C - Grundlagen und Konzepte
Modulare Programmierung und Bibliotheken

Stefan Grusche

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

2014-06-05

Gliederung

Modulare Programmierung

- Ansatz, Software in Module aufzuteilen
- Im Idealfall sind Module unabhängig
- Vorteile:
  - übersichtlichere Projekte
  - höhere Robustheit gegenüber Änderungen
  - leichtere Wartung
  - bessere Arbeitsteilung
  - Wiederverwendung von Code

Modulare Programmierung

- Module nutzen Prinzip der Kapselung
  - Implementation wird von Schnittstelle getrennt
  - Export-Schnittstelle stellt Funktionalität zur Verfügung
  - Interna bleiben verborgen ("Information Hiding")

Module in C

- In C ist ein Modul eine (seperat übersetzbare) Quelldatei
- Information Hiding mit `static`
- Header ermöglichen Trennung von Schnittstelle und Implementation

`static` und `extern`

- Schlüsselwörter vor Rückgabetyp bei Funktionen bzw. Typ bei Variablen
- Essenziell zur Umsetzung von Modulen
- `static`
  - Beschränkt Sichtbarkeit auf Datei (Funktionen, globale Variablen)
  - Beeinflusst Lebensdauer von lokalen Variablen
- `extern`
  - Macht globale Variable bzw. Funktion im ganzen Programm sichtbar
  - **implizit** gesetzt

Header

- Wichtig für Modularität
- Enthalten Deklarationen zur Verfügung gestellter Funktionen, Variablen etc.
- Definitionen in gleichnamiger .c-Datei oder in Bibliothek
- Werden dort eingebunden, wo Funktionalität benötigt wird
- **System-Header** stellen I/O-Operationen, Speichermanagement etc. zur Verfügung

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
```

Listing 1: Typischer Header

`#include`

- Präprozessor-Direktive
- Kopiert Inhalt des übergebenen Headers in Quelldatei
- `#include <Dateiname>` sucht in Standardordnern (/usr/include/, etc.)
- `#include "Dateiname"` sucht im Verzeichnis der Quelldatei

Include-Guards I

- Verschachteln von `#include` möglich
- Gefahr des mehrfachen Einfügens
  - Führt zu mehrfachem Deklarieren von Funktionen, Variablen etc.
  - Zirkuläre Abhängigkeiten leicht hervorzurufen

