7 Funktionsdeklarationen — Beispiel

```
#include <stdio.h>
#include <math.h>

double sinus(double);
/* oder: double sinus(double x); */

int main()
{
     double wert;

     printf("Berechnung des Sinus von ");
     scanf("%lf", &wert);
     printf("sin(%lf) = %lf\n",
          wert, sinus(wert));
     return(0);
}
```

```
double sinus (double x)
{
     double summe;
     double x_quadrat;
     double rest;
     int k;

     k = 0;
     summe = 0.0;
     rest = x;
     x_quadrat = x*x;

     while ( fabs(rest) > 1e-9 ) {
          summe += rest;
          k += 2;
          rest *= -x_quadrat/(k*(k+1));
     }
     return(summe);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
  - ➤ call by value
  - ➤ call by reference

call by value

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
  - ➥ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
  - ➥ die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne dass dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
  - ➥ die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Parameterübergabe an Funktionen (2)

call by reference

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen
  - ➥ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
  - ➥ wenn die Funktion den Wert des formalen Parameters verändert, ändert sie den Inhalt der Speicherzelle des aktuellen Parameters
  - ➥ auch der Wert der Variablen (aktueller Parameter) beim Aufrufer der Funktion ändert sich dadurch

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D.9 Programmstruktur & Module

1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
  - ◆ Top-down Entwurf / Prozedurale Programmierung
    - ➤ traditionelle Methode
    - ➤ bis Mitte der 80er Jahre fast ausschließlich verwendet
    - ➤ an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
  - ◆ Objekt-orientierter Entwurf
    - ➤ moderne, sehr aktuelle Methode
    - ➤ Ziel: Bewältigung sehr komplexer Probleme
    - ➤ auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

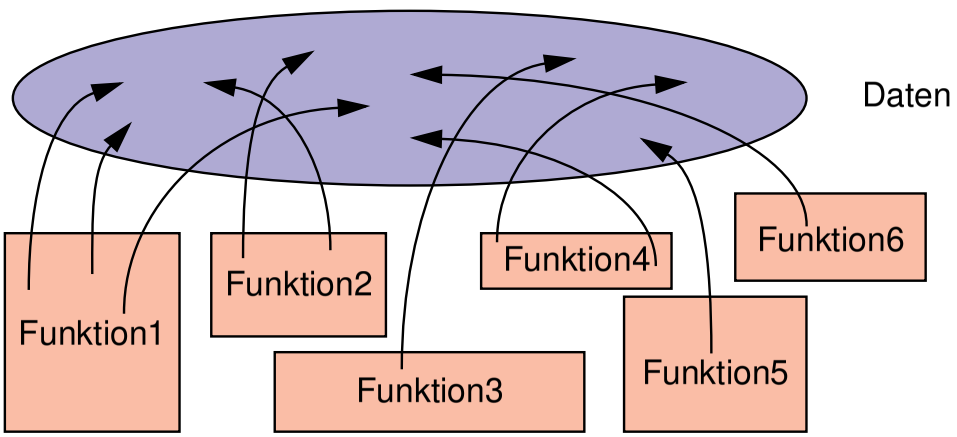
2 Top-down Entwurf

- Zentrale Fragestellung
  - ◆ was ist zu tun?
  - ◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?
    - ➤ Beispiel: Rechnung für Kunden ausgeben
      - – Rechnungspositionen zusammenstellen
        - – Lieferungsposten einlesen
        - – Preis für Produkt ermitteln
        - – Mehrwertsteuer ermitteln
      - – Rechnungspositionen addieren
      - – Positionen formatiert ausdrucken

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Top-down Entwurf (2)

- Problem:
  Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:
  Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten

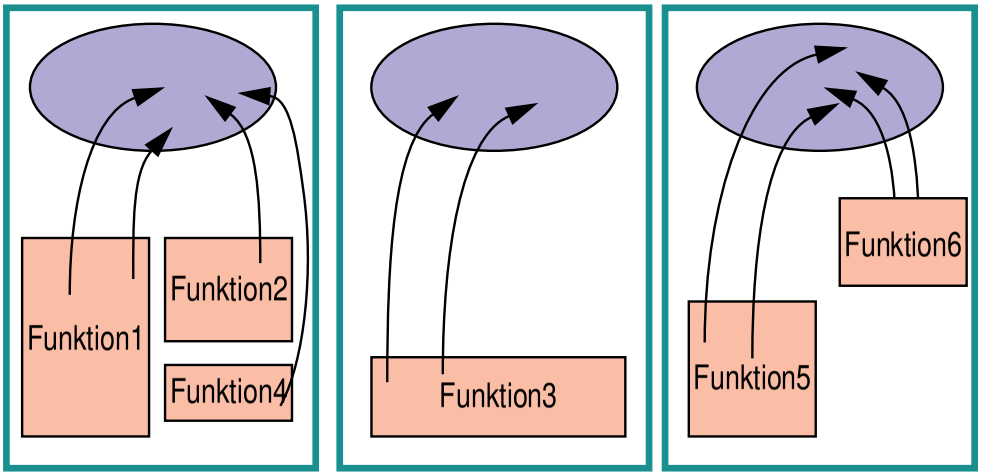


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Top-down Entwurf (3) Modul-Bildung

- Lösung:
  Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

  ➥ **Modul**



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefasst werden

  ➥ Modul
- Jede C-Quelldatei kann separat übersetzt werden (Option **`-c`**)
  - ➤ Zwischenergebnis der Übersetzung wird in einer **`.o`**-Datei abgelegt

