

6 Formatierte Ausgabe (2)

- die Zeichenkette **format** ist aufgebaut aus:
 - ↳ **einfachem Ausgabetext**, der unverändert ausgegeben wird
 - ↳ **Formatelementen**, die Position und Konvertierung der zugeordneten **Werte** beschreiben

Beispiele für Formatelemente:

Zeichenkette: `%[-][min][.max]s`
 Zeichen: `%[+][-][n]c`
 Ganze Zahl: `%[+][-][n][1]d`
 Gleitkommazahl: `%[+][-][n][.n]f`

[] bedeutet optional

Beispiel:

```
printf("a = %d, b = %d, a+b = %d", a, b, a+b);
```

7 C-Präprozessor — Kurzüberblick

- 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

8 C-Präprozessor — 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
```

9 C-Präprozessor — Einfügen von Dateien

- **#include** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein
- Syntax:

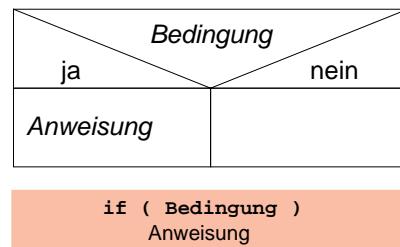

```
#include < Dateiname >
oder
#include "Dateiname "
```
- mit **#include** werden Header-Dateien mit Daten, die für mehrere Quelldateien benötigt werden einkopiert
 - Deklaration von Funktionen, Strukturen, externen Variablen
 - Definition von Makros
- wird **Dateiname** durch **< >** geklammert, wird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

D.7 Kontrollstrukturen

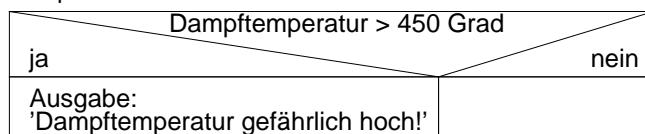
Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

- Bedingte Anweisung
 - ◆ einfache Verzweigung
 - ◆ mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
 - ◆ abweisende Schleife
 - ◆ nicht abweisende Schleife
 - ◆ Laufanweisung
 - ◆ Schleifensteuerung

1 Bedingte Anweisung

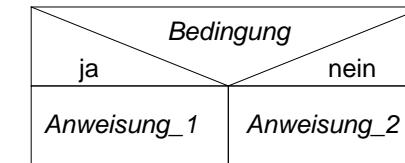


- Beispiel:



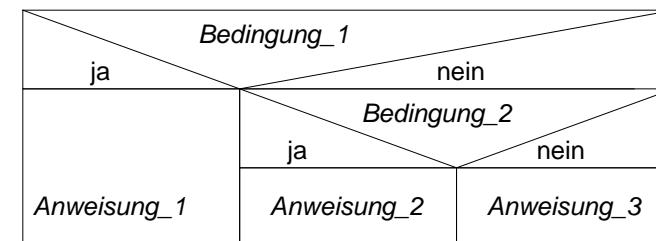
```
if ( temp >= 450.0 )  
    printf("Dampftemperatur gefährlich hoch!\n");
```

1 Bedingte Anweisung einfache Verzweigung



```
if ( Bedingung )  
    Anweisung_1  
else  
    Anweisung_2
```

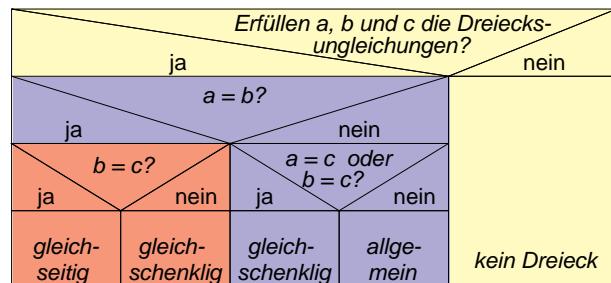
1 Bedingte Anweisung mehrfache Verzweigung



```
if ( Bedingung )  
    Anweisung_1  
else if ( Bedingung_2 )  
    Anweisung_2  
else  
    Anweisung_3
```

1 Bedingte Anweisung mehrfache Verzweigung (2)

- Beispiel: Eigenschaften von Dreiecken — Struktogramm



D.49

2 Fallunterscheidung

- Mehrfachverzweigung = Kaskade von if-Anweisungen
- verschiedene Fälle in Abhängigkeit von einem ganzzahligen Ausdruck



```

switch ( Ausdruck ) {
    case Wert_1:
        Anweisung_1
        break;
    case Wert_2:
        Anweisung_2
        break;
    ...
    case Wert_n:
        Anweisung_n
        break;
    default:
        Anweisung_x
}
  
```

D.51

1 Bedingte Anweisung mehrfache Verzweigung (3)

- Beispiel: Eigenschaften von Dreiecken — Programm

