Grundlagen der Informatik

Modularisierung

Prof. Dr. Peter Jüttner

Modularisierung

Programme

- sind meist (relativ) groß (> 1000-10000000 Zeilen)
- werden sehr oft in (örtlich und/oder organisatorisch) verteilten Entwicklungsteams erstellt
- nutzen sehr oft vorhandene Bibliotheken
- werden oft sequentiell oder parallel in verschiedenen Versionen entwickelt

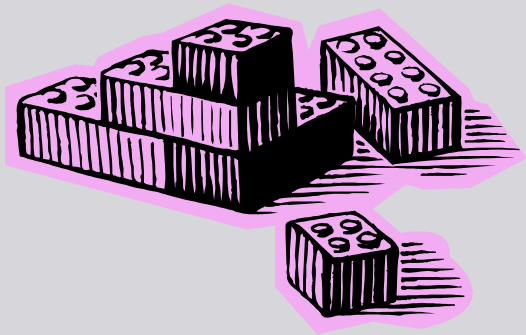
Modularisierung

Folgerung

➔ Eine ***Modularisierung*** ist notwendig

➔ Zerlegung in unabhängig voneinander zu erstellende SW-Teile (***Module***)

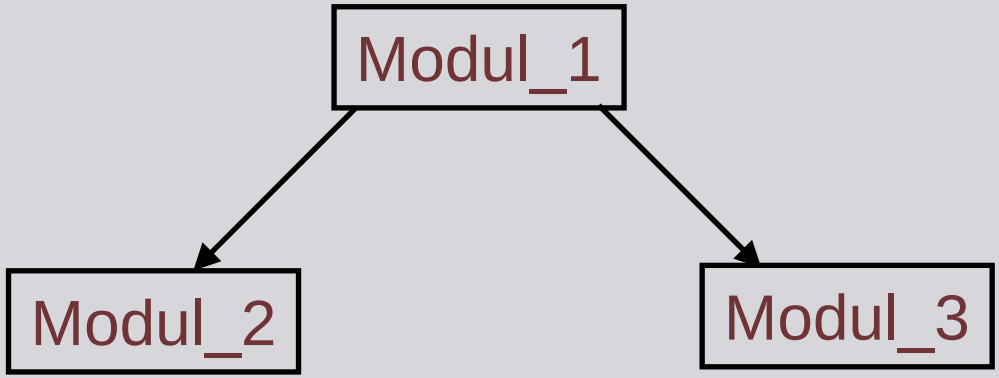
➔ Module sind getrennt voneinander übersetzbar



Modularisierung

Folgerung

➔ Grundlage der Zerlegung ist eine ***vor (!)*** der Codierung definierte ***SW Architektur***



➔ Die Gesamtfunktion des Programms wird durch das „Zusammenspiel“ der einzelnen Module realisiert