```
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"
```

Listing 2: Zirkuläre Abhängigkeit

Ergebnis...

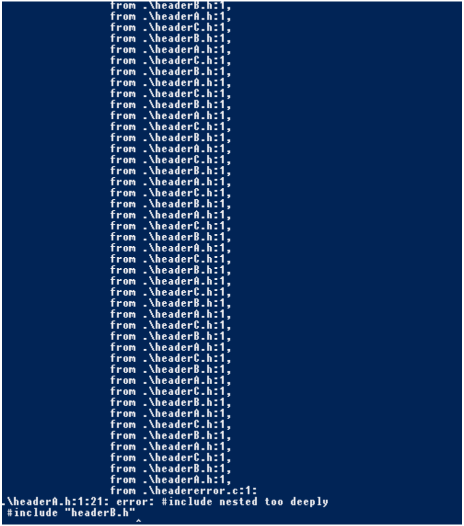

Figure: Compiler bricht ab...

Include-Guards I

- Lösung: `#ifndef`

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

Listing 3: #ifndef in Aktion

Include-Guards II

`#pragma once`

- Steht oben im Header
- Nach einmaligem Lesen bleibt Datei unangetastet
- Nicht von jedem Präprozessor unterstützt

`#import`

- Ersetzt `#include`
- Standardpraxis in Objective-C, unerwünscht in C

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

Table: C99-Standard-Header

stdalign.h	stdatomic.h	stdnoreturn.h	threads.h	uchar.h

Table: neue Header in C11

Der Linker

- Kombiniert Objektdateien zu ausführbarem Programm
- Löst alle Querbezüge auf (`extern`, Pointer)
- Gibt jeder Funktion, globalen Variable relative Adresse
- Erzeugt gemeinsamen Adressraum

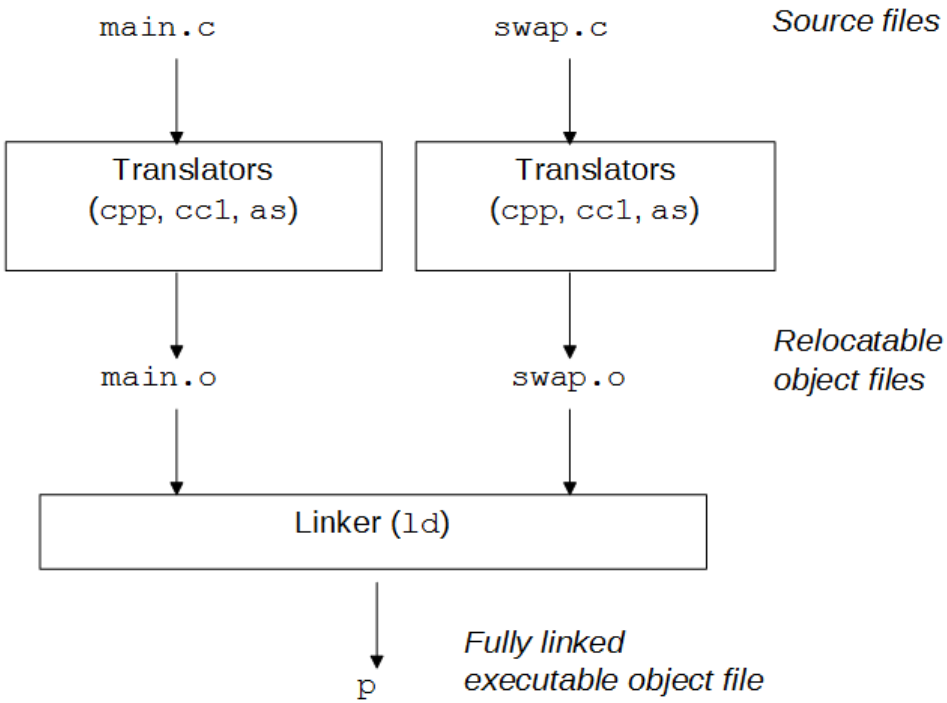

Figure: Linker [CSA14]

statische Bibliotheken

- Simples Archiv kompilierter Objektdateien mit Index
- Erzeugung mit ar:
  `ar rs libmylib.a funktion1.o funktion2.o`
- Linker sucht nach benutzten Funktionen und bindet Objektdateien fest ins Programm ein
- Ergebnis: Alleinstehende ausführbare Datei

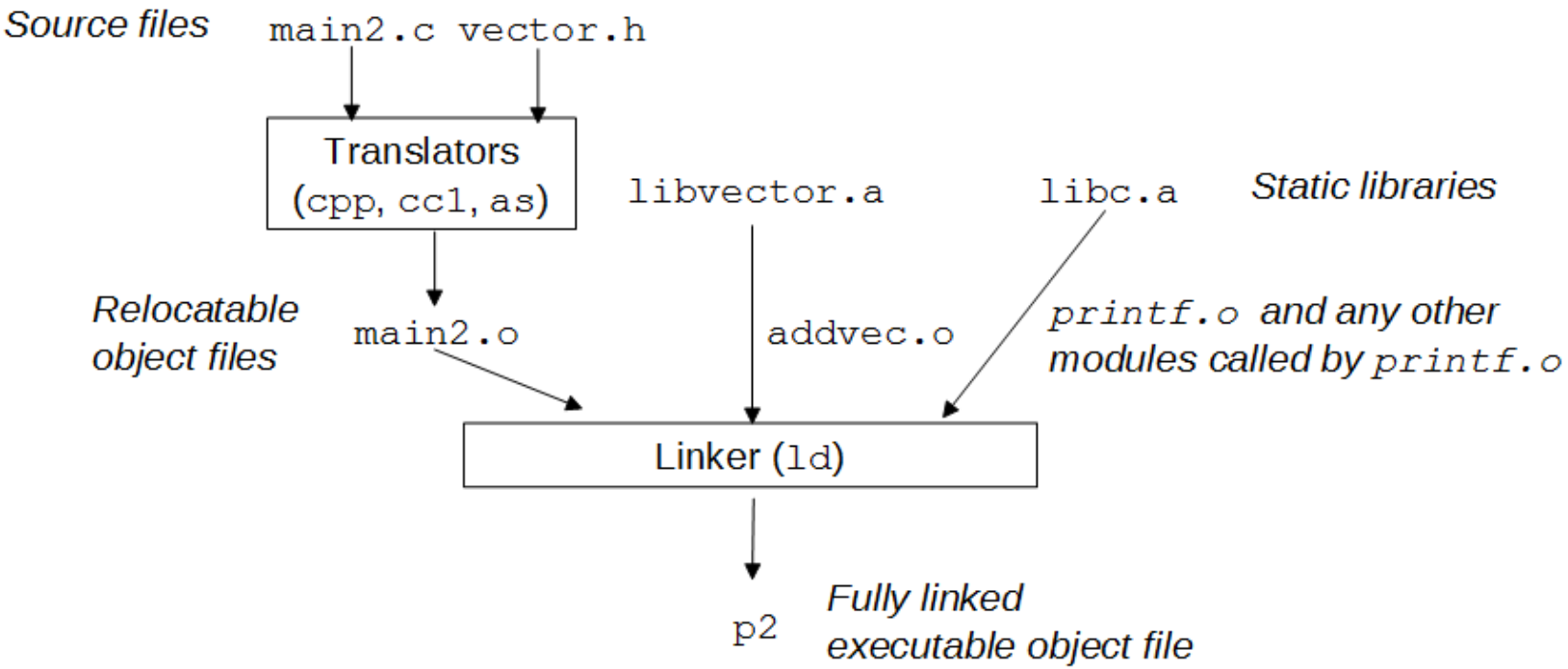

Figure: Statische Bibliotheken [CSA14]

dynamische Bibliotheken (DSOs)

- Linker vermerkt benutzte Funktionen, bindet keine ins Programm ein
- Beim Starten des Programms lädt Loader DSOs und referenziert Symbole