```

printf("Die Seitenlängen %f, %f und %f bilden ", a, b, c);

if ( a < b+c && b < a+c && c < a+b )
    if ( a == b )
        if ( b == c )
            printf("ein gleichseitiges");
        else
            printf("ein gleichschenkliges");
    else
        if ( a==c || b == c )
            printf("ein gleichschenkliges");
        else
            printf("ein allgemeines");
    else
        printf("kein");
    printf(" Dreieck");
  
```

D.50

2 Fallunterscheidung — Beispiel

```

#include <stdio.h>

int main()
{
    int zeichen;
    int i;
    int ziffern, leer, sonstige;
    ziffern = leer = sonstige = 0;

    while ((zeichen = getchar()) != EOF)
        switch (zeichen) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':           ziffern++;
            case ' ':
            case '\n':
            case '\t':           leer++;
            default:             sonstige++;
        }

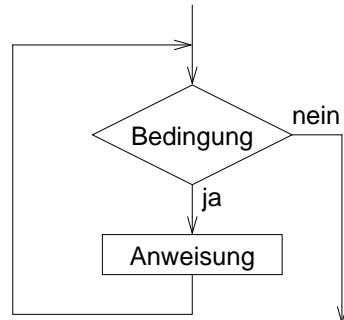
    printf("Zahl der Ziffern = %d\n", ziffern);
    printf("Zahl der Leerzeichen = %d\n", leer);
    printf("Zahl sonstiger Zeichen = %d\n", sonstige);
}
  
```

D.52

3 Schleifen

- Wiederholte Ausführung von Anweisungen in Abhängigkeit von dem Ergebnis eines Ausdrucks

4 abweisende Schleife

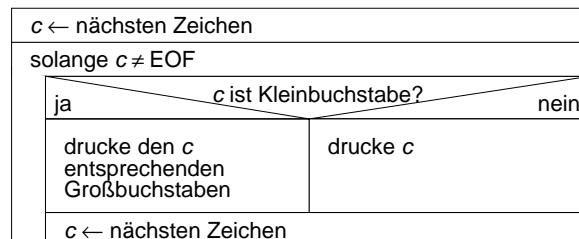


D.53

D.7 Kontrollstrukturen

4 abweisende Schleife (2)

- Beispiel: Umwandlung von Klein- in Großbuchstaben



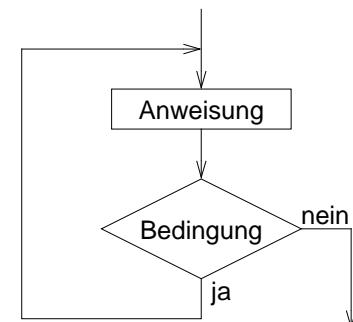
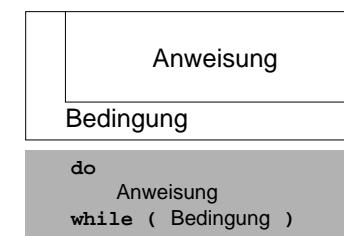
```
int c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
```

► abgekürzte Schreibweise

```
int c;
while ( (c = getchar()) != EOF )
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
```

D.54

5 nicht-abweisende Schleife



D.55

SPiC

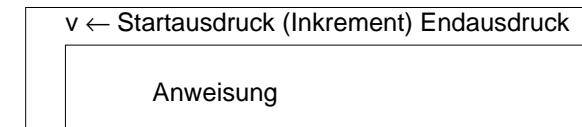
GdI2 - Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2008

D-CEinführung.fm 2008-04-17 10.16

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

6 Laufanweisung



```
for ( v = Startausdruck; v <= Endausdruck; v += Inkrement )
    Anweisung
```

allgemein:

```
for ( Ausdruck_1; Ausdruck_2; Ausdruck_3 )
    Anweisung
```

```
Ausdruck_1;
while (Ausdruck_2) {
    Anweisung
    Ausdruck_3;
}
```

SPiC

GdI2 - Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2008

D-CEinführung.fm 2008-04-17 10.16

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

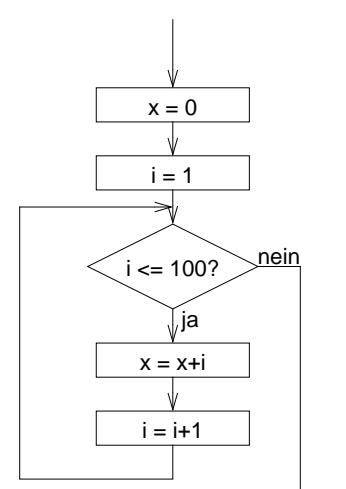
D.56

6 Laufanweisung (2)

- Beispiel: Berechne $x = \sum_{i=1}^{100} i$

