Vorkurs Informatik: Erste Schritte der Programmierung mit C++

Arne Nägel, Andreas Vogel, Gabriel Wittum
Lehrstuhl Modellierung und Simulation
Goethe-Center for Scientific Computing
Goethe Universität Frankfurt a.M.

1. Oktober 2014

STARKER START ins Studium

Agenda

Vom Lösen von Problemen mit dem Computer

Der Einsatz von Computern ist aus unserem Alltag nicht wegzudenken:

- Verschlüsseltes Versenden von Emails
- Bild- und Videokompression
- Crashtests bei der Fahrzeugentwicklung
- Virtuelle Wirkstoffentwicklung
- Visualisierung von CT-Daten
- Erfassen von Lagerbeständen über Datenbankabfragen
- Durchsuchen von Texten nach Schlüsselworten
- Steuerung von Robotern
- ...

Ihre Aufgabe

Problem $\Longrightarrow$ **Algorithmus** $\Longrightarrow$ **Implementierung** $\Longrightarrow$ **Lösung**

Was ist ein Algorithmus?

Definition 1 (Algorithmus)

Ein *Algorithmus* ist eine Menge von Regeln, durch deren Befolgung (in festgelegter Reihenfolge) ein bestimmtes Problem gelöst wird.

Was ist ein Algorithmus?

Definition 1 (Algorithmus)

Ein *Algorithmus* ist eine Menge von Regeln, durch deren Befolgung (in festgelegter Reihenfolge) ein bestimmtes Problem gelöst wird.

Beispiele:

- Rezept zum Backen eines Kuchens
- Ziehen eines Fahrscheins am Automaten
- Abschliessen der Wohnungstür
- ...

Was ist ein Programm?

Definition 2 (Programm)

Einen Algorithmus, der für einen Computer verständlich formuliert ist, nennt man *Programm*.

Was ist ein Programm?

Definition 2 (Programm)

Einen Algorithmus, der für einen Computer verständlich formuliert ist, nennt man *Programm*.

Bestandteile eines Programms:

- Variablen, beispielsweise:
  `m_mehl=200.0, n_eier=4, t_back = 20, temp=200`
- Anweisungen, beispielsweise:
  ```
  solange (finde_klumpen): knete_teig
  falls (schoen_knusprig): nimm_aus_ofen
  ```

Was ist Programmieren?

Herausforderung:

- Menschliche Sprache sehr komplex
  bzgl. Syntax und Semantik
- Instruktionssatz einer Recheneinheit
  im Computer vergleichsweise beschränkt
- Vermittlungswerkzeug: Programmiersprachen
- Im Rahmen Ihres Studiums werden Sie nicht nur Sprachen,
  sondern vor allem Konzepte erlernen

Was ist Programmieren?

Herausforderung:

- Menschliche Sprache sehr komplex
  bzgl. Syntax und Semantik
- Instruktionssatz einer Recheneinheit
  im Computer vergleichsweise beschränkt
- Vermittlungswerkzeug: Programmiersprachen
- Im Rahmen Ihres Studiums werden Sie nicht nur Sprachen,
  sondern vor allem Konzepte erlernen

Was ist Programmieren?

Herausforderung:

- Menschliche Sprache sehr komplex
  bzgl. Syntax und Semantik
- Instruktionssatz einer Recheneinheit
  im Computer vergleichsweise beschränkt
- Vermittlungswerkzeug: Programmiersprachen
- Im Rahmen Ihres Studiums werden Sie nicht nur Sprachen,
  sondern vor allem Konzepte erlernen

Lernen Sie Programmieren, nicht eine Programmiersprache!

Programmiersprachen

Es unterschiedliche Programmier-Ansätze (Paradigmen) und entsprechend auch unterschiedliche Programmiersprachen!

(deklarativ) logisch : *Prolog*, ...

(deklarativ) funktional : *Scheme*, Microsofts *F#*, *Haskell*, ...

(imperativ) prozedural : *Fortran*, *Pascal*, *C*, *Python*, ...

(imperativ) objektorientiert : *C++*, *Java*, *C#*, ...

Programmiersprachen

Es unterschiedliche Programmier-Ansätze (Paradigmen) und entsprechend auch unterschiedliche Programmiersprachen!

(deklarativ) logisch : *Prolog*, ...

(deklarativ) funktional : *Scheme*, Microsofts *F#*, *Haskell*, ...

(imperativ) prozedural : *Fortran*, *Pascal*, *C*, *Python*, ...

(imperativ) objektorientiert : *C++*, *Java*, *C#*, ...

Sprachen können **interpretiert** sein oder **in Maschinencode übersetzt** sein.

Was ist C++ ?

Zur Geschichte von C++ :

Abkömmling/Schwester von C

- 1980: Entwicklung der Sprache von Bjarne Stroustrup
- 1998: ISO/IEC/ANSI/DIN - Standard
- 2003: Korrekturen des Standards
- 2006: TR1 (Technical Report 1) - geplante Erweiterungen
- 2011: C++11 - Neuester C++ Standard

Literaturhinweise (Klassiker)

Kernighan, Brian W. ; Ritchie, Dennis M.: The C programming language. Prentice Hall, 1978

Stroustrup, Bjarne: The C++ programming language. Addison-Wesley, 1985 (online - neuere Auflage)

Literaturhinweise (Einsteiger und Fortgeschrittene)

- Stroustrup, Bjarne: Einführung in die Programmierung mit C++. Pearson Studium, 2010
- Willms, André: C++-Programmierung lernen : Anfangen, Anwenden, Verstehen. Addison-Wesley, 2008 . (online)
- Breymann, Ulrich: C++ : Einführung und professionelle Programmierung (9. Aufl). Hanser, 2007
  Breymann, Ulrich: Der C++-Programmierer : C++ lernen - professionell anwenden - Lösungen nutzen. Hanser, 2009
- Dieterich, Ernst-Wolfgang: C++. Oldenbourg, 2000 (online)

Nutzen einer Entwicklungsumgebung

Entwicklungsumgebungen

Eine Entwicklungsumgebung (IDE, integrated development environment) vereinfacht das Schreiben von Code, das Verwalten größerer Projekte und das Compilieren.

