Vorlesung

Objektorientiertes Programmieren in C++

Teil 8 - WS 2023/24

Detlef Wilkening
www.wilkening-online.de

15 Präprozessor, Compiler, Linker, ...

Bislang haben wir immer den gesamten Quelltext in eine Datei geschrieben. Das wird langsam unübersichtlich, und außerdem lässt sich so ja nichts wiederverwenden – darum soll das jetzt geändert werden. Leider betreten wir damit den noch dunkelsten Bereich von C++. C++ implementiert die Aufteilung des Codes auf mehrere Dateien u.a. mit sogenannten Header-Dateien, die dann von dem sogenannten Präprozessor zur eigentlichen Übersetzungs-Einheit vorverarbeitet werden. Wir werden dies in den nächsten Kapiteln im Detail besprechen. Aus heutiger Sicht ist dies kein modernes Verfahren mehr. Mit C++20 führt C++ Module ein, die die Header ablösen sollen. Aber C++20 ist gerade erst erschienen, und in der Praxis kenne ich keine Bibliothek und kein Projekt, das auf C++20 Module aufsetzt. Selbst die C++ Standard-Bibliothek selber liegt noch nicht in Modulen vor – dies ist erst für C++23 vorgesehen. Von daher werden wir uns in der Praxis noch jahrelang mit Headern rumschlagen müssen – darum besprechen wir nur diese.

15.1 Quelltext-Aufteilung, Header, Sourcen

Um zu überlegen, wie eine Aufteilung aussehen kann, lassen sie uns anschauen, welche „*Schnittstellen-Elemente*“ wir bislang kennen, und was für ihre Implementierung bzw. ihre Benutzung benötigt werden.

Unter dem Begriff „*Schnittstellen-Elemente*“ verstehen wir hier alle Elemente, die wir in unserem Code benutzen um ihn zu schreiben, also nach unserem augenblicklichen Wissenstand: Konstanten, Typen (Typ-Aliase, Aufzählungstypen, Klassen) und Funktionen.

Schnittstellen-Element	Benötigt mindestens
Konstanten	
Deklaration	---
Definition	---
Benutzung	Deklaration (oder Definition – je nachdem)
Typ-Aliase	
Definition	---

Benutzung	Definition
Aufzählungstypen	
Definition	---
Benutzung	Definition
Funktionen (freie)	
Deklaration	---
Definition (Implementierung)	---
Benutzung	Deklaration
Klassen	
Deklaration (s.u.)	---
Definition	---
Implementierung (Element-Funktionen)	Klassen-Definition
Benutzung	Klassen-Definition

Hiermit ist sofort offensichtlich, welche Dinge für die Benutzung der Schnittstellen-Elemente benötigt werden:

- Konstanten-Deklarationen oder –Definitionen
- Typ-Alias-Definitionen
- Enum-Definitionen
- Deklarationen von freien Funktionen
- Klassen-Definitionen

Wenn wir diese Dinge in extra Dateien legen könnten, bräuchten wir nur noch einen Mechanismus, der sie in einer anderen Datei bekannt macht. Dann könnten wir sie problemlos in unseren Quelltexten verwenden. Einen solchen Mechanismus gibt es in C++ in Form des Präprozessors mit der „include“-Anweisung – siehe nächstes Kapitel.

Diese *Extra-Dateien* für die Schnittstellen-Elemente nennt man in C++ Header, und sie haben typischerweise die Endung „*h“ oder „.hpp“. Alle C++ Header haben keine Endung.

Die Dateien mit den Implementierungen werden häufig Source-Dateien oder kurz Sourcen genannt, obwohl natürlich auch die Header Source-Code enthalten. Source-Dateien haben typischerweise die Endung „.cpp“ oder „.cc“.

Außerdem zeigt die Tabelle direkt, dass es wohl möglich sein könnte, die Implementierungen (Definition) der freien Funktionen und der Element-Funktionen in eigene Quelltexte auszulagern, denn diese werden für die Benutzung nicht benötigt. Dann stellt sich nur die Frage, wie einzelne unabhängige Quelltexte übersetzt (compiliert) werden, und für das eigentliche Ergebnis (das ausführbare Programm) zusammengeführt werden? Dies wird durch Compiler und Linker in C++ gemacht – siehe weiter unten.

