Software-Technik: Vom Programmierer zur erfolgreichen …

1. Von der Idee zur Software
2. Funktionen und Datenstrukturen
3. Organisation des Quellcodes
4. Werte- und Referenzsemantik
5. Entwurf von Algorithmen
6. Fehlersuche und –behandlung
7. Software-Entwicklung im Team
8. Abstrakte Datentypen: Einheit von Daten und Funktionalität
9. Vielgestaltigkeit (Polymorphie)
10. Entwurfsprinzipien für Software

Anhang A: Die Familie der C-Sprachen

Anhang B: Grundlagen der C++ und der Java-Programmierung

Software-Technik: Vom Programmierer zur erfolgreichen …

3 Organisation des Quellcodes
3.1 Modularisierung auf Dateiebene
3.2 Strukturierung jenseits von Dateigrenzen
3.3 Bibliotheken
3.4 Build-Management
3.5 Zusammenfassung

Lehrbuch: 1.8
Kompendium, 4. Auflage: 1.10

Folien mit gelben Punkten am oberen rechten Rand sind weniger wichtig für das Verständnis der nachfolgenden Kapitel.

Modularisierung auf Dateiebene

Die ersten Programme sind Ein-Dateien-Programme.

Je größer die Software wird, desto unpraktischer ist es, alles in einer Datei zu halten.

Viel praktischer ist es, Ordner und Dateien als Ordnungsmittel zu nutzen.

Aber auch aus technischer Sicht ist die Aufteilung in kleinere Portionen sinnvoll: Oft werden bei der Erweiterung eines Programms nur sehr lokal Änderungen vorgenommen.

Warum die unveränderten 99% jedes Mal neu compilieren? Das kostet unnötig Zeit.

Modularisierung auf Dateiebene (2)

Während wir bisher meistens Funktionen aufgerufen haben, die vorher in *derselben* Datei definiert wurden, wollen wir nun auch Funktionen aufrufen, die in *anderen* Dateien definiert wurden.

In C++ besteht die Lösung dieses Problems in der Wiederholung der Funktionsdeklarationen in der so genannten Header-Datei (auch H-Datei, Definitions-Datei, Schnittstellen-Datei genannt).

Diese Header-Dateien sind viel kürzer als die Quellcode-Dateien, denn sie enthalten nicht den Quelltext der Funktionen, sondern nur die Funktionsprototypen.

Durch das Nachladen (engl. *include*) dieser Header-Dateien wird der gewünschte Effizienzgewinn erzielt.

Modularisierung auf Dateiebene (3)

Problematisch an dieser Lösung ist, dass der Entwickler für die Konsistenz von Header- und Quelltext-Datei selbst verantwortlich ist (Prototyp in Header-Datei muss mit Funktionsdefinition in Quelltext-Datei übereinstimmen), es wird dem Entwickler Mehraufwand aufgebürdet.

In Java hat man daher einen anderen Weg gewählt, aus den Objekt-Dateien selbst werden die darin definierten Funktionalitäten ausgelesen, der Entwickler wird nicht weiter belastet.

Linker

Durch die Aufteilung des Objektcodes auf mehrere Dateien entsteht außerdem der Bedarf für einen weiteren Schritt in der Programmerstellung, nämlich das Zusammenfügen der verschiedenen Objektcode-Dateien zu einem Programm.

Diese Aufgabe übernimmt ein so genannter Linker. (Man spricht vom *Binden* der Objektdateien.)

Beispiel

Header-Datei „sum.h“ mit Schnittstellte

```
int summiere(const int arr[], int n);
```

Anwendung „anw.cxx“

```
#include "sum.h"
void f(...) {
...
   summiere(array,4);
...
}
```

Quellcode-Datei „sum.cxx“

```
#include "sum.h"

int summiere(const int arr[], int n) {
  int sum = 0;
  for (int i=0;i<n;++i) sum+=arr[i];
  return sum;
}
```

Aufbau und Verwendung von Headerdateien

A.c:
```
#include<iostream> using namespace std;
#include"B.h"
#include"C.h"
int main(void) {
  cout << 'A'; fB(); fC(); cout << endl; return 0;
}
```