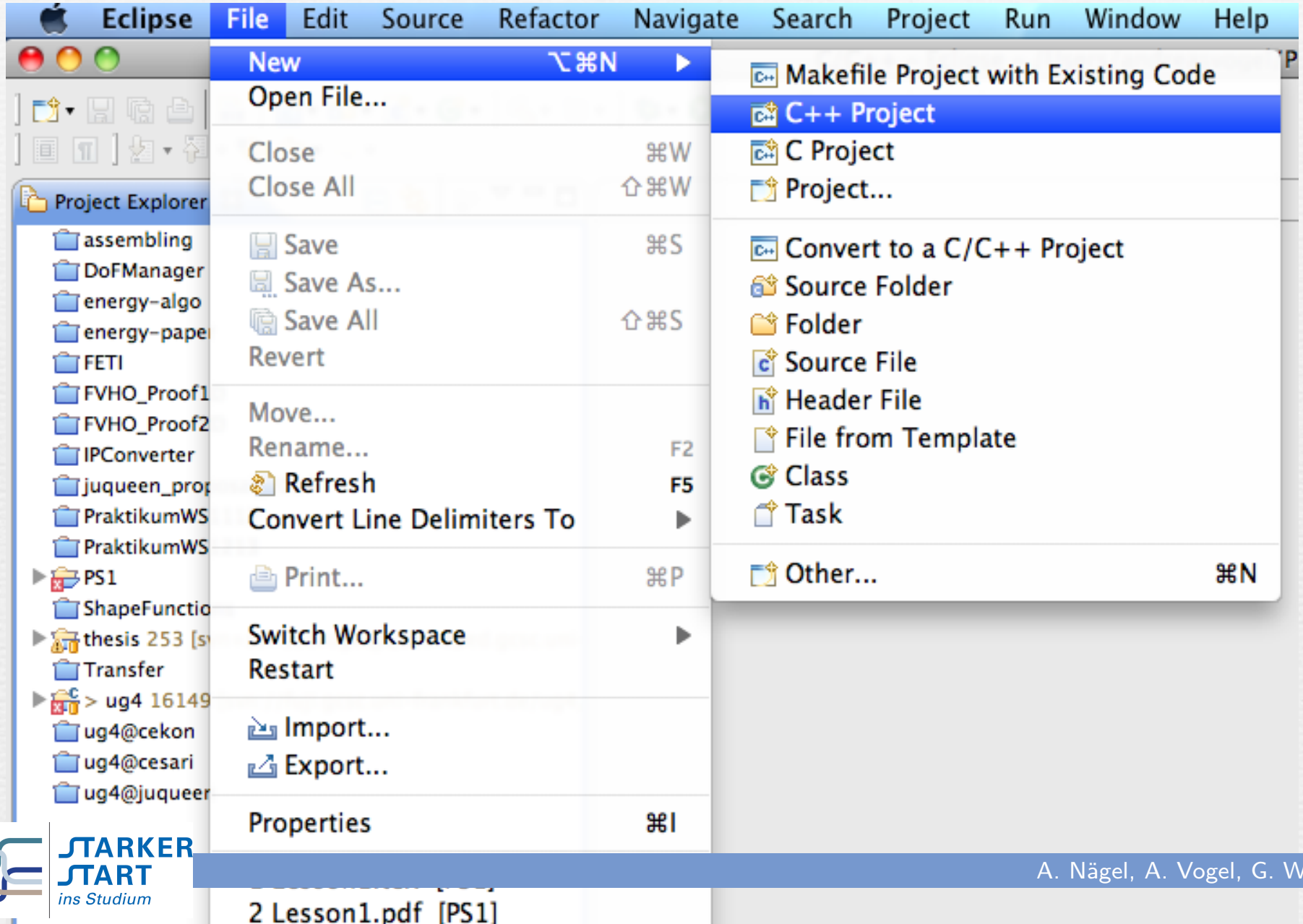
Bekannte Entwicklungsumgebungen sind:
Eclipse, Code::Blocks, Xcode, NetBeans, ...

Im folgenden wird die Umgebung **Eclipse** verwendet:

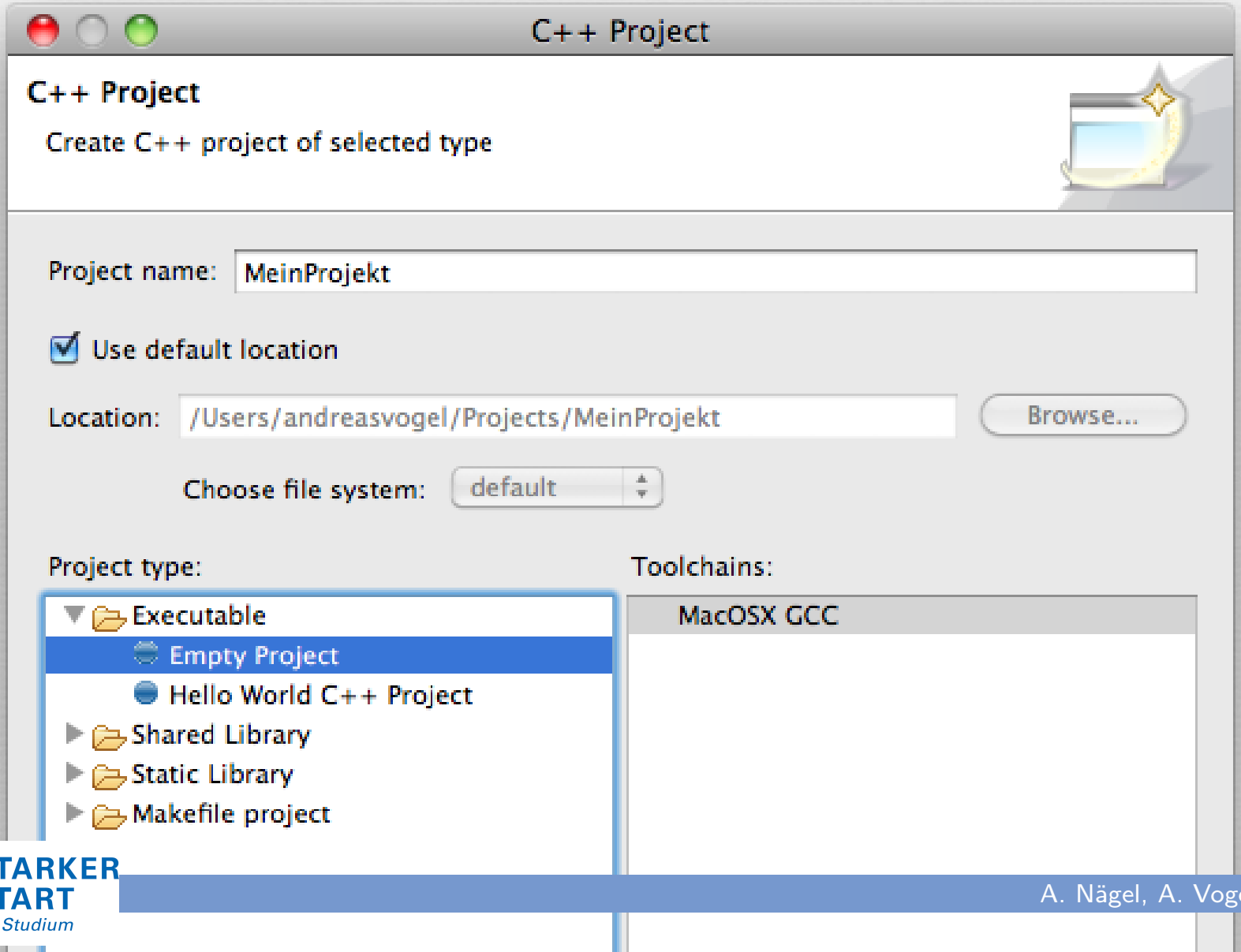
`www.eclipse.org/downloads/`

Für C++ Programme empfiehlt sich:
Eclipse IDE for C/C++ Developers.