15.2 Präprozessor

In C++ gibt es einen sogenannten Präprozessor, der am Anfang eines Compilevorgangs über den Quelltext läuft und alle Präprozessor Anweisungen verarbeitet. Präprozessor

Anweisungen sind alle die mit einem „#“ beginnen - augenblicklich kennen wir nur die include Anweisung „#include“.

Der Präprozessor ist ein ziemlich dummes Programm und macht eigentlich nichts anderes als einen etwas intelligenteren Textersatz (eine Art Suchen & Ersetzen).

Im Falle der include Anweisung sucht er in den sogenannten Include-Pfaden nach der entsprechenden Datei, fügt sie quasi 1:1 in unseren Quelltext ein, und speichert das Ergebnis als temporäre Datei (die bildet dann den eigentlichen Input für den Compiler).

Original-Dateien **Präprozessor macht daraus temporäre Datei „y.tmp“**

```
// Datei x.h

void f1();
void f2(int);
```

```
// Datei y.cpp

#include "x.h"

void g()
{
   ...
   f1();
   ...
}
```

=>

```
void f1();
void f2(int);

void g()
{
   ...
   f1();
   ...
}
```

Der Präprozessor unterstützt zwei Arten von Include’s:

- #include <header>
- #include "header"

Der Unterschied zwischen den beiden Includes ist die Suchstrategie nach den Dateien im Include-Pfad – auf Details soll hier aus Zeitmangel nicht eingegangen werden. Aus Faustregel kann man sagen: benutzen sie für die System-Header die <...> Variante, und für ihre eigenen Header die "..." Variante.

Bemerkung – es gibt noch viele weitere Präprozessor-Anweisungen – zwei weitere werden wir gleich im Kapitel über Header-Guards noch kennen lernen. Aber im Rahmen der Vorlesung war‘s das dann auch. In C++ ist der Präprozessor auch nicht mehr so wichtig wie in C, da in C++ viele typische C- Präprozessor Aufgaben durch leistungsfähigere C++ Sprachmittel übernommen werden.

15.3 Compiler

Der Compiler bekommt als Input den temporären Output des Präprozessors – in der Praxis

sieht man dies normalerweise nicht, es läuft transparent im Hintergrund ab. Aber in der Praxis hat auch jeder Entwicklungsumgebung eine Möglichkeit, sich den Präprozessor-Output anzuschauen.

Der Compiler übersetzt jetzt den Quelltext in Maschinencode – der Output des Compilers sind Objekt-Files, die unter Windows meist die Endung „*.obj“ und unter Linux meist „*.o“ haben. Ein Object-File ist noch kein ausführbares Programm, sondern nur der in Maschinensprache übersetzte Code eines Quelltexts.

Temporäre Datei „y.tmp“ **Compiler macht daraus das Object-File „y.obj“**

```
void f1();
void f2(int);

void g()
{
   ...
   f1();
   ...
}
```

=>

```
1001011011101110010101010101
0101000011110101110001000001
1 ... 110100010011
```

Hierbei sind mehrere Dinge wichtig:

- Jeder Übersetzungsvorgang, d.h. der Durchlauf von Präprozessor und Compiler, ist vollkommen unabhängig von anderen. Ein Compiler schaut nie nach links und rechts (in andere Dateien), selbst wenn er dann Dinge wissen könnte, die ihm helfen würden.
- Ein Compiler kümmert sich nicht um Vollständigkeit des Programms – im obigen Beispiel interessiert ihn nicht ob die Funktion „f1“ irgendwo implementiert ist, auch wenn sie benutzt wird. Da sie deklariert ist (Das ist immerhin das Versprechen dass es sie geben sollte!), ist die Benutzung für ihn definiert, und er compiliert den Quelltext.
- Ein Compiler kümmert sich nicht um unbenutzte Deklarationen und Definitionen. Im obigen Beispiel ist „f2“ deklariert, wird aber nicht benutzt. Dem Compiler ist das egal.

Ein Object-File kann noch kein ausführbares Programm sein, da meistens noch einige Dinge fehlen: all die deklarierten Dinge, die benutzt worden sind. Im Beispiel die Funktion „f1“. Der Compiler lässt an dieser Stelle im Maschinencode Platz für die echte Adresse, und schreibt in die sogenannte Import-Tabelle, was noch benötigt wird.

