Modulare Programmierung in C

Einführung in die Programmierung

Michael Felderer

Institut für Informatik, Universität Innsbruck

Entwicklung eines Programms

- Softwareprojekte werden in mehrere voneinander abgegrenzte Phasen unterteilt.
- Man erhält einen Softwareentwicklungslebenszyklus, einen so genannten **Software Development Life Cycle**.
- Es gibt zwar einen eindeutigen Ablauf eines Software Development Life Cycles, allerdings aber keine einheitliche Form der Darstellung.
- Typische Phasen (im historischen Wasserfallmodell)
  - Problemanalyse
  - Systementwurf
  - Programmentwurf
  - Implementierung und Test
  - Betrieb und Wartung

Phasen (1)

- **Problemanalyse**, auch Anforderungs- oder Systemanalyse genannt
  - Diese Analyse wird in Zusammenarbeit mit dem Auftraggeber durchgeführt.
  - Das Ergebnis ist eine Anforderungsbeschreibung (auch Pflichtenheft).
- **Systementwurf**
  - Hier werden die zu lösenden Aufgaben in so genannte **Module** aufgeteilt.
    - Ein Modul ist eine abgeschlossene funktionale (logische) Einheit einer Software, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.
    - Erhöht die Übersichtlichkeit und verbessert die Korrektheit und Zuverlässigkeit.
  - Das Ergebnis ist eine Systemspezifikation (Grundlage für die Implementierung).
- **Programmentwurf**
  - In dieser Phase werden die einzelnen Module weiter verfeinert, indem die Datenstrukturen festgelegt und Algorithmen entwickelt werden.
  - Das Ergebnis besteht aus mehreren Programmspezifikationen.

Phasen (2)

- **Implementierung** und **Test**
  - Hier werden die Module programmiert und anhand ihrer jeweiligen Spezifikation getestet (verifiziert).
  - Durch das Zusammensetzen der einzelnen Module erhält man das Programm.
- **Betrieb** und **Wartung**
  - Diese Phase umfasst die Pflege der Software, in der Erweiterungen und Änderungen eingebracht oder entdeckte Fehler behoben werden.
  - Unter Umständen führt dies wieder zu einer vorherigen Phase oder überhaupt zur Problemanalyse zurück, wodurch ein Zyklus entsteht.

Das Wasserfallmodell wird heute kaum mehr verwendet, die beschriebenen Aktivitäten kommen aber auch in anderen Modellen vor!

Etliche Erweiterungen bzw. Veränderungen werden mittlerweile in der Praxis verwendet (z.B. Agile Softwareentwicklung, Kanban)

Implementierung und Test

- Mit dieser Phase haben wir uns **teilweise** in dieser LV beschäftigt!
- Bisherige Programme waren recht klein und überschaubar.
  - Wenige Funktionen
  - Eine Datei
- Reale Programme bestehen aus hunderttausenden/millionen Zeilen Programmcode.
- Wie organisiert man diesen Code?
  - Eine Datei wird dabei nicht mehr ausreichend sein!
  - Größere Programme werden in mehrere Dateien aufgeteilt.
  - Zusammengehörende Logik wird in Module zusammengefasst.
  - Gegebenen falls werden mehrere Module zu einer Library kompiliert.

C und Software Engineering

- C unterstützt das sogenannte **Structured Design**
  - Dabei wird das Programm in kleinere Einheiten zerlegt, die untereinander möglichst wenig Querbeziehungen haben.
- In C kann man solche Einheiten mit **Funktionen** realisieren.
  - Diese Funktionen definieren eine Schnittstelle (Interface) zu ihrer Umgebung.
    - Parameter
    - Return Value
    - Globale Variablen (sollte vermieden werden!)
  - Durch den Aufruf von Funktionen entstehen Beziehungen und Abhängigkeiten.
    - Je kleiner die Schnittstelle, desto leichter lässt sich eine Implementierung austauschen.
  - Komplexere Problemlösungen (Funktionen) werden meist durch das Kombinieren von kleinere Problemlösungen (Funktionen) realisiert.

Modulares Design

- Die Funktionen selbst können wiederum in größere Einheiten zusammengefasst werden.
- Solche Einheiten bezeichnet man als Module.
- Ein Programm besteht dann in der Regel aus mehreren Modulen.

- Wie schon erwähnt können **Module** weiter zusammengefasst werden – zu **Libraries**.

