

# Systemprogrammierung

## Einführung in die Programmiersprache C

- Literatur zur C-Programmierung:
  - ◆ Darnell, Margolis. *C: A Software Engineering Approach*. Springer 1991
  - ◆ Kernighan, Ritchie. *The C Programming Language*. Prentice-Hall 1988

# Überblick

- ◆ Struktur eines C-Programms
- ◆ Datentypen und Variablen
- ◆ Anweisungen
- ◆ Funktionen
- ◆ C-Präprozessor
- ◆ Programmstruktur und Module
- ◆ Zeiger(-Variablen)
- ◆ sizeof-Operator
- ◆ Explizite Typumwandlung — Cast-Operator
- ◆ Speicherverwaltung
- ◆ Felder
- ◆ Strukturen
- ◆ Ein- /Ausgabe
- ◆ Fehlerbehandlung

# Struktur eines C-Programms

**globale Variablendefinitionen**

**Funktionen**

```
int main(int argc, char *argv[]) {  
    Variablendefinitionen  
    Anweisungen  
}
```

## ■ Beispiel

```
int main(int argc, char *argv[]) {  
    printf("Hello World!\n");  
    return(0);  
}
```

## ■ Übersetzen mit dem C-Compiler:

```
cc -o hello hello.c
```

## ■ Ausführen durch Aufruf von `./hello`

# Datentypen und Variablen

## ■ Datentyp := (<Menge von Werten>, <Menge von Operationen>)

- Literal            Wert im C-Quelltext (z. B. 4711, 0xff, 'a', 3.14)
- Konstante        Bezeichner für einen Wert
- Variable          Bezeichner für einen Speicherplatz,  
                          der einen Wert aufnehmen kann
- Funktion          Bezeichner für eine Sequenz von Anweisungen,  
                          die einen Wert zurückgibt

↳ Literale, Konstanten, Variablen, Funktionen haben einen (Daten-)Typ

## ■ Datentyp legt fest:

- Repräsentation der Werte im Rechner
- Größe des Speicherplatzes für Variablen
- erlaubte Operationen

## 3.1 Primitive Datentypen in C

- **Ganzzahlen/Zeichen:** `char`, `short`, `int`, `long`, `long long`
  - Wertebereich ist compiler-/prozessorabhängig  
es gilt:  $\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$
  - Zeichen werden als Zahlen im ASCII-Code (8 Bit) dargestellt
  - Zeichenketten (Strings) werden als Felder von `char` dargestellt
  
- **Fließkommazahlen:** `float`, `double`, `long double`
  - Wertebereich/Genauigkeit ist compiler-/prozessorabhängig
  
- **Leerer Datentyp:** `void`
  - Wertebereich:  $\emptyset$
  - Einsatz: Funktionen ohne Rückgabewert
  
- **Boolescher Datentyp:** `_Bool` (C99)
  - Bedingungsausdrücke (z. B. `if(...)`) sind in C aber vom Typ `int`!
  
- **Durch vorangestellte Typ-Modifizierer kann die Bedeutung verändert werden**
  - vorzeichenbehaftet: `signed`, vorzeichenlos: `unsigned`, konstant: `const`

## 3.2 Variablen

- Variablen werden definiert durch:
  - ◆ **Namen** (Bezeichner)
  - ◆ Typ
  - ◆ zugeordneten Speicherbereich für einen Wert des Typs  
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - ◆ **Lebensdauer**
  
- Variablenname
  - ◆ Buchstabe oder `_` ,  
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder `_`

## 3.2 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt (= dem Compiler bekannt gemacht)
  - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
  - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
  
- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich
  - ◆ Beispiele

```
int a1;  
float a, b, c, dis;  
int anzahl_zeilen=5;  
const char trennzeichen = ':';
```

## 3.2 Variablen (3)

- Position von Variablendefinitionen im Programm:
  - ◆ nach jeder "{"
  - ◆ außerhalb von Funktionen
  - ◆ ab C99 auch an beliebigen Stellen innerhalb von Funktionen und im Kopf von `for`-Schleifen
- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus Programmstruktur



