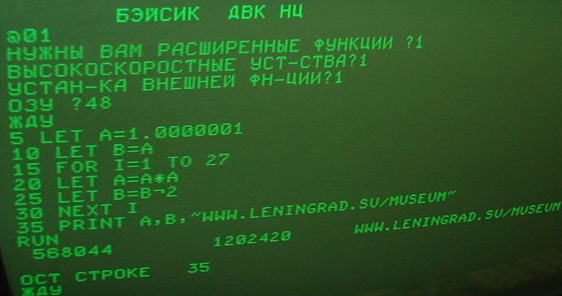


C++ Workshop

Addendum zum 3. Block: Header, 21.05.2012

Robert Schneider | 23. Mai 2012



```
БЭЙСИК  ДВК  НЧ
001
НУЖНЫ ВАМ РАСШИРЕННЫЕ ФУНКЦИИ ?1
ВЫСОКОСКОРОСТНЫЕ УСТ-СТВА?1
УСТАН-КА ВНЕШНЕЙ ФН-ЦИИ?1
03У ?48
ЖДУ
5 LET A=1.0000001
10 LET B=A
15 FOR I=1 TO 27
20 LET A=A*A
25 LET B=B-2
30 NEXT I
35 PRINT A,B,"WWW.LENINGRAD.SU/MUSEUM"
RUN
568044 1202420 WWW.LENINGRAD.SU/MUSEUM
ОСТ СТРОКЕ 35
ЖДУ
```

Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

1 Translation units

2 Include-Guards

1 Translation units

Der Compiler – also das Programm, welches den Quellcode liest, versteht und aufbereitet – arbeitet typischerweise auf einzelnen *translation units*. Wie der Name schon sagt, werden die translation units jede für sich vom Compiler übersetzt.

Der Compiler – also das Programm, welches den Quellcode liest, versteht und aufbereitet – arbeitet typischerweise auf einzelnen *translation units*. Wie der Name schon sagt, werden die translation units jede für sich vom Compiler übersetzt.

Translation unit (Standard, 2:1)

Der Programmtext (Quellcode) wird in sog. *source files* gehalten (typisch: .cpp, .cc). Eine *source file* zusammen mit allen Headern und eingebundenen *source files*, abzüglich aller bedingten Exklusionen (später), wird *translation unit* genannt.

Eine translation unit wird bei den üblichen Compilern ausgehend einer einzelnen Quellcode-Datei (.cpp, .cc) gebildet.

Wenn man eine leere Quellcode-Datei angelegt hat, so hat man dort zunächst nur die grundlegenden Sprachelemente zur Verfügung:

- built-in types: `bool`, `double`, `char`, `int` usw.
- Schleifen, Verzweigungen usw.
- Eigene Funktionen definieren und (weiter unten im Quelltext) aufrufen
- Eigene Typen (z.B. Klassen) definieren und (weiter unten) verwenden
- »Dinge« anlegen, `new`, `delete`

Wenn man eine leere Quellcode-Datei angelegt hat, so hat man dort zunächst nur die grundlegenden Sprachelemente zur Verfügung:

- built-in types: `bool`, `double`, `char`, `int` usw.
- Schleifen, Verzweigungen usw.
- Eigene Funktionen definieren und (weiter unten im Quelltext) aufrufen
- Eigene Typen (z.B. Klassen) definieren und (weiter unten) verwenden
- »Dinge« anlegen, `new`, `delete`

Die translation units sind voneinander unabhängig, also darf man nicht ohne weiteres auf Namen, Funktionen, Typen, »Dinge« usw. aus anderen translation units zugreifen.

Wann darf man was verwenden?

Offensichtlich darf man z.B. keine Klasse verwenden, die nirgendwo definiert wurde.

- Funktionen darf man verwenden, wenn sie weiter oben im Quelltext deklariert wurden (`void foobar();`)
- Klassen darf man instanziiieren, wenn sie weiter oben definiert wurden (die member wurden alle genannt)

Wann darf man was verwenden?

Offensichtlich darf man z.B. keine Klasse verwenden, die nirgendwo definiert wurde.

- Funktionen darf man verwenden, wenn sie weiter oben im Quelltext deklariert wurden (`void foobar();`)
- Klassen darf man instanziiieren, wenn sie weiter oben definiert wurden (die member wurden alle genannt)

Faustregel: Der Compiler muss die Form kennen, nicht aber den Inhalt. Bei Funktionen heißt dies, er muss die Signatur kennen (also Rückgabewert, Name und Parameter); bei Klasse heißt dies, er muss die Struktur der Klasse kennen (Basisklassen, member functions, data member, static members).

```
void check();    // deklariert die Funktion
class MyIncompleteType; // deklariert den Typen

class MyType    // definiert den Typen
{
public:
    int m;
    void test();
};

int main()
{
    check();    // OK, die Funktion wurde vorher deklariert
    MyIncompleteType m; // Fehler, der Typ wurde NICHT vorher definiert
    MyType mt;    // OK
    mt.test();    // OK, die Funktion wurde vorher deklariert
}

void check() { /* do something */ }
void MyType::test() { /* ... */ }
```

Mit dem Befehl `#include <HEADER_NAME>` in einer eigenen Zeile bindet man einen Header ein. ACHTUNG: Dieser Befehl ist gedacht für die C++ Standard-Bibliothek.