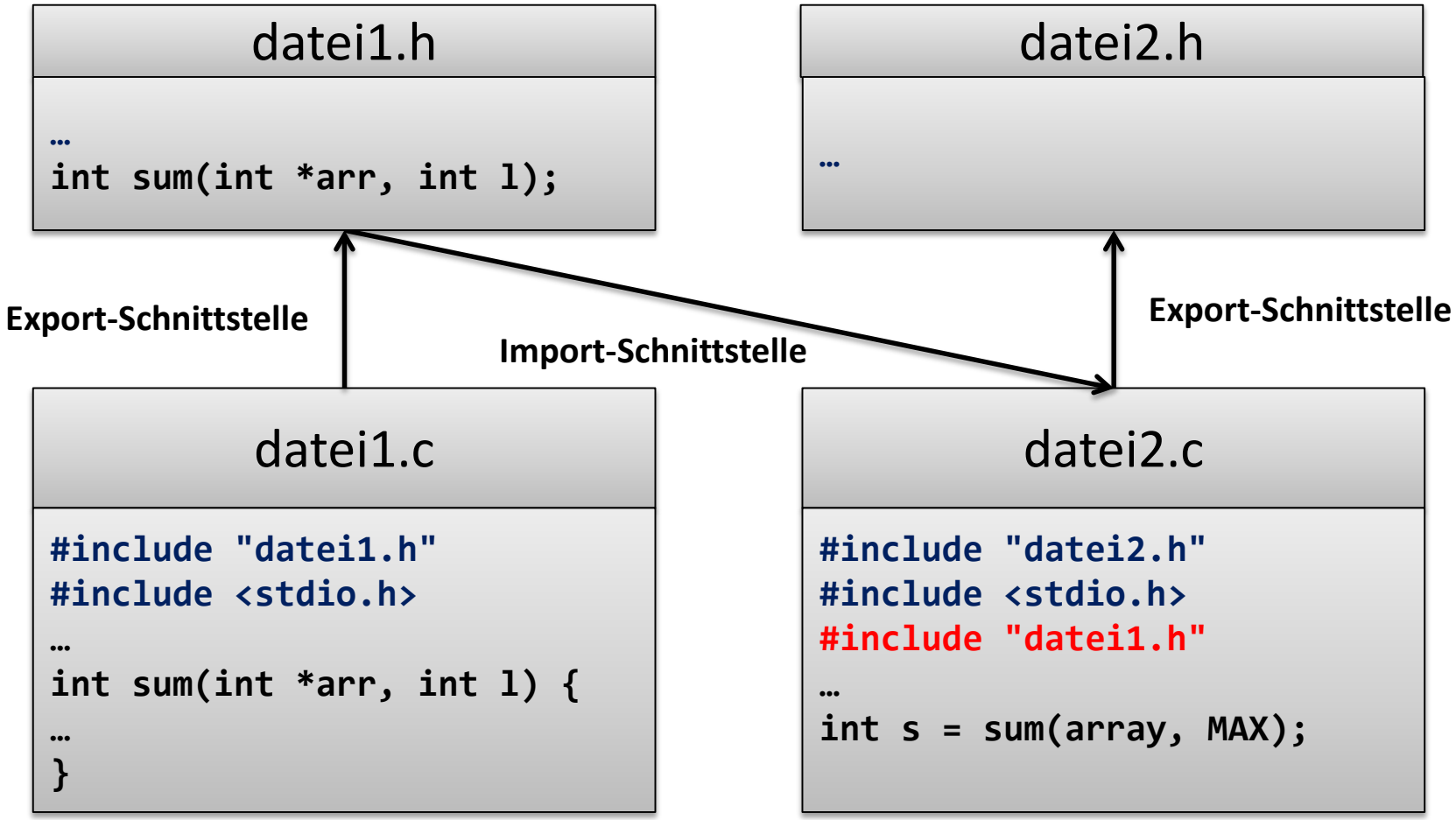
Modul – Kapselung und Information Hiding

- Kapselung
  - Kapselung der Daten und Funktionen als funktionale Einheiten.
  - Definition der Schnittstellen zwischen den einzelnen Einheiten (Interfaces).
- **Export-Schnittstelle**
  - Über diese Schnittstelle stellt ein Modul Ressourcen für andere Module zur Verfügung.
  - Alle anderen Interna sind verborgen (Information Hiding).
  - Entwickler kann bestimmen, was von einem Modul benutzt werden kann (Export-Schnittstelle) und was verborgen bleibt (Information Hiding).
- **Import-Schnittstelle**
  - Zur Implementierung eines Moduls kann man andere Module benutzen, die man in der Import-Schnittstelle auflistet.
    - Modul ist ersetzbar durch ein Modul gleicher Export-Schnittstelle.
    - Ob Export- und Import-Schnittstellen zusammenpassen wird vom **Linker** überprüft.

