?
    return(0);
}
```

Zu guter letzt: **Function pointers**

- Gängig in C
- Ermöglicht Verwendung von generischen Standard-Algorithmen, z.B. **Sortier-Algorithmus** derselbe für Integer, Double, Strings, etc.
- Typische Probleme in der Numerik: **Nullstellen**, **numerische Integration**, **Minimierung**, ...
- In **C++** selten verwendet, meist kann es wesentlich eleganter über **Klassen** und **Vererbung** erreicht werden (⇒ später)

```
#include <iostream>
#include <cstdlib>
// qsort: generelle Sortierfunktion,
//      Teil der C Standard-library
// qsort Deklaration:
//  #include <cstdlib>
//  void qsort(void* base, size_t nel, size_t wid, int (*compa)(const void *, const void *));
//  Benutzt pointer to function int (*compa),
//  muss vom User bereitgestellt werden
struct person {
    char name[100];
    char prename[100];
    int age; };
int test_age( const void *v1, const void *v2 );
int test_name( const void *v1, const void *v2 );
int main()
{
```

```

const int nstruct = 4;
struct person pa[nstruct];
strcpy(pa[0].name, "Boutemeur");
strcpy(pa[0].prename, "Madjid");
pa[0].age = 0x27;
strcpy(pa[1].name, "Duck");
strcpy(pa[1].prename, "Donald");
pa[1].age = 88;
strcpy(pa[2].name, "Kohl");
strcpy(pa[2].prename, "Helmut");
pa[2].age = 72;
strcpy(pa[3].name, "Duckeck");
strcpy(pa[3].prename, "Guenther");
pa[3].age = 0x28;
// sort by age: call qsort with function test_age
qsort( pa, nstruct, sizeof( struct person), test_age );
for ( int i = 0; i < nstruct; i++ ) {
    cout << i << " " << pa[i].name << endl;
}
// sort by name: call qsort with function test_name
qsort( pa, nstruct, sizeof( struct person), test_name );
for ( int i = 0; i < nstruct; i++ ) {
    cout << i << " " << pa[i].name << endl;
}
return(0);
}
int test_age(const void *v1, const void *v2 )    // compare age

```

```
{
    struct person *p1 = ( struct person * ) (v1); // cast arguments
    struct person *p2 = ( struct person * ) (v2);
    return( p1->age - p2->age);
}

int test_name(const void *v1, const void *v2 ) // compare name string
{
    struct person *p1 = ( struct person * ) (v1); // cast arguments
    struct person *p2 = ( struct person * ) (v2);
    int iret = 0, i = 0;
    while ( p1->name[i] != '\0' && p2->name[i] != '\0' && ( iret = p1->name[i] - p2->name[i] ) == 0 ) {
        i++;
    }
    return( iret );
}
```

1.11 I/O Basics

In C `printf(...)` bzw. `scanf(...)` Funktion für Aus- bzw. Eingabe. Praktisch für einfache, formatierte Ausgabe.

In C++ `cin >>` bzw. `cout <<` für Standard-Eingabe (Tastatur) bzw. Standard-Ausgabe (Bild-Schirm)

```
#include <iostream> // C++ std I/O functions
using namespace std; // declare namespace
int main()
{
    double a, b, c;
    cin >> a >> b >> c; // input: 3 doubles werden gelesen
    // output
    cout << a << ", " << b << ", " << c << endl; // Expliziter Zeilenvorschub mit endl
}
```

Liste von Eingabewerte lesen geht einfach mittels `while (cin >>...)` Schleife. Schleife wird solange durchlaufen bis Eingabe zu Ende ist oder Umwandlung nicht klappt.

// Programm readdouble

#include <iostream> *// C++ std I/O functions*

using namespace std; *// declare namespace*

int main()

{

double x, sum = 0.;

int n = 0;

// Lese Zahlen von Standard-input bis abgebrochen wird (-> cin >> x ergibt false)

while (cin >> x) {

 sum += x; *// summiere Werte*

 n++; *// zaehle Werte*

 }

}

Zwei Möglichkeiten zur Ein/Ausgabe aus/in Dateien:

- Umleiten von Std-In bzw. Std-Out beim Ausführen des Programms in Linux:

`./readdouble < eingabe.dat` Liest aus Datei "eingabe.dat" statt von Tastatur

`./readdouble < eingabe.dat > ergebnis.dat` ... Resultat wird jetzt in Datei "ergebnis.dat" geschrieben

- Oder man öffnet die Files direkt im C++ Programm mit den `ifstream`, `ofstream` Klassen.

```
#include <iostream> // C++ std I/O functions
#include <fstream> // C++ file I/O functions
using namespace std; // declare namespace
int main()
{
    ifstream inf("eingabe.dat"); // File fuer Eingabe oeffnen
    ofstream outf("ergebnis.dat"); // File fuer Ausgabe oeffnen
    double x, sum = 0.;
    int n = 0;
    // Lese Zahlen aus Datei bis Datei-ende wird (-> inf >> x ergibt false)
    while ( inf >> x ) { // statt cin jetzt inf
        sum += x; // summiere Werte
```

```
n++;           // zaehle Werte
outf << x << ", " << sum << ", " << n << endl; // Ausgabe in Datei, outf statt cout
}
}
```

Zur *Formatierung* gibt es in C++ die **I/O Manipulatoren**, damit kann Breite der Felder, Darstellungsart, Ausrichtung, u.v.a angegeben werden.

Die häufigsten Manipulatoren:

Manipulator	Wirkung	Betrifft
dec	Dezimaldarstellung	ganze Zahlen
hex	Hexadezimaldarstellung	dto
oct	Oktalдарstellung	dto
scientific	exponentialdarst. 1.2345e2	Gleitkomma
fixed	ohne exponent 123.45	dto
setprecision(int n)	n Nachkommastellen	dto
setw(int n)	minimale Ausgabebreite	alle
left	linksbündig	dto
right	rechtsbündig	dto

Beispiel: Fahrenheit–Celsius Tabelle formatiert ausgeben:

```
// print Fahrenheit->Celsius conversion table
#include <iostream>
#include <iomanip>
using namespace std; // declare namespace
int main()
{
    int lower(0), upper(300), step = 20;
    double fahr = 0., celsius;
    while ( fahr <= upper ) {
        celsius = (5.0/9.0) * (fahr-32.0);           // rechnen ...
        //      cout << fahr << " " << celsius << endl; // ausgeben ...
        cout << fixed << setw(6) << setprecision(0) << fahr // formatiert ausgeben
            << setw(10) << setprecision(3) << celsius << endl;
        fahr += step;
    } // end-while
} // end-main
/* Ausgabe:
    0   -17.778
   20   -6.667
   40    4.444
   ...
*/
```

1.12 Sonstiges

Namespaces

Ein nützliches Konzept in C++ sind die **namespaces** (= Namensräume). Damit lassen sich elegant Namenskonflikte vermeiden, d.h. die mehrmalige Verwendung von gleichen Namen für Variablen oder Funktionen in verschiedenen Programmbibliotheken, die im gleichen Programmbereich benutzt werden.

```
namespace PhysConst
{
    const double c          = 2.99792458e8;
    const double hbar       = 6.626068e-34;
    const double e          = 1.602176462e-19;
    ...
};

namespace MathConst
{
    const double pi         = 3.14159265358979;
    const double e          = 2.71828182845905;
    ...
};
```

...

// Verwendung

grad = rad * 180./MathConst::pi;

force = PhysConst::e * PhysConst::e / (r * r)

In älteren C++ Versionen waren die Bestandteile der sog. C++-Standard-Bibliothek (**std**) der voreingestellte (=default) Namespace, damit konnten direkt alle Elemente der std-Bibliothek benutzt werden, z.B. `cout`, `cin`, `sin()`, `sqrt()`,

Bei aktuellen C++ Versionen bzw. Compilern leider nicht mehr der Fall, der **namespace** muss angegeben werden !
Dazu i.W. 2 Möglichkeiten:

- Explizit durch Angabe des Namespaces bei jeder Verwendung, z.B.