Modularisierung

Prinzipien

- Elemente eines Moduls
  - Konstante
  - Variable
  - Funktionen und Prozeduren

  werden in anderen Modulen verwendet

Modularisierung

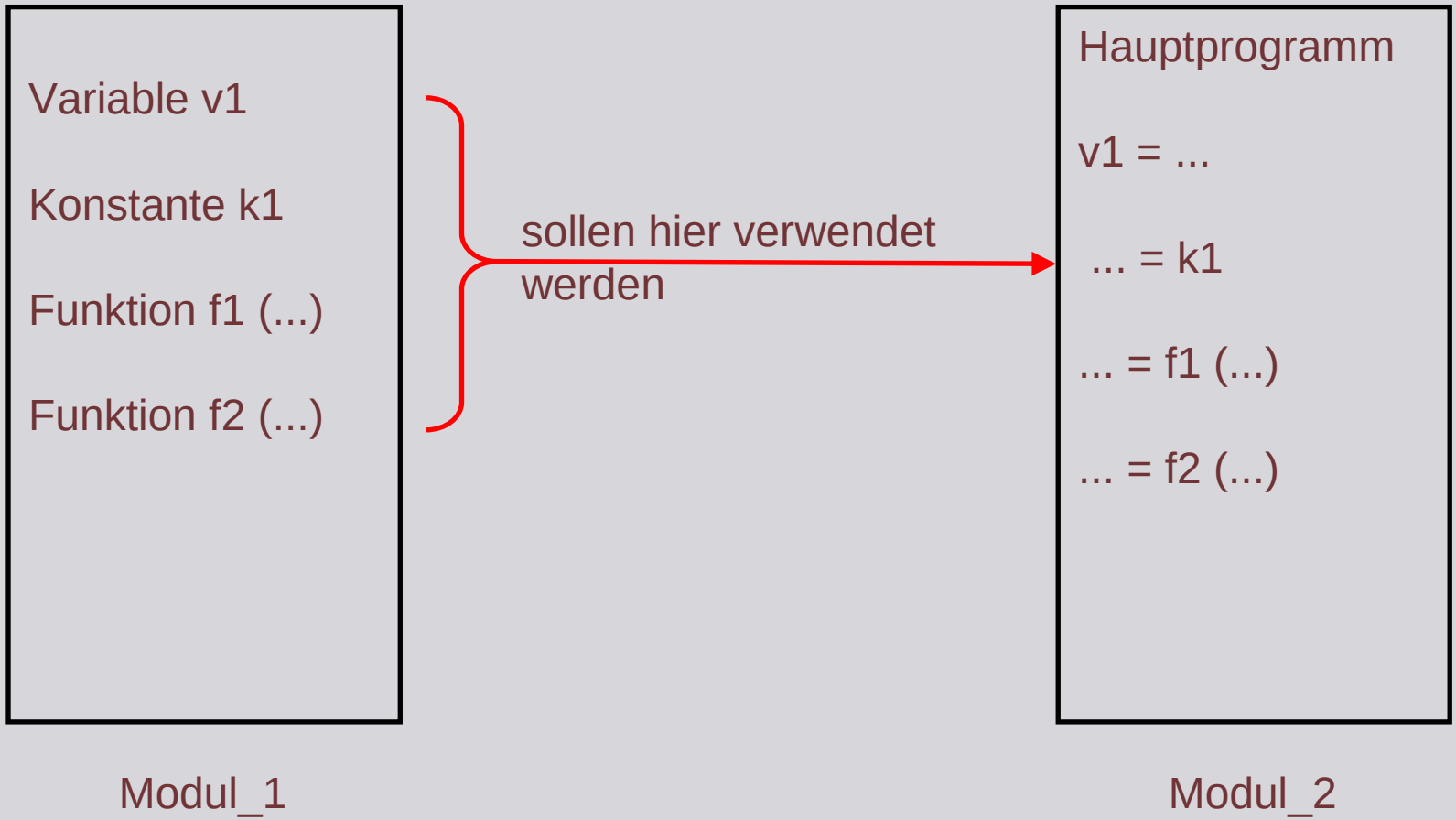
Prinzipien

- per #define definierte
  - Konstante
  - Makros

  werden in anderen Modulen verwendet

Modularisierung

Beispiel



Modularisierung

Realisierung in C

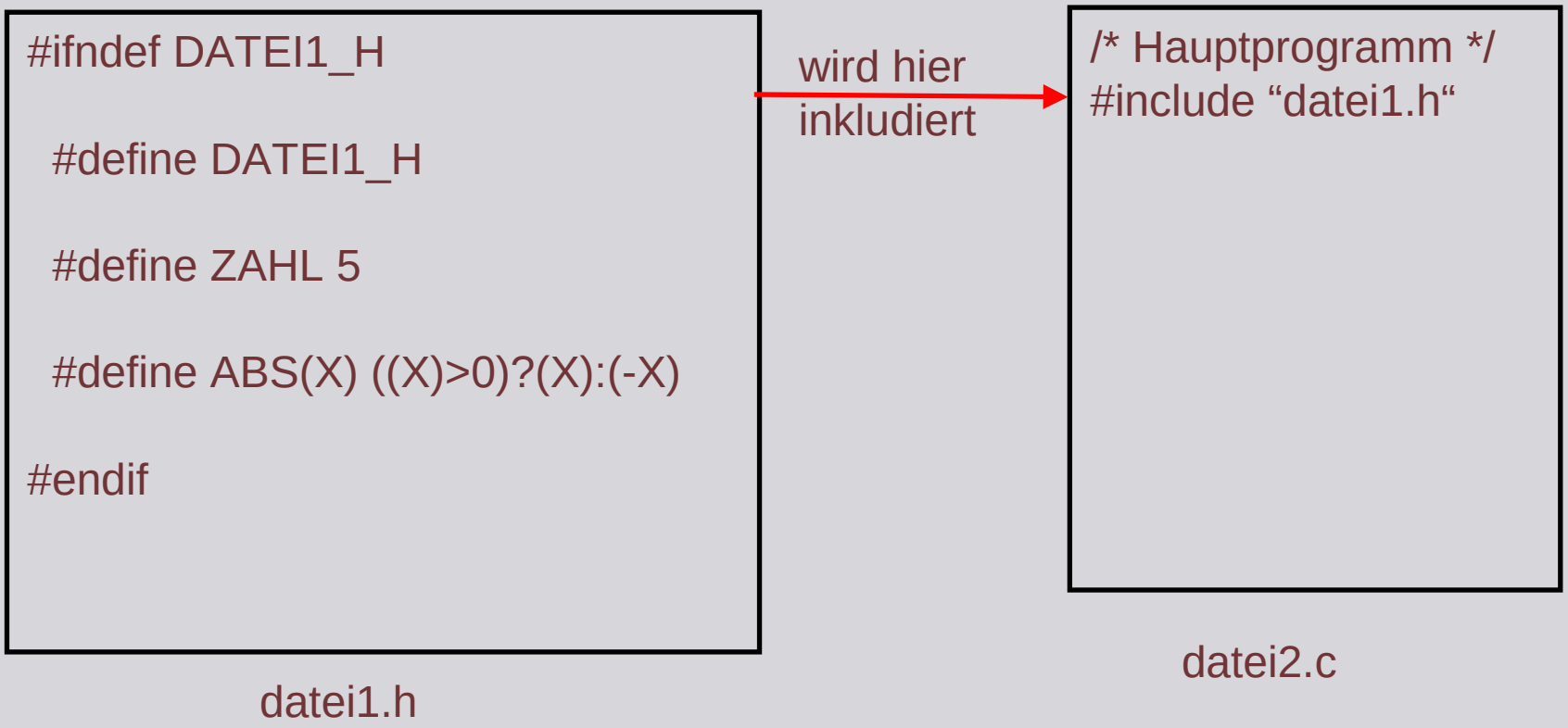
1. per #define festgelegt Größen und Makros
   - werden in einem sog. ***Headerfile*** (z.B. datei1.h*)) definiert
   - dieses Headerfile wird überall dort, wo benötigt, per #include-Anweisung (z.B. #include “datei1.h“) einkopiert

*) Headerfiles werden (traditionell) mit Suffix ***.h*** benannt

Modularisierung

Realisierung in C

1. per #define festgelegt Größen und Makros



Modularisierung

Realisierung in C

1. per #define festgelegt Größen und Makros

```
#ifndef DATEI1_H

  #define DATEI1_H

  #define ZAHL 5

  #define ABS(X) ((X)>0)?(X):(-X)

#endif
```

datei1.h

verhindert mehrfaches Includieren der Headerdatei in einer anderen Datei:
Beim 1. Includieren ist das Symbol DATEI1_H noch nicht definiert ➔ Header wird includiert und gleichzeitig wird das Symbol DATEI1_H definiert. Sollte die Headerdatei über andere Header erneut includiert werden, ist DATEI1_H definiert und der Rumpf wird vom Präprozessor überlesen