Mit dem Befehl `#include "HEADER_NAME"` in einer eigenen Zeile bindet man *bei allen gängigen Compilern* eigene Header-Dateien (.h, .hpp) ein. Der Standard spricht an dieser Stelle von *source files*.

Mit dem Befehl `#include <HEADER_NAME>` in einer eigenen Zeile bindet man einen Header ein. ACHTUNG: Dieser Befehl ist gedacht für die C++ Standard-Bibliothek.

Mit dem Befehl `#include "HEADER_NAME"` in einer eigenen Zeile bindet man *bei allen gängigen Compilern* eigene Header-Dateien (.h, .hpp) ein. Der Standard spricht an dieser Stelle von *source files*.

Die Befehle sind einfache copy&paste-Anweisungen: Der Quelltext der Header wird an die Stelle des Kommandos eingefügt (und weitere `#includes` abgearbeitet).

Innerhalb einer translation unit darf man bspw. eine Funktion auch dann nutzen, wenn in der translation unit *überhaupt keine Definition*, sondern nur eine Deklaration der Funktion vorhanden ist.

Der Linker bindet nach den Arbeiten des Compilers die translation units zusammen, dabei löst er die Verweise auf Funktionen auf. Wird eine Funktion in translation unit A nur deklariert, dieselbe Funktion in translation unit B auch definiert, so findet der Linker diese Definition und verwendet sie in A.

Üblicherweise (Konvention) teilt man den Quellcode seines Programms auf in logisch zusammenhängende Teile, die nicht zu groß sind. Diese wandern in separate Quellcode-Dateien (.cpp usw.). Um die Verbindung zwischen diesen unabhängigen translation units herzustellen, verwendet man Header. Hierin steht nur *das Minimum, welches zur Zusammenspiel der translation units erforderlich ist.*

Üblicherweise (Konvention) teilt man den Quellcode seines Programms auf in logisch zusammenhängende Teile, die nicht zu groß sind. Diese wandern in separate Quellcode-Dateien (.cpp usw.). Um die Verbindung zwischen diesen unabhängigen translation units herzustellen, verwendet man Header. Hierin steht nur *das Minimum, welches zur Zusammenspiel der translation units erforderlich ist*.

Im Falle von Funktionen ist das Minimum die Deklaration (Signatur), im Falle von Klassen ist es die Definition (Struktur).

Das Beispiel vorhin könnte etwa in drei Dateien aufgetrennt werden:

MyType.h

```
void check();  
class MyType  
{  
  public:  
    int m;  
    void test();  
};
```

MyType.cpp

```
#include "MyType.h"  
  
void check() { }  
void MyType::test() { }
```

program.cpp

```
#include "MyType.h"  
  
int main()  
{  
    check();  
    MyIncompleteType m;  
    MyType mt;  
    mt.test();  
}
```

Es werden zwei translation units gebildet: Eine aus *MyType.cpp* und eine aus *program.cpp*. Beide translation units enthalten den Header *MyType.h*, hierüber können in *program.cpp* die Funktionen mit den Definitionen in *MyType.h* verknüpft werden.

Die Header aus der Standard-Bibliothek von C++ müssen nicht zwangsläufig Dateien sein, deren Text kopiert und eingefügt wird. Es gibt lediglich Garantien, dass gewisse Dinge zur Verfügung stehen, wenn man sie „einbindet“.

Die Header aus der Standard-Bibliothek von C++ müssen nicht zwangsläufig Dateien sein, deren Text kopiert und eingefügt wird. Es gibt lediglich Garantien, dass gewisse Dinge zur Verfügung stehen, wenn man sie „einbindet“.

Bindet man den Header `<iostream>` ein, so steht u.a. `std::cin` und `std::cout` zur Verfügung.

- uneindeutige Datei-Namen (mehrere Dateien mit demselben Namen)
- Datei wird nicht gefunden (falsche Einstellungen vom Compiler / Präprozessor)
- es wurde vergessen, einen Header einzubinden
- ein Header wurde mehrmals eingebunden (gleich mehr dazu)

2 Include-Guards

Wenn das Projekt wächst, wächst freilich auch die Zahl an Header-Dateien. Ein weiteres Problem: Innerhalb einer translation unit gilt die one-definition-rule (es darf immer nur eine Definition geben). Durch mehrfache Einbindung desselben Headers kommt es daher oft zu Problemen.

Wenn das Projekt wächst, wächst freilich auch die Zahl an Header-Dateien. Ein weiteres Problem: Innerhalb einer translation unit gilt die one-definition-rule (es darf immer nur eine Definition geben). Durch mehrfache Einbindung desselben Headers kommt es daher oft zu Problemen.