B.h:
```
#ifndef _B_h
#define _B_h
void fB(void);
#endif
```

B.c:
```
#include<iostream> using namespace std;
#include"B.h"
#include"C.h"
void fB(void) { cout << 'B'; fC(); }
```

C.h:
```
#ifndef _C_h
#define _C_h
void fC(void);
#endif
```

C.c:
```
#include<iostream> using namespace std;
#include"C.h"
void fC(void) {cout << 'C';}
```

Problem der verschachtelten Include-Anweisungen

Syntaxfehler, da Mehrfachdefinition

```
const double PI = 3.14;
const double PI = 3.14;

int main() {
   double f = PI;
   }
```

Wie verhindert man das Problem, dass die Datei <cmath> **nicht** mehrfach indirekt eingebunden wird?

Datei: cmath

```
const double PI = 3.14;
```

Datei: main.cxx

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

int main() {
   double f = PI;
   }
```

Problem der verschachtelten Include-Anweisungen (2)

matrix.h:

```
#include <cmath>
#include "global.h“
```

ingenieur.h:

```
#include "matrix.h“
#include <cmath>
```

Brueckenberech.h

```
#include "matrix.h“
#include <cmath>
#include "ingenieur.h“
```

global.h:

```
#ifndef GLOBAL_H
#define GLOBAL_H
...
#endif
```

ingenieur.h:

```
#ifndef INGENI_H
#define INGENI_H
#include "matrix.h“
#include <cmath>
...
#endif
```

cmath:

```
#ifndef CMATH_H
#define CMATH_H
...
#endif
```

matrix.h:

```
#ifndef MATRIX_H
#define MATRIX_H
#include <cmath>
#include "global.h“
...
#endif
```

brueckenberech.h:

```
#ifndef BR_BER
#define BR_BER
#include "matrix.h“
#include <cmath>
#include "ingenieur.h“
...
#endif
```

Anwendung in der Netzplanung

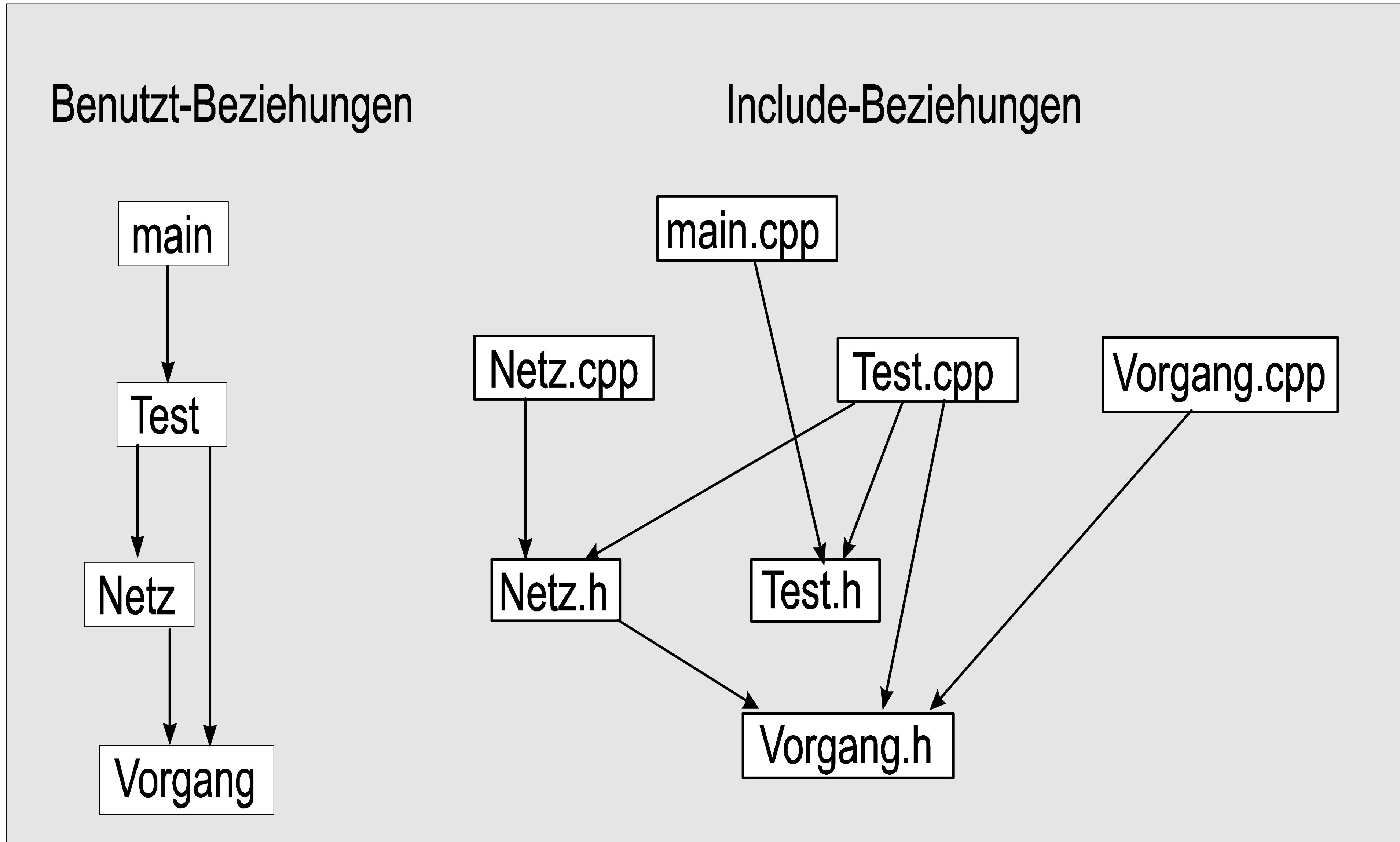
Vorgang.h

