# Einführung in die Programmierung

## mehr Ein- und Ausgabe, Programmfluss, Funktionen

Arvid Terzibaschian

# Klausurtermine

- Abstimmung beendet:
  - Einführung in die Programmierung
    - Freitag, 28.2.2014, wahrscheinlich 11:00
  - Basiskonzepte des Rechnerbetriebs
    - Freitag, 21.2.2014, wahrscheinlich 11:00
  - Kollisionen ? Bitte per E-Mail melden
- auch auf den Webseiten verfügbar
  - exakte Termine könnten sich wegen Raumplanung noch ändern

# mehr Ein- und Ausgabe

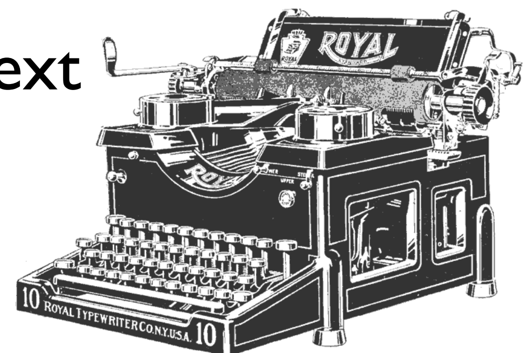
printf, scanf

# printf

- printf ermöglicht formatierte Ausgabe von Variablen und Text

```
int x = 17;
printf("Wert von x: %d\n",x);
```

- Platzhalter auch Ausgabe*format*
  - Ganzahlen: Octal, Hexadezimal, Dezimal, führende Nullen, …
  - Fließkomma: Festkommastellen, Exponentialschreibweise, kompakte Ausgabe, …
- Steuerzeichen formatieren den Fließtext
  - Ausrichtung,
  - Farbe (systemabhängig).



# Erweiterte Formatplatzhalter für printf

- Hexadezimal-Zahl (Basis 16): %x (%lx)

```
printf("3405691582 als Hexadezimalzahl ist: %x", 3405691582 );
```

- Oktal-Zahl (Basis 8): %o (%lo)

```
printf("501 als Oktalzahl ist: %o", 501);
```

- „kompakte" Fließkommazahl: %g (%lg)

```
printf("Pi kompakt ist: %lg", 3.141592653);
```

- Fließkommazahl in Exponentialschreibweise: %e (%le)

```
printf("Pi ist: %le", 3.141592653);
```

# Steuerzeichen – Escape Sequences

- Steuerzeichen
  - durch „escape sequences“ \[…] in Zeichenketten eingebaut
  - \n – newline
  - \r – carriage return
  - \\ – backslash
  - \t – tabstop
  - \ddd – oktaler Wert aus ASCII-Tabelle
  - \xhh – hexadezimaler Wert aus ASCII-Tabelle
  - \“ – Anführungszeichen
- Beispiel:

```
printf("Neue Zeile:\nTabstopp:\tund ein ASCII-Zeichen in HEX \x28"
         "und ein ASCII-Zeichen in Oktal \051\n");
        // beachte: Textkonstante geht über mehrere Zeilen!
```

# Erweiterte Format-Platzhalter für printf

- Ausgabe einer Fließkommazahl mit fester Anzahl Nachkommastellen [Prec]

```
printf(“Pi ist: %.2f“, 3.14159);        //2 Stellen nach 0
```

- Ausgabe einer Ganzzahl mit führenden Nullen [Prec]

```
printf(“%.3d“, 19); // 3 führende Nullen
```

- Ausgabe einer Zahl mit führenden Leerzeichen [Width]
  - für Fließkomma und Ganzzahl

```
printf(“%10d\n%10d“, 19,200031);
// Ausgabe wird mit Leerzeichen „aufgefüllt“
```

- Allgemeine Syntax: %[Width][.Prec]Typ

# Benutzereingaben: scanf

- scanf funktioniert als „Gegenstück“ zu printf:
  - formatiertes Einlesen von Benutzereingaben in Variablen
- Beispiel:
  - Einlesen einer Ganzzahl:

```
int n;
printf(“Bitte geben Sie eine ganze Zahl ein:“);
scanf(“%d“,&n);
```

  - Einlesen einer Fließkommazahl:

```
float x;
printf(“Bitte geben Sie eine Fliesskommazahl ein:“);
scanf(“%f“,&x);
```

