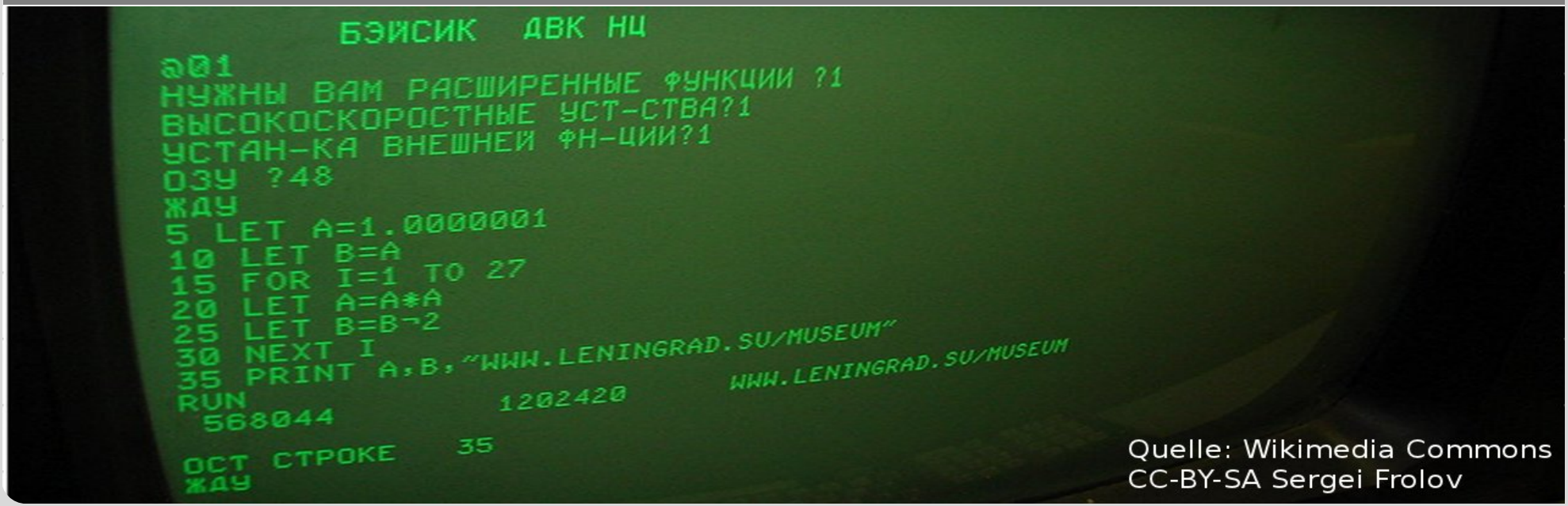
Karlsruher Institut für Technologie

C++ Workshop

4. Block, 24.05.2012

Christian Käser, Robert Schneider | 26. Mai 2012



Gliederung

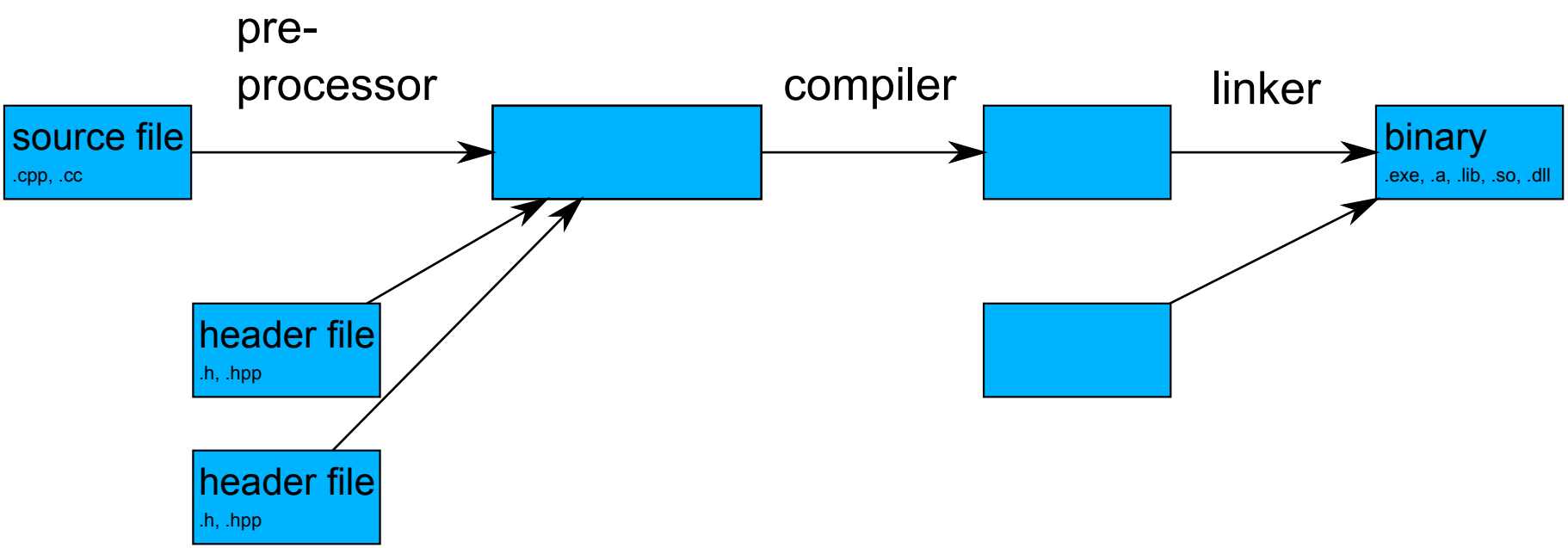
1. Übersetzungs-Phasen
   - Der Präprozessor
   - Der Compiler
   - Der Linker

Übersetzungs-Phasen (phases of translation)

Der Code durchläuft im Wesentlichen drei Arbeitsschritte:

1. Präprozessor
2. Compiler
3. Linker

Übersetzungs-Phasen



Präprozessor

Der Präprozessor ersetzt Teile des Quelltexts nach bestimmten Regeln.

- Einfügen von Dateiinhalten `#include`
  (der Präprozessor wird auch für die eingefügte Datei ausgeführt!)
- Suchen & Ersetzen `#define`
- Bedingtes Einfügen `#if`
- Abbrechen der Übersetzung `#error`

Präprozessor

Warum?

- Aufteilung des Projekts auf mehrere Dateien
- Verschiedene Build-Konfigurationen (z.B. Debug/Release)
- Unterschiedlicher Code für unterschiedliche Plattformen
- Automatische Codeerzeugung
- ...

Zusammenhang mit der Übersetzung

- Der Präprozessor wird durch Direktiven (Anweisungen) innerhalb des Quellcodes programmiert.