```
#ifndef VORGANG_HEADER
#define VORGANG_HEADER

struct Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
void initVorgang(Vorgang* v);
void ausgabeVorgang(const Vorgang* v);
bool istDurchfuehrbar(Vorgang *v);
void passeFruehanfAn(
         Vorgang* v, const Vorgang* vv);
void passeSpaetEndAn(
         Vorgang* v, const Vorgang* vn);

#endif /* VORGANG_HEADER */
```

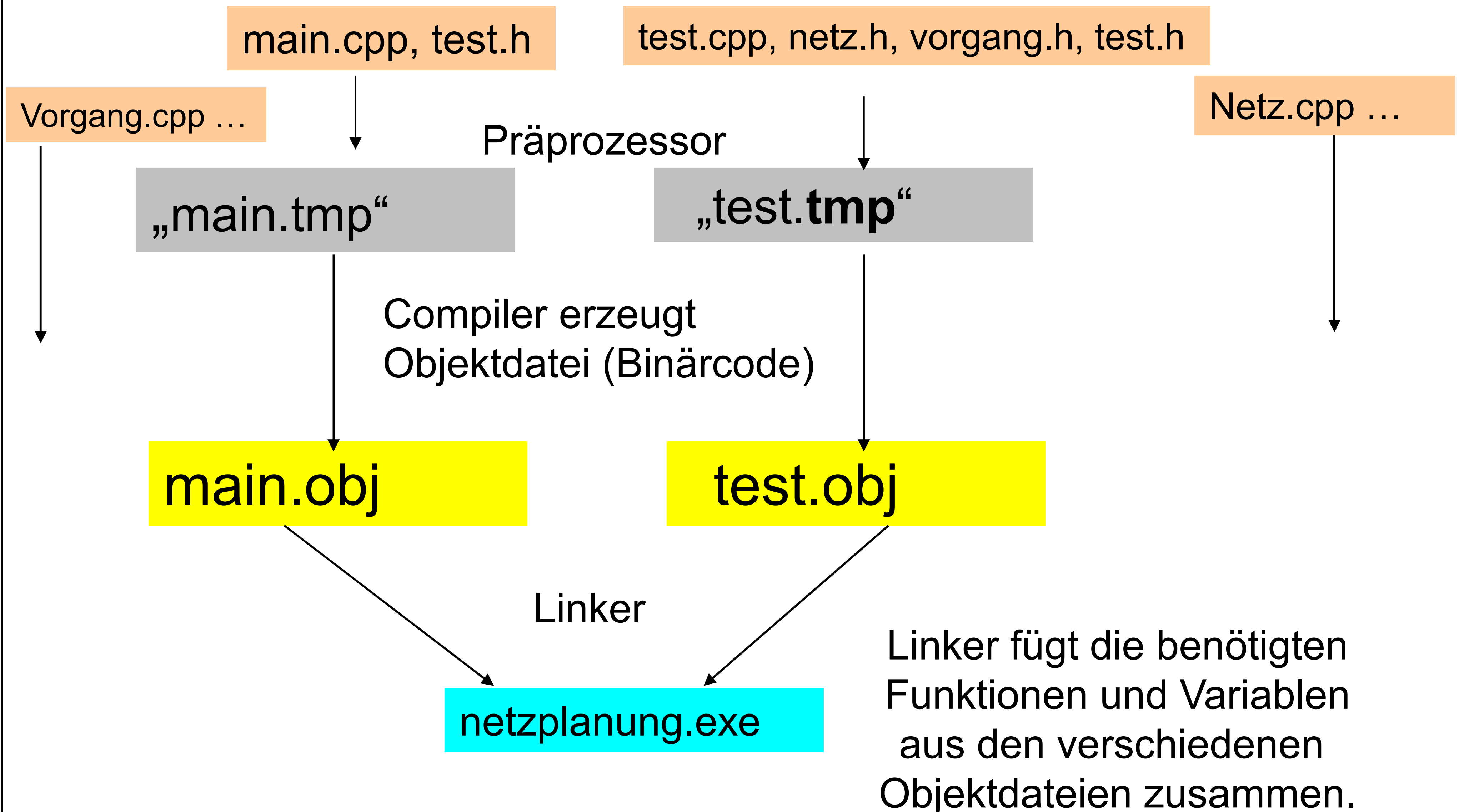
Include-Wächter,

Siehe Kompendium oder SW-Technik

Anwendung in der Netzplanung



Präprozessor, Compiler und Linker



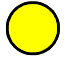
Erklärung/Demo cmake

Siehe
„..\Betreuer\Uebungen\CodeDerAufgabenLoesungen\ProjektErzeugungMitCmake.pptx"

Software-Technik: Vom Programmierer zur erfolgreichen …

3 Organisation des Quellcodes
3.1 Modularisierung auf Dateiebene
3.2 Strukturierung jenseits von Dateigrenzen
3.3 Bibliotheken
3.4 Build-Management
3.5 Zusammenfassung

Lehrbuch: 2,2
Kompendium, 4. Auflage: 3.6



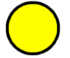
Sichtbarkeitsbereiche in ANSI/ISO-C

- Datei
- Funktion
- Block
- Funktionsprototyp / -deklaration
  f(int a; int b = a; int c = b);

In C++ gibt es daneben noch den Sichtbarkeitsbereich
der Klasse (class, struct, union)

Sowie den Namensraum

