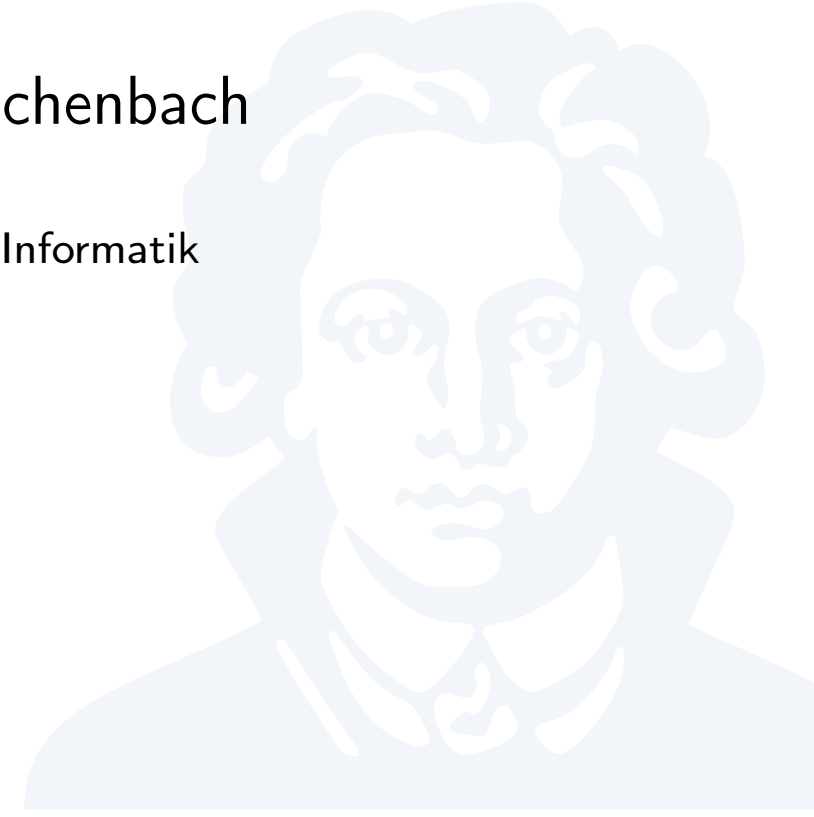
Einführung in die Systemprogrammierung
04

Prof. Dr. Christoph Reichenbach

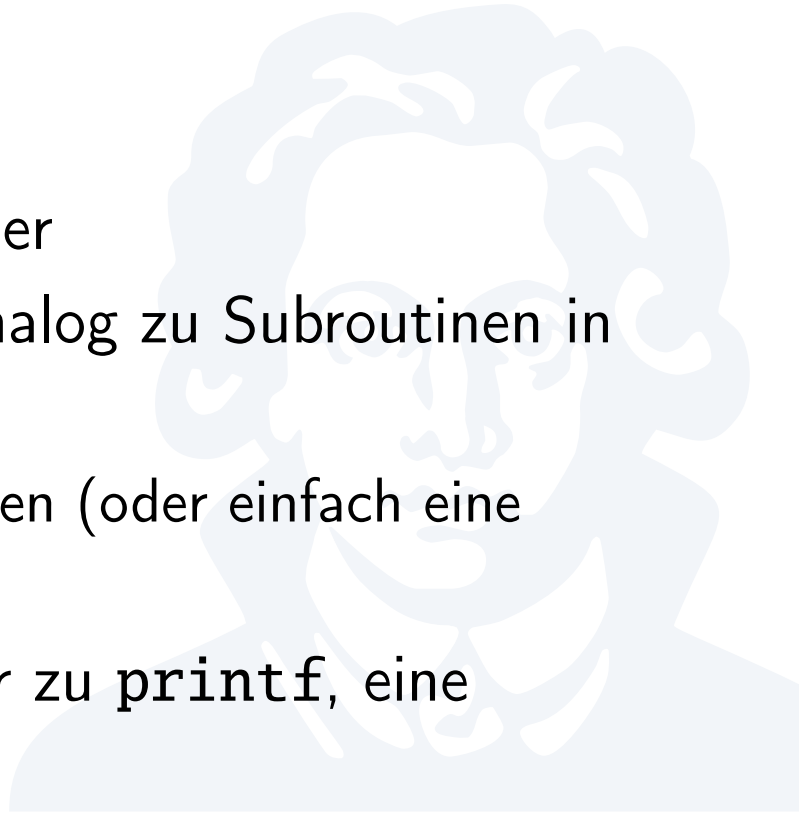
Fachbereich 12 / Institut für Informatik