- Präprozessor-Direktiven stehen in einer eigenen Zeile und beginnen mit einem #
- Die Quellcode-Dateien dienen zugleich als Eingabe für den Präprozessor. Die Ausgabe geht dann in die nächste Übersetzungsphase.
- Nach Präprozessor-Direktiven steht *kein* Semikolon ;

Zusammenhang mit der Übersetzung

- Der Präprozessor wird durch Direktiven (Anweisungen) innerhalb des Quellcodes programmiert.
- Präprozessor-Direktiven stehen in einer eigenen Zeile und beginnen mit einem #
- Die Quellcode-Dateien dienen zugleich als Eingabe für den Präprozessor. Die Ausgabe geht dann in die nächste Übersetzungsphase.
- Nach Präprozessor-Direktiven steht *kein* Semikolon ;

Zeilen, die nicht von Präprozessor-Direktiven betroffen sind, werden vom Präprozessor unverändert ausgegeben.

Die include-Direktive

include-Direktive

`#include "`*dateiname*`"`

Ersetzt diese Zeile durch den Inhalt der Datei *dateiname*.
Der Inhalt der Datei wird ebenfalls vom Präprozessor verarbeitet.

Die include-Direktive

include-Direktive

`#include "`*dateiname*`"`

Ersetzt diese Zeile durch den Inhalt der Datei *dateiname*.
Der Inhalt der Datei wird ebenfalls vom Präprozessor verarbeitet.