```
std::cout << "Ausgabe " << std::endl; std::sqrt(3.); ...
```

- Generell namespace verfügbar machen mit **using** Direktive

```
using namespace std;
```


const pointers

`const` kann sich auf

- Wert wo Pointer hinzeigt beziehen: `const double *p` oder `double const *p`
- auf pointer selbst: `double * const p`
- oder beides

// Beispiele fuer const pointers

```
const double pi = 3.1415;
```

```
double pdq = 1.2345;
```

```
const double *p = &pi;      // ok
```

```
double * const d = &pi;     // falsch
```

```
double * const e = &pdq;    // ok
```

```
const double * const f = &pi; // ok
```

```
*p = 2.7; // Fehler: p zeigt auf const, darf nicht ueberschrieben werden
```

```
p = &pdq; // ok: p selbst ist nicht const, kann also geaendert werden
```

```
*p = 2.7; // Fehler: p als Zeiger auf const deklariert
```

```
*e = 2.7; // ok
```

```
e = &pdq; // falsch
```

```
*f = 2.7; // falsch
```

```
f = &pdq; // falsch
```

Korrekte `const` Verwendung komplex, hilft aber schwer zu findende Fehler zu vermeiden!

Übergabe von Argumenten ans Programm

- Argumente können bei Programmaufruf an `main()` übergeben werden
`./readarg hallo wie geht es 1 2 345`
- `main(int argc, char** argv)` erhält Zahl der Parameter (`argc`) und pointer auf strings (`argv`)
mit den einzelnen Parametern: `argv[1]="hallo" ... argv[7]="345"`

```
#include <cstdlib>
#include <iostream>

double pow( double x, int n );

int main( int argc, char** argv )
{
    for ( int i = 0; i < argc; i++ ) {
        cout << "Argument " << i << " " << argv[i] << endl;
    }
}
```

C Präprozessor

- `#include <iostream>` “Holt” System-Header Datei (/usr/include/..)
- `#include "myhead.h"` “Holt” eigene Header Datei (aktuelles Verzeichnis)

- `#define MAX_LEN 100`
... `int a[MAX_LEN]` ...
`#define PI 3.1415`

Definition von Konstanten, gängig in C, in **C++** besser mit `const`

- `#define SQUARE(a) ((a)*(a))`
... `double x = SQUARE(7*2)` ...

Definition von Makros, gängig in C, in **C++** besser via function templates (später)

- `#ifdef LMU`
...
`#endif`

Bedingtes Kompilieren für Testen, Debuggen, system-spezifische Funktionen, ...

Kompilieren: `g++ -DLMU ...`

Was der C++ Compiler macht:

- Präprozessor 'included' angegeben Header-files, alle `#define` werden ersetzt
⇒ Ausgabe in temporäre Datei.
- Eigentlicher Compiler-Schritt, 'Maschinen'-code wird erzeugt
⇒ Objektfiles (.o-Endung)
- Linker: Systembibliotheken und ggf. weitere eigene Objektfiles oder Bibliotheken werden dazugebunden
⇒ ausführbares Programm, kann vom Betriebssystem gestartet werden.

1.13 Programme debuggen

Programmieren ist eine komplexe Angelegenheit. Nur die wenigsten schaffen es mehr als 10 Zeilen fehlerfrei zu schreiben. Erst recht nicht als Anfänger. Häufig meiste Zeit bei Programmentwicklung für Fehlersuche (=debuggen) und Korrektur.

Hilfreich Fehler zu klassifizieren:

- **Compiler– oder Syntax–Fehler:** Der Compiler kann ein Programm nur übersetzen wenn die Syntax des Programms korrekt ist, d.h. den C++ Regeln entspricht. Beispielsweise nicht deklarierte Variablen (`cannot resolve symbol ... variable ...`), fehlende Klammern, und vieles andere. Besonders für Anfänger häufigste Fehlerquelle und ziemlich lästig wegen kryptischer Fehlermeldungen des Compilers, dennoch einfachste Art von Fehler.

Ebenfalls zu dieser Kategorie zählen Fehler beim Linken oder Anlauf–Fehler beim Start wegen fehlender Bibliotheken.

Hier eine [Liste](#) gängiger Syntaxfehler mit näherer Beschreibung.

- **Laufzeit–Fehler** treten beim Ausführen des Programms auf (= run-time errors), z.B. Division durch Null, File existiert nicht, Speicherüberlauf, Zugriff auf nicht-existierende Array Elemente, etc. Oft führen sie zum Abbruch des Programms, aber gerade bei C++ häufig nicht an der Stelle, an der der eigentliche Fehler passiert, sondern erst viel später nach einer komplexen Sequenz von Folgefehlern. Nur manchmal offensichtlich, oft schwer und mühsam zu finden. Diszipliniertes Programmieren, z.B. Verwendung von Exceptions wichtig.

- **Semantik-Fehler** Programm kompiliert und läuft, tut aber nicht das was es soll. Mal einfach, kann aber auch Milliarden kosten (Ariane).

The First "Computer Bug"

Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.

Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP - MC ~~1.582642000~~ 2.130476415 (033) PRO 2 2.130476415
 (033) PRO 2 2.130476415
 connect 2.130676415

Relays 6-2 in 033 failed special speed test
 in relay " 11.00 test "

Relay
 2145
 Relay 3376

1100 Relays changed
 Started Cosine Tape (Sine check)
 1525 Started Multi-Adder Test.

1545



Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.
 1630 Antan started.
 1700 closed down.

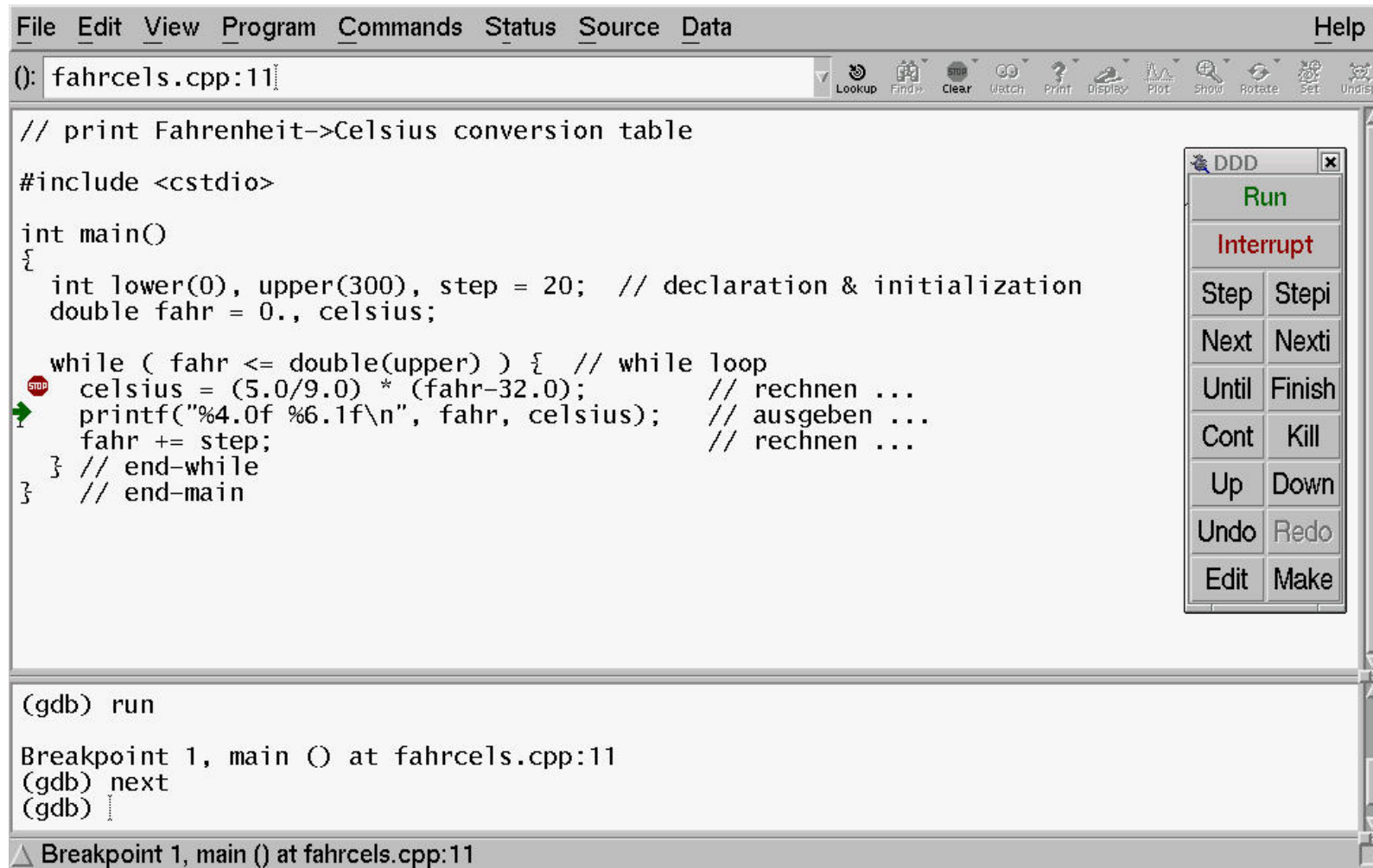
Meiste Zeit bei Programmentwicklung i.d.R. für (Laufzeit-/Semantik-) Fehlersuche.

Standardverfahren sind `printf`, `cout` statements an den kritischen Stellen

⇒ umständlich, zeitraubend

Viel eleganter mit *debugger*

- Programm läuft unter Kontrolle des Debuggers
- Zeile für Zeile dem Source-code nach
- oder *breakpoints* an den kritischen Stellen setzen und direkt dahin laufen.
- Inhalt von Variablen kann jederzeit angezeigt werden.
- Dazu: kompilieren mit `-g` option: `g++ -g -o fahr fahr.cpp`
Bewirkt dass zusätzliche Info in den kompilierten code geschrieben wird.
- debugger starten:
Entweder alte Kommandozeilen-version `gdb fahr`
Schöner die GUI Version: `ddd fahr`



The screenshot shows a GDB debugger window with the following components:

- Menu Bar:** File, Edit, View, Program, Commands, Status, Source, Data, Help.
- Toolbar:** Includes icons for Lookup, Find, Clear, Watch, Print, Display, Plot, Show, Rotate, Set, and Undis.
- Source Window:** Displays the file `fahrcels.cpp` at line 11. The code is as follows:

```
// print Fahrenheit->Celsius conversion table
#include <stdio>

int main()
{
    int lower(0), upper(300), step = 20; // declaration & initialization
    double fahr = 0., celsius;

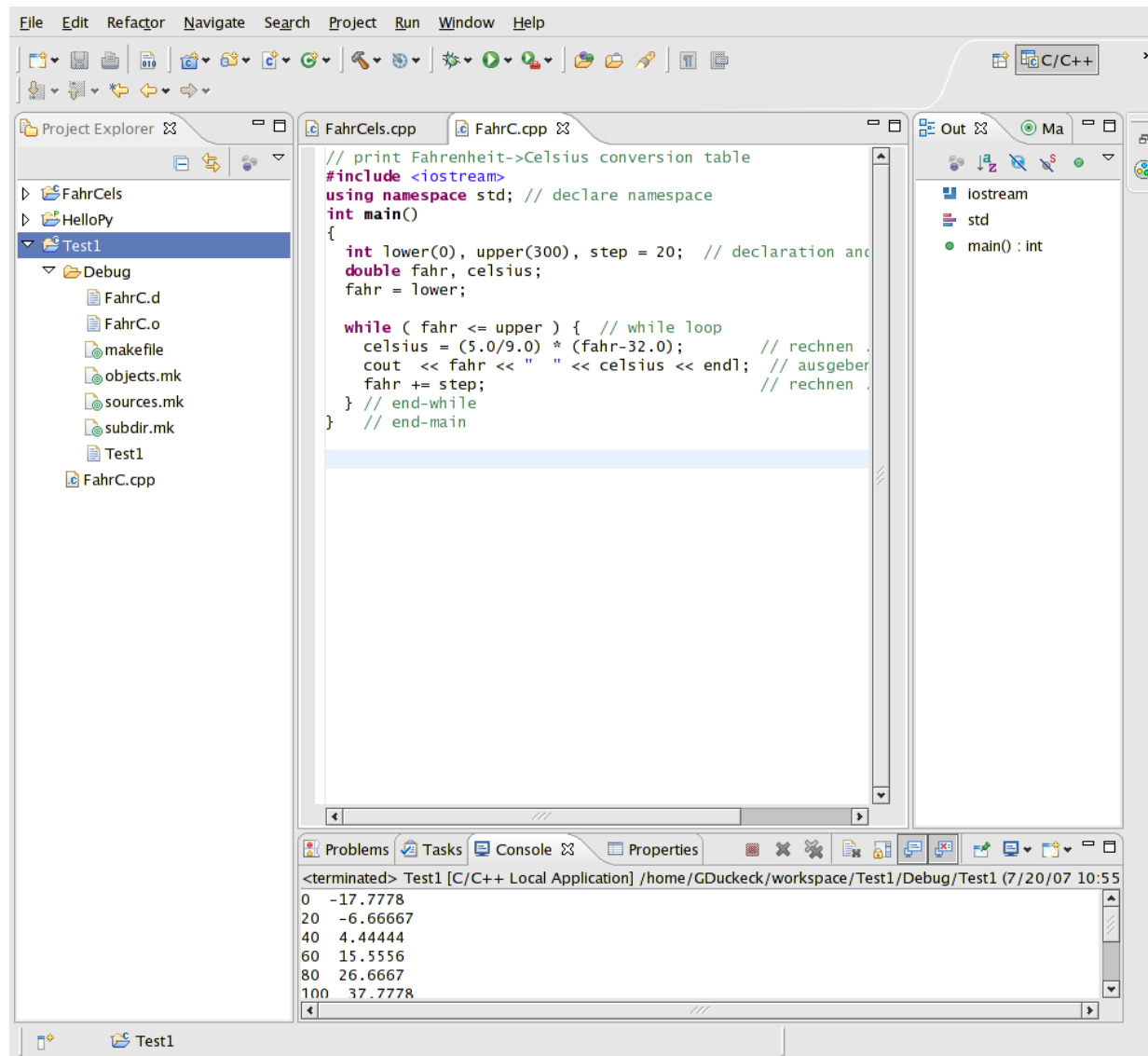
    while ( fahr <= double(upper) ) { // while loop
        celsius = (5.0/9.0) * (fahr-32.0); // rechnen ...
        printf("%4.0f %6.1f\n", fahr, celsius); // ausgeben ...
        fahr += step; // rechnen ...
    } // end-while
} // end-main
```

A red stop icon and a green arrow point to line 11, indicating a breakpoint.
- DDD Panel:** A floating window titled 'DDD' containing buttons: Run, Interrupt, Step, StepI, Next, NextI, Until, Finish, Cont, Kill, Up, Down, Undo, Redo, Edit, and Make.
- GDB Console:** Shows the following commands and output:

```
(gdb) run
Breakpoint 1, main () at fahrcels.cpp:11
(gdb) next
(gdb) |
```
- Status Bar:** Displays `△ Breakpoint 1, main () at fahrcels.cpp:11`.

Oder gleich mit integrierter Entwicklungsumgebung: Eclipse

Gibt es für C++, JAVA, Python, u.a., unter Linux, Windows, MacOS, ...



Eclipse am CIP

- Starten auf Kommandozeile:

`eclipse`

- Auswählen: File ⇒ New ⇒ C++ Project
- Projektname eintragen (**keine Leerzeichen!**), z.B. Executable ⇒ Hello World C++ Project auswählen ⇒ Next/Finish
- Project Explorer ⇒ src ⇒ XXX.cpp anklicken ⇒ öffnet Editor
- Run ⇒ Run as ...
- Run ⇒ Debug as ...

1.14 C++ Grundlagen Zusammenfassung

Soweit Schnelldurchgang/Basisgerüst für klassisches Programmieren in C/C++.

- Syntax
- Basic Datentypen
- Operationen, Kontrollstrukturen

Zwangsläufig unvollständig und oberflächlich

Aber keine Panik, auch wenn vieles unklar ist

- Grundlegende Sprachelemente tauchen immer wieder auf
- Man braucht Zeit zum verdauen und eigene Erfahrungen zu sammeln

⇒ fragen, üben, fragen, üben, fragen, üben, ...

Einige wichtige Bereiche wurden nur gestreift oder gar nicht behandelt:

- Weitere Preprozessor Funktionen, Conditionals
- C Standardfunktionen
 - input/output, filehandling, sockets
 - string Verarbeitung
 - mathematische Funktionen
- Vorrangregeln für Operatoren (Tip: immer (..) benutzen)
- bit Operationen
- Exceptions
- ...

⇒ **Siehe Literatur**

1.15 Aufgaben

Es wird nicht erwartet, dass Sie alle Aufgaben schnell während des Kurses runterschreiben. Es sind mit Absicht viele, es reicht wenn Sie etwa die Hälfte (gründlich) bearbeiten.

Grün markierte sind das Minimalprogramm, die sollten auf jeden Fall bearbeitet werden. Rot gekennzeichnete sind anspruchsvoller und v.a. für diejenigen gedacht, die schon Programmierkenntnisse mitbringen.

1. Warmlaufübungen

- Erstellen Sie das "Hello world" Programm im Editor, dann kompilieren und ausführen.
- Dasselbe mit der "Fahrenheit/Celsius" Konversion
modifizieren Sie es: Schrittweite, Bereich, umdrehen in Celsius \Rightarrow Fahrenheit.

2. Quadrat- und Kubik-Zahlen

- (a) Erstellen Sie ein Programm, dass die Quadrat- und Kubik-Zahlen von 1 bis 150 ausgibt und am Ende die Summe der Quadrat- bzw. Kubik-Zahlen
- (b) Demonstrieren Sie folgende mathematische Identität für $n = 1..200$:

$$\sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i \right)^2$$

3. Integer und Fließkommazahlen

Machen Sie sich mit der binär- bzw. hexadezimal-Darstellung von Zahlen ver-

traut.

(a) Drucken Sie Integerzahlen z.B. von -10 – 10:

```
#include <iostream>
#include <iomanip>
using namespace std; // declare namespace
int main()
{
    for ( int i=-10;i<=10;i++)
        cout <<dec <<setw(6) <<i <<setw(10) <<hex <<i <<endl;
    } // end-main
```

(b) Drucken Sie Floating-point-zahlen z.B. von -1/16 – 16:

```
#include <cstdio>
#include <cmath>
using namespace std; // declare namespace
int main()
{
    union { long l; double x; } u; // ugly hack
    for ( int i=-4;i<=4;i++) {
        u.x = pow(2., i);
        printf("%f %lX \n", u.x, u.l );
    }
    } // end-main
```

Hinweis: Rückgriff auf C-printf() um double Zahlen in hex auszugeben

(c) Lassen Sie das Programm `tdouble.cpp` laufen.

Warum "können Rechner nicht rechnen" ?

(d) Genauigkeit von `double` Operationen: Reduzieren Sie schrittweise

```
double eps = 1.;
```

```
while (...)
```

```
    eps /= 2.;
```

addieren Sie's zu

```
    onePlusEps = 1.0 + eps;
```

solange bis

```
    if ( onePlusEps == 1.0 ) ...
```

Analog für `float` (32-bit floating-point Zahlen)

4. Bit Operationen

Programmieren Sie die binäre Ausgabe von Integerzahlen mit Hilfe von Bit Operationen.

z.B.: 5 -> 101

Hinweis: Bit 15 in n abfragen mit z.B.: `if ((1 <<15) & n) != 0)`

5. Fibonacci-Zahlen

Fibonacci-Zahlen spielen eine wichtige Rolle in der Zahlentheorie und haben viele interessante Eigenschaften. Sie sind definiert als:

$$F_n = F_{n-1} + F_{n-2}; F_0 = 0, F_1 = 1$$

(a) Erstellen Sie eine Liste der Fibonacci-Zahlen. Bis zu welchem n kann man es in C++ mit `long` berechnen ?

(b) Demonstrieren Sie dass gilt:

$$F_{n+1} \cdot F_{n-1} - F_n^2 = (-1)^n$$

6. Prim-Zahlen

(a) Erstellen Sie ein Programm, das testet ob eine gegebene Zahl eine Primzahl ist.

(b) Erweitern Sie das Programm, so dass es abzählt wieviele Primzahlen es gibt, die kleiner als eine vorgegebene Zahl sind, z.B. 1 000 000.

Hinweis: Es geht nicht darum einen schnellen Algorithmus zu finden, machen Sie's so simpel wie möglich.

(c) **Sieb des Erasthones** ist ein klassisches Verfahren zur Bestimmung aller Primzahlen zwischen 2 und n . Der Algorithmus geht folgendermassen:

- Erstelle Liste aller Zahlen von 2 bis n
- Nimm schrittweise jede Zahl i dieser Liste, falls sie nicht gestrichen ist.
- Streiche alle Vielfachen von i aus der Liste

Am Ende bleiben genau die Primzahlen übrig in der Liste. *(Tipp: Im Programm Liste am besten als `bool` array implementieren, Test auf Vielfaches mit modulo Operator: `m%i == 0`)*

7. Lineare Algebra

(a) **Vektoroperationen:** Legen Sie 2 Arrays mit den Elementen $\{0.3, 1.8, -2.2\}$ bzw $\{-2.5, 3.8, 0.4\}$ an und berechnen Sie das Skalarprodukt und das Vektorprodukt.

(b) **Matrixmultiplikation:** Schreiben Sie ein Programm zur Multiplikation dieser beiden Matrizen:

```
double C[3][3] = { { 0.61, 0.24, 1.16 }, { 0.14, -0.82, 0.92 }, {  
-1.25, 0.96, -0.23 } };  
double D[3][3] = { { 0.40, -0.68, -0.68 }, { 0.65, -0.75, 0.23 }, {  
0.52, 0.51, 0.31 } };
```

8. Pointer

(a) Was macht folgender C++ Code?

```
#include <iostream>  
using namespace std;  
int main ()  
{  
    int i = 42, j = 1024;  
    int *p1 = &i, *p2 = &j;  
    *p2 = *p1 * *p2;  
    *p1 *= *p1;  
}
```

(b) Das Programm `tpointer.cpp` enthält eine kryptische Mischung aus Pointer- und Zahlen-Arithmetik, die aber für C Programme durchaus typisch ist. Gehen Sie das Programm durch und versuchen Zeile für Zeile

vorherzusagen was passiert. Anschliessend mit `cout ...` Ausgaben überprüfen oder im Debugger (`ddd`) laufen lassen.

9. Funktion für Fakultät

Schreiben Sie eine Funktion `fak(int n)`, zur Berechnung der Fakultät $n!$. Machen Sie jeweils eine Version die einen `long`, `float`, `double` Wert zurückgibt `lfak(n)`, `ffak(n)`, `dfak(n)`.

Bis zu welchem n lässt sich $n!$ bei den jeweiligen Typen berechnen ?

Verwenden Sie dabei unterschiedliche Algorithmen: Schleife und rekursive Aufrufe.

10. Power Funktion

Schreiben Sie eine Funktion `power(double x, int n)`, die ganzzahlige Potenzen (x^n) berechnet.

(a) Rufen Sie die `power(x,n)` Funktion in einem Hauptprogramm, das zunächst Werte für x sowie n von **stdin** (Tastatur) einliest. Das geht im Prinzip ganz einfach:

```
cin >>x; cin >>n;
```

(b) x und y sollen als Argumente übergeben werden: `main(int argc, char** argv)`

Dazu muss man die character strings in `argv` in `double` bzw `int` umwandeln, z.B. mittels:

```
double x = atof(argv[1]);  
int n = atoi(argv[2]);
```

11. Swap Funktion

Programmieren Sie eine Funktion `swap(sometype x, sometype y)`, die bei Aufruf `swap(a, b)` die Werte der Argumente vertauscht, d.h anschliessend enthält `a` den ursprünglichen Wert von `b` und umgekehrt.

12. Funktionen

Gehen Sie die Programme `funcarg.cpp` und `funcovl.cpp` durch, versuchen Sie die Ausgabe bei jedem Schritt vorausszusagen bevor Sie es laufen lassen.

13. Rekursive Funktion

Ein Paradebeispiel für Rekursion ist Euklid's Algorithmus zur Bestimmung des *Größten Gemeinsamen Teilers* zweier Zahlen $GGT(a, b)$.

$$GGT(a, b) = \begin{cases} GGT(b, a \bmod b) & \text{wenn } a \text{ nicht durch } b \text{ teilbar ist} \\ b & \text{sonst} \end{cases}$$

Erstellen Sie eine solche Funktion in C++

14. Lean Programming

Existierender C code ist oft knapp bis kryptisch; hier ein Beispiel

```
double x[10];
double *p = &x[10];
while ( p != x) *--p = 0.0;
```

Können Sie's nachvollziehen ?

Wem das zu läppisch ist soll's mal hiermit versuchen: `gedicht.c`

(Wer's nicht gleich versteht, einfach mal kompilieren und laufen lassen ... `gcc -o gedicht gedicht.c`

verwenden)

15. Zahlen einlesen und sortieren

In der Datei `numbers.dat` finden Sie eine Liste mit 100 Fließkommazahlen.

(a) Lesen die Zahlen in einen Array ein. (File-I/O in C++ siehe [I/O Basics](#))

(b) Finden Sie kleinsten und größten Wert.

(c) Verwenden Sie die C Standardfunktion `qsort` um die Zahlen zu sortieren und sortiert auszugeben.

16. Fehlersuche

(a) Im Verzeichnis `BadCode` finden Sie einige simple C++ source files, die entweder typische Fehler beim Kompilieren, Linken oder Ausführen verursachen, oder zumindest *Warning-Messages* erzeugen, wenn man zum Kompilieren die Optionen `-O -Wall` verwendet:

```
g++ -O -Wall -c xxx.C
```

Versuchen Sie jeweils den Fehler oder das Problem herauszufinden und zu korrigieren.

(b) Folgendes Programm zum Primzahlen-Zählen wurde sehr nachlässig geschrieben, es enthält etliche Compiler- und Syntax-fehler. Korrigieren Sie die Fehler und bringen Sie das Programm zum Laufen.