## 3.3 Verbund-Datentypen / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit
- Strukturdeklaration

```
struct person {  
    char name[20];  
    int alter;  
};
```

- Definition einer Variablen vom Typ der Struktur

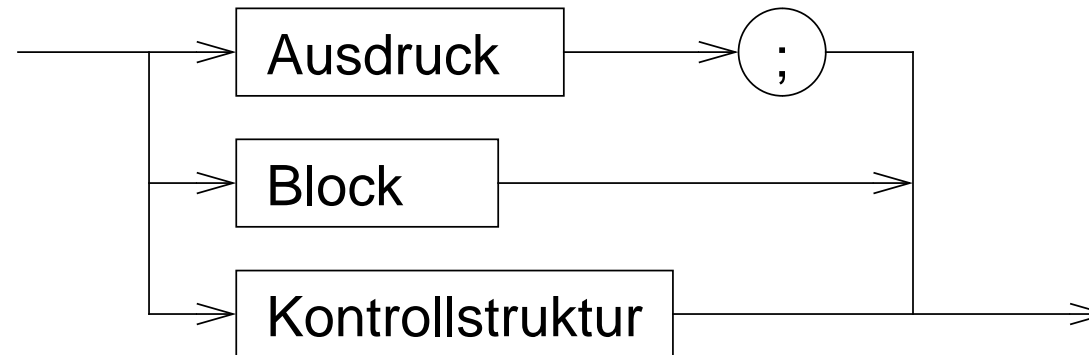
```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
p1.alter = 20;
```

# Anweisungen

Anweisung:



## 4.1 Ausdrücke - Beispiele

- ◆ `a = b + c;`
- ◆ `{ a = b + c; x = 5; }`
- ◆ `if (x == 5) a = 3;`

## 4.2 Blöcke

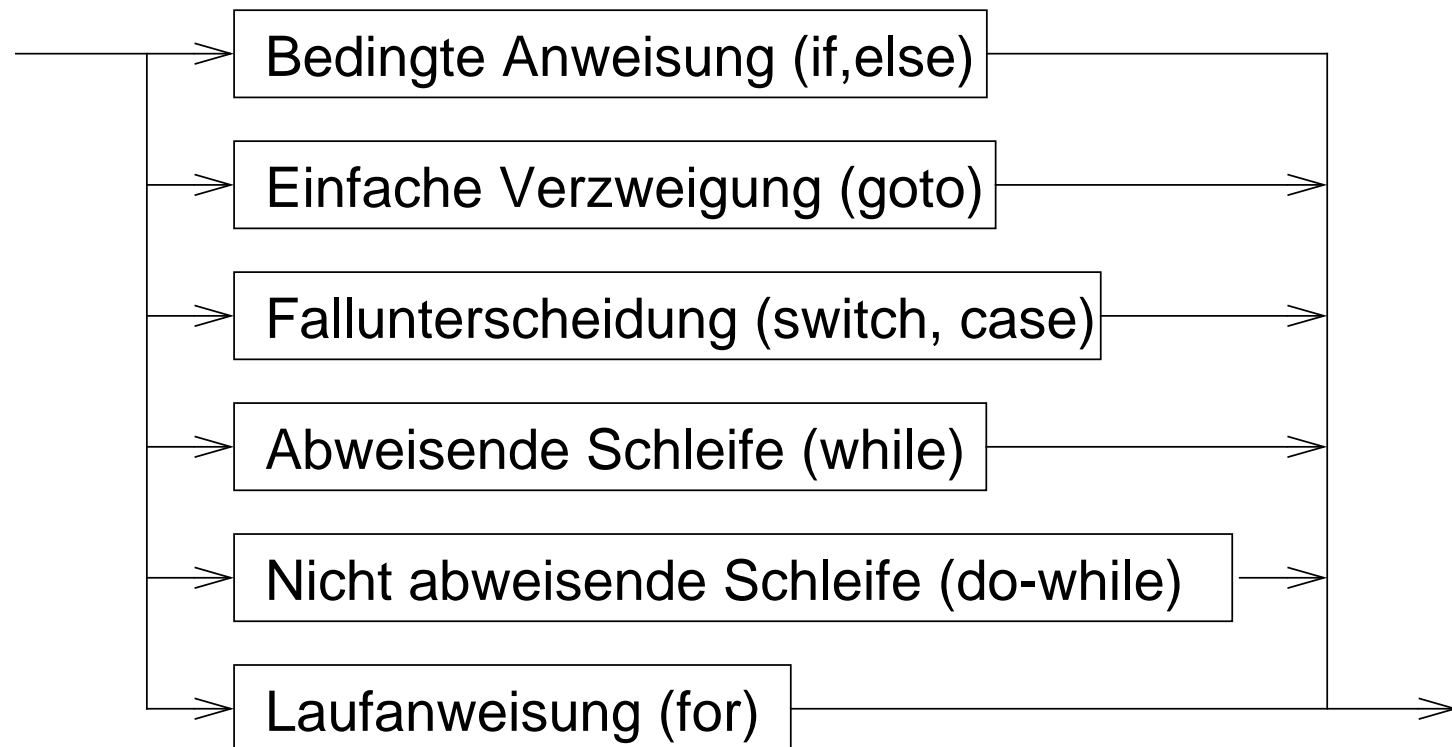
- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen

```
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}
```