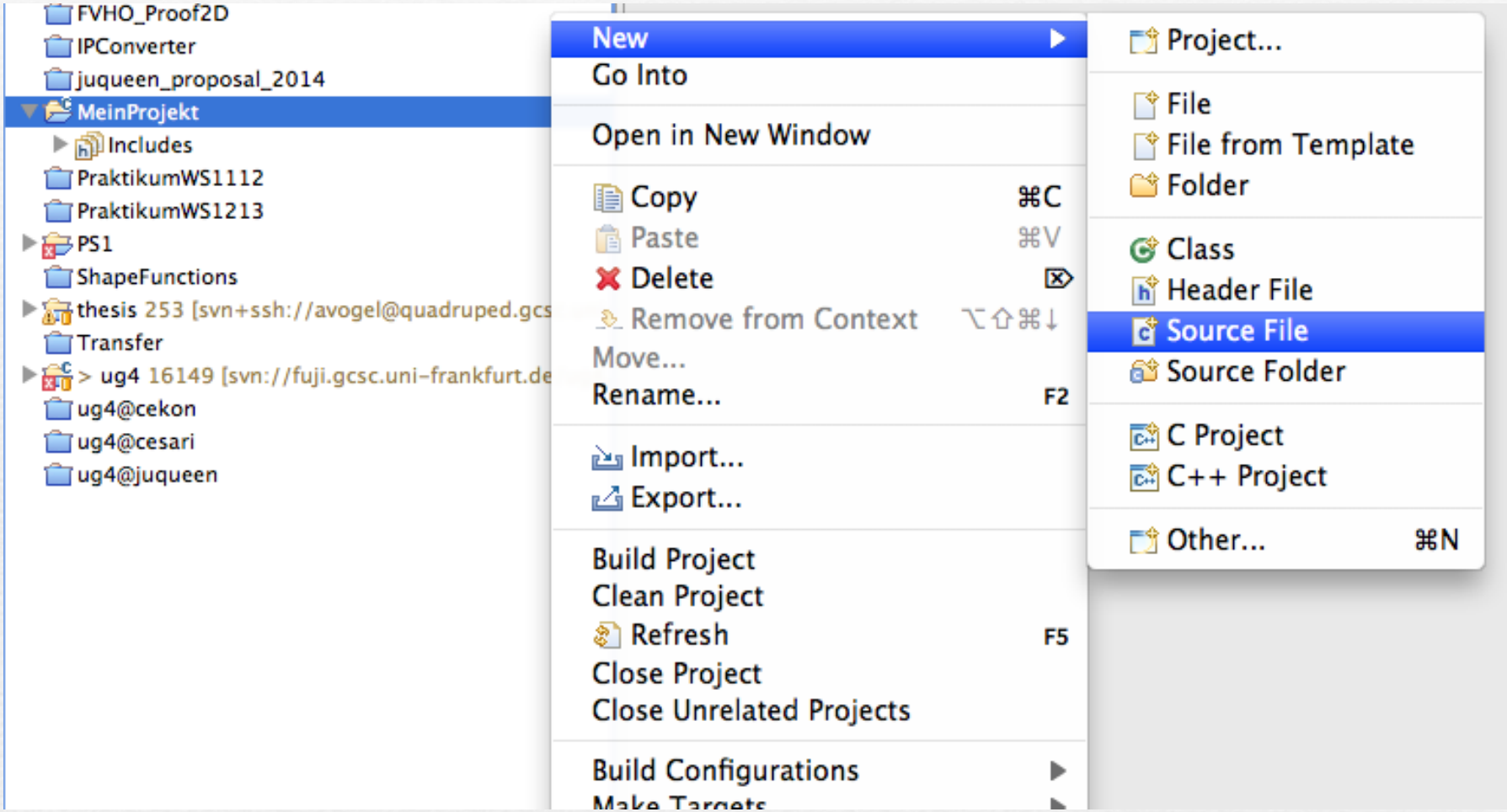
Neues Projekt anlegen



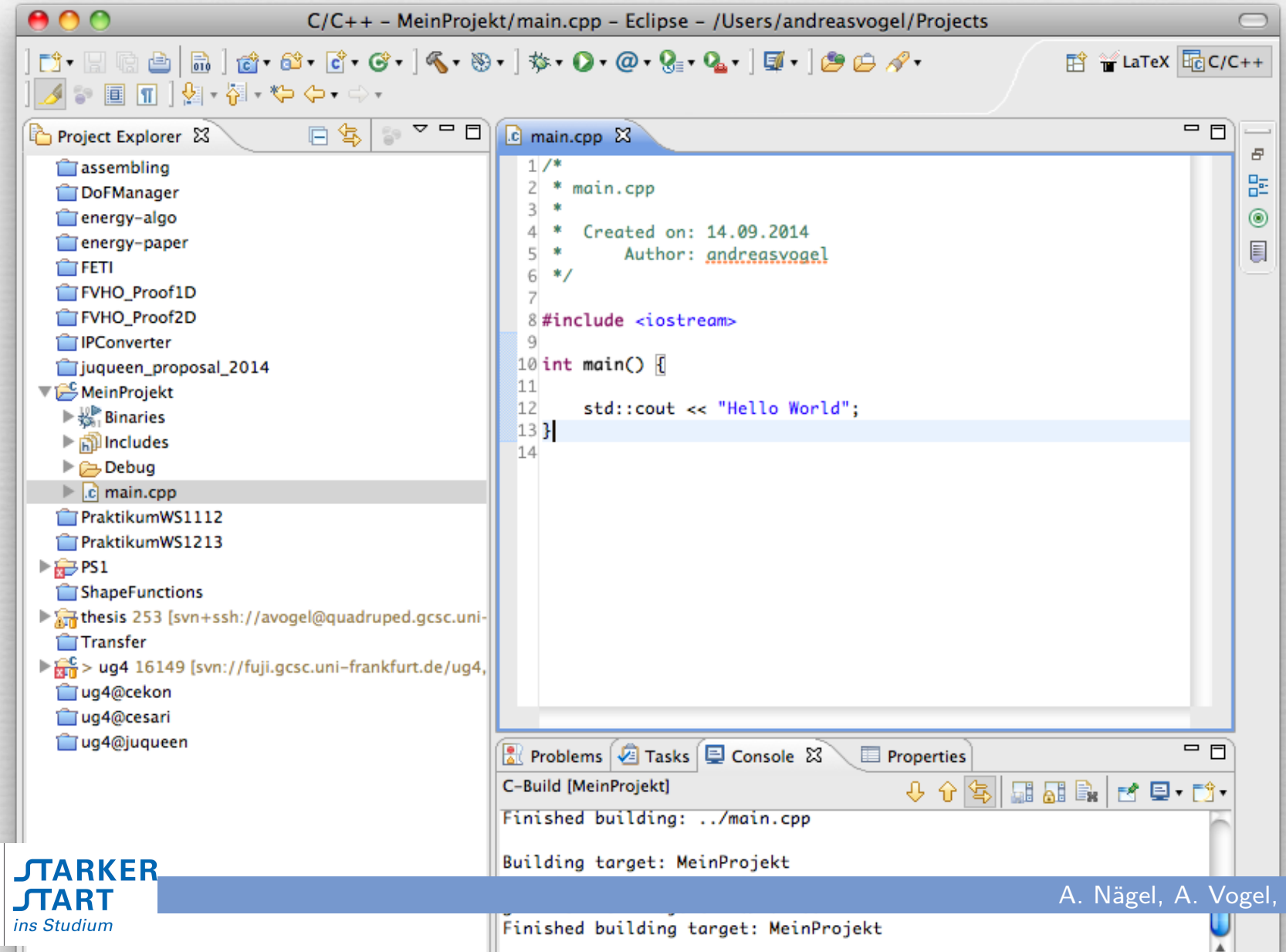
Neues Projekt anlegen



Neue Datei anlegen

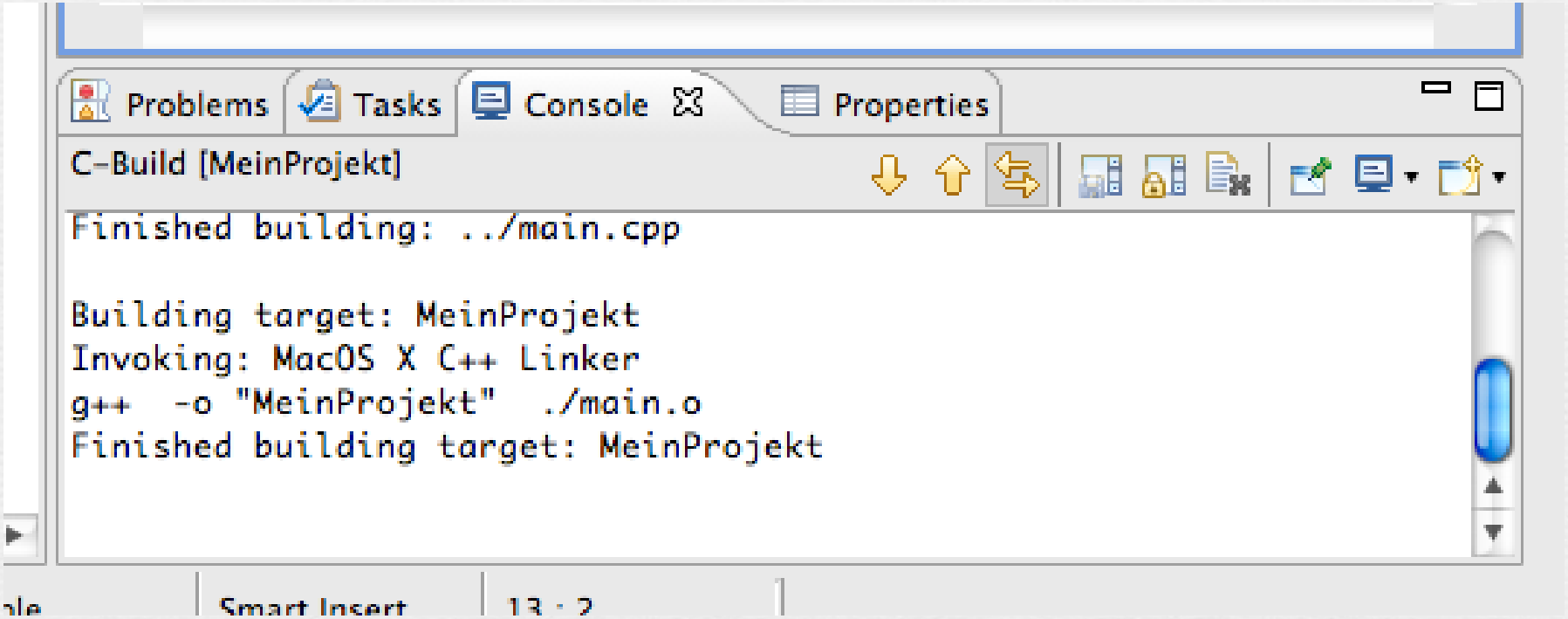


Dateien editieren



Compilieren

Compilieren (Hammer) und Ausführen (Play) kann man über die Schaltflächen in der Programmleiste durchführen:



Vom Quelltext zum Programm

Compiler

- Ein CPP-Programm ist **nur** Text.
  - Menschen können Text lesen.
  - Jeder beliebiger Editor kann verwendet.
- Computer können Text **nicht** direkt verstehen.
  - **Übersetzer** nötig: Compiler.
  - gpp ist ein Compiler (der GNU Compiler).
  - Viele weitere Compiler: icc (Intel), XL C++ (IBM), ...

Was genau macht der Compiler? Wie verwende ich ihn?

Erstes Programm

```
#include <iostream>

int main() {
  std::cout << "Hello world!\n";
}
```

helloworld.cpp

Ausführen:

```
$ g++ -o helloworld helloworld.cpp
$ ./helloworld
```

Aufgabe 3

Probieren Sie es (sowohl hier als auch im Folgenden ...) aus.

Datei-Endungen

Es gibt verschiedene für C++ relevante Dateitypen.

Suffix	Dateityp
.cpp / .cc	C++ -Quelldatei
.hpp / .hh /.h	C++ -Headerdatei
.o	Objektdatei
.a	Bibliothek (Library)
.c	C-Quelldatei
.h	C-Headerdatei

Ein Programmteil übersetzen: Compileren

Objektcode erzeugen:

```
$ g++ -c helloworld.cpp
$ ls
  helloworld.cpp helloworld.o
```

Text (*.cpp, *.cc) $\Rightarrow$ **Objektdatei** (*.o)

Aus **einer** Quelldatei wird **eine** Objektdatei mit direkten Steuerbefehlen erzeugt. Es kann viele Quelldateien geben.

```
#include <iostream>

int main() {
 std::cout<<"Hello world!";
}
```

helloworld.cpp

$\Rightarrow$

```
a_atexit_ZNSt8ios_base4Init
sonality_v0__ZSt3minImKT
e_main__GLOBAL__main
verify_groupingPmRKSsC0
_dsfgdf_susne
```

helloworld.o (hex-file)

Das Programm zusammenbinden: Linken

Ein ausführbares Programm erzeugen:

```
$ g++ -o MeinProgramm helloworld.o
$ ls
  helloworld.cpp helloworld.o MeinProgramm
```

Objektdateien (*.o, *.a) $\Rightarrow$ **Programm**

Aus vielen Objektdateien und Bibliotheken wird **eine** Programm erzeugt. Dies nennt man **linken** (zusammenbinden).

```
$ g++ -o MeinProgramm file1.o file2.o file3.o ...
```

Welche Instruktionen führt der Computer aus?

Assemblercode erzeugen:

```
$ g++ -S helloworld.cpp
$ ls
  helloworld.cpp helloworld.s
```

Text (*.cpp, *.cc) $\Rightarrow$ **Assemblerdatei** (*.s)

Aus einer Quelldatei wird eine Datei mit computerspezifischem Assemblercode (Maschineninstruktionen).

```
#include <iostream>

int main() {
 std::cout<<"Hello world!";
}
```

helloworld.cpp

$\Rightarrow$

```
LCFI5:
    movq    -8(%rbp), %rax
    movq    (%rax), %rax