Außerdem erstellt der Compiler eine sogenannte Export-Tabelle, in der alle Symbole stehen, die in diesem Object-File implementiert worden sind.

Das eigentliche Object-File „y.obj“ sieht also ungefähr folgendermassen aus:

Import

Adr	Symbol
0000:0062	void f1()

Export

Adr	Symbol
0000:0040	void g()

Code

Adr	Maschinencode	Pseudocode	Originalcode
0000:0000	....		
0000:0040	3F 7E	movem do-d4, sp	void g()
0000:0060	B3 C2 00 00 00 00	call f1	f1()
0000:0080	A2 44	ret	return

15.4 Linker

Der Linker hat nun die Aufgabe, aus all den Object-Files das ausführbare Programm zusammen zu setzen. Dazu muss er im einfachsten Fall:

- Die Code-Segmente aller Object-Files zusammenbinden
- Alle offenen Adressen (Import-Einträge) auflösen
- Die C++ Laufzeitumgebung dazubinden (Standard-Bibliothek und allgemeine Dinge, die zum Ablauf des Programms benötigt werden).
- Das Programm in einer Form abspeichern, die das OS auswerten kann.

Schauen wir uns das mal an einem Beispiel an: gehen wir mal davon aus, dass es neben dem Quelltext „y.cpp“ von oben noch eine Datei „z.cpp“ gibt, die folgenden Inhalt hat:

```
// Datei z.cpp

void g();

void f1()
{
   ...
}

int main()
{
   ...
   g();
   ...
   g();
   ...
   return 0;
}
```

Hieraus erzeugt das Gespann Präprozessor/Compiler folgende Objekt-Datei „z.obj“:

Import

Adr	Symbol
0000:0132	void g()
0000:0142	void g()

Export

Adr	Symbol
0000:0100	void f1()

Code

Adr	Maschinencode	Pseudocode	Originalcode
0000:0000	....		
0000:0100	3F 86	movem a0-a1, sp	void f1()
0000:0110	A2 44	ret	return
0000:0120	3F 92	movem a0-a7, sp	int main()
0000:0130	B3 C2 00 00 00 00	call g	g()
0000:0140	B3 C2 00 00 00 00	call g	g()
0000:0150	A2 44 00 00 00 00	ret 0	return 0

Der Linker fügt die beiden Objekt-Dateien jetzt zu einem lauffähigen Programm zusammen, und löst dabei alle Referenzen (hier „f1“ und „g“) auf.

Code

Adr	Maschinencode	Pseudocode	Originalcode
0000:0000	....		
0000:0010	3F 86	movem a0-a1, sp	void f1()
0000:0020	A2 44	ret	return
0000:0030	3F 92	movem a0-a7, sp	int main()
0000:0040	B3 C2 **00 00 00 70**	call g	g()
0000:0050	B3 C2 **00 00 00 70**	call g	g()
0000:0060	A2 44 00 00 00 00	ret 0	return 0
0000:0070	3F 7E	movem do-d4, sp	void g()
0000:0090	B3 C2 **00 00 00 10**	call f1	f1()
0000:00B0	A2 44	ret	return

Hinweis – in Wirklichkeit ist da noch etwas mehr zu machen, und die Adressen dürfen z.B. vom Linker noch gar nicht endgültig eingetragen werden – das macht erst der Loader des OS. Aber vom Prinzip her funktioniert das alles so.

15.5 Fehler

Wenn sie Compilerfehler bekommen, dann haben sie in der entsprechenden Datei (bzw. in den von ihr eingebundenen Dateien) einen syntaktischen Fehler gemacht.

Wenn sie Linkerfehler bekommen, dann sind alle ihre Dateien syntaktisch in Ordnung, aber die Auflösung der Symbole klappt nicht – typische Fehler:

- Symbol nicht vorhanden, d.h. sie haben ein Symbol deklariert und benutzt, aber nirgendswo implementiert.
- Symbol mehrfach vorhanden, d.h. sie haben ein Symbol mehrfach implementiert (zumindest sieht der Linker das so – siehe auch die Diskussion bei inline Funktionen weiter unten).

15.6 ODR und Header-Guards

Eine Sache muss noch geklärt werden – die ODR und ihre Konsequenzen. In C++ gibt es die sogenannte ODR „One-Definition-Rule“, die einfach nur besagt, dass eine Definition in einer Übersetzungseinheit nur einmal vorkommen darf. Auch wenn die Definition beim zweiten Mal identisch zur ersten ist, ist dies ein Compilerfehler.