Modulares Design in C

- Modul = C-Datei + dazugehörige Header-Datei
- Information Hiding mit `static`
  - Globale Variablen und Funktionen, die im Modul verborgen bleiben sollen (nur im Modul kann man darauf zugreifen), werden mit dem Schlüsselwort `static` versehen.
  - Diese Funktionen können aus anderen Dateien nicht aufgerufen werden.
- Trennung von Schnittstelle und Implementierung mit **Header-Datei**
  - Die Header-Datei enthält die Funktionsprototypen und entspricht damit der Schnittstelle.
    - Eine Header-Datei kann auch noch globale Variablen, Structs, etc. **deklarieren**.
  - Die C-Datei, die die Funktionen einer Header-Datei realisiert, stellt die Implementierung dar.
  - C- und Header-Datei bilden eine logische Einheit.
    - Sollte sich auch im Dateinamen ausdrücken.
    - Gleicher Name, unterschiedliche Dateinamenserweiterung.

Modulares Design in C (schematisches Beispiel)



Datei **datei2.c** benutzt Teile (zum Beispiel Funktionen) aus **datei1.c**. Mit Hilfe von **datei1.h** werden die Funktionsprototypen der **datei1.c** in **datei2.c** bekannt gemacht.

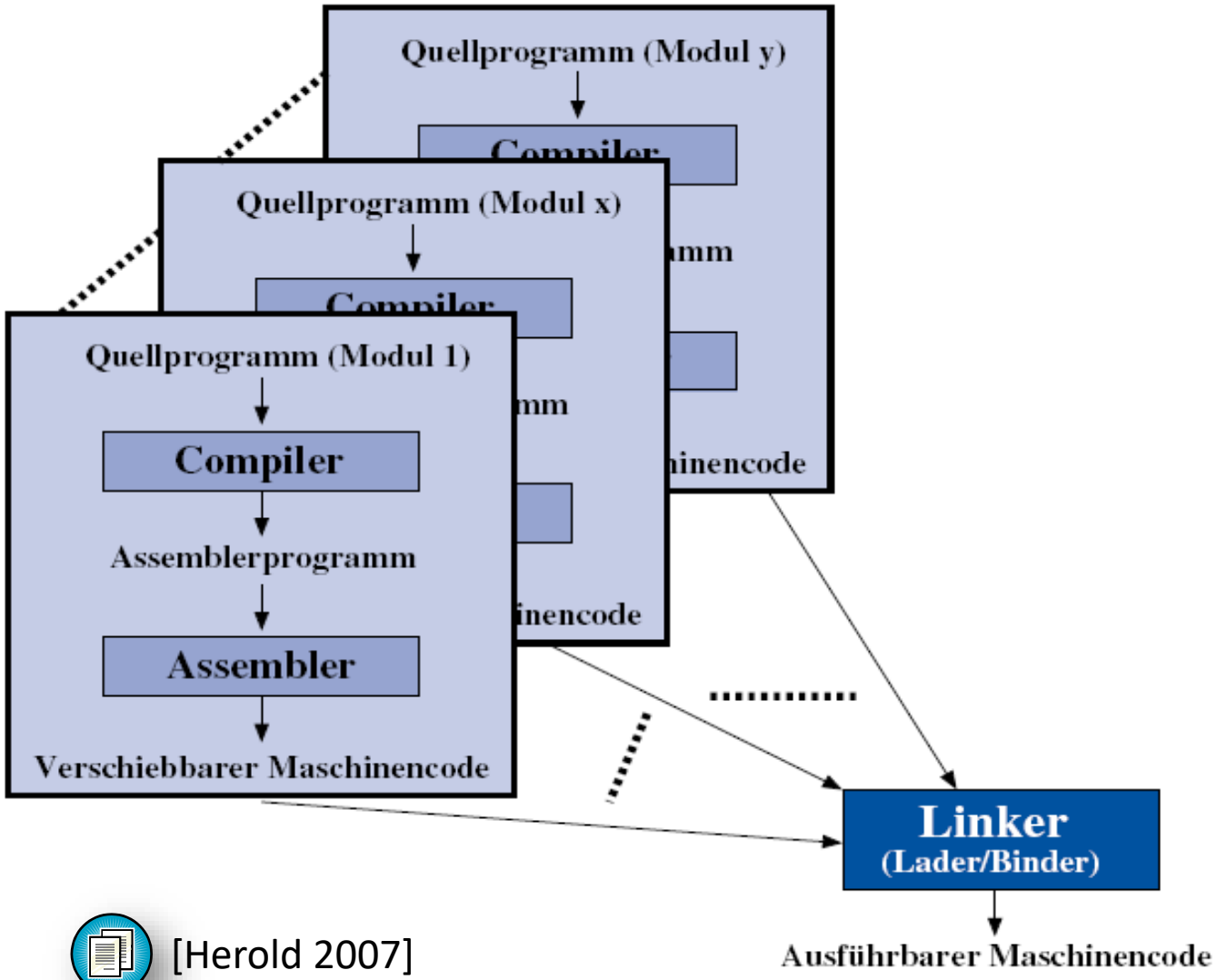
Einfache Regeln für Header-Dateien

- Jede Funktion, die **außerhalb** der Datei benutzt werden soll, in der sie **definiert** ist, erhält einen **Prototypen** in einer **Header-Datei.**
- Jede Datei, die einen **Aufruf einer Funktion** (nicht in der Datei definiert) enthält**, inkludiert** die entsprechende Header-Datei.
- Die Quelldatei, die die Definition einer Funktion enthält, **soll** die Header-Datei ebenfalls inkludieren (erstes include).

Programme aus mehreren Dateien in C

- C unterstützt eine getrennte Übersetzung
- Jedes Modul kann für sich getrennt übersetzt werden (**Translation-Unit**).
- Unterschiedliche Module können sowohl zu unterschliedlichen Zeitpunkten, als auch gleichzeitig, übersetzt werden.
- Die Übersetzung eines Moduls resultiert in eine Objekt-Datei (binär).
- Anschließend, nachdem alle Translation-Units übersetzt wurden, können die Objekt-Dateien mithilfe des **Linkers** yu einem lauffähigen Program (Executable) verbunden werden.
- Alternative kann der **Linker** statt einem Executable auch eine Library erzeugen. Die ist dann zwar nicht ausführbar, kann aber von anderen Programmen verwendet werden. Mehr dazu in Betriebssysteme.

Linker (1)