Es gibt noch die Variante `#include <`*dateiname*`>`.

Die include-Direktive

Gängige Compiler unterscheiden zwischen beiden Varianten folgendermaßen:

- Die "-Variante ist für Dateien des eigenen Projekts. Der angegebene Pfad wird relativ zum Pfad der aktuellen Datei interpretiert.
- Die <>-Variante ist für Dateien aus fremden Bibliotheken. Sie werden an vorher festgelegten Orten gesucht. (z.B. für die Std-Lib, boost, Qt usw.)

Beispiel: include

myheader.h

```
int square(int p)
{
        return p*p;
}
```

main.cpp

```
#include "myheader.h"

int main()
{
        int i = square(42);
}
```

Beispiel: include

Ergebnis

```
int square(int p)
{
        return p*p;
}

int main()
{
        int i = square(42);
}
```

Die define-Direktive

define-Direktive

```
#define NAME TOKEN0 TOKEN1.....
```

Definiert ein Makro mit dem Namen `NAME`, die darauf folgenden Token sind optional. Trifft der Präprozessor nach der Definition eines Macros auf das Token `NAME`, so ersetzt er es durch die Token `TOKEN0 TOKEN1.....`

Beispiel: define-Direktive

```
#include <iostream>

#define MYMACRO myInt

int main()
{
        int MYMACRO = 42;
        int MYMACRO_NOT = 3;

        if(42 == MYMACRO)
        {
                std::cout << myInt << "MYMACRO";
        }
}
```

Die define-Direktive

Makros können auch Parameter haben (function-like macro):

```
#define SQUARE(x) (x*x)
#define CUBE(VAR) (VAR*VAR*VAR)
#define DIST(x, y) std::sqrt(SQUARE(x) + SQUARE(y))

int main()
{
        int foo = SQUARE(42);

        int myX = 5;
        int myY = 3;
        int myDist = DIST(myX, myY);
}
```

ACHTUNG! Reine Textersetzung!

Die define-Direktive

Makros können auch Parameter haben (function-like macro):

```
#define MIN(x, y) (x < y) ? x : y
#define MIN2(x, y) ((x < y) ? x : y)

int main()
{
        int foo = MIN(23, 42) + 5;
        int bar = MIN2(23, 42) + 5;
        return 0;
}
```

ACHTUNG! Reine Textersetzung!

Die define-Direktive

Wann benutze ich Makros bzw. `#define`?

Die define-Direktive

Wann benutze ich Makros bzw. `#define`?

So selten wie möglich!

Stattdessen lieber typedef, Konstanten, Funktionen, etc.

Bedingtes Einfügen

if-, else- und endif-Direktiven

```
#if constant_expression_0
function_variation1();
#else
function_variation2();
some_other_function();
#endif
```

Abhängig vom Wert von `constant_expression_0` wird entweder der Code zwischen `#if` und `#else` oder zwischen `#else` und `#endif` eingefügt.

Beispiel: Bedingtes Einfügen

```
#if defined(WIN32)
#include <winsock2.h>
#else
#include <sys/net.h>
// and more headers
#endif
```

Beispiel: Bedingtes Einfügen

```
#ifdef WIN32
#include <winsock2.h>
#else
#include <sys/net.h>
// and more headers
#endif
```

Beispiel: Bedingtes Einfügen

```
#include <iostream>

#define MY_ONE 1

int main()
{
#if 1 != MY_ONE
        std::cout << "MY_ONE is NOT equal to 1";
#else
        std::cout << "MY_ONE is equal to 1";
#endif
}
```

Der Compiler

Übersetzt von Quellcode in Maschinensprache.

Alles was wir in vorigen Workshops behandelt haben, beschreibt gültige Eingaben für den Compiler (Quellcode).

(Da C++ kompliziert ist, ist es weder einfach noch hilfreich, eine Übersicht über das zu geben, wie seine Eingabe und Ausgabe zusammenhängt.)