Hiermit kann der globale Sichtbarkeitsbereich (außerhalb von Funktionen) strukturiert werden.



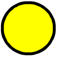
Namespaces -- Motivation

In verschiedenen Projekten wird jeweils eine mathematische Bibliothek aufgebaut mit Funktionen wie sin, cos, tan ...

CAD.h
```
double sin(double x);
double cos(double x);
```

flugfuehrung.h
```
double sin(double y);
double cos(double y);
double cosh(double y);
```

neueAnwendung.cxx
```
#include "CAD.h"
#include "flugfuehrung.h"


/* sin und cos sind doppelt
   definiert.
   Compilerfehler */
sin(1.57);
```



Namespaces – Motivation (2) Lösung

Durch die Einführung von Namensräumen wird das Problem vermieden.

CAD.h

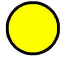
```
namespace CAD {
double sin(double x);
double cos(double x);
}
```

flugfuehrung.h

```
namespace FF {
double sin(double y);
double cos(double y);
double cosh(double y);
}
```

neueAnwendung.cxx

```
#include "CAD.h"
#include "flugfuehrung.h"


/* sin und cos sind nun
   jeweils unterscheidbar */
FF::sin(1.57);
```

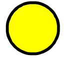


Konstrukte für Namensbereiche

namespace identifier {declaration-sep}
dient zur Zusammenfassung aller Deklarationen in diesem Namensraum.