## 4.3 Kontrollstrukturen

- Kontrolle des Programmablaufs in Abhängigkeit vom Ergebnis von Ausdrücken

Kontrollstruktur:



## 4.4 Kontrollstrukturen — Schleifensteuerung

### ■ break

- ◆ bricht die umgebende Schleife bzw. `switch`-Anweisung ab

```
int c;

do {
    if ( (c = getchar()) == EOF ) break;
    putchar(c);
} while ( c != '\n' );
```

### ■ continue

- ◆ bricht den aktuellen **Schleifendurchlauf** ab
- ◆ setzt das Programm mit der Ausführung des Schleifenkopfes fort

# Funktionen

- **Funktion =**  
Programmstück (Block), das mit einem **Namen** versehen ist und dem zum Ablauf **Parameter** übergeben werden können
- Funktionen sind die elementaren Bausteine für Programme
  - ↳ verringern die **Komplexität** durch Zerteilen umfangreicher, schwer überblickbarer Aufgaben in kleine Komponenten
  - ↳ erlauben die **Wiederverwendung** von Programmkomponenten
  - ↳ verbergen **Implementierungsdetails** vor anderen Programmteilen (**Black-Box-Prinzip**)

## 5.1 Funktionsdefinition

---

- Schnittstelle (Ergebnistyp, Name, Parameter)
- + Implementierung

## 5.2 Beispiel Sinusberechnung

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

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);
}
```

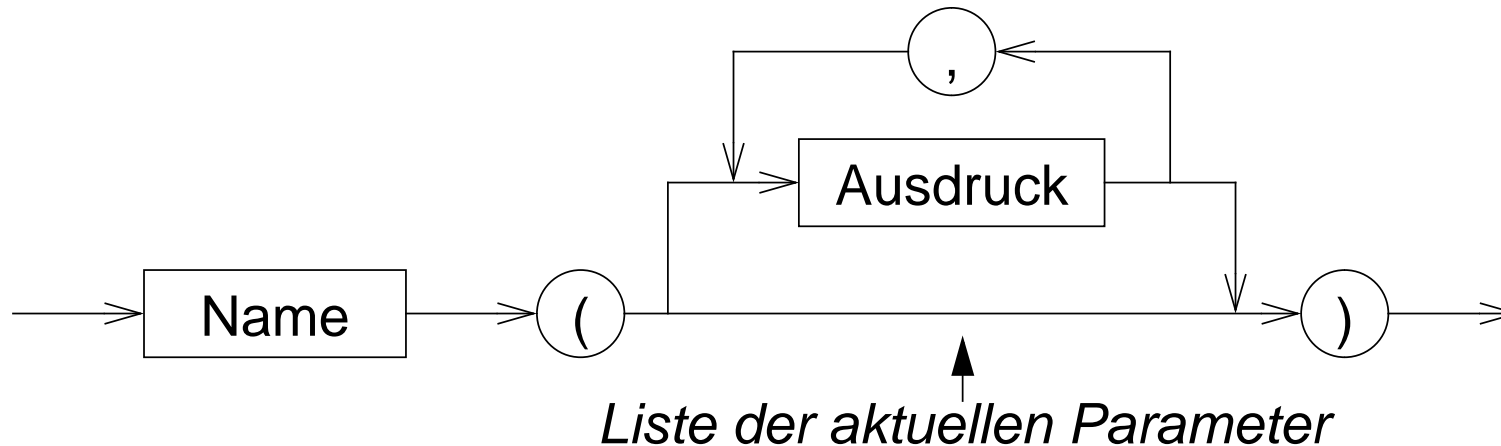
```
int main()
{
    double wert;

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

- beliebige Verwendung von `sinus` in Ausdrücken:

```
y = exp(tau*t) * sinus(f*t);
```

## 5.3 Funktionsaufruf



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird  
↳ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt



## 5.4 Regeln

- Funktionen werden global definiert
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
- rekursive Funktionsaufrufe sind zulässig
  - ↳ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

```
int fakultaet(int n)
{
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```

## 5.4 Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
  - = Rückgabebetyp und Parametertypen müssen bekannt sein
  - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
  
- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
  - Funktionswert vom Typ **int**