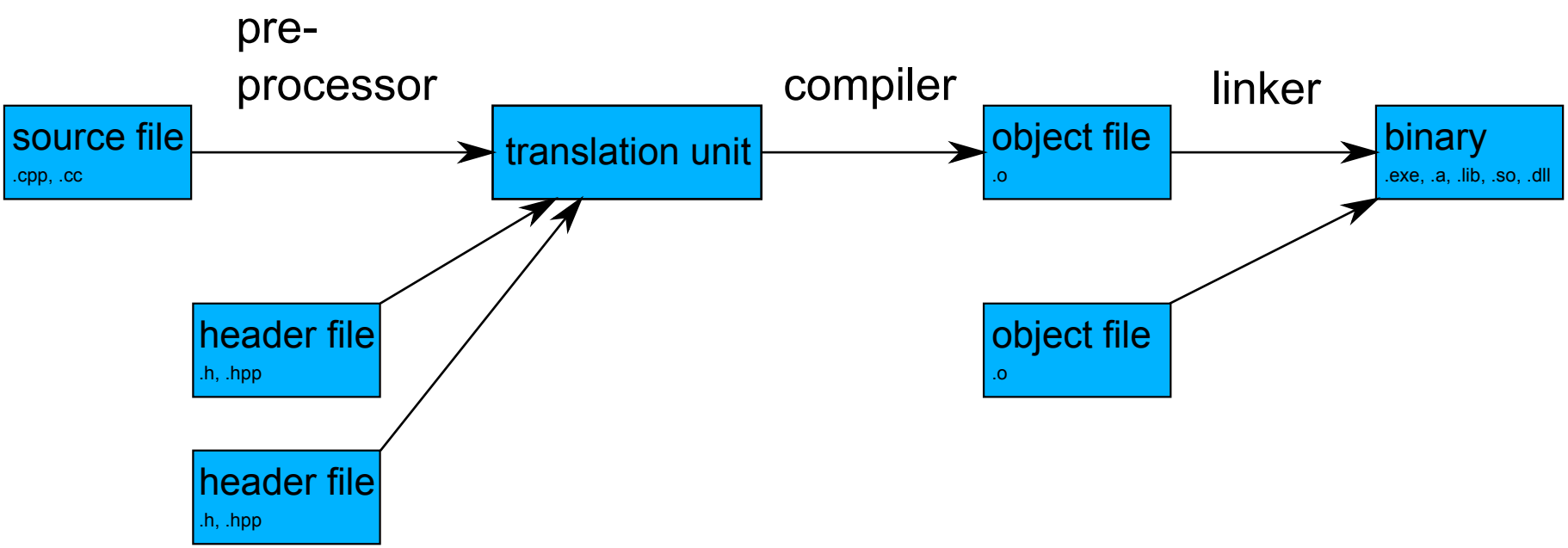
Die translation unit

Bisher wurden sowohl vom Präprozessor als auch vom Compiler alle .cpp Dateien getrennt voneinander behandelt.
Der Präprozessor arbeite nun auf einer .cpp-Datei. Er fügt dann Weiteres ein bzw. ersetzt (`#include`, `#define`) und lässt manches aus (`#if`, `#else`).

translation unit

Die Ausgabe des Präprozessors ist dann also die source file (.cpp) plus alle eingebundenen Dateien abzüglich den Auslassungen und mit den Macro-Ersetzungen.
Das Entstandene nennt man *translation unit*.

Übersetzungs-Phasen



Linkage

Eine translation unit wird dann vom Compiler tatsächlich *übersetzt*, also kompiliert. Es werden alle translation units des Projektes übersetzt und anschließend verknüpft (gelinkt).

linkage (Standard, 3.5:2)

Hat in einer translation unit der Name eines ≫Dinges≪, Referenz, Funktion, Typ, template oder namespace *external linkage*, so darf auf das Etwas, das der Name benennt, auch in anderen translation units zugegriffen werden.

Im Normalfall hat ein Name in einem namespace (also nicht innerhalb einer Klassen oder Funktion) external linkage.