Nun gut, aber wo ist das Problem?

Durch indirektes includieren kann es leicht passieren, das Definitionen doppelt in Übersetzungseinheiten vorhanden sein. Darum sollte in jedem Header ein sogenannter Header-Guard vorhanden sein.

```
// Header "x.h"

#ifndef X_H
#define X_H

class x
{
};

#endif
```

Ein Header-Guard besteht aus mehreren Präprozessor-Anweisungen.

- Die Anweisung „#ifndef X_H“ fragt ab, ob das Präprozessor-Makro „X_H“ **nicht** definiert ist. Wenn es nicht definiert ist, dann wird der Präprozessor-If-Block betreten – der enthält dann den normalen Header-Inhalt.
- Zuerst wird das Präprozessor-Makro „X_H“ aber definiert – mit der Anweisung „#define X_H“.
- Beendet wird der Präprozessor-If-Block betreten mit der Anweisung „#endif“.

Wird der Header „x.h“ jetzt zweimal eingebunden, dann wird er beim ersten Einbinden ganz normal vollständig includiert, da das Makro nicht definiert ist – aber es wird jetzt auch definiert. Bei weiteren Includes ist das Makro nun aber definiert, und daher weist die Anweisung „#ifndef X_H“ den Präprozessor ab, und der Präprozessor-If-Block wird kein weiteres Mal eingebunden. Ergebnis: die Definiton ist nur einmal vorhanden, die ODR ist erfüllt.

Damit Header-Guard-Makros eindeutig sind, gibt es eine einfache Konvention für ihre

Benamsung:

- Header-Guard-Makros werden – wie alle Präprozessor-Makros – **per Konvention** GROSS geschrieben.
- Header-Guard-Makros entsprechen dem Namen der Datei, möglicherweise inklusive Verzeichnis-Namen (bei verschachtelten Strukturen).
- Die Namensteile von Header-Guard-Makros werden durch „_“ getrennt.
- An das Header-Guard-Makro wird noch der Postfix „_H“ angehängt.

Achtung – leider recht verbreitet, aber trotzdem gefährlich und falsch, ist die Konvention dem Header-Guard-Makro noch das Präfix „_“ oder „__“ vorzustellen. Ein solcher Makroname – z.B. „_X_H“ bzw. „__X_H“ – ist in C++ reserviert!

- Namen mit einem Underscore am Anfang sind für den Compiler und die Entwicklungsumgebung reserviert.
- Namen mit zwei Underscore am Anfang sind für die Sprache reserviert – z.B. „__FILE__“ oder „__LINE__“.

15.7 Quelltext-Aufteilung

Pro „logischem Modul“ (z.B. einzelne Klasse mit möglicherweise zugehörigen freien Funktionen) sollte es einen Header und eine Implementierungs-Datei geben.

Der Header enthält Konstanten, Typ-Defintionen (Typedef’s, Enums, Klassen,...) und Deklarationen der freien Funktionen. Die Implementierungs-Datei includiert als erstes den Header und enthält die Implementierungen der Element-Funktionen und freien Funktionen des Headers.

15.8 Inline

15.8.1 Thema „Performance“

C++ hat u. a. das Ziel: **effizienter Code, schnelle Programme**.

Selbst wenn Funktionsaufrufe in C++ nicht viel Zeit kosten, so bergen sie doch immer einen gewissen Overhead in sich. Gerade bei sehr kleinen einfachen Funktionen kann dieser Overhead einen nicht vernachlässigbaren Anteil darstellen.

In C++ gibt es **inline**-Funktionen, die ganz normale Funktionen sind, aber vom Compiler direkt an der Aufrufstelle expandiert werden können. Dabei wird die normale Semantik von Funktions-Aufrufen vollständig gewahrt.

15.8.2 Freie Funktionen

Um eine Funktion **inline** zu machen, schreiben Sie das Schlüsselwort **inline** vor die Funktions-Deklaration und -Definition.


```
inline int max(int, int);

inline int max(int a, int b)
{
   return a>b ? a: b;
}

int m = max(i++, j++);
```

```
Ausgabe
i: 18
j: 43
m: 42
```

Hinweis - der Compiler ist nicht gezwungen eine mit **inline**-deklarierte Funktion an der Aufrufstelle zu expandieren. Dieses Schlüsselwort ist nur eine *Bitte* bzw. ein *Hinweis* an den Compiler. Dem Compiler ist es freigestellt einen Funktionsaufruf nicht zu inlinen bzw. nur einen Teil der Aufrufe zu inlinen.