```
% cc -c main.c          (erzeugt Datei main.o)
% cc -c f1.c            (erzeugt Datei f1.o)
% cc -c f2.c f3.c       (erzeugt f2.o und f3.o)
```

- Das Kommando **`cc`** kann mehrere **`.c`**–Dateien übersetzen und das Ergebnis — zusammen mit **`.o`**–Dateien — binden:

```
% cc -o prog main.o f1.o f2.o f3.o f4.c f5.c
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Module in C

!!! `.c`–Quelldateien auf keinen Fall mit Hilfe der `#include`–Anweisung in andere Quelldateien einkopieren

- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
  - ➤ Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu ***Header*-Dateien** zusammengefasst
  - ◆ *Header*-Dateien werden mit der `#include`–Anweisung des Präprozessors in C-Quelldateien einkopiert
  - ◆ der Name einer *Header*-Datei endet immer auf `.h`

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Gültigkeit von Namen

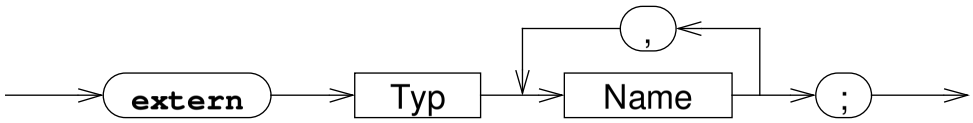
- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
  1. Global im gesamten Programm (über Modul- und Funktionsgrenzen hinweg)
  2. Global in einem Modul (auch über Funktionsgrenzen hinweg)
  3. Lokal innerhalb einer Funktion
  4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
  - ➤ eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
  - ➤ eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Varibalen in umgebenden Blöcken

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden
  (**`extern`–Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;
extern char c;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Globale Variablen (2)

- **Probleme mit globalen Variablen**
  - Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
  - Funktionen können Variablen ändern, ohne dass der Aufrufer dies erwartet (Seiteneffekte)
  - Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➥ **globale Variablen möglichst vermeiden!!!**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Globale Funktionen

- Funktionen sind generell global
  (es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden
  (= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **`extern`** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:

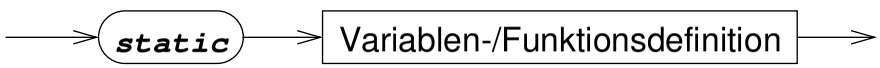
```
double sinus(double);
float power(float, int);
```

- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
  - ➤ "vertragliche" Zusicherung an den Benutzer des Moduls

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
  - ➤ Schlüsselwort ***`static`*** vor die Definition setzen



  - ➤ **`extern`**-Deklarationen in anderen Modulen sind nicht möglich
- **`Die static`**-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer *static* definiert werden
  - ➤ sie werden dadurch nicht Bestandteil der Modulschnittstelle
    (= des "Vertrags" mit den Modulbenutzern)

