Modulare Programmierung & Bibliotheken

– Proseminar: C-Grundlagen-&-Konzepte –

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Stefan Grusche
E-Mail-Adresse:	3grusche@informatik.uni-hamburg.de
Matrikelnummer:	6527160
Studiengang:	B.Sc. Informatik
Betreuer:	Michael Kuhn

Hamburg, den 29. 05. 2014

Inhaltsverzeichnis

1 Motivation / Einleitung

1.1 Modulare Programmierung

Der Begriff des modularen Programmierens beschreibt ein Programmierparadigma, bei dem ein Softwareprojekt in verschiedene Module aufgeteilt wird. Diese Module sind funktional möglichst unabhängig voneinander und haben einige Vorteile gegenüber Ansätzen, die ohne Module auskommen.
Besonders große Softwareprojekte mit tausenden Zeilen Code können mit Modulen in kleinere, verständlichere Einheiten aufgeteilt werden. Dies ermöglicht es auch, dass Programmierer auf verschiedene Module verteilt werden und so sich die Arbeit leicht teilen können, ein Programmierer ist dann z.B. für die GUI zuständig, ein anderer für die interne Berechnung. Bei Änderungen am Quellcode muss nun nicht mehr das ganze Projekt auf einmal kompiliert werden, dies kann für einzelne Module unabhängig getan werden, was den Zeitaufwand stark verringert. Ein Modul kann zusätzlich auch alleine auf Fehler geprüft werden.
Modularisierung ermöglicht die Wiederverwendung von Code, indem man ein Modul aus einem Projekt später in einem anderen leicht wiederverwenden kann. Dies ermöglicht auch den Aufbau von Funktionsbibliotheken, die dies noch weiter vereinfachen.

Module nutzen das Prinzip der Kapselung, bei dem die eigentlichen Implementation von der Schnittstelle getrennt werden. So kann der Modulentwickler einerseits die Implementation austauschen, während sich für den Anwender die Nutzung nicht ändert und andererseits kann er selbst bestimmen, welche Teile des Moduls von außen nutzbar sind (Export-Schnittstelle) und was für Betrachter von außen versteckt bleibt (Information Hiding).

1.2 Umsetzung in C

In C bilden die Quelldateien (C-Dateien) die Module. Diese bilden so seperat übersetzbare Einheiten.
Indem man globale Funktionen und Variablen in seinen Dateien mit `static` deklariert, kann man die Sichtbarkeit dieser auf die eigene Datei beschränken und so das Prinzip des "Information Hiding" umsetzen. Symbole, die für die Nutzung von außen freigegeben sind, werden mit `extern` deklariert, womit sich deren Sichtbarkeit auf das komplette Programm erweitert. Dies ist in C implizit gesetzt, alle Symbole ohne explizite Deklaration der Sichtbarkeit sind in C also extern verfügbar.

2 Header

2.1 Der Header

Sogenannte Header-Dateien dienen in C zur Trennung von Implementation und Schnittstelle.
In diesen stehen üblicherweise Deklarationen von Funktionen und Variablen, die anderen Dateien zur Verfügung gestellt werden sollen. Außerdem kann man in diesen auch Definitionen von Makros, Konstanten oder (sehr) kleinen Funktionen finden (diese müssen dann mit `static` definiert werden, um Fehler beim Linken zu vermeiden).
Ein Header gehört normalerweise zu einer C-Datei, die die Implementation für die deklarierten Symbole liefert, die Implementationen können sich aber auch in eigenen Funktionsbibliotheken befinden. Sogenannte "System-Header" werden vom C-Standard bereitgestellt und liefern Funktionen, die Speichermanagement, I/O-Operationen, Typumwandlungen u.ä. ermöglichen. Die Implementationen dieser Header-Funktionen findet man in der C-Standard-Bibliothek, die normalerweise zum Betriebssystem gehört.