13. Mai 2014



Hallo, Welt!

```
main()
{
  printf("Hallo, Welt!\n");
}
```

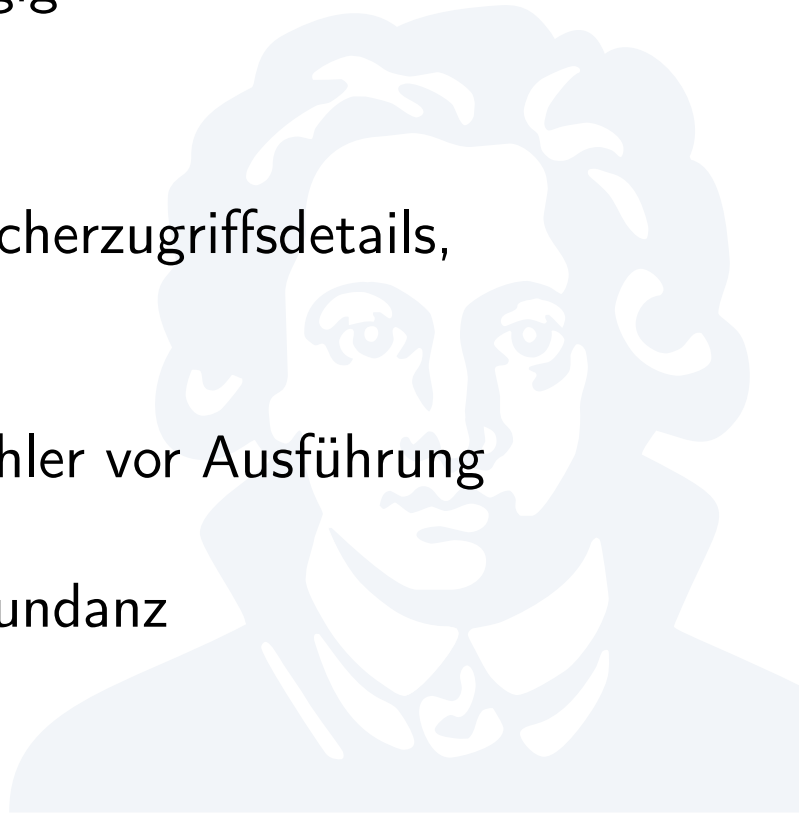


- `main`: Einsprungpunkt für den Lader
- `printf`: Name einer *Funktion* (Analog zu Subroutinen in MIPS-Assembler)
  - `printf` gibt formatierte Ausgaben (oder einfach eine Zeichenkette) aus.
- `("Hallo, Welt!\n")`: Parameter zu `printf`, eine auszugebende Zeichenkette

Eigenschaften von C

C ist...

- *ein portabler Assembler*:
  - Größtenteils Maschinenunabhängig
  - Relativ geringe Abstraktion
- *eine Hochsprache*:
  - Abstrahiert über Registern, Speicherzugriffsdetails, Maschineninstruktionsauswahl
  - Hat Standardbibliothek
  - Kann bestimmte Programmierfehler vor Ausführung entdecken
  - Erleichtert Vermeidung von Redundanz



Komponenten der Sprache C

Präprozessor

Kernsprache:

Deklarationen
weisen Namen Bedeutungen zu

Ausdrücke
beschreiben Berechnungen

Typen
klassifizieren Ausdrücke und Deklarationen

Anweisungen
ändern Zustände

Kontrollstrukturen
organisieren den Kontrollfluß

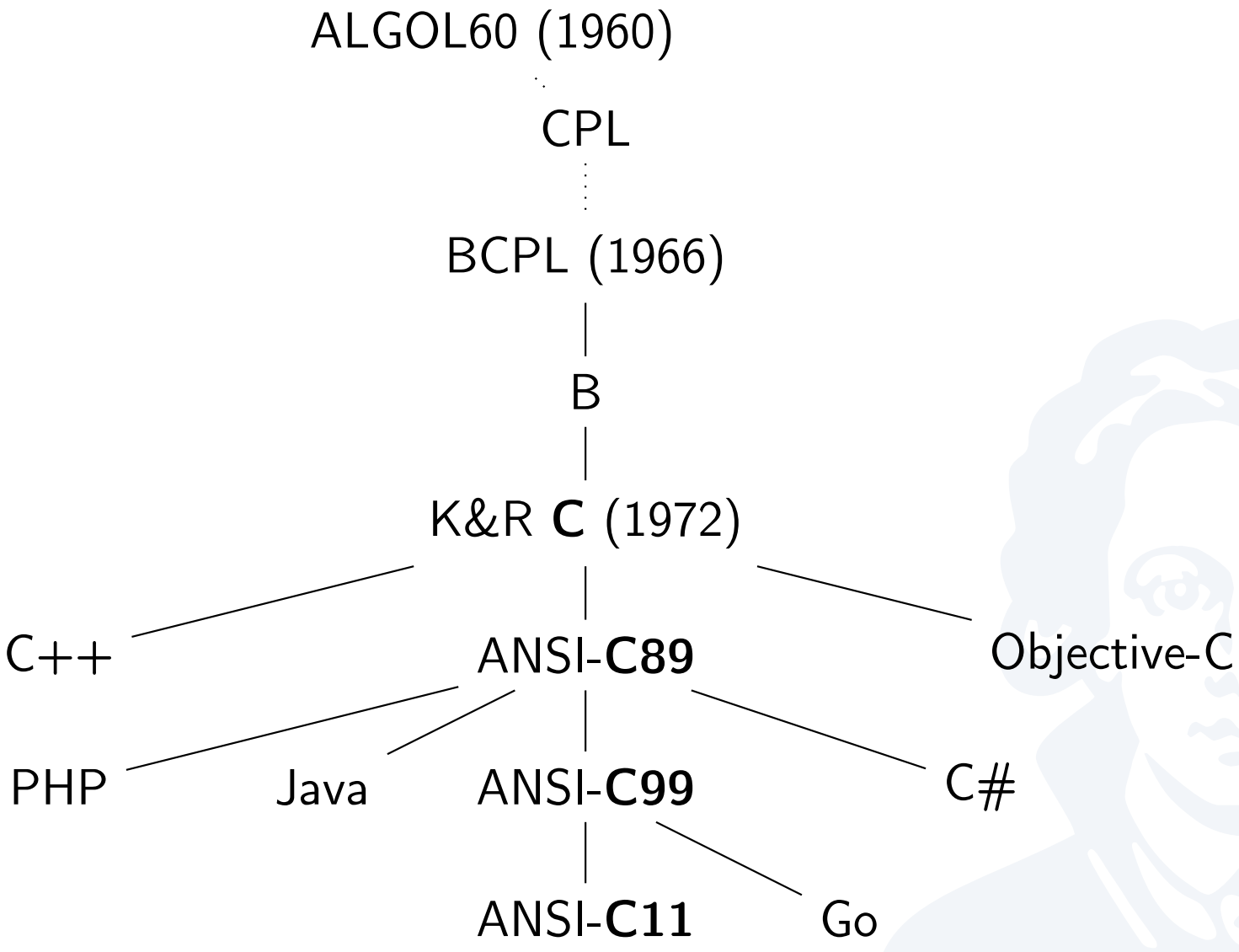
Nicht-Eigenschaften von C

C ist...