!!! das Schlüsselwort **`static`** gibt es auch bei lokalen Variablen
(mit anderer Bedeutung! - zur Unterschiedung ist das hier beschriebene *static* immer kursiv geschrieben)

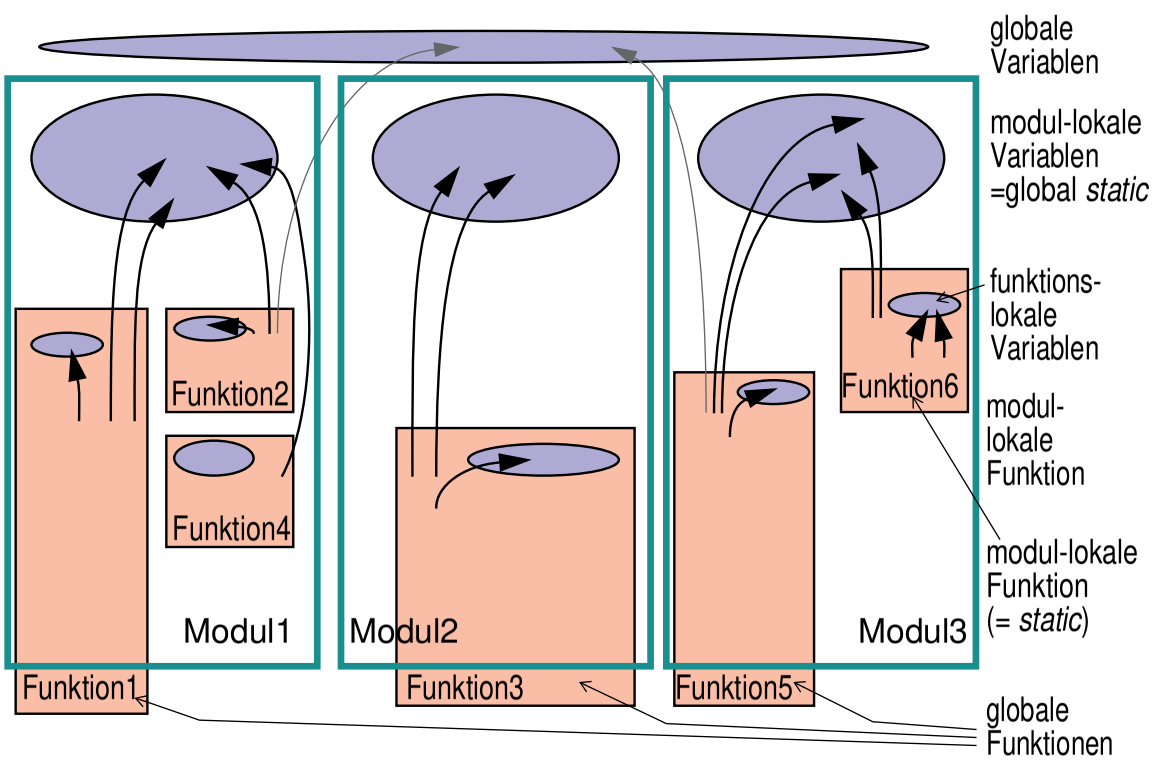
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluss auf die Zugreifbarkeit von Variablen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Gültigkeitsbereiche — Übersicht



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
    - ➤ statische (**`static`**) Variablen
  - Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
    - ➤ dynamische (**`automatic`**) Variablen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

9 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - ➤ der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - ➥ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - ➤ die Initialisierung wird bei jedem Eintritt in den Block wiederholt
  - **!!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert** (= irgendwas)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

9 Lebensdauer von Variablen (2)

static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort **`static`** eine **Lebensdauer über die gesamte Programmausführung** hinweg
  - ➥ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
  - !!! Das Schlüsselwort ***`static`*** hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
  - ➤ die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
  - ➤ erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