```
bool isPrime(long i)
{
    for( long test = 2; test < i; test++) {
        if(i%test == 0) {
```

```
        return false;
    }
}
return true;
}

int Main()
{ // count number of prime numbers smaller than n_loops
    long n_loops = 50000,
    Long n_primes;
    for( long i = 0, i < n_loops; i++) {
        if( isprime(i) ) {
            n_primes++;
        }
        cout << n_primes << " primes found " << endl;
    }
}
```

2 Klassen und Objekte

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

2.1 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, ...
- ...

Beliebig komplexe Datenstrukturen auch schon mit C `struct` möglich. C++ Klassen zusätzlich Verknüpfung Daten und Funktionen.

Historische Entwicklung:

Unstructured Programming: Ein Hauptprogramm + Daten

Procedural Programming: Ein Hauptprogramm + globale Daten + kleinere Funktionen (prozedures)

Modular Programming: Ein Hauptprogramm + globale Daten + Module mit lokalen Daten und Unterprozeduren

Object-oriented: Daten und Prozeduren integriert in Klassen, keine globalen Daten, kein eigentliches Hauptprogramm, Objekte kommunizieren direkt

Im Prinzip natürlicher & intuitiver Ansatz:

⇒ Alltag **Datum** & **Methode** verknüpft

Auto ..., Fahren, Schalten, Blinken, ...

Wiesn ..., Maß, Schunkeln, Brechen, ...

Fußball ..., Blutgrätsche, Schwalbe, ...

2.2 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.y + 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

⇒ **Grundkonzept für Objektorientiertes Programmieren**

2.3 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

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 );  
}
```

2.4 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

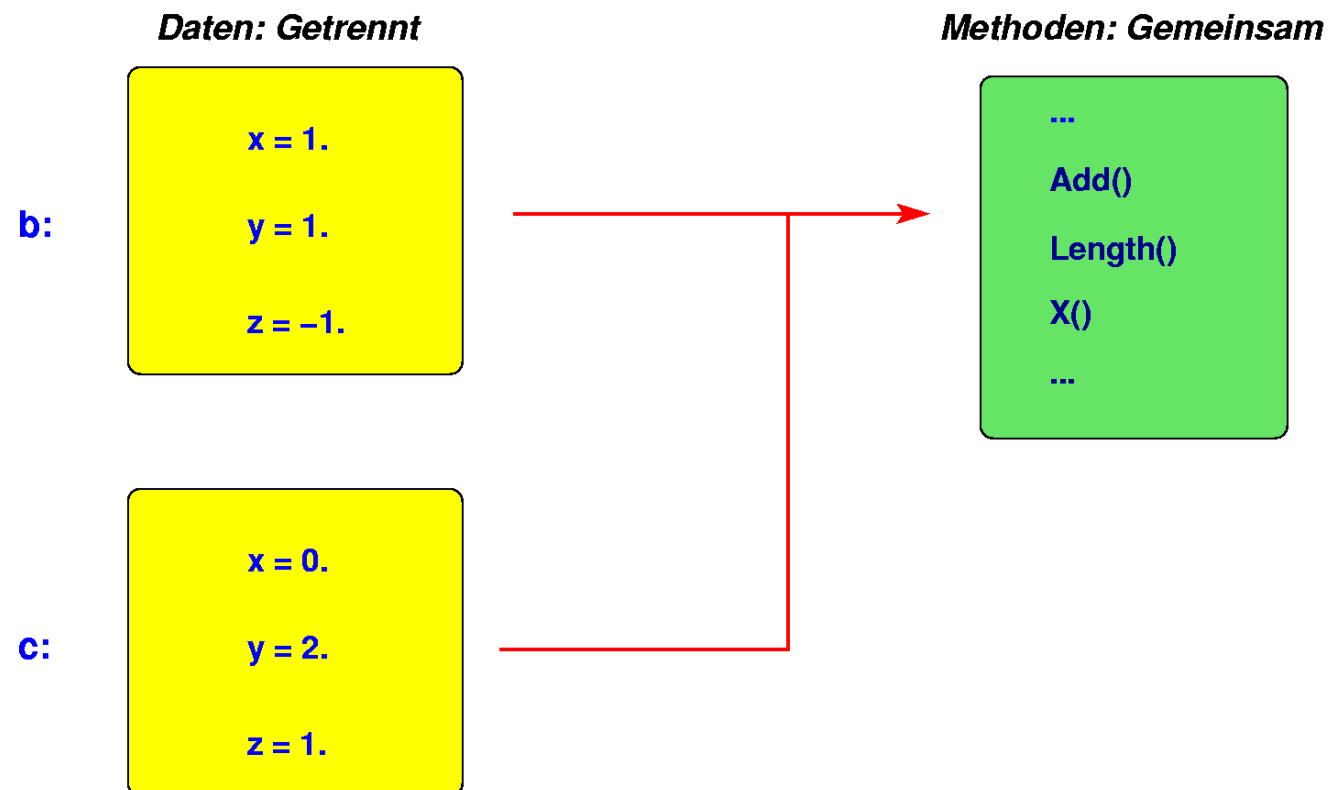
```
double My3Vector::Length() { // get length of Vector  
return( sqrt( x*x + y*y + z*z) );
```

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

```
ThreeVector b(1.,1.,-1.);
```

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

Memory-Model



Zugriff auf Member-Variablen in Member-Funktionen

ThreeVector a(-1.,0.5,0.5);

ThreeVector b(1.,1.,-1.);

a.Length();

```
public double Length() {  
    return( sqrt( this.x*this.x + this.y*this.y +  
                  this.z*this.z );  
}
```

b.Length();

```
public double Length() {  
    return( sqrt( this.x*this.x + this.y*this.y +  
                  this.z*this.z );  
}
```

Member-Variablen und Funktionsargumente in Member-Funktionen

ThreeVector a(-1.,0.5,0.5);

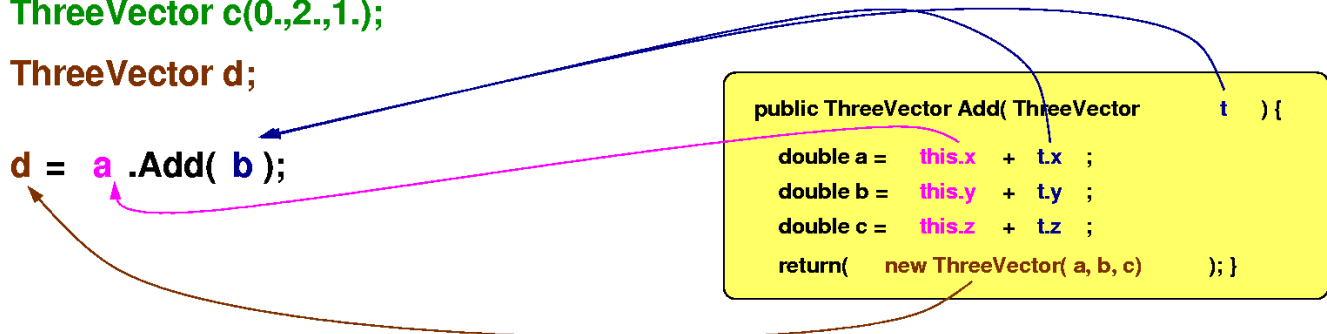
ThreeVector b(1.,1.,-1.);