Praxis - damit der Compiler die Funktion bei der Compilation des Aufrufs expandieren kann, muss er zu diesem Zeitpunkt den Code der **inline**-Funktion kennen. Daher werden inline-Funktionen fast immer in den Header-Dateien implementiert.

15.8.3 Element-Funktionen

Viele Funktionen in einem gut designten C++ Programm sind sehr kurz, und damit ideale Kandidaten für inline-Funktionen. Z.B. sind durch einfache Zugriffsfunktionen die Daten sauber gekapselt, aber jeder Zugriff führt zu einem kleinen Overhead: dem Funktions-Aufruf.

In Bezug auf **inline** sind Element-Funktionen ganz normale Funktionen:

- entweder schreiben sie das Schlüsselwort **inline** vor Deklaration und Definition, oder
- sie implementieren die Definition direkt in der Klassen-Definition (in diesem Fall ist das Schlüsselwort gar **inline** nicht nötig - dies wird auch „impliztes Inline“ genannt - s.u.

```
class date
{
public:
   int day() const { return day_; }       // implizites inline ohne Semikolon
   int month() const { return month_; };  // implizites inline mit Semikolon
   inline int year() const;               // explizites inline

   // Rest wie bisher...
};

inline void date::year() const
{
   return year_;
}
```

Bemerkung - werden Element-Funktionen direkt in der Klassendefiniton definiert, so erlaubt die Syntax von C++, dass das abschliessende Semikolon wegfallen kann, da die abschliessende geschweifte Klammer die Definition eindeutig beendet – siehe Beispiel.

Hinweis - dies gilt auch für spezielle Element-Funktionen wie z.B. Konstruktoren, Destruktoren oder Operatoren, auch z.B. in Verbindung mit „virtual“ oder Initialisierungslisten.

15.8.4 Implizites Inline bei freien Funktionen

Neben dem impliziten Inline von Element-Funktionen gibt es einen weiteren Fall von implizitem Inline bei freien Funktionen: kennt der Compiler beim Compilieren eines Funktions-Aufrufs die Funktions-Definition, so darf er sie inline expandieren, auch wenn sie nicht so deklariert ist.

Da stellt sich sofort die Frage, was das Schlüsselwort inline eigentlich soll, wenn der Compiler eine Funktion eh automatisch inline expandieren kann, sobald er ihre Implementierung kennt? Denn genau das ist ja auch die Voraussetzung, dass inline überhaupt funktionieren kann.

Würde die Funktion aber nicht inline deklariert, so wäre sie schnell mehrfach im Programm definiert, was zu Linker-Fehlern führen würde.