  - 1. Parameter vom Typ **int**
  - ↳ **schlechter Programmierstil → fehleranfällig**
  - ↳ **ab C99 nicht mehr zulässig**

## 5.5 Funktionsdeklaration

- soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden (Prototyp)

- ◆ Syntax:

```
Typ Name ( Liste formaler Parameter );
```

- ▶ Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!

- ◆ Beispiel:

```
double sinus(double);
```

## 5.6 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);
}
```

## 5.7 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
  - call by value (wird in C verwendet)
  - call by reference (wird in C **nicht** verwendet)
  
- call-by-value: 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

# C-Präprozessor

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Präprozessor bearbeitet
- Anweisungen an den Präprozessor werden durch ein #-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Präprozessoranweisungen ist unabhängig vom Rest der Sprache
- Präprozessoranweisungen werden nicht durch ; abgeschlossen!
- wichtigste Funktionen:
  - #define**                    Definition von Makros
  - #include**                    Einfügen von anderen Dateien

## 6.1 Makrodefinitionen

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die `#define`-Anweisung definiert
- Syntax:

```
#define Makroname Ersatztext
```

- eine Makrodefinition bewirkt, dass der Präprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt
- Beispiel:

```
#define EOF -1
```

## 6.2 Einfügen von Dateien

- `#include` fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein

- Syntax:

```
#include <Dateiname >  
oder  
#include "Dateiname "
```

- mit `#include` werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden, einkopiert
  - Deklaration von Funktionen, Strukturen, externen Variablen
  - Definition von Makros
- wird **Dateiname** durch `< >` geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch `" "` geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)



# Programmstruktur & Module

## 7.1 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

## 7.2 Top-down Entwurf

### ■ Zentrale Fragestellung

◆ was ist zu tun?