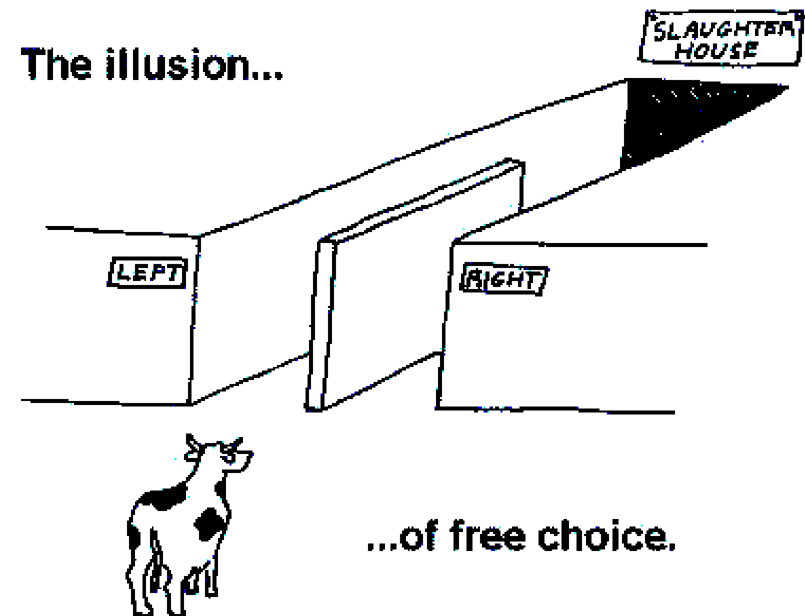
- `&x` bedeutet „Speicheradresse der Variablen x“
  - scanf benötigt **Adresse** von x (und nicht den Wert) um Eingabe dort abzulegen

# Programmfluss

Kontrollstrukturen und Schleifen

# Entscheidungen zur Programmlaufzeit

- Problem:
  - Sie möchten ein Programm schreiben, das sich z.B. abhängig von der Uhrzeit oder einer Nutzereingabe anders verhält!
- Lösung: Kontrollstrukturen
  - Kontrollstrukturen führen bestimmte Anweisungsblöcke nur unter bestimmten Bedingungen aus.
- Kontrollstrukturen elementar für alle imperativen Programmiersprachen
  - Syntax und Semantik in den meisten Sprachen sehr ähnlich