    cmpq    %rax, %rdx
    jmp     L4
```

helloworld.s

Was macht #include?

Der #include Befehl kopiert den Text einer Datei vor dem Übersetzen an seine Stelle. Per Konvention werden die inkludierten Dateien mit ***.h, *.hpp oder *.hh** benannt.

```
#include "someCode.h"
int main() {
 std::cout<<"Hello world!";
}
```

helloworld.cpp

```
struct SomeCode {};
```

someCode.h

```
  $ g++ -E helloworld.cpp
 [ ... ]
struct SomeCode {};
# 2 ‘‘helloworld.cpp‘‘ 2
int main() {
 std :: cout << ‘‘Hello world!\n‘‘;
```

Was macht #include?

Mit dem `#include` Befehl werden üblicherweise Biblioteken und andere Dateien eingebunden, deren Funktionalität man verwenden möchte.

Beispiel: I/O Library, Daten Ein-/Ausgabe

```
#include <iostream>  // Einbinden der I/O-Bibliothek

int main() {
std::cout << "Schreibt heraus ...";
std::cout << "... und dann noch ein Absatz hier" << std::endl;
std::cout << "Kann " << "man " << "auch stueckeln...";
std::cout << "Oder Zahlen wie die Vier: " << 4 << " mit  einbinden";
int wert;
std::cin >> wert; // Lese Wert ein
}
```

Compileroptionen

Jeder Compiler hat verschiedene **Optionen**, die über die Konsolenzeile spezifiziert werden können. Diese können das Übersetzungsverhalten beeinflussen.

Optionen für den g++ beginnen mit einem Bindestrich "–".

Wichtige Optionen:

- `-c`: Compilieren, d.h. Objektdatei erzeugen
- `-o`: Linken, d.h. ausführbare Datei erzeugen
- `-std=c++11`: C++11 Standard einschalten
- `-O0` ... `-O3`: Compiler-Optimierungen einschalten
- `-g`: Debug-Modus
- `-Wall`: alle Warnungen ausgeben

Der Anfang des Programms

Jedes C++ -Programm beginnt an einer Funktion mit dem Namen `main`:

```
int main() {
 // ... das Programm beginnt hier.
}
```

main.cpp

Im Quellcode muss es genau eine solche Funktion geben.

Kommentare

Kommentare werden in C++ durch zwei Arten realisiert:

- //: Zeilenkommentar
  (Ignoriert Rest bis Ende der Zeile)
- /* ... */: Alles zwischen den Zeichen ist Kommentar

```
int main() {
 int x = 5; // ... dies hier ist ein Zeilenkommentar
 int y = 6; /* dies hier ist auch ein Kommentar */ int z = 7;
  /* Dies ist ein
  mehrzeiliger
   Kommentar */
}
```

Variablen und elementare Datentypen

Typisierte Sprache

C++ ist ein **typisierte** Sprache:

Jede Variable muss mit einem Datentyp deklariert werden.

```
{
 int eineGanzzahl = 4;
 float eineFliesskommazahl = 1.3453e-13;
 char einBuchstabe = 'a';
}
```

Variablen

Variablen werden in C++ folgendermaßen **deklariert**:

`<typ> <bezeichner1>, <bezeichner2>, ...;`

Achtung: Deklarierte Variablen haben einen **undefinierten Wert**. Dies kann gewünscht sein – man sollte sich dessen aber bewusst sein.