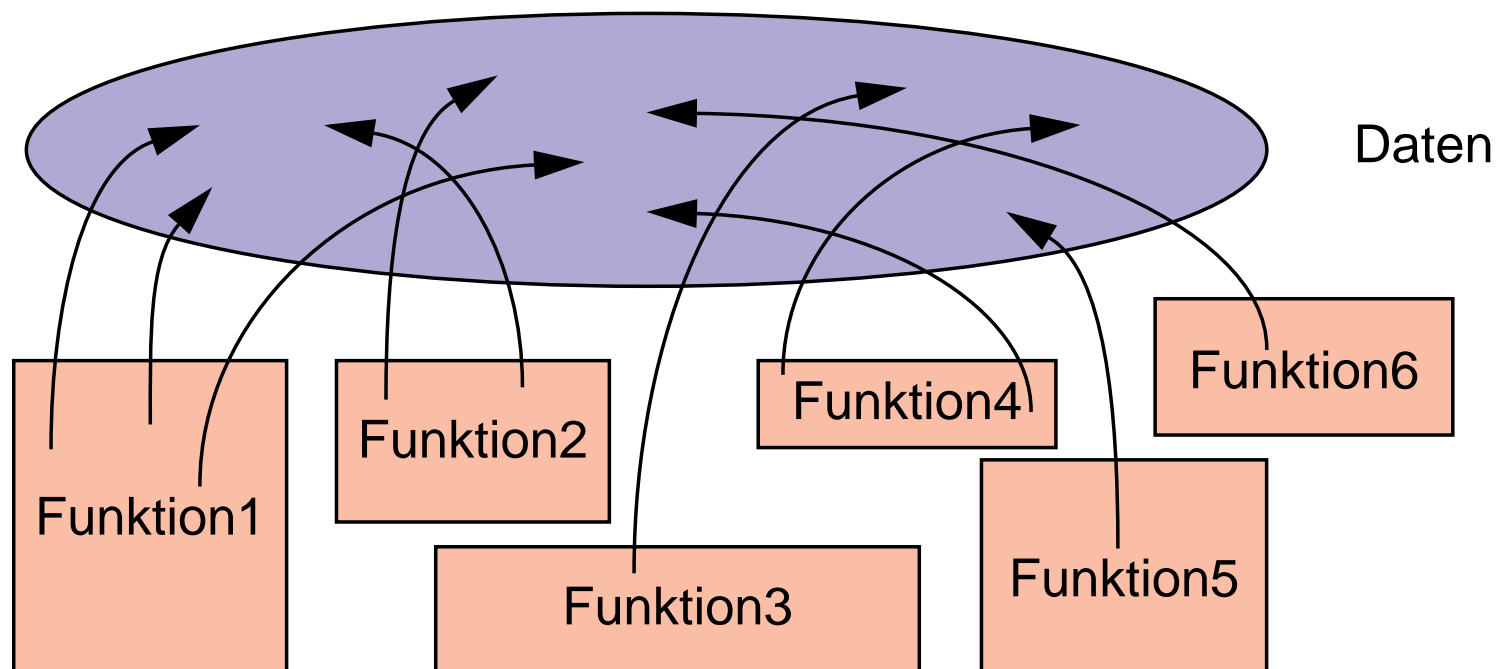
◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?

➤ Beispiel:

- Rechnung für Kunden ausgeben
  - Rechnungspositionen zusammenstellen
    - Lieferungspositionen einlesen
    - Preis für Produkt ermitteln
    - Mehrwertsteuer ermitteln
  - Rechnungspositionen addieren
  - Positionen formatiert ausdrucken