Modularisierung

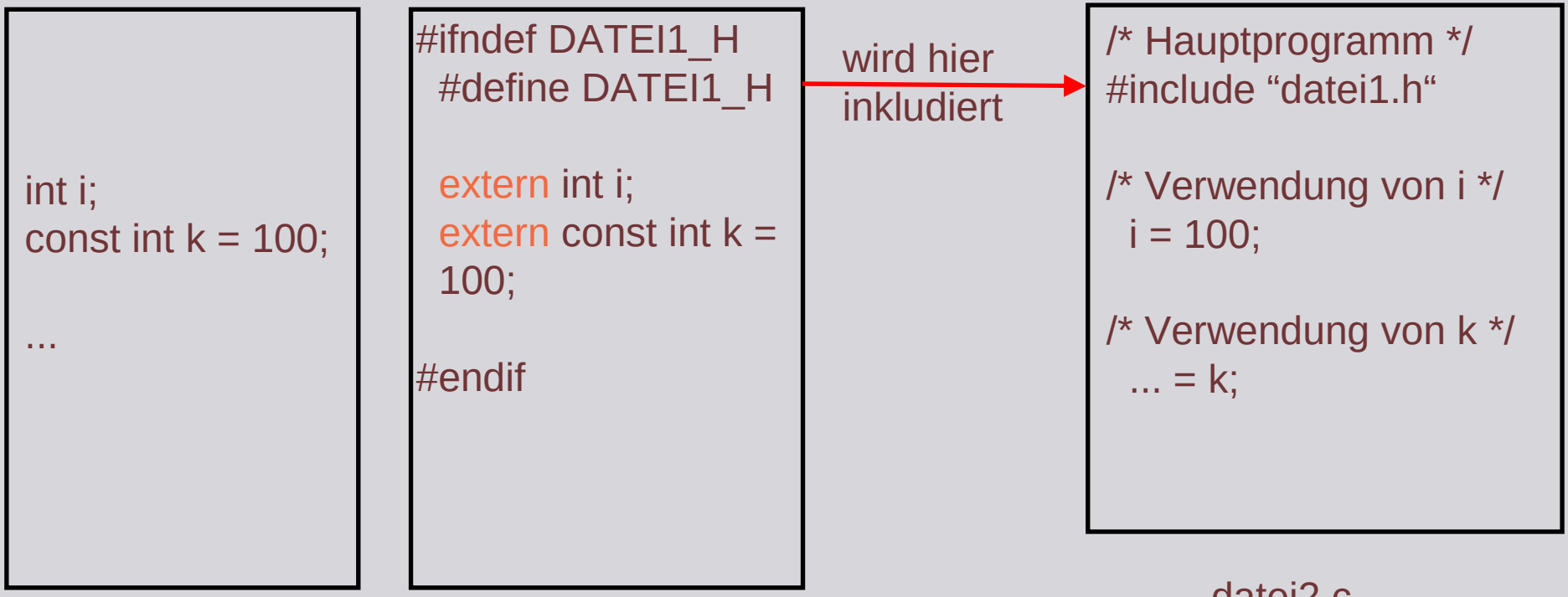
Realisierung in C

2. Eine Konstante (per const definiert) oder Variable
   - wird in einer .c-Datei (Datei1.c bzw. Dati1.cpp) global (!) definiert (wie gewohnt)
   - und über eine ***extern-Deklaration*** in einem Headerfile (Datei1.h) in anderen Modulen (z.B. Datei2.c bzw. Datei2.cpp) per include bekannt gemacht (wird auch als ***Exportieren*** bezeichnet)
   - kann dann (wie gewohnt) in Datei2.c gelesen oder geschrieben werden