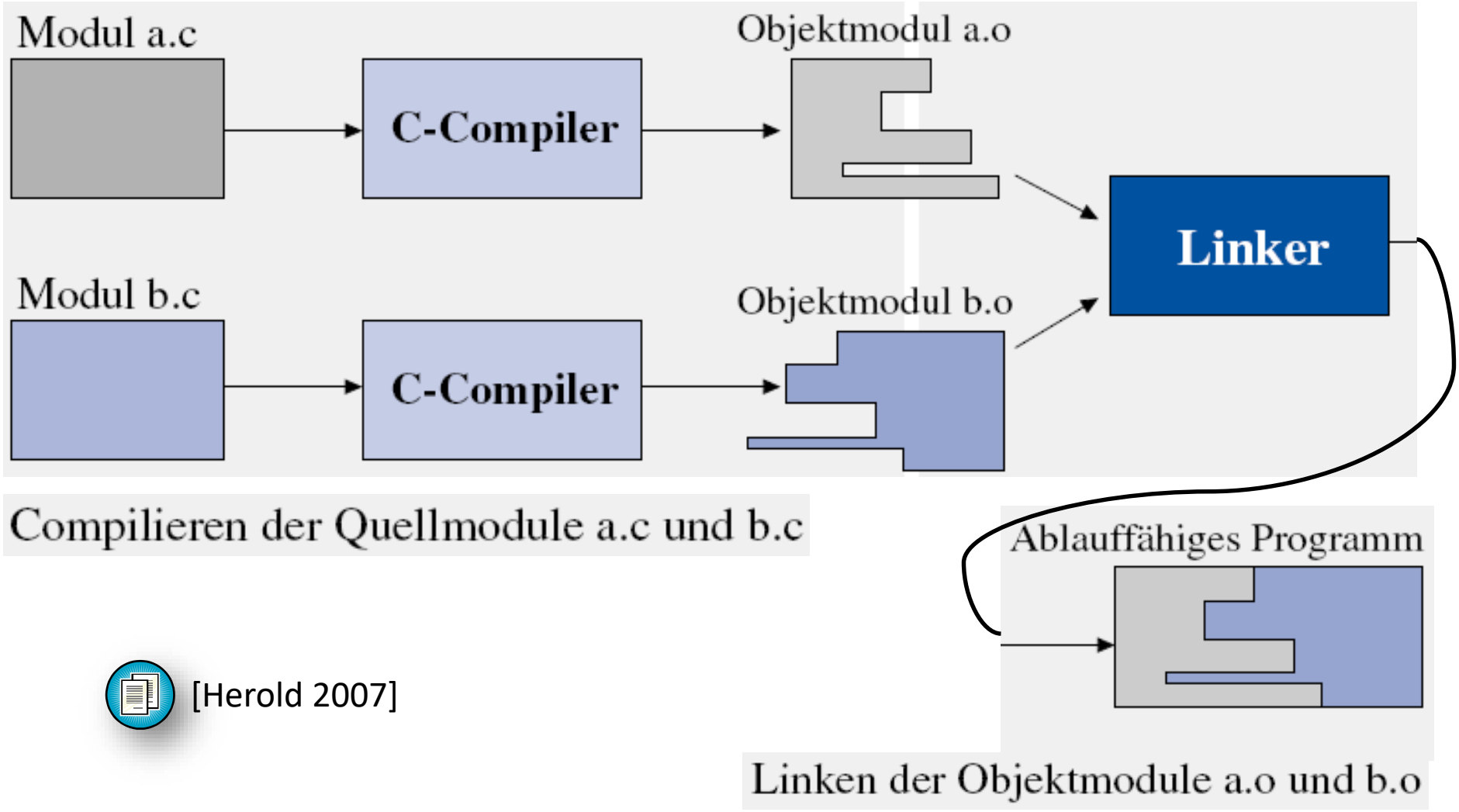


[Herold 2007]

Linker (2)

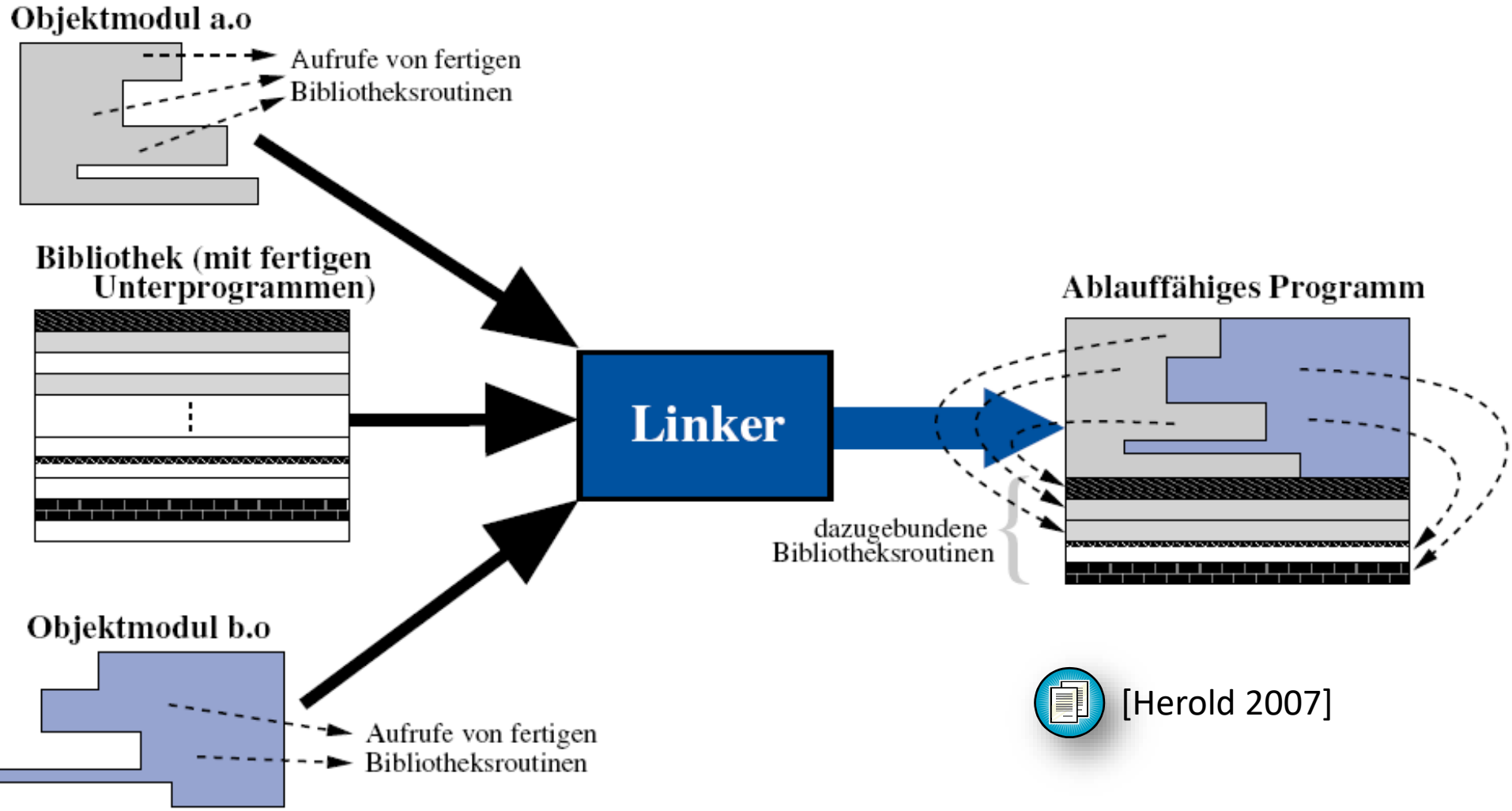
- Der Linker ist normalerweise ein eigenes Programm, das unabhängig vom Compiler ist und die vom Compiler übersetzten Objekt-Dateien als input erhält.
- Diese Dateien erkennt man an der Endung .o (oder .obj).
- Ohne Sondereinstellungen nimmt der Linker auch die C Standardlibrary hinzu. Hier findet sich dann der entsprechende Code für die Funktionen der Standard Library (z.B. `printf`).
- Werden Funktionen aus anderen Libraries verwendet (z.B. GMP), so muss dem Linker explizit mitgeteilt werden, dass er diese Library ebenfalls verwenden soll (-lgmp beim GCC aufruf)