```
x ← 0
i ← 1(1)100
    x ← x + i
```

```
x = 0;
for ( i=1; i<=100; i++)
    x += i;
```



D.57

7 Schleifensteuerung

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

```
char 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

D.8 Funktionen

1 Überblick

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

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

1 überblick (2)

- ▶ Funktionen dienen der Abstraktion
- Name und Parameter abstrahieren
 - ▶ vom tatsächlichen Programmstück
 - ▶ von der Darstellung und Verwendung von Daten
- Verwendung
 - ◆ mehrmals benötigte Programmstücke können durch Angabe des Funktionsnamens aufgerufen werden
 - ◆ Schrittweise Abstraktion (**Top-Down- und Bottom-Up-Entwurf**)

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

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);
}
```

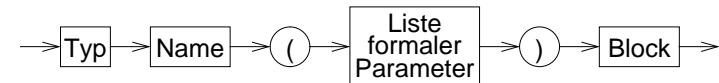
- beliebige Verwendung von `sinus` in Ausdrücken:

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

```
int main()
{
    double wert;

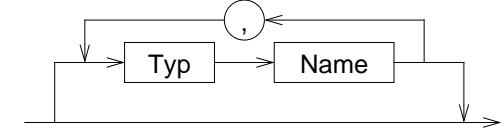
    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
    return(0);
}
```

3 Funktionsdefinition (2)



■ Liste formaler Parameter

- ◆ **Typ:** beliebiger Typ
- ◆ **Name:** beliebiger Bezeichner
- ◆ die formalen Parameter stehen innerhalb der Funktion für die Werte, die beim Aufruf an die Funktion übergeben wurden (= **aktuelle Parameter**)
- ◆ die formalen Parameter verhalten sich wie Variablen, die im **Funktionsrumpf** definiert sind und mit den aktuellen Parametern vorbelegt werden



3 Funktionsdefinition

- Schnittstelle (Typ, Name, Parameter) und die Implementierung

- ◆ Beispiel:

```
int addition ( int a, int b ) {
    int ergebnis;
    ergebnis = a + b;
    return ergebnis;
}
```



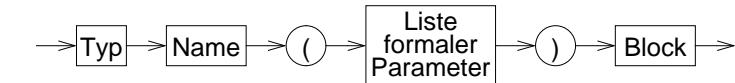
■ Typ

- ◆ Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
- ◆ beliebiger Typ
- ◆ **void** = kein Rückgabewert

■ Name

- ◆ beliebiger Bezeichner, kein Schlüsselwort

3 Funktionsdefinition (3)



■ Block

- ◆ beliebiger Block
- ◆ zusätzliche Anweisung

return (Ausdruck);

oder

return;

bei **void**-Funktionen

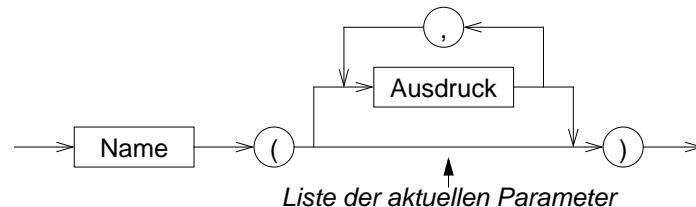
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muss mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

4 Funktionsaufruf

- Aufruf einer Funktion aus dem Ablauf einer anderen Funktion

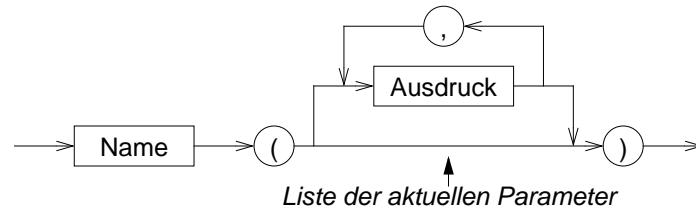
◆ Beispiel:

```
int main ( ) {
    int summe;
    summe = addition(3,4);
    ...
}
```



- Jeder Funktionsaufruf ist ein Ausdruck
- **void**-Funktionen können keine Teilausdrücke sein
 - ◆ wie Prozeduren in anderen Sprachen (z. B. Pascal)

4 Funktionsaufruf (2)



- 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 Beispiel

```
float power (float b, int e)
{
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    return(prod);
}
```

```
float x, y;
y = power(2+x,4)+3;
```

```
float x, y, power;
{
    float b = 2+x;
    int e = 4;
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    power = prod;
}
y=power+3;
```

≡

6 Regeln

- Funktionen werden global definiert
 - keine lokalen Funktionen/Prozeduren wie z. B. in Pascal
- **main()** ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
 - Ergebnis vom Typ **int** - wird an die Shell zurückgeliefert (in Kommandoprozeduren z. B. abfragbar)
- rekursive Funktionsaufrufe sind zulässig
 - eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

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

6 Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabetyp und Parametertypen müssen dem Compiler 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**

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);
}
```