## 7.2 Top-down Entwurf (2)

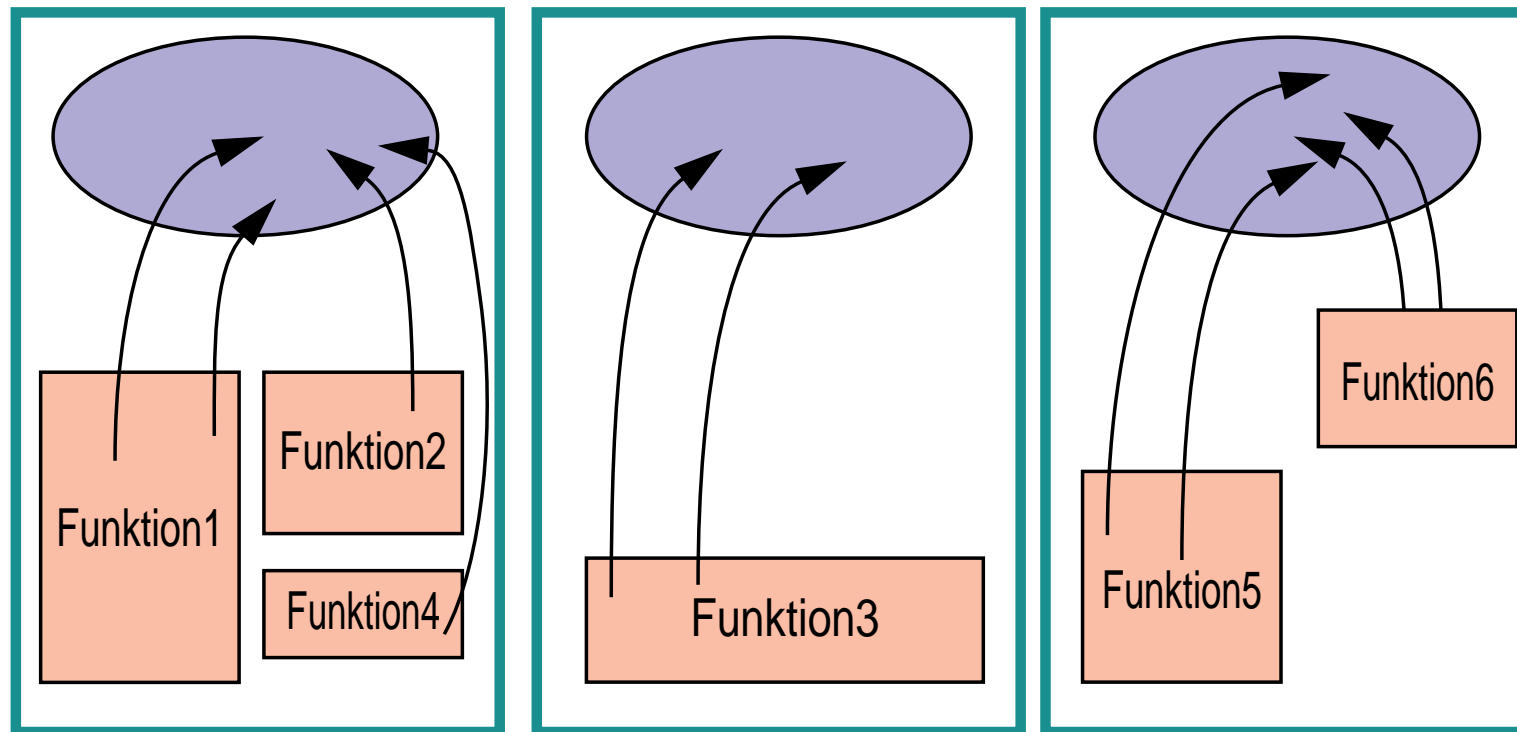
- Problem:  
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:  
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



## 7.2 Top-down Entwurf (3) — Modul-Bildung

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

➔ **Modul**



## 7.3 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 prog.c           (erzeugt Datei prog.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 prog.o f1.o f2.o f3.o f4.c f5.c
```

## 7.3 Module in C (2)

- !!! **.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**

## 7.4 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 Variablen in umgebenden Blöcken

## 7.5 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;
```