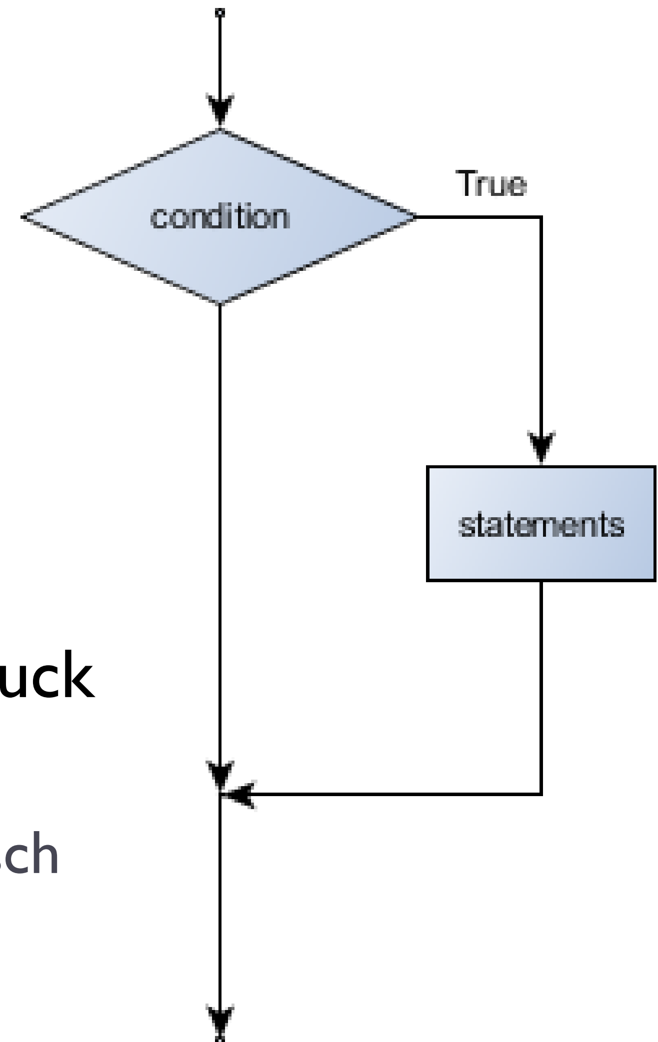


# Kontrollstrukturen: If-Anweisung

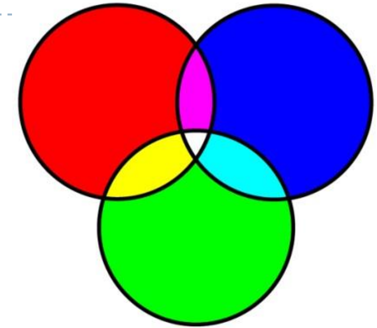
- einfachste Kontrollstruktur

```
if(condition) {
 // Anweisungsblock
}
```

- Semantik:
  - Wenn „condition“ *wahr,* dann führe Anweisungsblock aus
- „condition“ ist ein boolescher Ausdruck
  - kann nur „wahr“ oder „falsch‘ sein
  - keine Schlüsselworter für wahr und falsch
    - 0 als „falsch“ (false) definiert
    - **alles** andere als „wahr“ (true) definiert



# Einschub: boolesche Ausdrücke in C



- boolesche Ausdrücke in C kompatibel mit arithmetischen Ausdrücken
  - sie werden intern als Ganzzahlen dargestellt
- semantisch aber von Arithmetik verschieden: Boolesche Logik:

| Vergleichsoperatoren | Bedeutung |
|---|---|
| A > B, B < A | A größer als B, B kleiner als A |
| A >= B, B <= A | A größer gleich B , B kleiner gleich A |
| A == B, A != B | A gleich B, A ungleich B |
| **Boolesche Operatoren** | **Bedeutung** |
| A && B | wahr, wenn A und B wahr |
| A \|\| B | wahr, wenn A oder B, oder A und B wahr |
| !A | wahr, wenn A falsch |

# Beispiele für If-Anweisung mit booleschen Ausdrücken

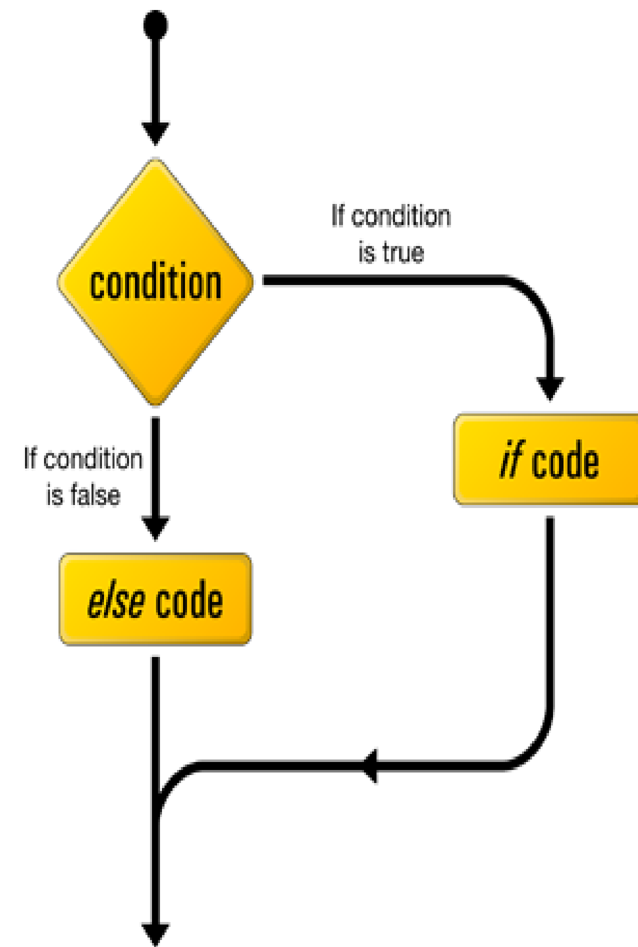
int n = 40;    int k = 20;    float pi = 3.1416;

- `if(n > 40) { … }                // wahr?`
- `if(k < n && pi > 3) { … }       // wahr?`
- `if(pi*pi > 10) { … }            // wahr?`
- `if(1 && 0) { … }                // wahr?`
- `if(!1 && 0 != 0 + 1 > 1 || 1)// wahr?`
  - `Im Zweifel immer Klammern setzen!`