Listing 2.1: Typischer Header

```
1  #ifndef HEADER_H_
2  #define HEADER_H_
3
4  #define PI 3.1415
5
6  extern double gleitkommazahl;
7  extern int zaehler;
8
9  extern void zaehle();
10 extern double addierePi(int zahl);
11
12 static int max(int a, int b) {
13     (a > b) ? a : b;
14 }
15
16 #endif
```

2.2 #include

`#include` ist eine sogenannte Präprozessor-Direktive, die es ermöglicht, den Inhalt von Header-Dateien in eine andere Quelldatei einzubinden, sodass in dieser die im Header deklarierten Symbole nun verfügbar sind. `#include` macht dabei nicht mehr, als den Text aus der übergebenen Datei an die Stelle im Quelltext zu kopieren, an der dieses `#include` steht.
Es gibt zwei verschiedene Arten an `#include`-Statements:

- `#include <Dateiname>` für System-Header. Die spitzen Klammern sorgen dafür, dass der Präprozessor in Systemverzeichnissen nach dem Header sucht.
- `#include "Dateiname"` für eigene Header. Der Präprozessor sucht hier zuerst im Verzeichnis der Quelldatei.

2.3 Include-Guards

#include-Statements können verschachtelt werden, d.h., dass ein Header einen anderen Header einbinden kann, der wiederum einen weiteren Header einbindet. Dies ist eine leichte Fehlerquelle, da einer der eingebundenen Header einen der vorher schon eingebundenen Header einbinden könnte, wodurch Symbole mehrmals in der gleichen Datei deklariert werden, was beim Linken für Fehler sorgt.
Des weiteren könnten zirkuläre Abhängigkeiten auftreten, bei denen ein Header einen Header einbindet, der einen Header einbindet, der den ersten Header wieder einbindet, sodass der Präprozessor dann in einer Endlosschleife versucht, die #include-Statements aufzulösen. Dies führt nach kurzer Zeit zu einem Abbruch beim Ausführen des Präprozessors, da der Compiler ein Limit festlegt, wie tief diese #include-Verschachtelungen reichen dürfen.

Listing 2.2: Zirkuläre Abhängigkeit

```
1  /** Datei: headerA.h */
2
3  #include "headerB.h"
4
5  /** Datei: headerB.h */
6
7  #include "headerC.h"
8
9  /** Datei: headerC.h */
10
11 #include "headerA.h"
```

Es gibt mehrere Möglichkeiten, solche Fehler zu vermeiden. Die sauberste Umsetzung ist mittels der Präprozessor-Direktive `#ifndef HEADER-ALIAS` mit einem direkt folgenden

`#define HEADER-ALIAS` im Kopf des Headers (`#endif` muss am Ende des Headers folgen). Hier wird erst überprüft ob ein Makro mit dem Namen HEADER-ALIAS schon definiert ist, ist dies nicht der Fall, wird dies definiert und der Präprozessor geht den Quelltext weiter normal durch. Wurde ein Makro mit diesem Namen schon definiert, was der Fall ist, wenn dieser Header in der Datei schon einmal eingebunden wurde, springt der Präprozessor über den Quelltext und bindet nichts weiter ein.
Im `gcc` ist dies soweit optimiert, dass Header mit `#ifndef`, die schon einmal durchlaufen wurden, überhaupt nicht mehr gelesen werden.

Listing 2.3: `#ifndef` beugt dem Fehler vor

```
1 #ifndef STACK_H_
2 #define STACK_H_
3
4 /**
5     Deklarationen etc.
6 */
7
8 #endif /** STACK_H_ */
```

Es gibt zwei weitere Möglichkeiten, von deren Nutzung aber abgeraten wird:

- `#pragma once`
  - Steht im Kopf des Headers
  - Verhindert, dass Dateien wiederholt gelesen werden
  - Ist aber nicht von jedem Präprozessor unterstützt
- `#import`
  - Ersetzt #include
  - Verhindert ebenfalls das mehrfache Lesen einer Datei
  - Bildet die Standardpraxis in Objective-C, ist in C aber nicht gerne gesehen