## 7.5 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**

## 7.5 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

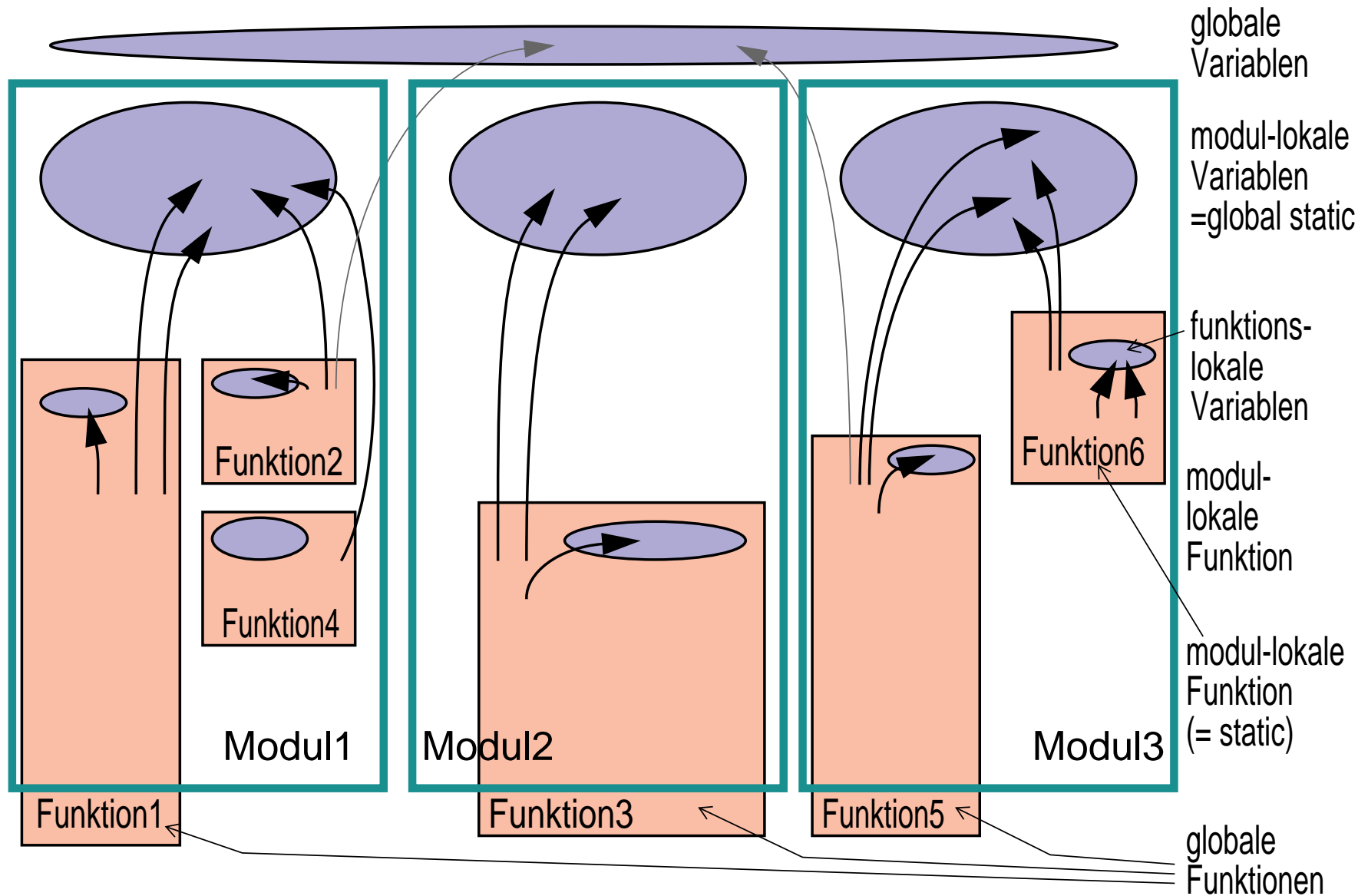
## 7.6 Einschränkung der Gültigkeit auf ein Modul

- 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
  - Beispiel: `static int a;`
  - ↳ `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! - dort jeweils *kursiv* geschrieben)

## 7.7 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

# 7.8 Gültigkeitsbereiche — Übersicht



## 7.9 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

## 7.9 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)**

## 7.9 Lebensdauer von Variablen (3)

### **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