Modularisierung

Realisierung in C

2. Eine Konstante (per const definiert) oder Variable



Modularisierung

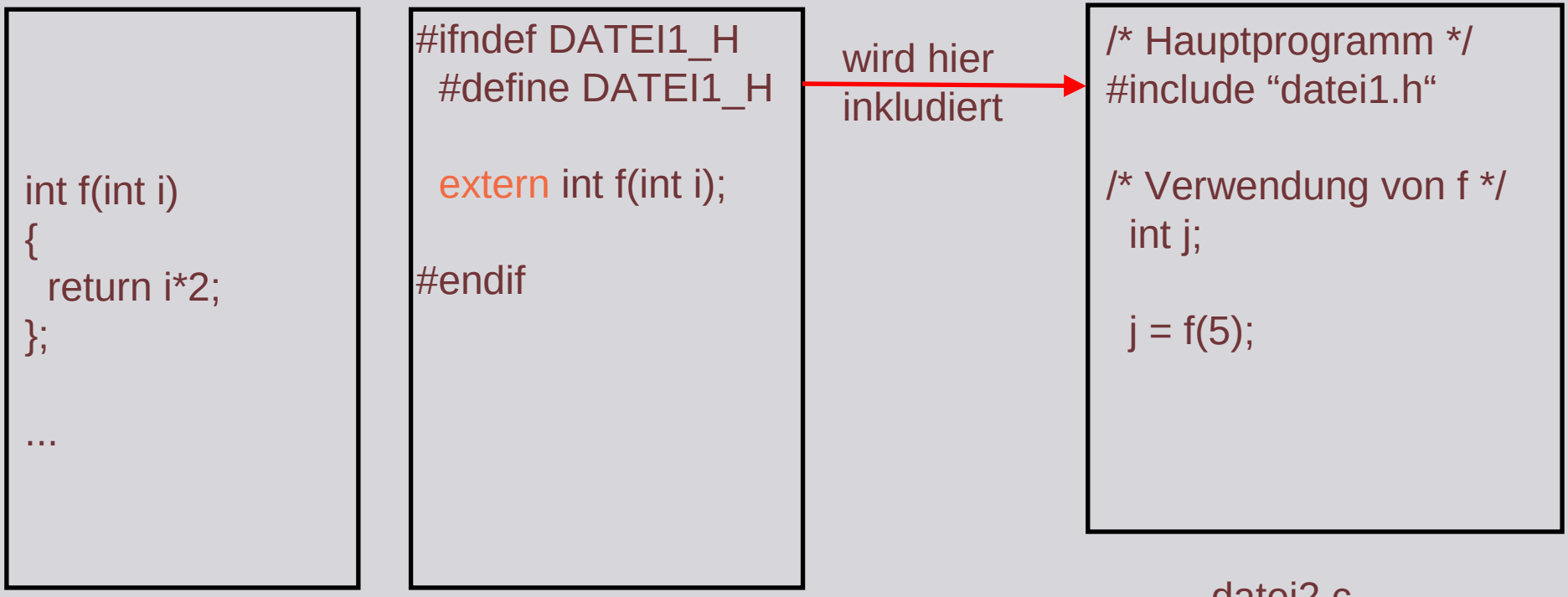
Realisierung in C

3. C-Funktion oder Prozedur
   - wird in einer .c-Datei (Datei1.c bzw. Datei1.cpp) global (!) definiert (wie gewohnt)
   - und über eine extern-Deklaration in einem Headerfile (Datei1.h) in anderen Modulen (z.B. Datei2.c bzw. Datei2.cpp) per include bekannt gemacht
   - kann dann (wie gewohnt) in Datei2.c aufgerufen werden