ThreeVector c(0.,2.,1.);

ThreeVector d;

d = a.Add(b);

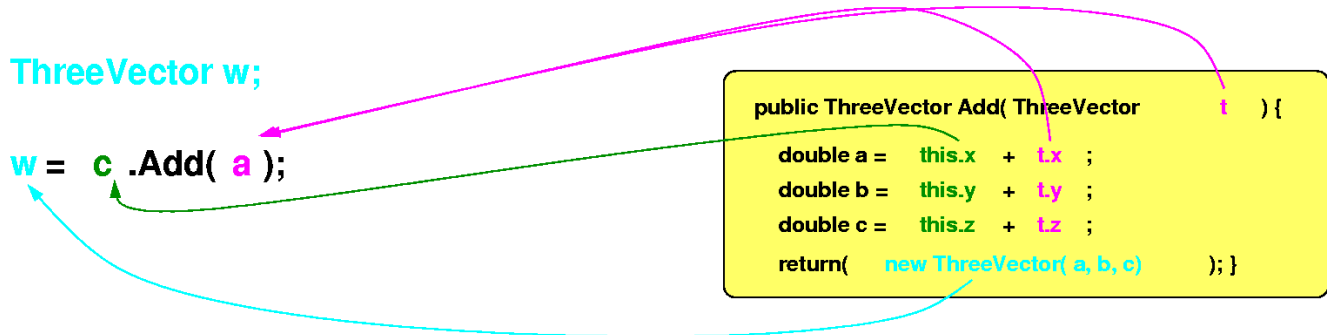
```
public ThreeVector Add( ThreeVector t ){  
    double a = this.x + t.x ;  
    double b = this.y + t.y ;  
    double c = this.z + t.z ;  
    return( new ThreeVector( a, b, c ) );  
}
```



ThreeVector w;

w = c.Add(a);

```
public ThreeVector Add( ThreeVector t ){  
    double a = this.x + t.x ;  
    double b = this.y + t.y ;  
    double c = this.z + t.z ;  
    return( new ThreeVector( a, b, c ) );  
}
```



2.5 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.

2.6 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.)`;

2.7 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.

2.8 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 übergeben.

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 \star für `My3Vector`: `a \star b ;`

Skalar- oder Kreuz-Produkt ? Nicht sinnvoll zu entscheiden \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;
```

⇒ += 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.

2.9 non-member function and operators

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

```
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, ...

2.10 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.);
```

2.11 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;
};
};
```

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.

2.12 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(My3Vector & v);`
und Verwendung: `My3Vector c(b);`
- **= 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 automatische Erzeugung ausreichend:

- **Copy constructor** und **= Operator** legen neues Objekt an und kopieren die Member-Variablen (`double x, y, z` bei `My3Vector`). Für einfache Klassen meist ok, solange kein **dynamic** memory angelegt wird (z.B. `new double[100]`) \Rightarrow dann eigene Implementierung !
- Analog **destructor**, v.a. nötig wenn mit `new ...` Speicher reserviert wurde. Entsprechender `delete ...` Aufruf im **destructor** um Speicherbereich wieder zurückzugeben.

2.13 Übung: Klasse BigInt

Ziel: Klassenbibliothek für beliebig grosse Integer Zahlen

” 41237539784637218904682394617984678146857817557”

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
⇒ 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;
}
```


2.14 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

⇒ 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;
...
}
```

2.15 Klassen erweitern – 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(); // get length of vector
    double T();
    double X();
    double Y();
    double Z();
    MyLVector Add( MyLVector p ) ; // add two vectors
    double Angle( MyLVector p ); // angle between two vectors
    double Mass() ; // get mass of 4-vector
};
```

Viele Gemeinsamkeiten mit *My3Vector* 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(); // get length of vector
    double T();
    double X();
    double Y();
    double Z();
    MyLVector Add( MyLVector p ); // add two vectors
    double Angle( MyLVector p ); // angle between two vectors
    // ....
};
```

⇒ 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( MyLVector p )
```

```
{
```

```
    return( vec3.Angle(p.vec3)); // re-use code
```

```
}
```

```
double MyLVector::X()
```

```
{
```

```
    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(); // get mass of 4-vector
    double Mass( MyLVector p); // 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() // Method for mass:
{
```

```
// caution: direct access to My3Vector x,y,z doesn't work with private !  
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, im Zweifelsfall Aggregation verwenden.

Bei Verwendung von *Polymorphismus* (nächster Kurs ...) ist vor allem konsistentes *Verhalten* wichtig bei Vererbung, weniger die Funktionalität.

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.

2.16 Aufgaben

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 anschließend 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. String Klasse

Die C++ `string Klasse` stellt eine Vielzahl nützlicher Methoden zur Textanalyse bzw. Modifikation zur Verfügung. In `kant.txt` finden Sie eine elektronische Fassung von Immanuel Kant's "Kritik der reinen Vernunft".

(a) Wieviele Zeilen enthält der Text ?

(b) Finden Sie die String Funktion die Gross-Buchstaben in Klein-Buchstaben umwandelt und transformieren sie damit den ganzen Text.

(c) Wie oft kommt das Wort `Vernunft` in dem Text vor ?

Hinweis: Mit `getline(istream in, string s)` kann man eine ganze Zeile in einen String einlesen. Rückgabewert kann für File-Ende Test genommen werden (liefert 0).

7. 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.

8. **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`.

9. **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](#).

10. 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](#).

Modifizieren Sie das Programm, z.B. zufällige Starthöhe, zufällige Variation des Schubfaktors, etc.

3 **Templates und STL**

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

3.1 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> 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);
}
```

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;
```

```
}
```

3.2 Template Klassen

Analog für Klassen Definitionen:

```
#include <iostream>
using namespace std;
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_di(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);  
..
```

3.3 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**: z.B. suchen, sortieren etc.

Grundlegende Problematik:

- N Datentypen
- M Container
- K Algorithmen

⇒ $N * M * K$ Klassen & Funktionen nötig

Abhilfe durch STL:

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

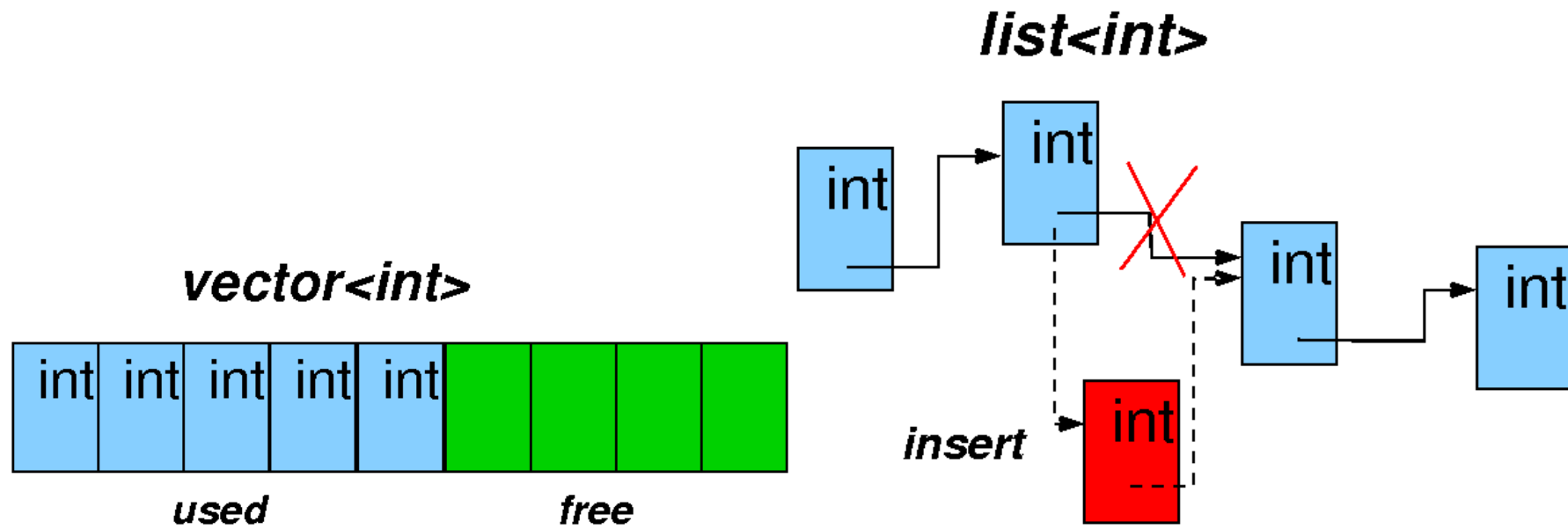
Wir müssen uns auf einführende Beispiele beschränken ...