Man kann die Variablen auch folgendermaßen **initialisieren**:

`<typ> <name1> = <wert1>, <name2> = <wert2>, ...;`

oder auf eine dieser Möglichkeiten:

```
// ein paar integrale Zahlen (Integer):
int a;      // undefinierter Anfangswert
int b = 2;  // Wert: 2
int c {3};  // Wert: 3
int d {};   // default Wert (fÃ¼r int: 0)
```

Namenskonvention

Für die Bezeichner (von z.B. Variablen, Funktionen, . . .) gelten folgende Regeln:

- Folge von Buchstaben, Zahlen und Unterstrich ('_').
- Anfang darf keine Zahl sein.
- Groß-/Kleinschreibung wird beachtet.
- Dürfen keine Schlüsselwörter der Sprache sein.
- Prinzipiell beliebig lang (Compilerbeschränkt, z.B. <256).

```
int 1abc;       // falsch (Keine Zahl am Anfang)
int abc def;    // falsch (Leerzeichen)
int MyProperName, my_proper_name; // richtig
```

Elementare Datentypen

Typ	Bytes*	Inhalt	Werte
`void`	-	kein Typ	
`char`	1	Buchstabe	'A', 'b', ...
`bool`	1	Boolean	`false, true`
`short (int)`	2	Ganze Zahl	$-2^{15} \ldots 2^{15}$ $(= 32768)$
`int`	4		$-2^{31} \ldots 2^{31}$ $(\approx 2 \cdot 10^9)$
`long (int)`	8		$-2^{63} \ldots 2^{63}$
`float`	4	Fließkommazahl	$\pm 1.2 \cdot 10^{-38} \ldots \pm 3.4 \cdot 10^{38}$
`double`	8		$\pm 2.2 \cdot 10^{-308} \ldots \pm 1.8 \cdot 10^{308}$
`long double`	12		$\pm 3.4 \cdot 10^{-4932} \ldots \pm 1.2 \cdot 10^{4932}$

*) kann Compilerabhängig sein.

**) Integer auch als `unsigned` (z.B. `unsigned int`).

Wie groß ist der Datentyp?

Der `sizeof` Operator gibt die Größe des Datentyps in Bytes an.

```
#include <iostream>
int main(){
std::cout << "Size (int): " << sizeof(int) << std::endl;
std::cout << "Size (double): " << sizeof(double) << std::endl;
}
```

Aufgabe 4

Schreiben sie ein Programm, das die Größe der Datentypen für Ihren Compiler ermittelt.

Maschinengenauigkeit

Die Bibliothek `limits` gibt die Extremwerte an.

```
#include <iostream>
#include <limits>
int main(){
std::cout << "Max (int): " << numeric_limits<int>::max() << std::
    endl;
std::cout << "Min (int): " << numeric_limits<int>::min() << std::
    endl;
...
}
```

Aufgabe 5

Schreiben sie ein Programm, das die Extremwerte der Integertypen ermittelt.

Konstante Variablen

Das const Schlüsselwort macht eine Variable unveränderbar.

```
int main(){
 float Pi = 3.1415926;
 Pi = 3; // Korrekt: Pi kann veraendert werden

 const float pi = 3.1415926;
 pi = 3; // Fehler: pi kann nicht veraendert werden

 const float halfpi = pi/2.0; // Arithmetik bei Zuweisung erlaubt
}
```

Durch Verwenden von const können viele Fehler zur Compile-Zeit vermieden werden.

Nutzen Sie const, wenn etwas nicht verändert werden soll.

Operatoren von C++

Zuweisungsoperator

Das (einfache) “=“ Zeichen weist einen Wert zu.

```
...
int a, b, c;
a = b = c = 5;          // setzt den Wert der Variablen
a = (b = (c = 5));      // macht dasselbe
c = 5; b = c; a = b;    // macht dasselbe
...
```

Man beachte den Unterschied:

Eine **Initialisierung** ist die Wertzuweisung bei Erzeugung einer Variablen.
Eine **Zuweisung** ist die Wertzuweisung einer existierenden Variablen.

Arithmetische Operatoren

Arithmetische Operatoren:

Op.	Beschreibung	Beispiel
+	Addition	i + 5
–	Subtraktion	7 – j
*	Multiplikation	8 * 4
/	Division	9 / 3
%	Modulo	5 % 2

Kurznotation:

Op.	Kurzform	Langform
+=	i += 3	i = i + 3
–=	i –= 3	i = i – 3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3

Vergleichsoperatoren

Vergleichsoperatoren:

Op.	Beschreibung	Beispiel
>	größer	a > b
>=	größer gleich	a >= b
<	kleiner	a < b
<=	kleiner gleich	a <= b
==	gleich	a == b
!=	ungleich	a != b

Achtung bei Fließkommazahl: Rundungsfehler können auftreten! Dann ist ==, != mit Vorsicht zu verwenden.

Logische Operatoren

Logische Operatoren:

Op.	Beschreibung	Beispiel
!	logische Negation	`!(3 < 4) // false`
&&	logisches UND	`(3 > 4) && (3 < 4) // false`
||	logisches ODER	`(3 > 4) || (3 < 4) // true`

Info: Bei UND/ODER wird der rechte Operand nur ausgwertet, wenn das Ergebnis nicht schon durch den linken Operanden feststeht.

Inkrement/Dekrement

Die Operatoren ++ und -- erhöhen bzw. erniedriegen. Steht der Operator vor der Variable (prefix), so wird der Wert erst erhöht und dann die Variable verwendet. Steht er dahinter (postfix), wird der Wert der Variable erst verwendet und dann benutzt.

```
int a = 5, b = 5, c = 5;
c++;              // nun gilt: c = 6
int d = ++a;  // nun gilt: d = 6, a = 6
int e = b++;  // nun gilt: e = 5, b = 6
```

Achtung: postfix muss i.A. eine Kopie anlegen. Daher ist prefix schneller.

Mathematische Funktionen

Viele mathematische Funktionen für `float/double` finden sich in der Bibliothek `cmath`. Darunter:

`sqrt(x)`, `exp(x)`, `log(x)`, `pow(x,y)`, `fabs(x)`, `fmod(x)`, `ceil(x)`, `floor(x)`, `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`
und Konstanten: `M_E` $= e$ und `M_PI` $= \pi$.

```
#include <iostream>
#include <cmath>
int main(){
 std::cout << std::sin(2*M_PI);
}
```

Typumwandlung der Standardtypen

Von **impliziter Typumwandlung** spricht man, wenn eine Variable eines Typs mit einem Wert einen anderen Typs zugewiesen wird. Dies geht nur, wenn eine solche Umwandlung sinnvoll (und von C++ zugelassen) ist.

Bei Zahlen sind solche Umwandlungen immer möglich, wenn kein Informationsverlust auftritt. (z.B. `int` nach `double`).
Bei Informationsverlust wird meist eine Warnung ausgegeben. Man sollte soetwas aber vermeiden. (z.B. `double` nach `int`, Rundung)

```
int main(){
  int a = 1;
  double b = 4.99;

  double c = a; // ok
  int d = b;    // Wert wird implizit abgerundet: 4
  int e {d};    // Warnung: Wert wird abgerundet: 4
}
```

Blöcke und Gültigkeitsbereiche

Anweisung und Block

- Eine **Anweisung** ist ein Ausdruck gefolgt von einem Semikolon.
- Ein **Block** ist eine Gruppierung von Anweisungen.
- Ein Block wird durch die geschweiften Klammern { } gebildet.
- Wo eine Anweisung steht, kann immer auch ein Block stehen.
- Ein Block braucht kein schließendes Semikolon.
- Blöcke können geschachtelt werden.

```
int a = 4; // Eine Anweisung
{ // Block
  int b = 4;
  int c = 4; int d = 4;
}
```

Gültigkeitsbereich von Variablen

- Eine Variable ist nur nach Deklaration und innerhalb eines Blocks gültig.
- Innerhalb eines Blocks darf darf der Name nur einmal deklariert werden.
- Variablen bleiben bei inneren Blöcken sichtbar.
- Innere Blöcke können Variablen äußerer Blöcke überdecken.

```
{ // Block Anfang
  int a,b = 4;
  { // innerer Block
   int a = 3;
   std::cout << a << b; // a = 3, b = 4;
  }
  std::cout << a << b; // a = 4, b = 4;
} // Block Ende
```

Namespace

Durch **Namespaces** können Namen gruppiert werden.