Modularisierung

Realisierung in C

3. C-Funktion oder Prozedur



Modularisierung

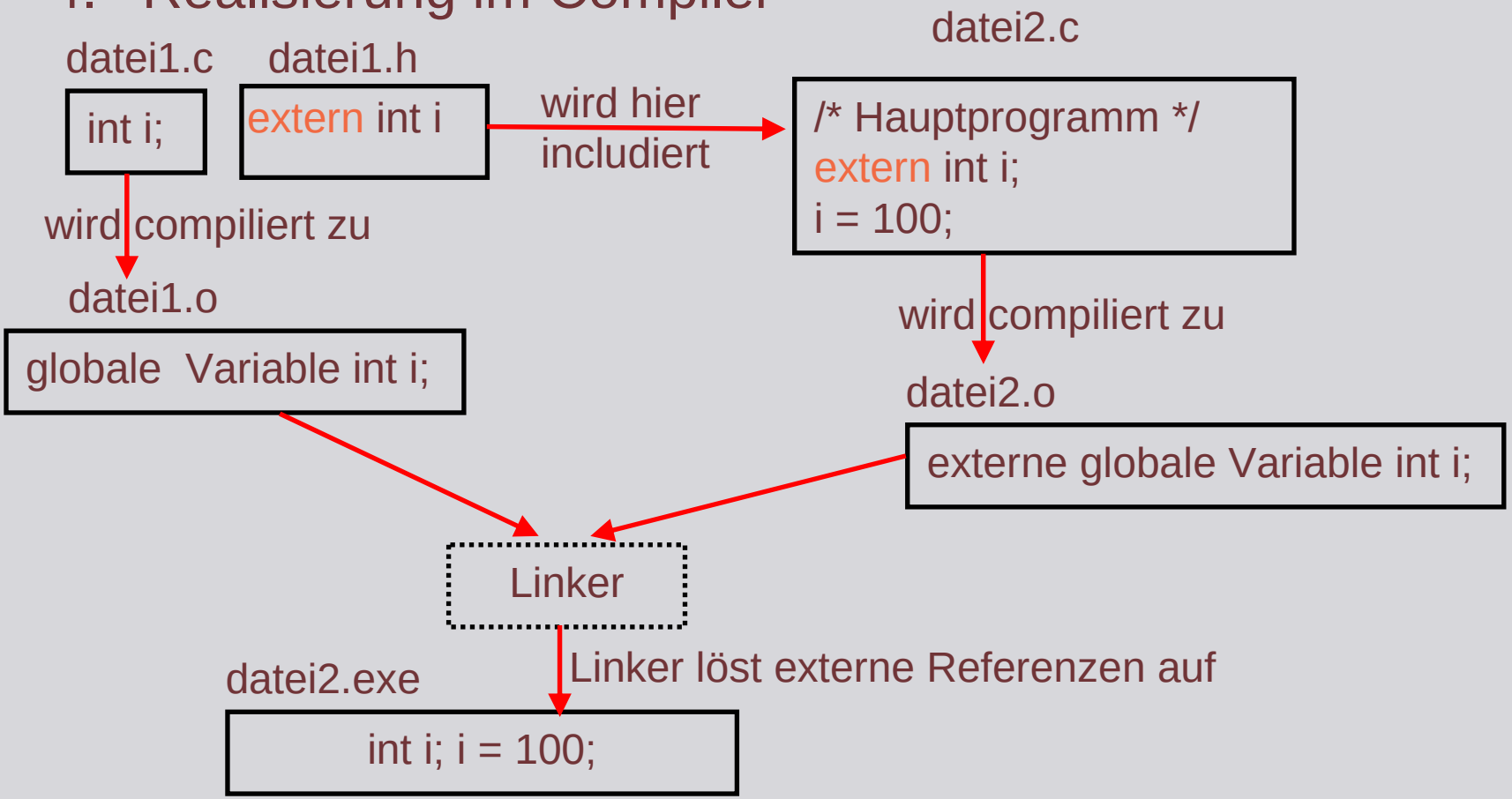
Realisierung in C

4. Realisierung im Compiler
   - für „normale“, d.h. nicht importierte Variable wird Speicherplatz angelegt
   - für per #include importierte Variable wird kein Speicherplatz angelegt, sondern nur eine externe Referenz
   - externe Referenzen werden vom Linker, wenn das Programm zusammengebaut wird, aufgelöst