2.4 Übersicht der Standard-Header in C

assert.h	fenv.h	limits.h	signal.h	stdint.h	tgmath.h
complex.h	float.h	locale.h	stdarg.h	stdio.h	time.h
ctype.h	inttypes.h	math.h	stdbool.h	stdlib.h	wchar.h
errno.h	iso646.h	setjmp.h	stddef.h	string.h	wctype.h
stdalign.h*	stdatomic.h*	stdnoreturn.h*	threads.h*	uchar.h*	

Tabelle 2.1: C-Header, markiert mit * sind Neuzugänge in C11

3 Bibliotheken

3.1 Linker

Der Linker ist ein Programm, dass beim Erzeugen eines Programms eine essentielle Rolle spielt; er sorgt dafür, dass aus mehreren Quelldateien ein lauffähiges, ausführbares Programm wird.
Er verbindet übergebene Objektdateien und löst dabei alle Referenzen auf, die in den Dateien bestehen, also vor allem `extern`-referenzierte Funktionen und Variablen, sowie Pointer, die der Entwickler benutzt hat. Der Linker führt beim Durchgehen der Objektdateien eine Liste der deklarierten und definierten Symbole, Symbole, die bei der Erzeugung Speicherplatz zugeordnet bekommen, gelten als starke Symbole (strong; alle Symbole, die definiert werden), alle anderen gelten als schwach (weak; alle Symbole, die lediglich deklariert werden).
Der Linker versucht nun allen schwachen Symbolen gleichnamige starke zuzuweisen, sollte zu einem schwachen kein starkes Symbol existieren, oder sollte es mehrere starke geben, bricht der Linker mit einem Fehler ab.
Läuft alles richtig, hat der Linker für die verschiedenen Objektdateien einen gemeinsamen Adressraum erstellt und alle Funktionen und Variablen haben eine relative Adresse, die sie nutzbar machen.

3.2 statische Bibliotheken

Statische Bibliotheken sind einfache Archive von Objektdateien, die mit einem Index versehen werden. Sie bilden eine Möglichkeit viele Funktionen an einem zentralen Ort immer wieder benutzbar zu machen.
Eine statische Bibliothek wird mit einem *Archiver* erstellt, folgender Befehl bewirkt dies mit `gcc`:
`ar rs libmylib.a funktion1.o funktion2.o`
`rs` sorgt dafür, dass eine Bilbiothek mit Index erstellt wird, lässt man das ”s” weg, wird kein solcher Index erzeugt. `libmylib.a` ist der Name der erzeugten Bibliothek, C-Bibliotheken beginnen nach Konvention mit `lib` und enden entweder auf `.a`(statische Bibliothek) oder `.so`(dynamische Bibliothek). Danach werden dann beliebig viele kompilierte Objektdateien übergeben, die die Bibliothek später aufbauen werden.
Zu dieser Bibliothek erstellt man eine oder mehrere Header-Dateien, die die Funktionen aus der Bibliothek verfügbar machen. Bindet man eine dieser Funktionen in sein Programm ein, sucht der Linker beim Linken auch in der Bibliothek, falls diese auf der Kommandozeile übergeben wurde (dazu später mehr).

Findet der Linker eine benutze Funktion in der Bibliothek, bindet er die entsprechende Objektdatei fest ins Programm ein. Als Ergebnis erhält man so eine alleinstehende ausführbare Datei, die direkt benutzt werden kann.