# Kontrollstrukturen: If-Else-Anweisung

- If-Anweisung mit alternativem Anweisungsblock (Else), falls Bedingung nicht erfüllt ist

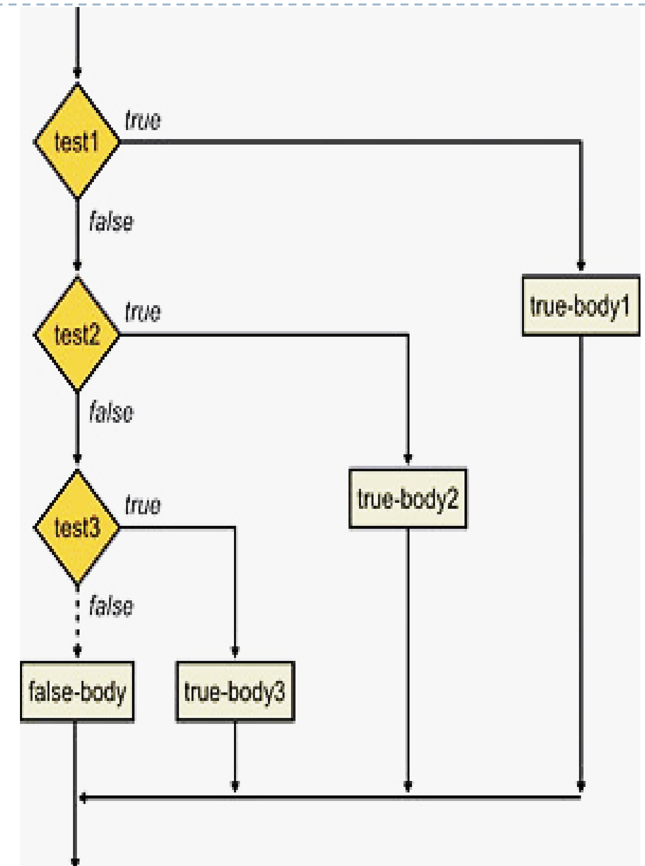
```
if(condition) {
  // Anweisungsblock
  printf(„condition is true“);
}
else {
  // alternativer Anw.-block
  printf(„condition is false“);
}
```



# Kontrollstrukturen: If-Else-If-Kette

- Verkettung von If-Else-Anweisungen

```
if(condition1) {
        // Anweisung(en)
}
else if(condition2){
        // Anweisung(en)
}
…
else if(conditionN) {
         // Anweisung(en)
}
…
else {
         // Anweisung(en)
}
```




- führt ersten Anweisungsblock mit erfüllter Bedingung aus
  - **maximal 1 Anweisungsblock wird ausgeführt!**
- letzter Else-Block ist optional

# Kontrollstrukturen: Switch-Case

- direkte Verzweigung, abhängig von einem **Wert**
  - nur für Variablen mit ganzzahligem Datentyp
  - Ersatz von If-Else-Kette für Fallunterscheidungen

```
int a = …;
switch(a) {
   case 10:
      printf("a ist 10");
      break;
   case 100:
      printf("a ist 100");
      break;
   default:
         printf("a ist ???");
      break;
}
```

# Schleifen

- Problem: Berechne Summe der Zahlen von 1 bis n
- mit Kontrollstrukturen nur bedingt möglich:

```
int n = 10; int s = 0;
if(n >= 1)
      s += 1;
if(n >= 2)
      s += 2;

...
if(n >= 10000)
      s += 10000;
```

- Lösung: Schleifen!

# Schleifen: While-Schleife

- einfachste Schleife:

```
while(condition) {
  // Anweisung(en) [Schleifenrumpf]
}
```

- führt Schleifenrumpf wiederholt aus, solange "condition" wahr ist
  - Schleifenrumpf *kann* condition beeinflussen
- Endlosschleife: while(1) {};
- Summe der Zahlen 1 bis N

```
int i = 0, n = 25, sum = 0;
while(i < n) {
      sum += i;
      i += 1;
}
```

# Laufvariablen und Inkrementierung

- Summe der Zahlen 1 bis N

```
int i = 0, n = 25, sum = 0;
while(i < n) {
        sum += i;
        i += 1;
}
```