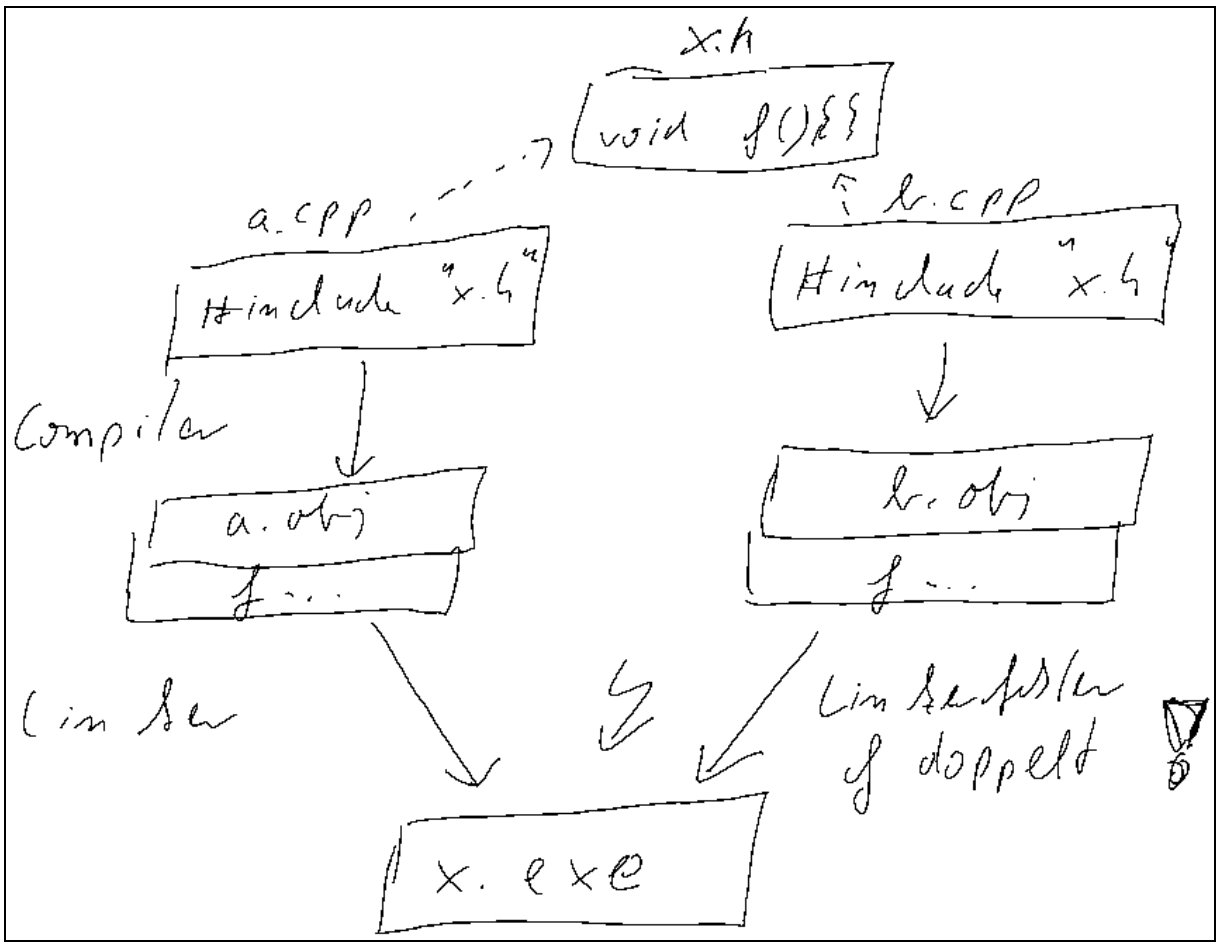

Abb. 15-1 : Linker-Fehler bei Funktionen im Header ohne „inline“

Bzgl. der Link-Spezifikation sorgen inline-Funktionen für eine sogenannte interne Link-Spezifikation, d.h. die Funktion wird nicht aus dem Objekt-File exportiert. Trotzdem muss das Compiler/Linker-Gespann dafür sorgen, dass die gesamte normale Funktions-Semantik

vollständig erhalten bleibt (dies betrifft Themen, die wir noch nicht kennen, z.B. Funktions-Adressen, oder static Funktions-Variablen).

15.9 Bibliotheken

15.9.1 Was ist eine Bibliothek?

Nehmen wir an, jemand programmiert einige schöne Dinge, die von allgemeinem Interesse sind. Z.B. eine Sammlung von Klassen und Funktionen zur Verwaltung, Darstellung, Manipulation, Anzeige, usw. von Daten, Zeiten, Zeitstempeln, usw.

Die komplette Implementierung besteht sicher aus einigen Header- und einigen Implementierungs-Dateien, z.B.

- date.h und date.cpp
- time.h und time.cpp
- utils.h und utils.cpp

Wollen sie diese Dinge nun in einem Projekt von ihnen nutzen, so gibt es eine einfache Möglichkeit: „ihnen werden die Header und Implementierungen zur Verfügung gestellt, und sie binden diese Dateien ganz normal in ihr Projekt ein“. Dies ist aber keine schöne Vorgehensweise, da sie nun das gesamte Handling vieler fremder Dateien gewonnen haben. Und eigentlich interessiert sie die Implementierung und all das Drumherum doch gar nicht, sie wollen das doch einfach nur nutzen. Außerdem kann es sein, dass der Implementierer ihnen seine Implementierung nicht geben will, da sie hochgeheime Algorithmen enthalten. Was dann?

Die Header muss er ihnen immer geben. Das geht nicht anders, denn die Header enthalten alle Deklarationen und Definitionen, die der Compiler benötigt um den Code zu compilieren.