Linkage

Eine translation unit wird dann vom Compiler tatsächlich *übersetzt*, also kompiliert. Es werden alle translation units des Projektes übersetzt und anschließend verknüpft (gelinkt).

linkage (Standard, 3.5:2)

Hat in einer translation unit der Name eines ≫Dinges≪, Referenz, Funktion, Typ, template oder namespace *external linkage*, so darf auf das Etwas, das der Name benennt, auch in anderen translation units zugegriffen werden.

Im Normalfall hat ein Name in einem namespace (also nicht innerhalb einer Klassen oder Funktion) external linkage.

Aber: Da zuvor schon kompiliert wird, muss dem Compiler gesagt werden, dass dieses Etwas überhaupt existiert.

Siehe: `https://github.com/downloads/kit-cpp-workshop/workshop-ss12-03/addendum-header.pdf`

Beispiel: external linkage

translation unit *square.cpp*

```
int square(int p)
{
        return p*p;
}
```

Beispiel: external linkage

translation unit *square.cpp*

```
int square(int p)
{
        return p*p;
}
```

translation unit *main.cpp*

```
int square(int);

int main()
{
        int i = square(42);
}
```

Klassen und Linkage

member functions von Klassen (Methoden) sind auch nur Funktionen, d.h. sie dürfen auch in anderen translation units verwendet werden.
Dem Namen einer Klassen selbst schreibt man auch linkage zu. Hat der Name einer Klasse external linkage, so bedeutet dies nur, dass die Namen der member functions external linkage haben.

Klassen und Linkage

member functions von Klassen (Methoden) sind auch nur Funktionen, d.h. sie dürfen auch in anderen translation units verwendet werden.
Dem Namen einer Klassen selbst schreibt man auch linkage zu. Hat der Name einer Klasse external linkage, so bedeutet dies nur, dass die Namen der member functions external linkage haben.

translation unit *square.cpp*

```
struct Square
{
        int compute(int);
};

int Square::compute(int p)
{
        return p*p;
}
```

Klassen und Linkage

member functions von Klassen (Methoden) sind auch nur Funktionen, d.h. sie dürfen auch in anderen translation units verwendet werden.
Dem Namen einer Klassen selbst schreibt man auch linkage zu. Hat der Name einer Klasse external linkage, so bedeutet dies nur, dass die Namen der member functions external linkage haben.

translation unit *square.cpp*

```
struct Square
{
        int compute(int);
};

int Square::compute(int p)
{
        return p*p;
}
```

translation unit *main.cpp*

```
struct Square
{
        int compute(int);
};

int main()
{
        Square s;
        int i = s.compute(42);
}
```

Linkage-Modifier

Im Normalfall hat ein Name in einem namespace (also nicht innerhalb einer Klassen oder Funktion) external linkage.

Man kann die Linkage durch die Schlüsselwörter `extern` und `static` sowie durch unbenannte namespaces beeinflussen:

Linkage-Modifier

Im Normalfall hat ein Name in einem namespace (also nicht innerhalb einer Klassen oder Funktion) external linkage.

Man kann die Linkage durch die Schlüsselwörter `extern` und `static` sowie durch unbenannte namespaces beeinflussen:

static im Bezug auf Linkage

Schreibt man vor die Definition eines ≫Dinges≪, einer Referenz oder einer Funktion `static`, so hat der Name *internal linkage*.

Linkage-Modifier

Im Normalfall hat ein Name in einem namespace (also nicht innerhalb einer Klassen oder Funktion) external linkage.

Man kann die Linkage durch die Schlüsselwörter `extern` und `static` sowie durch unbenannte namespaces beeinflussen:

static im Bezug auf Linkage

Schreibt man vor die Definition eines »Dinges«, einer Referenz oder einer Funktion `static`, so hat der Name *internal linkage*.

unnamed namespaces

Alles innerhalb eines unbenannten namespaces `namespace { /*...*/ }` hat erst einmal *internal linkage*. Dies „vererbt“ sich bspw. auf alle darinnen deklarierten Funktionen.