3.3 dynamische Bibliotheken

Dynamische Bibliotheken funktionieren etwas anders als statische Bibliotheken. Der Linker bindet keine Objektdateien aus einer dynamischen Bibliothek direkt ins Programm ein, sondern vermerkt nur, dass die Referenz aufgelöst wurde.
Beim Starten eines so erstellten Programms schaltet sich nun der *dynamische Linker* ein; dieser gehört zum Betriebssystem und lädt alle in der ausführbaren Datei vermerkten dynamischen Bibliotheken in den Speicher. Im Programm benutzte Symbole werden nun entweder direkt beim Programmstart referenziert, oder bleiben unaufgelöst, solange sie noch nicht benutzt werden (eine sogenannte "lazy" Referenzierung).
Der größte Vorteil von dynamischen Bibliotheken ist, dass sie nur einmal im Speicher liegen, aber von mehreren Programmen gleichzeitig verwendet werden können. Besonders wichtige Bibliotheken verbrauchen so nicht unnötig viel Speicher.
Beim dynamischen Linken verringert sich auch die Größe der Programme, da nicht mehr alle Funktionen direkt integriert sind, außerdem lassen sich Implementationen leicht austauschen, ohne sein Programm wieder linken zu müssen.

Um dynamische Bibliotheken erstellen zu können, braucht man sogenannten positionsunabhängigen Objektcode ("position independent code", *PIC*), da man im Voraus nicht weiß, wo die Dateien im Speicher landen werden. PIC nutzt einen sogenannten "Global Offset Table", der alle Variablen mit Offsets enthält. Solchen Code erstellt man beim Kompilieren mit dem Compiler-Flag `-fpic` bzw. `-fPIC`:

```
gcc -c -fPIC code.c mehrcode.c
```

Hat man solche Objektdateien, ist das Erstellen der Bibliothek nun nicht mehr schwierig, das Compiler-Flag `-shared` verbindet nun die Objektdateien zu einem sogenannten *Shared Object* (Dateiendung .so):

```
gcc -shared -o libmylib.so funktion1.o funktion2.o
```

Shared Objects, Onjektdateien und ausführbare Dateien unterliegen unter Linux dem **ELF**-Standard, der den Dateien einiges an Zusatzinformationen hinzufügt, die die Verwendung (besonders das benutzen an verschiedenen Stellen im Speicher) erleichtern, unter anderem bei dynamischen Bibliotheken (*Shared Object*) eine Symboltabelle mit korrespondierender Stringtabelle, die alle Symbole mit Werten und deren Namen enthält.

3.4 Sichtbarkeit und Performance

Da bei dynamischen Bibliotheken alle nicht explizit lokal deklarierten Symbole automatisch in der Symboltabelle landen, können Symbole, die gar nicht für die Nutzung außerhalb der Bibliothek vorgesehen sind, viel Platz im Speicher einnehmen, da die Tabellen komplett geladen werden.
Außerdem kann so ein Programmierer unerwünschterweise auf diese Symbole zugreifen, und der Compiler kann den Code nicht gut optimieren, da er die globalen Symbole immer über den "Global Offset Table" (s.o.) anspricht und so Systemaufrufe verbraucht. Außerdem könnte er lokale Symbole unter Umständen "inlinen".
Die beste Möglichkeit all dies zu verhindern, ist es, wie in normalen C-Dateien alle nur Datei-weit benutzten Symbole mit `static` zu definieren. Dies spart Systemaufrufe, da die Variablen nun direkt angesprochen werden und der Compiler kann Code nach Bedarf "inlinen".
Eine weitere von `gcc` unterstützte Möglichkeit, ist das Festlegen einer Bibliotheks-weiten Sichtbarkeit mit der Compiler-Option `-fvisibility`.
`-fvisibility=default` legt die Sichtbarkeit auf die Standardvariante des Compilers fest, `-fvisibility=hidden` versteckt alle Symbole standardmäßig. Betroffen sind hiervon alle Symbole, die keine explizite Sichtbarkeitsbestimmung haben.
Man könnte dies also z.B. verwenden, indem man die Sichtbarkeit `-fvisibility=hidden` festlegt und alle Symbole, die von außen benutzt werden sollen, explizit als sichtbar definiert (Dies ist aber nicht die beste Möglichkeit, da so unschöne Header entstehen können und vielleicht Symbole übersehen werden).
Verbunden mit `-fvisibility` ist die Möglichkeit, die Sichtbarkeit pro Symbol festzulegen. Hierzu steht `__attribute__((visibility("Sichtbarkeit")))` zur Verfügung, das im Code ähnlich wie `extern` oder `static` verwendet wird.
Die Sichtbarkeit `hidden` verhindert dabei nur die Verwendung des Symbols außerhalb der Bibliothek, zwischen den Objektdateien der Bibliothek können diese Symbole im Gegensatz zu `static` weiterhin verwendet werden.