- Referenzierung passiert sofort oder erst bei Gebrauch des Symboles ("lazy")
- **Liegt einmal im Speicher, von mehreren Programmen benutzbar**

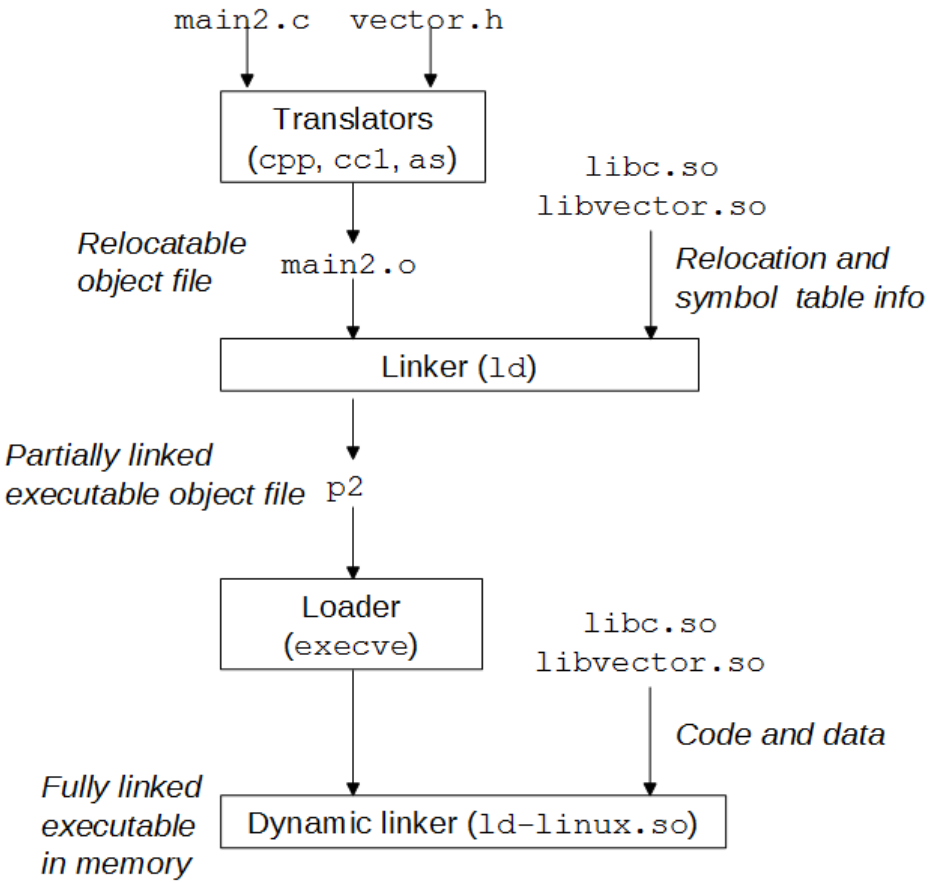

Figure: Dynamische Bibliotheken [CSA14]

dynamische Bibliotheken

- Zur Erstellung muss positionsunabhängiger Objektcode vorliegen: `gcc -c -fPIC code.c mehrcode.c`
- Erstellen der Bibliothek selbt sehr einfach: `gcc -shared -o libmylib.so funktion1.o`
- Unter Linux nutzen dynamische Bibliotheken den ELF-Standard

Sichtbarkeit und Performance

- Alle globalen Symbole einer DSO kommen in ELF-Symboltabelle
- Nicht zu nutzende Symbole nehmen mit Namen viel Speicher weg
- Compiler benötigt mehr Systemaufrufe, könnte an anderen Stellen optimieren

`static` benutzen

- `static` für dateiweit genutzte Symbole
- Verringert Anzahl der Systemaufrufe
- Compiler nutzt unter Umständen Inlining

globale Sichtbarkeit

- Compiler-Option `-fvisibility` reguliert Sichtbarkeit in DSOs
- `-fvisibility=default` übernimmt Standardeinstellung
- `-fvisibility=hidden` macht alle nicht sichtbar deklarierten Symbole außerhalb einer DSO unsichtbar

Sichtbarkeit pro Symbol

- `__attribute__((visibility("<Sichtbarkeit>")))` legt Sichtbarkeit pro Symbol fest
- <Sichtbarkeit> mit gewünschter Sichtbarkeit zu ersetzen (`default`, `hidden`, `protected`, `internal`)

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

Listing 4: Code ohne explizite Sichtbarkeiten [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  }
```