- Laufvariable/Schleifenzähler oft i genannt (Iteration)
- Für i += 1 gibt es extra Operator
  - Inkrementoperator ++
  - Dekrementoperator --
- Inkrementoperator verantwortlich für Bezeichnung von C++

# Laufvariablen und Inkrementierung

- Zwei Arten von Inkrementoperator
  - Post-Inkrement: i++
    - i wird **nach Berechnung** der *arithmetischen* Anweisung erhöht

  - Pre-Inkrement: ++i
    - i wird **vor Berechnung** der *arithmetischen* Anweisung erhöht

  - maximal ein ++/--Operator pro Variable pro arithmetischem Ausdruck möglich, sonst ist Verhalten undefiniert!

# Beispiel: Summe der Zahlen 1 bis N

- kurz:

```
int n = 25, s = 0;
while(n--) {
    s += n+1;
}
```

- kürzer
  - Wie funktioniert das?

```
int n = 25, s = 0;
while((s+=n) && --n) {}
```

- … noch kürzer?

```
int n = 25;
int s = n*(n+1)/2;
```

  - Mathematik oft kürzeste und effektivste Lösung

# Schleife: do-while

- Überprüfung der Bedingung **nach** dem Schleifendurchlauf
  - Gegenstück zur While-Schleife

```
do {
        // Anweisung(en)
} while(condition);
```

  - Summe der Zahlen von 1 bis N:

```
int n = 25, s = 0;
do {
 s += n--;
} while(n > 0);
```

# Schleife: typische Eigenschaften

```
int n = 25, s = 0;
while(n > 0) {
    s += n;
    ++n;
}
```

- Initialisierung
- Bedingung
- Rumpf
- Aktualisieren der Laufvariablen

Gibt es ein Sprachkonstrukt, welches generische Schleife direkt abbildet?

# Schleife: for-schleife

- Die For-Schleife besteht aus
  - Initialisierung,
  - Bedingung,
  - Aktualisierung der Laufvariable und
  - Anweisungsrumpf.

```
for(init;condition;update) {
        // Anweisung(en) [Rumpf]
}
```

- „Verwaltungsoverhead" im Schleifenkopf definiert
  - nur „relevanter Code" im Rumpf
- Beispiel:

```
int i,s;
for(i = 0, s = 0;i<25;++i) {
    s += i;
};
```

# Programmfluss in Schleifen: break

- vorzeitiges Verlassen der Schleife
- hilfreich bei komplexen Schleifenbedingungen, die z.B. zwischendurch im Schleifenrumpf überprüft werden

- Beispiel:

```
while(1) {
        // berechne Spiellogik
        if(gameOver)
                break;
        // zeichne Grafik
        if(grafikkarteZuHeiss)
                break;
        // verarbeite Nutzereingaben
        if(pressedESC)
                break;
        …
}
```

# Programmfluss in Schleifen: continue

- continue setzt das Programm fort mit nächstem Schleifen*durchlauf* wie folgt:
  - bei For-Schleifen:
    - Zuerst Aktualisierungsanweisung ausgeführt,
    - dann condition geprüft,
    - dann Fortsetzung bei erster Anweisung im Rumpf.
  - bei While-Schleifen:
    - Zuerst condition geprüft,
    - dann Fortsetzung bei erster Anweisung im Rumpf.
  - bei Do-While-Schleifen
    - Fortsetzung bei erster Anweisung im Rumpf.

# Funktionen und Parameter

# Funktionen

- Funktionen sind elementarer Bestandteil von Hochsprachen
  - Funktionen stellen verfügbare Lösungen für Teilprobleme bereit
    - printf, scanf, sin, exp, sqrt …
  - Funktionen ermöglichen Abstraktion
    - Divide-And-Conquer-Prinzip:
      komplexe Probleme in einfache zerlegen
  - Funktionen erhöhen Wiederverwendbarkeit von Code
    - Zerlegung in unabhängige Module
  - Funktionen ermöglichen einfachere Wartbarkeit
    - können (im Idealfall) unabhängig voneinander verändert werden
    - verschiedene Teams können einfacher an verschiedenen Teilen arbeiten
  - Funktionen ermöglichen Rekursion
  - …

# Funktionen in C

- Funktionsdefinition:

*Rückgabetyp* **Name** Parameterliste

```
double sum(double x,double y) {
        int z = x+y;
        return z;
}
```