Linker (3)



[Herold 2007]

Linker (4)



[Herold 2007]

Beispiel (stack.h)

```
#ifndef EXAMPLE_STACK_H_
#define EXAMPLE_STACK_H_

enum stack_errors {
      STACK_OK,
      STACK_FULL,
      STACK_EMPTY
};

enum stack_errors stack_push(int);
enum stack_errors stack_pop(int*);

#endif
```

Bedingte Übersetzung!
Beim ersten Einbinden ist EXAMPLE_STACK_H_ noch nicht vorhanden.
Genannt Header Guards, oft wird auch #pragma once verwendet

Beispiel (stack.c)

```c
#include "stack.h"

#include <stdio.h>
#include <stdlib.h>

#define MAX 5

static int array[MAX];
static int topOfStack = 0;

static void print_error(enum stack_errors errNr) {
	static char *errorStrings[] = { "OK", "Stack is full!", "Stack is empty!" };
	char *errString =  (errNr < 0 || errNr > STACK_EMPTY) ? "Undefined error!" : errorStrings[errNr];
	fprintf(stderr, "Error nr. %d: %s\n", errNr, errString);
}

enum stack_errors stack_push(int number) {
	if (topOfStack == MAX) {
		print_error(STACK_FULL);
		return STACK_FULL;
	} else {
		array[topOfStack++] = number;
		return STACK_OK;
	}
}

enum stack_errors stack_pop(int* number) {
	if (topOfStack == 0) {
		print_error(STACK_EMPTY);
		return STACK_EMPTY;
	} else {
		*number = array[--topOfStack];
		return STACK_OK;
	}
}
```

Beispiel (stacktest.c – benutzt die Stack-Implementierung)

```c
#include <stdio.h>
#include <stdlib.h>

#include "stack.h"

int main(void) {
	int count, run, number;
	enum stack_errors status = STACK_OK;
	printf("How many numbers should be read?: ");
	scanf("%d", &count);
	for (run = 0; run < count && status == STACK_OK; run++) {
		printf("Please insert a number: ");
		scanf("%d", &number);
		status = stack_push(number);
	}
	if (status == STACK_OK) {
		printf("Printing all numbers:\n");
		for (; run > 0 && status == STACK_OK; run--) {
			status = stack_pop(&number);
			printf("Number %d was %d\n", run, number);
		}
	}
	return EXIT_SUCCESS;
}
```

Literatur

- [Herold 2007] H. Herold, B. Lurz, J. Wohlrab, **Grundlagen der Informatik**, Pearson, 1. Auflage, 2007