Listing 5: Gleicher Code mit `__attribute__` [Dre]

Standard C Library

- Standard-C-Bibliothek mit ungefähr 200 Funktionen
- Liefert Implementationen zu Standard-Headern
- Mathematische Funktionen in libm definiert
- Verfügbar als statische (`libc.a`) und dynamische (`libc.so`) Bibliothek
- Funktionsumfang gering im Vergleich zu anderen Sprachen

- Benutzte Bibliotheken müssen Linker übergeben werden
  `gcc -o executable -Iinclude -L. -lmylib -lm`
- **-I** Header-Pfad **-L** Bibliotheks-Pfad **-l**<**name**> genutze Bibliothek
- -lc implizit
- **Reihenfolge wichtig!**
- Dynamische Bibliotheken werden bevorzugt

Zusammenfassung

- Modularität hilft bei Strukturierung von Softwareprojekten
- Umsetzung in C mit `extern`, `static` und Headern
- Include-Guards schützen vor Fehlern
- Funktionen können in statischen oder dynamischen Bibliotheken gesammelt werden


Creating And Using Static And Shared "C" Libraries.
http://www.cpp-home.com/tutorials/108_1.htm, Mai 2014.


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


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


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


Drepper, Ulrich: *How To Write Shared Libraries*.


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


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


Program Library HOWTO.
http://tldp.org/HOWTO/Program-Library-HOWTO/index.html, Mai 2014.


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


ld.
https://sourceware.org/binutils/docs/ld/index.html, Mai 2014.


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


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


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


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