Es gibt verschiedene Varianten, damit umzugehen. Am gängigsten ist Folgende:

- Jeder Header inkludiert diejenigen Headers, die er benötigt – etwa die Typen der Parameter von Funktionsdeklarationen oder die Typen von data members von Klassen. Damit enthält eine translation unit immer alle Header, die benötigt werden.
- In die Header werden sog. *include guards* eingebaut, die verhindern, dass ein Header zweimal in derselben translation unit landet.

Es kommt am Ende nur darauf an, dass alles in der translation unit gelandet ist – ABER: bei komplexen Projekten ist es nicht leicht, selbst alle Abhängigkeiten zu finden, und es muss auch in der richtigen Reihenfolge in der translation unit gelandet sein: In einem Header A sei eine Funktion deklariert, die einen Typen T verwende. Dann muss der Header B, in welchem T definiert wurde, *vor* der Funktionsdeklaration eingebunden werden.

Es kommt am Ende nur darauf an, dass alles in der translation unit gelandet ist – ABER: bei komplexen Projekten ist es nicht leicht, selbst alle Abhängigkeiten zu finden, und es muss auch in der richtigen Reihenfolge in der translation unit gelandet sein: In einem Header A sei eine Funktion deklariert, die einen Typen T verwende. Dann muss der Header B, in welchem T definiert wurde, *vor* der Funktionsdeklaration eingebunden werden.

Daher ist es günstig, in den Headern selbst in den ersten Zeilen die nötigen `#include`-Anweisungen stehen zu haben, die alle Header inkludieren, die dieser Header benötigt.

MyType.h

```
class MyType
{
public:
    int m;
};
```

MyDepType.h

```
#include "MyType.h"

class MyDepType
{
public:
    void foobar();

private:
    MyType m;
};
```

Selbst wenn in der translation unit nur »Dinge« vom Typ `MyDepType` und keine »Dinge« vom Typ `MyType` direkt verwendet werden, muss dennoch der Header *MyType.h* eingebunden sein, da sonst dem Compiler die Struktur der Klasse `MyDepType` unbekannt ist. Auch muss diese Einbindung *vor* der Definition von `MyDepType` geschehen. Daher schreibt man die notwendige `#include`-Anweisung gleich in den *MyType.h*-Header.

include guards basieren auf der bedingten Einbindung. Diese wiederum wird durch den sogenannten Präprozessor durchgeführt (ein einfaches Programm, das sich vor dem Compiler den Quelltext anschaut).

Grundlegend schreibt man in Headern:

```
#ifndef EINDEUTIGER_NAME
```

```
#define EINDEUTIGER_NAME
```

```
// eigentlicher Header-Text
```

```
#endif // EINDEUTIGER_NAME
```

Der Text für `EINDEUTIGER_NAME` sollte dabei eindeutig sein für alle Header, die in die translation unit eingebunden werden. *Tipp von Christian Käser:* Dateinamen, Projektnamen und relativen Pfad verwursten. Es gelten dieselben Beschränkungen wie etwa für Variablennamen.

Funktionsweise von include guards (1)

Der Präprozessor lebt in seiner eigenen Welt aus Suchen&Ersetzen. Zusätzlich dazu kann er noch rudimentäre Variablen verwalten. Trifft ein Präprozessor auf eine if-then-else-Verzweigung, so fügt er das passende Textstück (entweder then oder else) in die translation unit ein:

```
#ifndef UNIQUE_NAME_FOR_HEADER
// then (UNIQUE_NAME_FOR_HEADER ist NICHT definiert)

#else
// else (UNIQUE_NAME_FOR_HEADER IST definiert)

#endif
```

Es geht hierbei wirklich um stupides Text-Einfügen: Entweder der Text zwischen der `#if`-Zeile und der `#else`-Zeile wird verwendet, oder der Text zwischen der `#else`-Zeile und der `#endif`-Zeile. Es kann der else-Teil auch weggelassen werden (nicht jedoch das `#endif`).

Funktionsweise von include guards (2)

- Der Präprozessor prüft, ob eine Variable *nicht* existiert (`#ifndef` = if not defined).
- Falls ja (sie existiert nicht), legt er diese Variable an (`#define HE...`) und bindet den eigentlichen Header-Text in die translation unit ein.
- Schließlich wird die Verzweigung noch beendet (`#endif`).

Funktionsweise von include guards (2)

- Der Präprozessor prüft, ob eine Variable *nicht* existiert (`#ifndef` = if not defined).
- Falls ja (sie existiert nicht), legt er diese Variable an (`#define HE...`) und bindet den eigentlichen Header-Text in die translation unit ein.
- Schließlich wird die Verzweigung noch beendet (`#endif`).

Der Präprozessor arbeitet begrenzt auf translation units:

- Beim ersten einbinden des Headers *in diese translation unit* ist die Variable noch nicht definiert.
- Der Präprozessor geht in den *then*-Fall des if-then-else, definiert die Variable und bindet den Header ein.
- Beim nächsten Einbinden dieses Headers *in diese translation unit* wird nichts mehr eingebunden, da der else-Zweig leer ist.