- Gruppierung durch `namespace <namensraum> {...}`.
- Alle Bezeichner lauten dann `<namensraum>::<bezeichner>`.
- Durch `using namespace <namespace>;` kann man für alle Bezeichner eines Files den Namensraum angeben.
- Wichtiger Namensraum: `std` (Standard-Library)

```
namespace MyNameSpace {
 int a = 3;
}

int main(){
 int a = 4;
 std::cout << "a: " << a; // a = 4
 std::cout << "a: " << MyNameSpace::a; // a = 3
}
```

using-Deklaration

Man kann sich viel Tipparbeit sparen, indem man in einer ganzen Datei einen Namensraum verwendet. Dazu dient die `using` Anweisung.

Diese geht für einen ganzen Namensbereich:

```
using namespace std;
```

oder auch nur für ausgewählte Teile:

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

Kontrollstrukturen

Verzweigung

Die Verzweigung hat die folgende Syntax:

```
if ( <logischer Ausdruck> )
    <Anweisung>
else
    <Anweisung>
```

```
if(a > 3) a = 2;
else a = 4;

if(a > 3){
  a = 2;
} else {
  a = 4;
}
```

Fallunterscheidung

Die Fallunterscheidung hat die folgende Syntax:

```
switch ( <Ausdruck> ) {
    case <konstanter Ausdruck 1>:
        <Anweisung 1> [break;]
    ...
    case <konstanter Ausdruck N>:
        <Anweisung N> [break;]
    default:
        <Anweisung default>
}
```

```
switch(a){
  case 2: std::cout << "Fall 2"; break;
  case 1: std::cout << "Fall 1";
  case 3: std::cout << "Fall 3"; break;
  default: std::cout << "Default Fall";
}
```

Zählzyklus

Die For-Schleife hat die folgende Syntax:

```
for ( <Start>; <Condition>; <Increase> )
      <Anweisung>
```

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

Abweisender Zyklus

Die While-Schleife hat die folgende Syntax:

```
while ( <logischer Ausdruck> )
      <Anweisung>
```

```
int i = 0;
while( i < 10 ){
  std::cout << i++ << std::endl;
}
```

Nichtabweisender Zyklus

Die Do-While-Schleife hat die folgende Syntax:

```
do
      <Anweisung>
while ( <logischer Ausdruck> );
```

```
int i = 0;
do{
  std::cout << i++ << std::endl;
}while ( i < 10 );
```

break und continue

- `break`: Sofortiges Verlassen der `switch`, `while`, `do-while`, `for` Anweisung.
- `continue`: Abbruch aktueller Zyklus und Fortfahren mit nächsten Zyklus in `while`, `do-while`, `for` Anweisung.

```
int i = 0;
while( true ){
 if( i == 10 ) break;
 std::cout << i << std::endl;
}

for(int i = -10; i < 10; ++i){
 if( i < 0 ) continue;
 std::cout << i << std::endl;
}
```

Beispiele

Aufgabe 6

Schreiben Sie ein Programm, das die Zahlen von 0 bis 2 in Schritten von 0.1 herausschreibt. Machen Sie dies unter Verwendung von:

- for-Schleife
- while-do-Schleife
- do-while-Scheife

Aufgabe 7

Schreiben Sie ein Programm, das eine Zahl in eine römische Ziffer umwandelt, sofern sich diese mit einem Buchstaben schreiben lässt. Geben Sie andernfalls eine Information darüber aus, dass dies nicht möglich ist.

Benutzerdefinierte und zusammengesetze Typen

Aufzählungstypen

Enums (Enumeration) sind benutzerdefinierte Aufzählungstypen. Sie dienen dazu sinnvolle (menschenlesbare) Nummerierungen zu machen.

`enum [class] Typname { <Name1> [=<Wert1>], <Name2> ...};`

```
enum Farbe {rot = 0, blau, gelb};
int main(){
  Farbe x = rot;
  x = blau;
}
```

Strukturen

Ein **Struct** definiert einen neuen Datentyp welcher Komponenten unterschiedlichen Typs vereint.

```
struct <Name> {
      <Typ1> Name1;
      <Typ2> Name2;
      ...
};
```

```
struct Complex {
  float re; // Realteil
  double im; // Imaginaerteil
  bool isComplex;
};
int main(){
  Complex x;
  x.re = 2.2; x.im = 1.7; x.isComplex = true;
}
```

Union

Alle Komponenten einer **Union** werden auf *demselben* Speicherplatz abgelegt. (sehr selten benötigt)

```
union <Name> {
      <Typ1> Name1;
      <Typ2> Name2;
      ...
};
```

```
union Zahl {
  int i;
  float f;
  double d;
};
int main(){
  Zahl x;
  x.i = 123; x.f = 1.7; x.d = 2.3;
}
```

Array

Ein **Array** fasst Daten desselben Typs zusammen. Es lässt sich als ein Vektor betrachten.

`<Typ> <Bezeichner> [<Größe>];`

```
int main(){
  int vFeld[5]; // Array der Groesse 5
  vFeld[0] = 2; // Zugriff erstes Element
  vFeld[4] = 3; // Zugriff letztes Element
}
```

Ein Array der Größe N wird von $0, \ldots, N-1$ gezählt.
Der Zugriff eines Arrays ist über den Operator [ ].

Typedef

Mit Hilfe von typedef kann man einen Alias von einem Typ deklarieren. typedef <Typ> <TypAlias>;

```
typedef int MyInt;
typedef float Point2d[2];
typedef struct {double x,y,z;} Point3d;
int main(){
  MyInt a = 4;    // a ist ein int
  Point2d p2d:    // p2d ist 3in ein float-array der Groesse 2
  Point3d p3d;
}
```

Beispiele

Aufgabe 8

Schreiben Sie ein Programm, das ein Struct und eine Union definiert, die beide genau die folgenden Datentypen enthalten: bool, int, double. Lassen Sie von Ihrem Programm ausgeben, wie groß die Datentypen sind.

Aufgabe 9

Schreiben Sie ein Programm, das einen Vektor der Länge N mit den Zahlen 1 bis N füllt. Berechnen Sie die Norm dieses Vektors.

Referenzen und Zeiger

Referenz

Eine **Referenz** ist ein Alias (Pseudoname) für eine Variable. Sie ist ein Verweis auf die Variable und kann genauso wie diese benutzt werden. Ein Variable hat dann mehrere Namen.