Statt der Source-Dateien könnte er ihnen aber auch die Objekt-Dateien geben. Dann bräuchten sie die Sourcen nicht mehr zu compilieren, und der Implementierer muss seine Sourcen nicht herausgeben.

Aber auch das Handling vieler hunderter Objekt-Dateien ist nicht wirklich angenehm – schöner wäre doch **eine einzige Datei**, die den Inhalt aller Objekt-Dateien enthält. Und genau dies ist eine Bibliothek. Es ist eine Datei, die die vollständige Implementierung enthält, und vom Linker zu ihrem Programm dazugebunden werden kann.

Achtung – spätestens mit den Bibliotheken haben wir den Rahmen von ISO C++ komplett verlassen. Der Standard kennt den Begriff einer Bibliothek oder einer Library in diesem Sinne nicht, und äußert sich auch nicht zu dieser Thema. Er beschränkt sich vollständig auf die Sprache und die Standard-Bibliothek. Praktische Dinge wie eben auch die Erstellung und Benutzung von Bibliotheken werden von ihm zurzeit vollkommen ausgeklammert, und sind dem entsprechend auch sehr plattform- und compiler-spezifisch.

Hinweis – in der Praxis gibt es statische und dynamische Bibliotheken. All dies ist interessant und auch sehr wichtig, übersteigt den Rahmen der Vorlesung aber bei weitem.

Was müssen sie also nun machen, wenn sie:
1. eine fremde Bibliothek nutzen wollen, bzw.
2. eine eigene Bibliothek schreiben wollen?

15.9.2 Fremde Bibliothek nutzen

Mit dem Wissen dieses Kapitels sollte ihnen jetzt klar sein, dass eine fremde Bibliothek im Normalfall zwei Dinge mitbringt:
- Header, die sie zum compilieren benötigen
- und einer Bibliotheks-Datei, die sie beim Linken angeben müssen.

Im Einzelfall kann eine Bibliothek natürlich noch andere Dinge enthalten, und sie kann natürlich auch aus mehreren Bibliotheks-Dateien bestehen. Aber konzeptionell kann man sich das so vorstellen.

Die Header – typischerweise in einem Verzeichnis gruppiert – legen sie an eine allgemeine Stelle in ihre Verzeichnis-Struktur, und geben diesen Pfad für den Compiler an. Die Bibliotheks-Datei legen sie z.B. neben die Header, und geben diese beim Linken mit an.

Bei vielen IDE`s können sie sowohl allgemeine als auch projekt-spezifische Header und Bibliotheken angeben. Allgemeine stellen sie einmal in der IDE ein, und sie stehen damit automatisch für alle Projekte zur Verfügung. Dies empfiehlt sich für Bibliotheken, die sie häufig bis immer benutzen. Andere Bibliotheken benutzen sie nur manchmal, und diese sollten dann projekt-spezifisch hinzugefügt werden.

Hinweis – für die Standard-Bibliothek müssen sie sich um nichts kümmern. diese Header und Bibliotheks-Dateien sind dem Compiler und Linker automatisch per Default bekannt. Nichtsdestotrotz kann man sie natürlich in der Praxis auch ändern, d.h. dem Compiler bzw. Linker eine andere Standard-Bibliothek mitgeben.

Achtung – manche Bibliotheken haben keine explizite Bibliotheks-Datei. Dies ist dann der Fall, wenn alle Definitionen in den Headern vorhanden sind. Dies passiert häufig im Zusammenhang mit inline-Funktionen und vor allem Templates. Ein Beispiel dafür sind große Teile der Boost-Bibliotheken, bei denen sie nur die Header angeben müssen, und sie damit fertig sind.

15.9.3 Eigene Bibliotheken erstellen

Um eigene Bibliotheken zu erstellen müssen Sie dem Compiler/Linker einfach nur angeben, dass Sie kein Programm sondern eben eine Bibliothek erstellen wollen. Als Ergebnis erhalten Sie dann eben kein Executable, sondern eben eine Bibliotheks-Datei. Die jeweilige

Target-Angabe ist natürlich compiler-spezifisch.

Achtung – in einer Bibliothek sollte im Normallfall natürlich keine Main-Funktion vorhanden sein, da Sie das Programm ja weiterhin selber schreiben wollen.