3.4 Container

- `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.

Abgesehen vom normalen Array können alle Container (weitestgehend) in gleicher Weise benutzt werden (Zugriff, einfügen etc.), nur der Aufwand für die nötigen Operationen kann stark variieren.

Z.B., wenn in einem Vektor am Beginn etwas eingefügt wird müssen alle Elemente verschoben werden.



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: 1. Element im Container.
- `end()` Ende

Wichtig: `end()` zeigt auf das “virtuelle” Element **nach** dem letzten Element, also auf **N+1** bei **N** Elementen im Container.

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
    }
}
```

Für praktische Arbeit ist **STL-vector** besonders wichtig. Sollte (fast) immer anstelle des Standard C Arrays verwendet werden!

Ausnahmen sind u.U. lokal verwendete, direkt initialisierte Arrays oder bei sehr speicher-/zeit-kritischen Anwendungen mit mehr-dimensionalen Arrays.

3.5 Algorithmen

find :

```
#include <iostream>
#include <cstring>      // C std string ops
#include <algorithm>    // STL algorithms, find ...
using namespace std;
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.

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

sort :

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    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;    // Laenge ausgeben
    sort( vec1.begin(), vec1.end() ); // alphapetisch sortieren von Anfang bis Ende
    for ( vector<char>::iterator vc = vec1.begin(); vc != vec1.end(); vc++ ) {
        cout << *vc ; // de-referencing
    }
    cout << endl; // Zeilenende
}
```

- 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.

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() );

...
```

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 );
```

3.6 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 initialisiert.

- Viele praktische und sehr flexible Algorithmen verfügbar

```
copy, accumulate, 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 Array oder Matrix emulieren

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

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];
};
```

3.7 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
}
```

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
```

3.8 Aufgaben

1. Template

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

2. STL-Container

Spielen Sie mit den vorgestellten Beispielen zu den verschiedenen Containern, Zugriffsfunktionen, Iteratoren und Algorithmen.

3. Prim-Zahlen cont.

Das Primzahl-Programm ([Aufgabe 1.6](#)), das abzählt wieviele Primzahlen es gibt, die kleiner als eine vorgegebene Zahl sind (z.B. 1 000 000), lässt sich wesentlich effizienter gestalten. Dazu trägt man die Primzahlen nacheinander in einen STL-Vektor und testet für alle Kandidaten nur noch die Division mit den schon bekannten Primzahlen.

4. Zahlen sortieren

Implementieren Sie [Aufgabe 1.14](#), aber diesmal mit STL Funktionen für das Speichern und Sortieren.

(a) Zunächst noch "klassisch", d.h. mit einer **for** oder **while** loop über den Container.

(b) Dann ohne jegliche Loop, mit den STL **iterator/copy** Methoden.

5. Dreier-Vektoren sortieren

Container und Algorithmen funktionieren auch für eigene Klassen/Objekte. Testen Sie das speichern in vector und sortieren mit zufällig erzeugten Dreier-Vektoren.

Vorher aber noch den `<` Operator für Dreier-Vektoren implementieren !