In C++ werden Referenzen über ein angehängtes &-Zeichen an den Datentyp realisiert.

```
...
int a = 1;
int& b = a; // Referenz von a
b = 3; // a ist nun 3
std::cout << "Wert von a: "<< a << std::endl;
...
```

const-Referenz

Referenzen können auch als konstant deklariert werden. Dann kann man den Wert über die Referenz nicht ändern.

```
...
int a = 1;
const int& b = a; // const-Referenz von a
a = 2; // Ok: a ist nicht const
b = 3; // Fehler: die Referenz ist konstant
...
```

Pointer

Auf Variablen kann man nicht nur direkt, sondern auch über **Zeiger** (Pointer) zugreifen. Ein Pointer zeigt auf die Speicherstelle, an der die Variable gespeichert ist. Die Syntax ist:

```
<Typ>* <Bezeichner>;
```

Oder auch: `<Typ> *<Bezeichner>;` Die Speicheradresse einer Variablen

bestimmt der **Adressoperator** "&":

```
&<Bezeichner>
```

Die Variable, auf die ein Pointer zeigt, erhält man durch den **Dereferenzoperator** (Zugriffsoperator) "*":

```
*<Pointer>
```

Pointer (Beispiel)

```
{
  int i = 5, j = 10; // zwei Variablen vom Typ int
  int* p;  // ein Pointer auf ein Variable vom Typ int (
    uninitialisiert)
  p = &i; // p zeigt auf i
  std::cout << p; // ... gibt die Speicheradresse aus, auf die p
    zeigt

  std::cout << *p; // ... gibt den Wert der Variablen i aus

  p = &j; // p zeigt nun auf j
  std::cout << *p; // ... gibt nun den Wert der Variablen j aus
}
```

NULL - Pointer

Es gibt eine spezielle Zeigerkonstante 0 (NULL in C), welche auf die (hexadezimale) Speicheradresse 0x0 (= nil) verweist und bzgl. welcher eine Zeigervariable getestet werden kann. Diese dient dazu ungültigen Speicher darzustellen.

```
{
  double d = 1.345;
  double* p1 = 0, p2 = &d;

  if (p1 != 0)
    std::cout << *p1; // p1 ist kein gÃ¼ltiger Zeiger, nicht (!)
    verwenden

  if (p2 != 0)
    std::cout << *p2; // p2 ist ein gÃ¼ltiger Zeiger, kann verwendet
     werden
}
```

Zeiger und Arrays

Ein Array nutzen linearen Speicher, d.h. ein im Index nachfolgendes Element ist physisch im nachfolgenden Speicherbereich abgelegt.

Zeigervariablen lassen sich als Arraybezeichner nutzen und umgekehrt.

```
{
  int a[3] = {4,5,6};
  int* p;

  p = &a[0];          // p zeigt nun auf den Anfang des Arrays
  std::cout << p[1]; // gibt den Wert an der 2ten Position des
    Arrays

  std::cout << a[1] << p[1] << *(a+1) << *(p+1); // immer dasselbe
}
```

Zeiger auf Konstanten und konstante Zeiger

Auch Zeiger können `const` sein. Dies bedeutet, dass der Zeiger (d.h. die Speicherstelle, auf die er zeigt) nicht verändert werden kann. Dies muss man unterscheiden von Zeigern auf konstante Variablen. Bei diesen ist der Zeiger veränderbar, aber der hinterliegende Wert ist konstant.

```
{
  const int a = 1;
  int b = 2;
  const int* p1 = &a; // ok, variabler Zeiger, Wert nicht
    veraenderbar
  const int* p2 = &b; // ok, variabler Zeiger, Wert nicht
    veraenderbar
  int* p3 = &a;       // Fehler: int* auch const int

  int* const p4 = &b;    // ok, Konstanter Zeiger, Wert veraenderbar
  *p4 = 5;               // ok, Zeiger unveraendert, Wert geaendert
  p4 = &a;               // Fehler: Zeiger (Speicherstelle) nicht
    veraenderbar
```

Zeigerarithmetik

Mit Zeigern lassen sich arithmetische Operationen durchführen:

- `==`, `!=`, `<`, `>`, `<=`, `>=`: Vergleiche bzgl. der Speicherstelle
- `+` / `-`: Addition, Subtraktion (z.B. Speicherabstand)
- `p + N` := `p + N*sizeof(<typ>)`
- `++`, `--`: Inkrement, Dekrement

Mehrdimensionale Arrays

Es lassen sich auch mehrdimensionale Arrays erzeugen.

```
{
  const int NumRows = 2;
  const int NumCols = 3;
  int a[NumRows][NumCols] = {{1,2,3},{4,5,6}};

  for(int i = 0; i < NumRows; ++i){
    for(int j = 0; j < NumCols; ++j){
      std::cout << a[i][j] << ' ';
    }
    std::cout << std::endl;
  }
}
```

Speicherverwaltung

Speicher selbst anlegen

Bisher: Speicher für Variablen wird innerhalb des Blocks angelegt und am Ende wieder freigegeben.

Man kann Speicher auch selber verwalten. Der Speicher wird dann dynamisch allokiert und **muss** vom Benutzer selbst wieder freigegeben werden. (keine Garbagecollection)

Allokieren: `<Typ>* p = new <Typ>;`
Freigeben: `delete p;`

Allokieren: `<Typ>* p = new <Typ>[<Size>];`
Freigeben: `delete[] p;`

Allokation (Beispiel)

```
{
 int* a;
 {
   a = new int[10]; // Speicher wird angelegt
 }
 a[2] = 5; // Speicher immer noch gÃ¼ltig

 delete[] a; // Speicher wird freigegeben
}
```

Allokation (Achtung)

```
{
 {
   int* a = new int[10]; // Speicher wird angelegt
 }

 // Pointer a nicht mehr vorhanden, aber Speicher weiter belegt
}
```

Der Speicher **muss von Benutzer** wieder freigegeben werden!

Funktionen

Funktionen

Eine Funktion ...

- erledigt eine abgeschlossene Teilaufgabe
- macht das Programm übersichtlicher
- macht Codeabschnitt wiederverwendbar
- macht Funktionalität (z.B. als Bibliothek) auslagerbar

Definition:

```
<ReturnTyp> <FunktionsName> ( <Parameterliste> ) {
   <Anweisungen>
}
```

Deklaration/Definition einer Funktion

Die Kombination aus Funktionsname und Parameterliste wird **Signatur** genannt.

Eine Funktion muss genau einmal **definiert** werden. Die Definition einer Funktion besteht aus Signatur und Funktionsrumpf. Eine

Funktion kann beliebig oft **deklariert** werden. Die Deklaration einer Funktion besteht nur aus der Signatur.

Funktionsdefinition (Beispiel)

```
// Deklaration
int Add(int x, int y);

int main(){
  Add(3,4); // nutze die Funktion
}

// Definition
int Add(int x, int y){
 return x+y;
}
```

Damit eine Funktion verwendet werden kann, muss nur die Deklaration geben sein. Die Definition kann später erfolgen.

Überladene Funktionen

Es kann mehrere Funktionen mit demselben Funktionsnamen geben. Dann muss sich jedoch die Parameterliste unterscheiden (Returntyp reicht nicht). Die Funktion wird **überladen** genannt.

```
int Add(int x, int y);

double Add(double x, double y);  // ok, anderer Typ
int Add(int x, int y, int z);    // ok, verschieden Zahl Parameter

double Add(int x, int y);        // Fehler: nur Returntyp anders
```

Variablen in Funktionen

Für die Gültigkeits- und Sichtbarkeitsregeln für Variablen gelten diesselben wie üblich - eine Funktion ist ein Block.

Die Variablen der Parameterlist sind lokale Variablen (d.h. nur innerhalb des Blocks sichtbar).

Ausnahme: **Statische** Variablen. Ist eine Variable als `static` deklariert, so wird sie beim ersten Aufruf initialisiert und der Wert bleibt auch bei weiteren Aufrufen erhalten.

```
void print(){
  static int val = 1;
  std::cout << "val: " << val++; // Aufrufe: 1, 2, 3, ...
}
```

Call by Value

Sind die Typen in der Parameterliste kein Pointer oder Referenzen, so wird beim Aufruf die Variable **kopiert** (Call by Value).

```
void AddOne(int x){
  x += 1;
  std::cout << "x: " << x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Ausgabe: 5
}
```

Das übergebene Objekt wird **nicht verändert**.

Call by Reference

Sind die Typen in der Parameterliste Referenzen, so wird beim Aufruf die Variable die **Referenz** übergeben (Call by Reference).

```
void AddOne(int& x){
  x += 1;
  std::cout << "x: " << x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Ausgabe: 6 (!!)
}
```

Das übergebene Objekt wird **verändert**. Es hat in der Funktion nur einen anderen Namen.