Listing 3.1: Code ohne explizite Sichtbarkeiten [Dre]

```
1 int last;
2
3 int next () {
4   return ++last;
5 }
6
7 int index (int scale) {
8   return next () << scale;
9 }
```

Listing 3.2: Gleicher Code mit `__attribute__` [Dre]

```
1 int last __attribute__ ((visibility ("hidden")));
```

```
2
3 int __attribute__ ((visibility ("hidden"))) next () {
4   return ++last;
5 }
6
7 int index (int scale) {
8   return next () << scale;
9 }
```

3.5 Nutzung von Bibliotheken

Um Bibliotheken beim Kompilieren überhaupt benutzen zu können, müssen sie dem Compiler als Option übergeben werden. Bibliotheken stehen in der Kommandozeile üblicherweise mit dem Flag `-l` am Ende, das texttt"lib" am Anfang des Dateinamens und die Dateiendung werden nicht mit angegeben.
Befinden sich die Bibliotheken nicht im Verzeichnis der Quelldateien, muss dieses mit `-L` angegeben werden, gleiches gilt für Header, deren Verzeichnis mit `-I` übergeben wird. Beim Übergeben der Bibliotheken muss auf die Reihenfolge geachtet werden, der Linker geht diese von links nach rechts durch, deshalb müssen Bibliotheken, die Symbole aus anderen benutzen, vor diesen stehen, da sonst die Referenz nicht aufgelöst werden kann. Am Ende eines Übersetzen-Befehl wird `-lc` für das Hinzufügen der C-Standard-Bibliothek von `gcc` implizit hinzugefügt.
Ein Beispiel für ein Kommando mit allen erwähnten Optionen:
`gcc -o executable -Iinclude -Llibraries -lmylib -lm`

Literaturverzeichnis

[CAU14] *Creating And Using Static And Shared CLLibraries*. http://www.cpp-home.com/tutorials/108_1.htm, Mai 2014.

[CPP14] *The C Preprocessor*. https://gcc.gnu.org/onlinedocs/gcc-4.9.0/cpp/, Mai 2014.

[CSA14] *CSAPP: Original Line Drawings*. http://csapp.cs.cmu.edu/public/figures.html, Mai 2014.

[DBSG11] Dausmann, Manfred, Ulrich Bröckl, Dominik Schoop Joachim Goll: *C als erste Programmiersprache*. Viewig+Teubner Verlag, 7th , 2011.

[Dre] Drepper, Ulrich: *How To Write Shared Libraries*.

[GBU14] *GNU Binary Utilities*. https://sourceware.org/binutils/docs/binutils/index.html, Mai 2014.

[GCC14] *GCC Manual: Code Gen Options*. https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Code-Gen-Options.html#Code-Gen-Options, Mai 2014.

[HOW14] *Program Library HOWTO*. http://tldp.org/HOWTO/Program-Library-HOWTO/index.html, Mai 2014.

[Kle14] Klemens, Ben: *C im 21. Jahrhundert*. O'Reilly Verlag, 2nd , 2014.

[LD14] *ld*. https://sourceware.org/binutils/docs/ld/index.html, Mai 2014.

[PC06] Prinz, Peter Tony Crawford: *C In A Nutshell*. O'Reilly Verlag, 1st , 2006.

[PMG14] *Programmieren mit GCC*. http://www.oreilly.de/german/freebooks/rlinux3ger/ch132.html#13776, Mai 2014.

[WIK14] *C Standard Library - Wikipedia*. http://en.wikipedia.org/wiki/C_standard_library, Mai 2014.

[WPM14] *Modulare Programmierung - Wikipedia*. http://de.wikipedia.org/wiki/Modulare_Programmierung, Mai 2014.