Funktionsrumpf

- Funktionsname
  - eindeutig
  - Buchstaben, Ziffern,_
- Parameterliste
  - Datentyp Name[, Datentyp Name[, …]]
  - Parameter sind nur innerhalb der Funktion „sichtbar“
- Rückgabewert
  - hat bestimmten Datentyp
  - „return [Wert];“ setzt Rückgabewert und verlässt Funktion
  - Funktionen ohne Rückgabewert haben Datentyp „void“
    - werden auch Prozeduren genannt

# Funktionen in C: Definition vs. Deklaration

- Funktionsdefinitionen
  - beschreiben Schnittstelle UND **Verhalten**
  - mit Quellcode des Funktionsrumpfes
    - Implementierung bekannt

```
int sqr(int x) {
        return x*x;
}
```

- Funktionsdeklarationen
  - beschreiben nur **Schnittstelle (Interface)**
  - ohne Quellcode des Funktionsrumpfes
    - Implementierung irrelevant
    - Black-Box-Prinzip

```
int sqr(int x);
```

**In C üblich:**

Definitionen (Implementierungen) in .c – Dateien
Deklarationen (Schnittstellen) in .h - Dateien

# Funktionen in C: Definition vs. Deklaration

**Definition (Implementierung): box.c**

```
int box_volume(int a,int b,int c) {
    return a*b*c;
};

int box_area(int a,int b,int c) {
    return 2*(a*b + a*c + b*c);
}
```

**Deklaration (Interface): box.h**

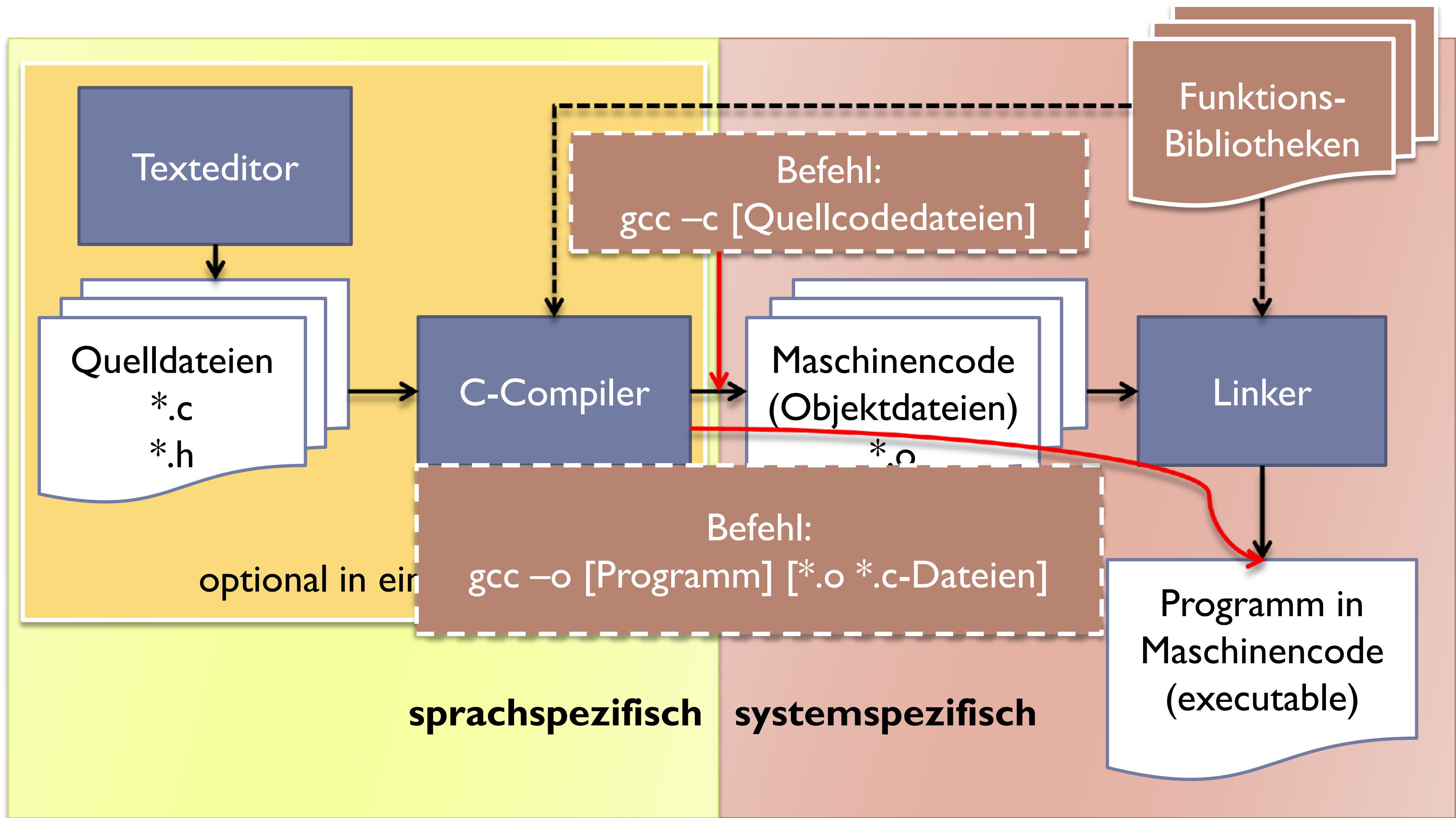
```
/* calculate volume of a box with
side-lengths a,b,c */
int box_volume(int a,int b,int c);

/* calculate surface area of a box
with side-lengths a,b,c */
int box_area(int a,int b,int c);
```