Modularisierung

Realisierung in C

4. Realisierung im Compiler



Modularisierung

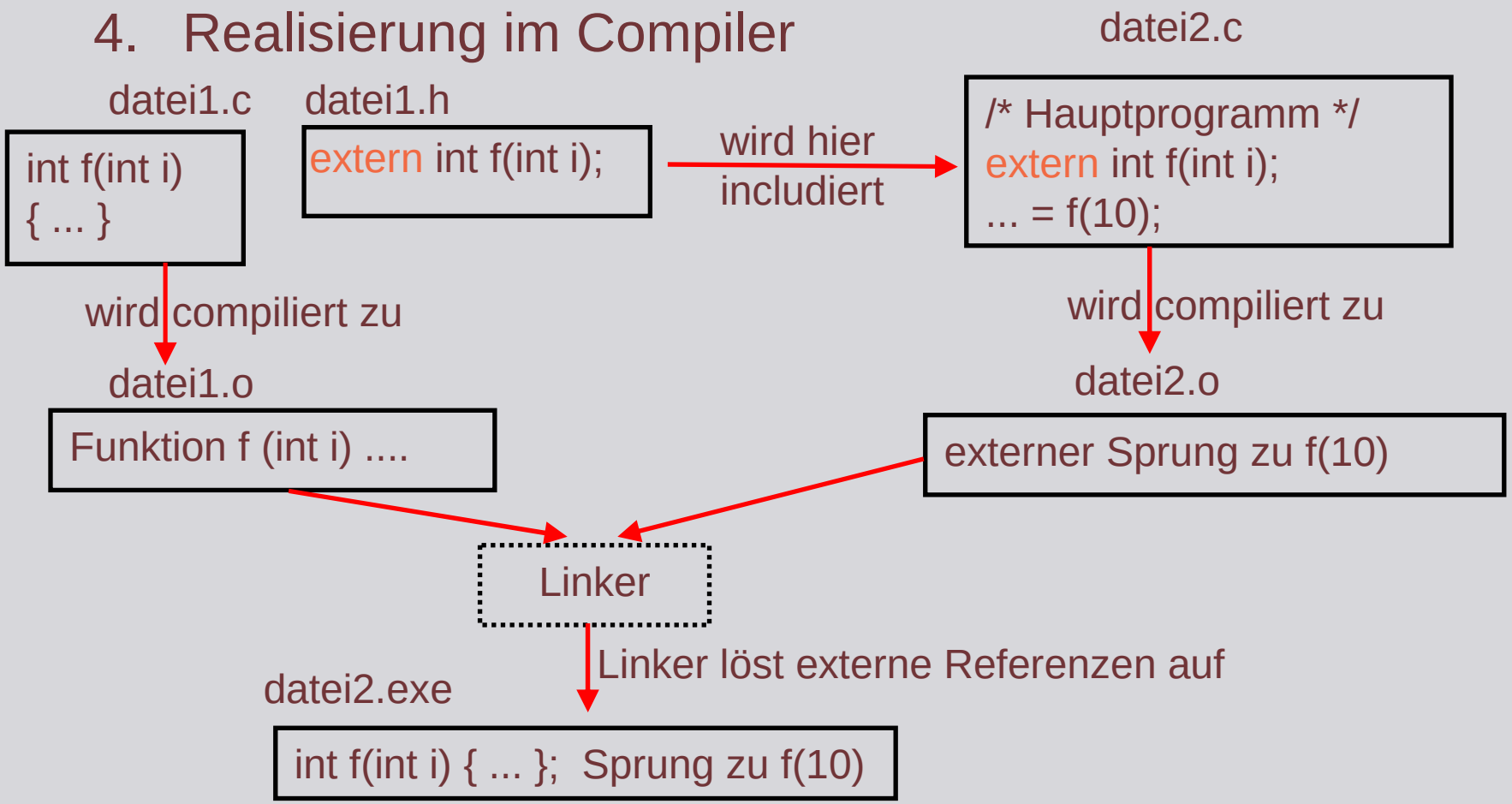
Realisierung in C

4. Realisierung im Compiler
   - für „normale“, Funktionen und Prozeduren wird Code generiert
   - für per #include importierte Funktionen wird kein Code angelegt, sondern nur eine externe Aufruf-Referenz
   - externe Referenzen werden vom Linker, wenn das Programm zusammengebaut wird, aufgelöst

Modularisierung

Realisierung in C

4. Realisierung im Compiler



Modularisierung

Realisierung in C

5. zu beachten ...
   - globale Namen müssen eindeutig sein
   - der Compiler unterscheidet i.d.R. die ersten 31 Buchstaben eines Bezeichners
   - Der Linker unterscheidet manchmal nur die ersten 8 Zeichen eines Bezeichners
   - irgendein Modul muss das Hauptprogramm (main()) enthalten