Call by Pointer

Sind die Typen in der Parameterliste Pointer, so wird beim Aufruf die Variable die **Speicherstelle** übergeben.

```
void AddOne(int* x){
  *x += 1;
  std::cout << "x: " << *x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(&a);
  std::cout << "a: " << a; // Ausgabe: 6 (!!)
}
```

Das übergebene Objekt wird **verändert**. Es hat in der Funktion nur einen anderen Namen (analog zu Call by Reference).

Übergabe von Arrays

Auch Arrays können übergeben werden.

```
void AddOne(int[] x){
  x[0] += 1;
  std::cout << "x: " << x[0]; // Ausgabe: 6
}

int main(){
  int a[20]; a[0] = 5
  AddOne(a);
  std::cout << "a: " << a[0]; // Ausgabe: 6 (!!)
}
```

Übergabe von mehrdimensionalen Arrays

Auch mehrdimensionale Arrays können übergeben werden.
Pointer auf mehrdimensionale Arrays müssen immer alle Größen bis auf die erste explizit angeben.

```
void SomeFkt(int (*x)[100]){
  x[0] += 1;
  std::cout << "x: " << x[0]; // Ausgabe: 6
}

int main(){
  int a[20][100];
  int (*pArray)[100] = a; // passender Pointer
  SomeFkt(pArray);
}
```

Variable mehrdimensionale Arrays

Ist die Größe eines zweidimensionalen Arrays nicht zur Compile-Zeit bekannt, so geht man geschickterweise anders vor:
Man speichert jede Zeile der Matrix in einem 1d-Array, das man sich per `new` anlegt. Die Matrixzeilen fasst man dann in einem Pointer-Array zusammen – ebenfalls per `new` angelegt.

```
{
  int z = 3, s = 4;              // nicht konstante Weiten
  int** mat = new int*[z];       // ein Array von int* (Zeilen)
  for(int i = 0; i < z; ++i)     // fÃ¼r jede Zeile
    mat[i] = new int[s];         // allokiere Speicher fÃ¼r Zeile

  std::cout << mat[2][3];        // Zugriff
}
```

Obacht bei Call by Reference

Werden Parameter per Reference oder als Pointer übergeben, muss man sich bewusst sein, dass sie verändert werden können.

Möchte man keine Kopie machen (z.B. bei großen Objekten), diese jedoch nicht verändern lassen, so kann man sie als **`const` Referenz** übergeben. Der Compiler überprüft dann, ob dies tatsächlich der Fall ist. Der Aufrufer der Funktion kann sich also sicher sein, dass sein Objekt nicht verändert wird.

```
void AddOne(const int& x){
  x[0] += 1; // Fehler: Darf nicht veraendert werden
}

int main(){
  int a = 5
  AddOne(a);
}
```

Default Parameter

Parameter einer Funktion können mit Defaultwerten deklariert werden. Diese werden verwendet, wenn die Parameter nicht angegeben werden.

```
bool IsSmall(double x, double tol = 1e-8){
  if(x < tol) return true; else return false;
}

int main(){
  std::cout << IsSmall(1e-15); // true
  std::cout << IsSmall(1e-15, 1e-18); // false
}
```

Rückgabewerte

Der Rückgabe einer Funktion kann auch by Value oder by Reference geschehen. Bei Rückgabe per Referenz muss man darauf achten, dass die Variable auch nach der Funktion noch existiert. Daher niemals lokale Variablen per Referenz zurückgeben!

```
int& Increment_Good(int& x){
  return ++x;
}

int& Increment_Bad(int x){
  return ++x;
}

int main(){
  int a = 1;
  std::cout << Increment_Good(a); // ok
  std::cout << Increment_Bad(a);  // Fehler: RÃ¼ckgabevariable ("x")
     existiert nicht mehr
}
```

inline-Funktionen

Ein Funktionsaufruf kostet Zeit: Parameter müssen kopiert werden, das Programm springt an eine andere Stelle und nach der Funktion wieder zurück und der Stack muss angepasst werden. Diesen Aufwand möchte man gerne bei sehr kleinen Funktionen vermeiden.

Deshalb kann man eine Funktion als `inline` deklarieren. Dies schlägt dem Compiler vor, dass er diese Funktion direkt an der Stelle einsetzen soll und intern somit keinen Funktionsaufruf durchführen soll. Dies ist nur eine Empfehlung für den Compiler, wird jedoch oftmals vorgenommen.

```
inline int Add(int x, int y) { return x+y;}

int main(){
  std::cout << Add(3,4);  // inline Funktionsaufruf
}
```

Zeiger auf Funktionen

Funktionen sind ebenfalls physikalisch im Speicher abgelegt. Man kann sich folglich einen Zeiger auf eine Funktion halten.

```
int max(int a, int b) {if(a>b) return a; else return b;}
int min(int a, int b) {if(a<b) return a; else return b;}

{
  int (*pF)(int,int);  // pF ist ein Zeiger auf eine Funktion

  pF = &max;
  std::cout << (*pF)(3,4); // gibt Maximum

  pF = &min;
  std::cout << (*pF)(3,4); // gibt nun Minimum
}
```

Beispiele

Aufgabe 10

Schreiben Sie Überladungen der Funktion Subtract, die zwei Zahlen subtrahiert. Machen Sie dies für int, float und double. Testen Sie durch Ausgaben, dass tatsächlich die gewünschte Überladung aufgerufen wird.

Aufgabe 11

Schreiben Sie eine Funktion, die zwei double-Arrays übergeben bekommt und diese zusammenaddiert, falls beide dieselbe Länge haben.

Aufgabe 12

Schreiben Sie eine Funktion, die eine dynamisch allokierte Matrix übergeben bekommt und auf eine Einheitsmatrix setzt (Diagonale 1, sonst 0). Testen Sie diese Funktion durch eine Ausgabe.

Modulares Programmdesign

Modulares Programmdesign

Bei größeren Programmen stellt sich die Frage, wie das Programm sinnvoll in kleinere Teile aufgeteilt werden kann.

Grundsätzlich sollten verschiedene, nicht voneinander abhängige Funktionalitäten in **verschiedene Dateien** aufgeteilt werden. Dabei kann man durchaus ähnliche Funktionalität in einer Datei gruppieren.

Deklarationen sollten von der Implementierung abgespalten werden. Die Implementierung wird in einer *.cpp Datei umgesetzt, die Deklaration sollte in einer *.h Datei vorgenommen werden.

Andere Programmteile brauchen dann nur den sogenannten Header (*.h) einbinden, d.h. nur die Deklaration.

Beispiel

```
int Add(int x, int y);
```

add.h

```
#include "add.h"
int Add(int x, int y){
return x+y;
}
```

add.cpp

```
#include "add.h"
int main(){
std::cout << Add(5,6);
}
```

myProc.cpp

Vorteil:

- Programm ist modular aufgebaut
- Programmteile können einzeln übersetzt werden
- Implementierungen können ausgetauscht werden
- ilierzeiten werden reduziert

Include-Guards

Oftmals inkludieren Dateien viele andere *.h Dateien und auch Header inkludieren weitere Header. Damit am Ende nicht die Deklarationen oft kompiliert werden, hat es sich eingebürgert sogenannte **Include-Guards** zu verwenden. Diese sorgen dafür, dass jeder Header nur einmal inkludiert wird.

```
#ifndef __H__SOME_HEADER__
#define __H__SOME_HEADER__

// .. Deklarationen stehen hier

#endif // end __H__SOME_HEADER__
```

someHeader.h

Die Wahl der Include-Guards muss eindeutig sein.

Beispiele

Aufgabe 13

Lagern Sie Ihre Funktionen Subtract und Add in externe Dateien aus. Verwenden Sie diese zum Test in einer separaten Datei, in der die main-Funktion definiert ist.

Aufgabe 14

Überlegen Sie sich, wie `inline`-Funktionen in einem modularen Konzept umgesetzt werden müssen.