```
namespace Math
{
const double PI = 3.14;
double sin(double x) {...}
double cos(double x) {...}
double tan(double x)
   {return (sin(x) / cos(x)); }
}
```

```
#include <cmath>

const double PI= 3.14159;
int main()
{
cout << PI << Math::PI;
cout << Math::sin(PI)
     << sin(Math::PI);
}
```



Konstrukte für Namensbereiche (2)

namespace org-namespace-name {declaration-sep}

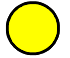
dient zur Erweiterung eines vorhandenen Namensraums

```
namespace Math {
   const double e=2.7182;
}
```

Auch der Standard-Namensraum (std) kann so erweitert werden.

```
namespace std
{
   struct Stack {...};
}
```

Dieses Konstrukt wird hauptsächlich verwendet, um einen Namensraum auf mehrere Dateien zu verteilen.

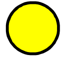


Konstrukte für Namensbereiche (3)

namespace identifier = namespace-name;

dient zur Einführung von Alias (Zweitnamen), z.B. Abkürzungen für bestehende Namensräume

```
namespace MeineZufallsZahlenBibliothek { ...}
namespace MZB = MeineZufallsZahlenBibliothek;
```



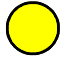
Konstrukte für Namensbereiche (4)

using namespace namespace-name;

Hiermit kann der bezeichnete Namensraum auf den aktuellen erweitert werden, d.h. alle Bezeichner des angegebenen Namensraums können im aktuellen ohne Qualifizierer verwendet werden, z.B.

```
namespace Math
{
const double PI = 3.14;
double sin(double x) {...}
double cos(double x) {...}
double tan(double x)
   {return (sin(x) / cos(x)); }
}
```

```
using namespace Math;
// ab hier kann man sin und cos
// anstatt Math::sin und Math::cos
// verwenden
```

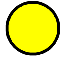


Konstrukte für Namensbereiche (4-2)

Das using-Konstrukt wird meist verwendet, um den Standardnamensraum std verwenden zu können.

```
#include <list>
#include <iostream>
#include <cstdlib>
using namespace std;
```

Hiermit können wir also alle Deklarationen der C++-Standardbibliothek, die im Namensraum std liegen, wie früher unqualifiziert benutzen.



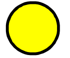
Konstrukte für Namensbereiche (5)

using unqualified-id;

Mit diesem Konstrukt können gezielt einzelne Deklarationen eines Namensbereichs entqualifiziert werden.

```
namespace Math
{
const double PI = 3.14;
double sin(double x) {...}
double cos(double x) {...}
double tan(double x)
   {return (sin(x) / cos(x)); }
}
```

```
using Math::sin;
using Math::cos;

cout << sin(Math::PI) << cos(2);
cout << tan(4); // Syntaxfehler
```



Konstrukte für Namensbereiche (6)

Anonymer Namensbereich: namespace {...}

Mit diesem Konstrukt kann der Sichtbarkeitsbereich von Deklarationen auf die aktuelle Übersetzungseinheit (Datei) beschränkt werden, sodass globale static Deklarationen überflüssig sind.

```
int random;
namespace{
   const int MAX_SPIELER=14;
   void PrintName() {...}
}
void InitRandom() {...}
```

MAX_SPIELER und PrintName sind nur in der aktuellen Datei sichtbar.
Sie werden jedoch ohne Präfix benutzt.

random und InitRandom sind ggf. auch in anderen Übersetzungs-einheiten sichtbar.

Gemeinsame Aufgabe

Teilen Sie den Code zur Implementierung des struct Vektor auf verschiedene Dateien Vektor.h, Vektor.cxx; Test.h; Test.cxx und main.cxx auf.

Verwenden Sie Namensräume.

Für die Schnellen: Entsprechendes für struct Matrix

überspringen