Beispiel: Linkage-Modifier

```
int square(int);

static int cube(int);

namespace bmp
{
    bool check(int, int);

    namespace
    {
        void print();
        static void test();
    }
}
```

Beispiel: Linkage-Modifier

```
int square(int);                // global namespace  => external linkage

static int cube(int);           // static            => internal linkage

namespace bmp
{
    bool check(int, int);       // (named) namespace => external linkage

    namespace
    {
        void print();           // unnamed namespace => internal linkage
        static void test();     // static            => internal linkage
    }
}
```

2 Modularisierung

Aufteilung in Dateien

Projekte werden schnell zu groß für einzelne Dateien.

Wie fein unterteilen?

- Eine Datei pro Funktion/Methode?
- Eine Datei pro Klasse?
- Eine Datei pro Namespace?
- Ganz anders?

Aufteilung in Dateien

Faustregel

- Namespaces entsprechen Ordnern
- Klassen entsprechen Dateien
- Gegebenenfalls zusätzliche Dateien für globale Makros

Source und Header

In C++ wird in der Regel zwischen Source-Dateien (.cpp) und Header-Dateien (.h oder .hpp) unterschieden.

- Header-Dateien enthalten Klassen- und Funktionsprototypen
- Source-Dateien enthalten die Implementierung dazu

Beispiel: Source und Header

person.h

```
class Person
{
private:
        std::string firstname
        std::string lastname;
public:
        std::string getName();
};
```

person.cpp

```
#include "person.h"

std::string Person::getName()
{
        return firstname
        + " " + lastname;
}
```

Inkludieren von Headern in Header

Freilich darf man auch in Header-Dateien `#include`-Direktiven stehen haben. Man verwendet dies bspw., wenn man einen Typen in einem Header benötigt:

pair.h

```
struct pair
{
    int foo
    int bar;
};
```

Inkludieren von Headern in Header

Freilich darf man auch in Header-Dateien `#include`-Direktiven stehen haben. Man verwendet dies bspw., wenn man einen Typen in einem Header benötigt:

pair.h

```
struct pair
{
    int foo
    int bar;
};
```

MyClass.h

```
#include "pair.h"

int compute(pair);

struct MyClass
{
    pair member;
};
```

Inkludieren von Headern in Header

Freilich darf man auch in Header-Dateien `#include`-Direktiven stehen haben. Man verwendet dies bspw., wenn man einen Typen in einem Header benötigt:

pair.h

```
struct pair
{
    int foo
    int bar;
};
```

MyClass.h

```
#include "pair.h"

int compute(pair);

struct MyClass
{
    pair member;
};
```

program.cpp

```
#include "pair.h"
#include "MyClass.h"

int main()
{
    MyClass m;
    compute(m.member);
}
```

Include Guards

Lösung: in jedem Header ein Makro definieren, das anzeigt, ob die Datei schon eingebunden wurde.

pair.h

```
#ifndef PAIR_H
#define PAIR_H

struct pair
{
    int foo
    int bar;
};

#endif
```

Include Guards

Lösung: in jedem Header ein Makro definieren, das anzeigt, ob die Datei schon eingebunden wurde.

pair.h

```
#ifndef PAIR_H
#define PAIR_H

struct pair
{
    int foo
    int bar;
};

#endif
```

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include "pair.h"

int compute(pair);

struct MyClass
{
    pair member;
};

#endif
```

Include Guards

Lösung: in jedem Header ein Makro definieren, das anzeigt, ob die Datei schon eingebunden wurde.

pair.h

```
#ifndef PAIR_H
#define PAIR_H

struct pair
{
    int foo
    int bar;
};

#endif
```

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include "pair.h"

int compute(pair);

struct MyClass
{
    pair member;
};

#endif
```

program.cpp

```
#include "pair.h"
#include "MyClass.h"

int main()
{
    MyClass m;
    compute(m.member);
}
```

3 Praxis

Towel-Day!

Quelle: Wikimedia Commons CC-BY-SA 3.0 Beny Shlevich

Praxis!

- Aufgabe 1: Ein einfaches Schachspiel
- Aufgabe 2: Mehr Features

`https://github.com/kit-cpp-workshop/workshop-ss12-04`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`