Modularisierung

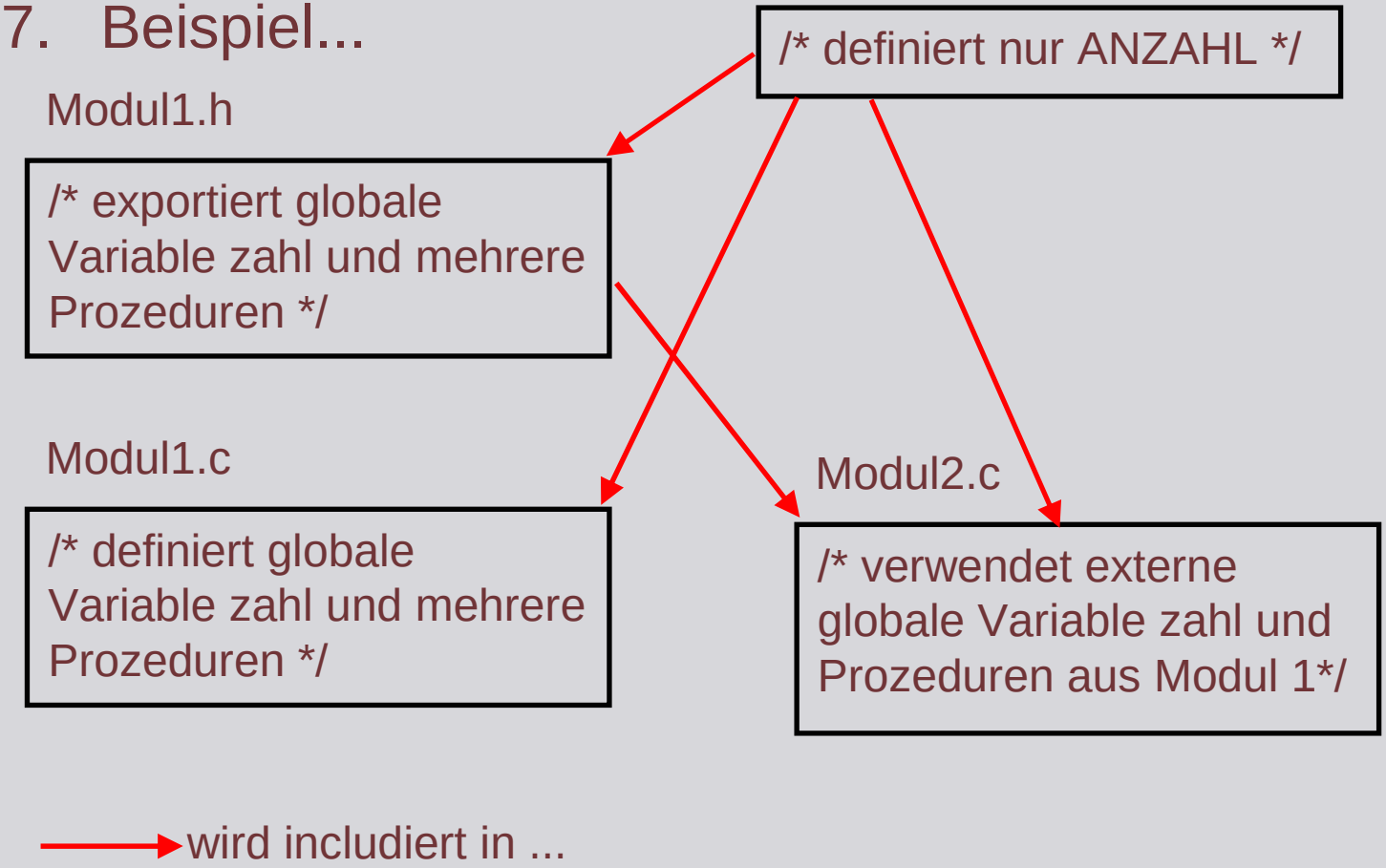
Realisierung in C

6. Programmierstil ...
   - Verwendung von Funktionen und Prozeduren modulübergreifend ist OK
   - globale Variable möglichst vermeiden und Information über funktionale Schnittstellen austauschen
     - ➔ Trennung von Schnittstelle und Implementierung
     - ➔ änderungsfreundlicher
     - ➔ Schutz vor unerwünschtem Überschreiben

Modularisierung

Realisierung in C

7. Beispiel...



Modularisierung

Realisierung in C

7. Beispiel...

Global_Header.h

```
#ifndef GLOBAL_HEADER1_H
  #define GLOBAL_HEADER1_H
  #define ANZAHL 100
#endif
```

Modul1.h

```
#ifndef HEADER1_H
#define HEADER1_H
  #include "Global_Header.h"
  extern int zahl;
  extern void einlesen_text(char t[ANZAHL]);
  extern void einlesen_zahl(int* i);
  extern void einlesen_zahl(float* f);
#endif
```

Modularisierung

Realisierung in C

7. Beispiel...

Modul1.c

```
#include "Global_Header.h"
#include <stdio.h>
int zahl;
void einlesen_text(char t[ANZAHL])
{ printf("Bitte Text eingeben\n");
  gets(t);
};
void einlesen_zahl(int* i)
{ printf("Bitte Zahl eingeben\n");
  scanf("%d",i);
};
void einlesen_zahl(float* f)
{ printf("Bitte Zahl eingeben\n");
  scanf("%f",f);
};
```

Modularisierung

Realisierung in C

7. Beispiel...

Modul2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Global_Header.h"
#include "Modul_1.h"
void ausgeben_text(char t[ANZAHL])
{ printf("Textausgabe:\n%s\n",t); };
void ausgeben_zahl(int i)
{ printf("Textausgabe:\n%d\n",i); };
int main()
{ char text[ANZAHL];
  einlesen_text(text);
  einlesen_zahl(&zahl);
  ausgeben_text(text);
  ausgeben_zahl(zahl);
  ...
};
```

Modularisierung

Zum Schluss dieses Abschnitts ...