10 Getrennte Übersetzung von Programmteilen — Beispiel

- Hauptprogramm (Datei **`fplot.c`**)

```
#include "trig.h"
#define INTERVALL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
        char c;
        double i;

        printf("Funktion (Sin, Cos, Tan, cOt)? ");
        scanf("%x", &c);

        switch (c) {
        ...
        case 'T':
          for (i=-PI/2; i < PI/2; i+=INTERVALL)
                printf("%lf %lf\n", i, tan(i));
          break;;
        ...
        }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

10 Getrennte Übersetzung — Beispiel (2)

- *Header*-Datei (Datei **`trig.h`**)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```

- Trigonometrische Funktionen (Datei **`trigfunc.c`**)

```
#include "trig.h"

double tan(double x) {
        return(sin(x)/cos(x));
}

double cot(double x) {
        return(cos(x)/sin(x));
}

double cos(double x) {
        return(sin(PI/2-x));
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

10 Getrennte Übersetzung — Beispiel (3)

- Trigonometrische Funktionen — Fortsetzung (Datei **`trigfunc.c`**)

...

```
double sin (double x)
{
        double summe;
        double x_quadrat;
        double rest;
        int k;

        k = 0;
        summe = 0.0;
        rest = x;
        x_quadrat = x*x;

        while ( fabs(rest) > 1e-9 ) {
                summe += rest;
                k += 2;
                rest *= -x_quadrat/(k*(k+1));
        }
        return(summe);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D.10 Vom C-Quellcode zum laufenden Programm

1 Übersetzen - Objektmodule

- 1. Schritt: Präprozessor
  - entfernt Kommentare, wertet Präprozessoranweisungen aus
    - fügt include-Dateien ein
    - expandiert Makros
    - entfernt Makro-abhängige Code-Abschnitte (*conditional code*) Beispiel:

```
#define DEBUG
...
#ifdef DEBUG
    printf("Zwischenergebnis = %d\n", wert);
#endif DEBUG
```

  - Zwischenergebnis kann mit `cc -P datei.c` als `datei.i` erzeugt werden oder mit `cc -E datei.c` ausgegeben werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Übersetzen - Objektmodule (2)

- 2. Schritt: Compilieren
  - übersetzt C-Code in Assembler
  - Zwischenergebnis kann mit `cc -S datei.c` als `datei.s` erzeugt werden
- 3. Schritt: Assemblieren
  - assembliert Assembler-Code, erzeugt Maschinencode (Objekt-Datei)
  - standardisiertes Objekt-Dateifomat: ELF (Executable and Linking Format) (vereinfachte Darstellung) - in nicht-UNIX-Systemen andere Formate
    - Maschinencode
    - Informationen über Variablen mit Lebensdauer static (ggf. Initialisierungswerte)
    - Symboltabelle: wo stehen welche globale Variablen und Funktionen
    - Relokierungsinformation: wo werden welche "nicht gefundenen" globalen Variablen bzw. Funktionen referenziert
  - Zwischenergebnis kann mit `cc -c datei.c` als `datei.o` erzeugt werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Binden und Bibliotheken

- 4. Schritt: Binden
  - Programm `ld` : (*linker*), erzeugt ausführbare Datei (*executable file*)
    - ➤ ebenfalls ELF-Format (früher a.out-Format oder COFF)
  - Objekt-Dateien (.o-Dateien) werden zusammengebunden
    - ➤ noch nicht abgesättigte Referenzen auf globale Variablen und Funktionen in anderen Objekt-Dateien werden gebunden (Relokation)
  - nach fehlenden Funktionen wird in Bibliotheken gesucht
  - **statisch binden**
    - ➤ alle fehlenden Funktionen werden aus Bibliotheken genommen und in die ausführbare Datei einkopiert
- Ergebnis: ein ausführbares Programm

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Laden des ausführbaren Programms

- Windows oder Linux
  - Erzeugen einer Umgebung zur Ausführung des Programms:
    - ➥ Prozess
  - **dynamisch binden**
    - ➤ Funktionen, die von mehreren Programmen gemeinsam genutzt werden (*shared libraries, dll*), werden erst beim Start des Programms hinzugeladen
- Mikrocontroller
  - Übertragen des statisch gebundenen Programms in den Programmspeicher des Mikrocontrollers
    - ➥ Flashen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.