6 Regeln (2)

- Funktionsdeklaration
 - ◆ soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden
- ◆ Syntax:

Typ Name (Liste formaler Parameter);

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

- ◆ Beispiel:

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

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

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

D.9 Programmstruktur & Module

1 Softwaredesign

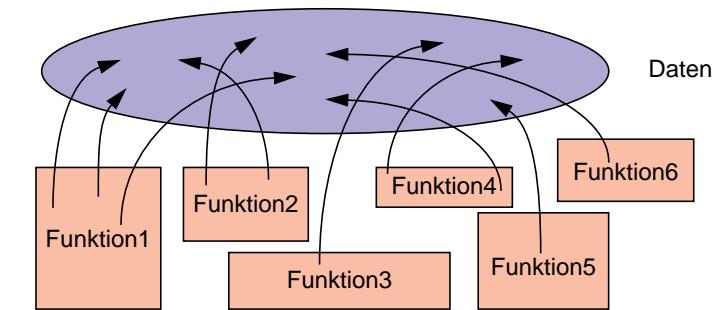
- Grundsätzliche Überlegungen über die Struktur eines Programms vor Beginn der Programmierung
- Verschiedene Design-Methoden
 - ◆ Top-down Entwurf / Prozedurale Programmierung
 - traditionelle Methode
 - bis Mitte der 80er Jahre fast ausschließlich verwendet
 - an Programmiersprachen wie Fortran, Cobol,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

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

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



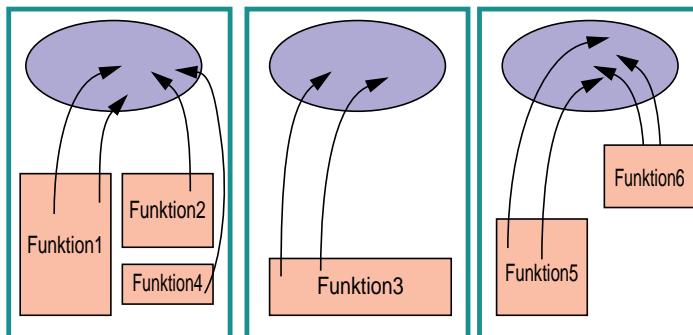
2 Top-down Entwurf (3)

Modul-Bildung

■ Lösung:

Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

► **Modul**



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
```

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

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

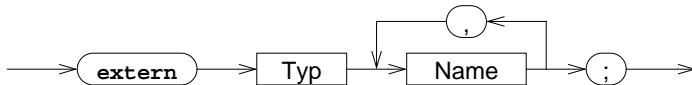
■ Überdeckung bei Namensgleichheit

- 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

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;
```

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!!!**

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

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

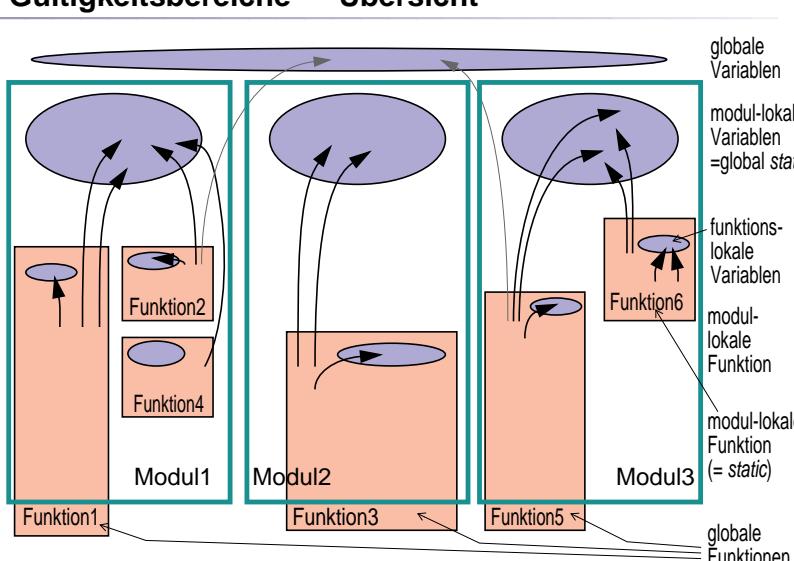
➤ sie werden dadurch nicht Bestandteil der Modulschnittstelle
(= des "Vertrags" mit den Modulbenutzern)

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

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

8 Gültigkeitsbereiche — Übersicht



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

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)

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

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));
}
```

10 Getrennte Übersetzung von Programmteilen — Beispiel

Hauptprogramm (Datei fplot.c)

```
#include "trig.h"
#define INTERVAL 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+=INTERVAL)
            printf("%lf %lf\n", i, tan(i));
        break;;
    ...
    }
}
```

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);
}
```