- zum *Aufrufen* der Funktionen ist nur box.h nötig (Interface!)
- zum *Ausführen* benötigt man eine Implementierung der Funktionen in Maschinencode
  - Linker verbindet Maschinencodeteile
  - Maschinencode oft in Bibliotheken abgelegt
    - auch von anderen Programmiersprachen

**anderes Programm: call_box.c:**

```
#include "box.h"
int main() {
    int a = 2;b = 3;c = 4;
    int v = box_volume(a,b,c);
    printf("Vol: %d\n",v);
}
```

# Erinnerung: Vom Quellcode zum Programm



# Funktionen: Rekursion

- Fakultät berechnen 1)

```
long fac(long n) {
    long r;
    for(r = 1;n>1;--n) {
        r *= n;
    }
    return r;
}
```

- Schleife zum Berechnen
- #Funktionsaufrufe = 1
  - kaum „Overhead“
- für einige Probleme schwer zu implementieren
  - viele Zwischenergebnisse
  - Trial-and-Error-Algorithmen

- Fakultät berechnen 2)

```
long fac(long n) {
    if(n > 1)
        return fac(n-1)*n
    else
        return 1;
}
```

- Rekursion zum Berechnen
- #Funktionsaufrufe = n
  - Overhead durch Funktionsaufrufe
- für viele Probleme gute Modellierung
  - Suchprobleme, Sortierung

# Fakultät rekursiv

- fac(4) = 24

```
long fac(long n) {
    if(n > 1)
        return fac(n-1)*n
    else
        return 1;
}
```

```
long fac(4) {
    if(4 > 1)
        return fac(4-1)*4
    else
        r
}
```

24

```
long fac(
    if(3
        return fac(3-1)*3
    else
}
```

6

```
long fac(2)
    if(2 >
        return fac(2-1)*2
    else
        r
}
```

2

```
long fac(
    if(1
        r    urn fac(1-1)*1
    else
        return 1;
}
```

1

# Funktionen: Parameterübergabe

| Art der Parameter-übergabe | Beschreibung | Sprachen |
|---|---|---|
| Call-By-Value | Funktionen erhalten nur Werte<br>• es können keine Seiteneffekte auftreten<br>• große Werte müssen bei jedem Aufruf kopiert werden<br>• keine Rückgabe über diese Parameter an aufrufende Funktion | C, Java, C++, PHP, JavaScript, C#, … fast alle imperativen Programmiersprachen! |
| Call-By-Reference | Funktionen erhalten einen Verweis auf Variablen<br>• sie können u.U. Werte ändern und an Aufrufer zurückgeben (Seiteneffekte!)<br>• große Werte müssen nicht kopiert werden<br>• Aufruf arithmetischen Ausdrücken nicht mgl. | C++, Java (C nur bedingt!), …<br>fast alle Programmiersprachen! |
| Call-By-Name | • symbolische Ersetzung im Rumpf der Funktionen | C++, .NET, deklarative & logische Sprachen (LISP, Mathematik) |

# Vielen Dank!

- Bei Fragen einfach eine Mail an:
  - arvid@cs.uni-potsdam.de