```
#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
```

6. BigInt mit Vektoren

Benutzen Sie Vektoren für die dynamische BigInt Version.

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.

Schluss

Das war ein Schnelldurchgang C/C++ ...

- C/C++ Grundlagen
- Klassen/Objekte/Vererbung
- Templates/STL

Vieles fehlt noch

- File & Network I/O
- Polymorphism / virtual Functions
- Sets & Hash-Maps
- Function Objects
- ...

Das wichtigste ist üben, üben, üben, ... am besten in konkreten Anwendungen

Hardware Programmierung, Statistikkurs, Java, Objektorientiertes Programmieren, Numerik, ...

4 Kurzeinführung in Linux

4.1 Links zu Linux Tutorials

Einige Tutorials zu Linux gibt es z.B. unter:

- [SelfLinux](#)
- [UNIX Tutorial for Beginners](#)

4.2 Die Linux X-Benutzeroberfläche

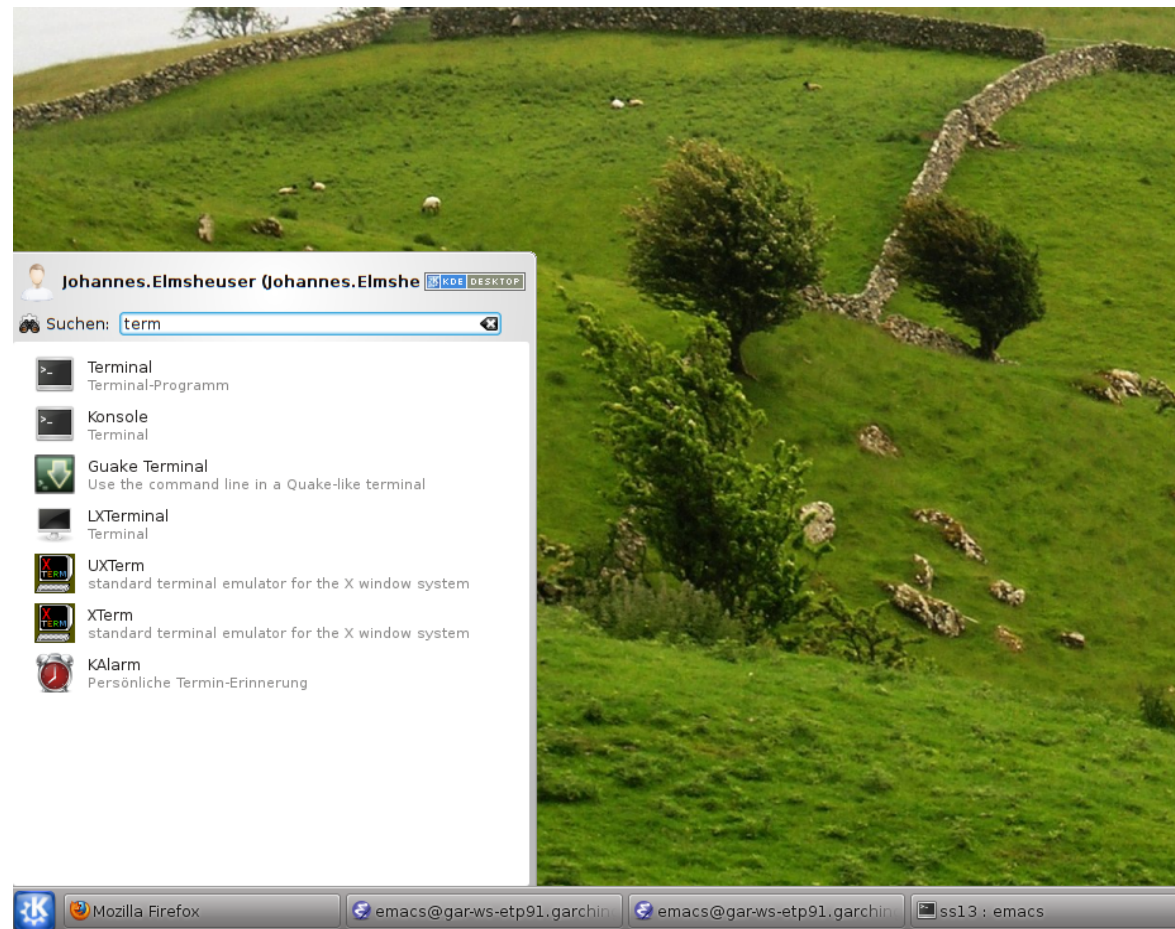
Die beliebtesten Benutzeroberflächen bzw. Fenstermanager auf Linuxsystemen sind: KDE bzw. GNOME. Diese können auf dem login-screen unter "Menü" → "Session type" zwischen verschiedenen Fenstermanager aussuchen. Wählen Sie entweder "[KDE](#)" bzw. "[GNOME Classic](#)".

4.3 Terminalfenster starten

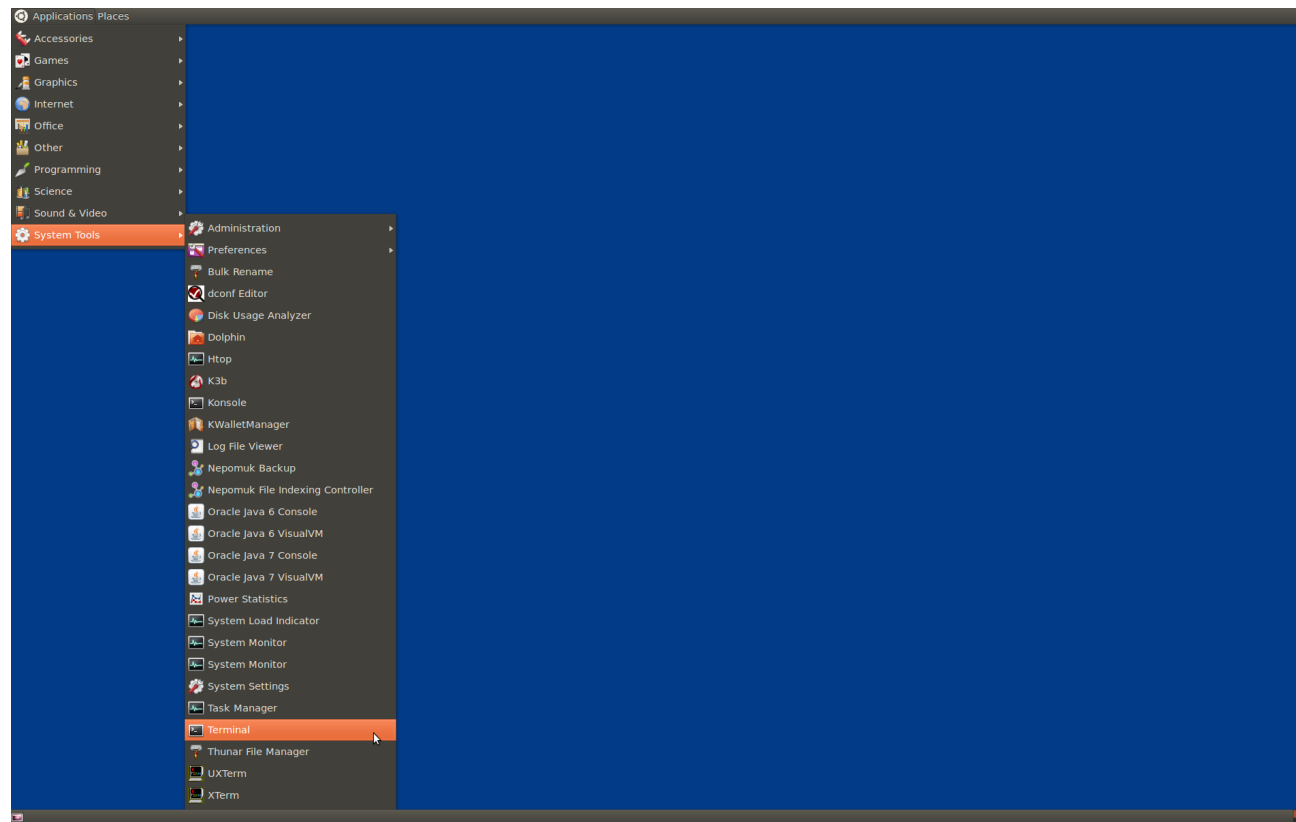
Nachdem Sie eingeloggt sind, starten Sie ein Terminalfenster, mit dem Sie verschiedene Befehle auf der Kommandozeile eingeben können:

- Unter KDE klicken Sie in der rechten unteren Bildschirmcke auf den blauen "K"-Knopf und tippen in das danach erscheinende Suchfeld des Startmenüs: "term". Klicken Sie anschliessend auf den "Konsole" Menüeintrag und ein

Terminalprogramm wird gestartet.



- Unter GNOME classic klicken Sie in der oberen rechten Bildschirmcke auf "Applications" und anschliessend im "System Tools"-Menü auf den Eintrag "Terminal":



- Es öffnet sich ein Terminalfenster, in dem Sie Befehle auf der sog. bash Kommandozeile eingeben und ausführen können:



4.4 Befehle im Terminalfenster

Auf der bash Kommandozeile können Befehle eingegeben werden, um Programme zu starten oder z.B. mit dem Dateisystem zu interagieren.

Einfache Befehle:

- Das aktuelle Arbeitsverzeichnis anzeigen:

pwd

- Den Inhalt des aktuellen Verzeichnis anzeigen:

ls

- Den Inhalt des aktuellen Verzeichnis als Liste anzeigen:

ls -l

- Den Inhalt des aktuellen Verzeichnis als Liste mit versteckten Dateien anzeigen:

ls -al

- Den Inhalt des aktuellen Verzeichnis als Liste sortiert nach Änderungsdatum anzeigen:

ls -rtl

- In das Homeverzeichnis wechseln:

cd

- Ein neues Verzeichnis anlegen:

mkdir mycode

- In das neue Verzeichnis wechseln:

cd mycode**Weitere Befehle:**

- Eine leere Datei anlegen:

touch test.txt

- Eine Datei löschen:

rm test.txt

- Eine Datei aus dem WWW herunterladen:

wget <http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckurs/source/numbers.dat>

- Den Inhalt einer Datei vollständig anzeigen:

cat numbers.dat

- Den Inhalt einer Datei interaktiv anzeigen (Verlassen mit "q", Scrollen mit Pfeiltasten):

less numbers.dat

- Die Anzahl der Zeilen einer Datei anzeigen:

wc -l numbers.dat

- Ein leeres Verzeichnis löschen:

rmdir mytestdir

- Das aktuelle Verzeichnis kann mit "." angesprochen werden:

ls .

- In das übergeordnete Verzeichnis kann mit ".." angesprochen werden:

ls ..

- In das übergeordnete Verzeichnis wechseln:

cd ..

- Eine Datei von einem Verzeichnis in das aktuelle Verzeichnis kopieren:

cp /path/to/somefile .

- Eine Datei "somefile" vom Verzeichnis "/path/from" in das Verzeichnis "/path/to" kopieren:

cp /path/from/somefile /path/to/

Programme starten:

- Ein Programm starten Sie einfach durch Eingabe des Befehls auf der Kommandozeile. Dadurch wird die Kommandozeile für weitere Eingaben blockiert. Starten Sie deshalb sämtliche interaktiven Programme wie Editoren etc. immer mit einem zusätzlichem **&** am Ende der Befehlszeile, um die Kommandozeile wieder für neue Befehle freizugeben. Starten Sie den KDE Editor z.B. mit:

kate &

Befehlseingabe:

- Auf vorher eingegebene Befehle kann mit der Pfeil-nach-oben bzw. Pfeil-nach-unten Taste zugegriffen werden.
- Kommandozeilenvervollständigung: Lange Programmnamen können mit Hilfe der Tabulatortaste vervollständigt werden, d.h. Sie müssen nicht immer lange Programmnamen oder Dateinamen eintippen, sondern brauchen nur die Anfangsbuchstaben eintippen und nach Drücken der Tabulatortasten kann die Befehlszeile vervollständigt werden.

Eingabe-/Ausgabeumleitung:

- Die Ausgabe eines Programms oder eines beliebigen Befehls kann vom Bildschirm des Terminalfensters in eine Datei mit `>` umgeleitet werden:

`ls -rtl > out.txt`

- Die Eingabe in ein Programm kann anstatt von der Tastatur von einer Datei mit `<` umgeleitet werden:

`cat < numbers.dat`

Editoren:

- KDE Editor:

`kate`

- GNOME Editor:

`gedit`

- Fortgeschrittene Editoren:

`emacs` oder `vi`

Entwicklungsumgebungen und Debugger:

- Java, C++, Python Entwicklungsumgebung:

eclipse

- C++ Entwicklungsumgebung:

kdevelop

- Qt und C++ Entwicklungsumgebung:

qtcreator

- Graphischer Debugger:

ddd

GNU C++ Compiler:

- Ein C++ Programm kompilieren und linken in einem Schritt:

g++ -o mytest mytest.cpp

- Ein C++ Programm kompilieren:

g++ -c mytest.cpp

- Ein zusammengesetztes C++ Programm kompilieren und linken:

g++ -o TLVector MyLVector.cpp My3Vector.cpp

Verzeichnisse archivieren:

- Das aktuelle Verzeichnis in eine Datei archivieren und packen:

tar cvzf myfile.tar.gz .

- Ein Archivdatei in aktuelle Verzeichnis entpacken:

tar xvzf myfile.tar.gz