- *nicht typsicher*: C-Programme können Speicherinhalte Zeichen, Zahlen, Zeiger usw. fehlinterpretieren
  - Das C-Typsystem hilft, die gröbsten Fehler zu vermeiden
- *nicht objektorientiert*: C ist eine rein *prozedurale* Sprache
- *nicht automatisch effizient*:
  - Langsame Algorithmen in C sind immer noch langsam
- *nicht groß*:
  - Spracheigenschaften orientieren sich an typischen Prozessorfähigkeiten

C unterscheidet sich stark von C++ und anderen Verwandten

C und seine Verwandtschaft



„K&R“: Brian Kernighan, Dennis M. Ritchie (Bell Labs)

C-artige Sprachen in der Industrie

Verbreitung gemäß „Tiobe-Index“

- Basiert auf Anzahl von Webseiten über Programmiersprachen

Rang	Sprache	Verbreitung
1	**C**	16.926%
2	**Java**	16.907%
3	**Objective-C**	11.791%
4	**C++**	5.986%
5	(Visual) Basic	4.197%
6	**C#**	3.745%

`tiobe.com`, Stand Mai 2014

Einfache Berechnungen

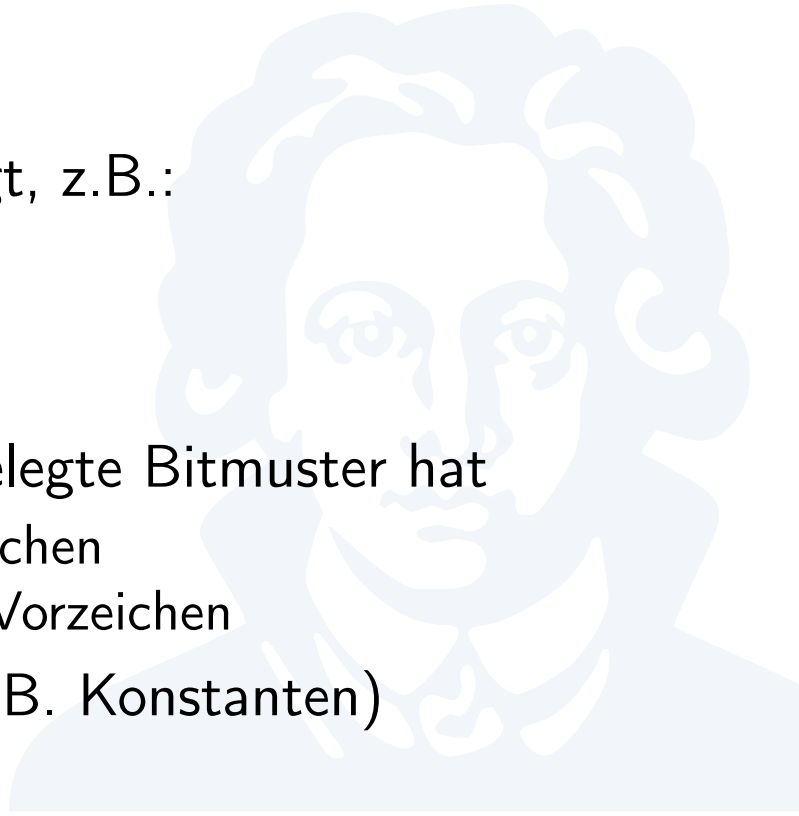
```
#include <stdio.h>                    } Präprozessoranweisung

main()  // Dies ist ein Kommentar
{       /* Dies ist auch ein Kommentar */
  int ergebnis;   // Deklariere Variable 'ergebnis'
  int wert = 5;   // Deklariere vorinitialisierte Variable   } Deklarationen
  ergebnis = wert * 7;  // Berechne, weise 'ergebnis' zu
  printf("Ergebnis = %d\n",                                  } Anweisungen
         ergebnis);
}
```

- `int` *n*: deklariert Zahlvariable *n*, vorzeichenbehaftet
- Deklarierte Variablen können initialisiert werden
- Zuweisungen per „`ziel = ausdruck`“

Variablen-Deklarationen (1/2)

$$\underbrace{\texttt{int}}_{\textit{Typ}}\ \underbrace{\texttt{name}}_{\textit{Bezeichner}}\ \underbrace{\texttt{= ausdruck}}_{\textit{Initialisierung (optional)}}\ \texttt{;}$$

- Typen geben an:
  - *wieviele Bytes* die Variable belegt, z.B.:
    `short`: 2 oder mehr
    `int`: 2 oder mehr
    `long`: 4 oder mehr
  - *welche Bedeutung* das dort abgelegte Bitmuster hat
    - `short`, `int`, `long`: mit Vorzeichen
    - `unsigned short`, ...: ohne Vorzeichen
  - *welche Benutzung* erlaubt ist (z.B. Konstanten)

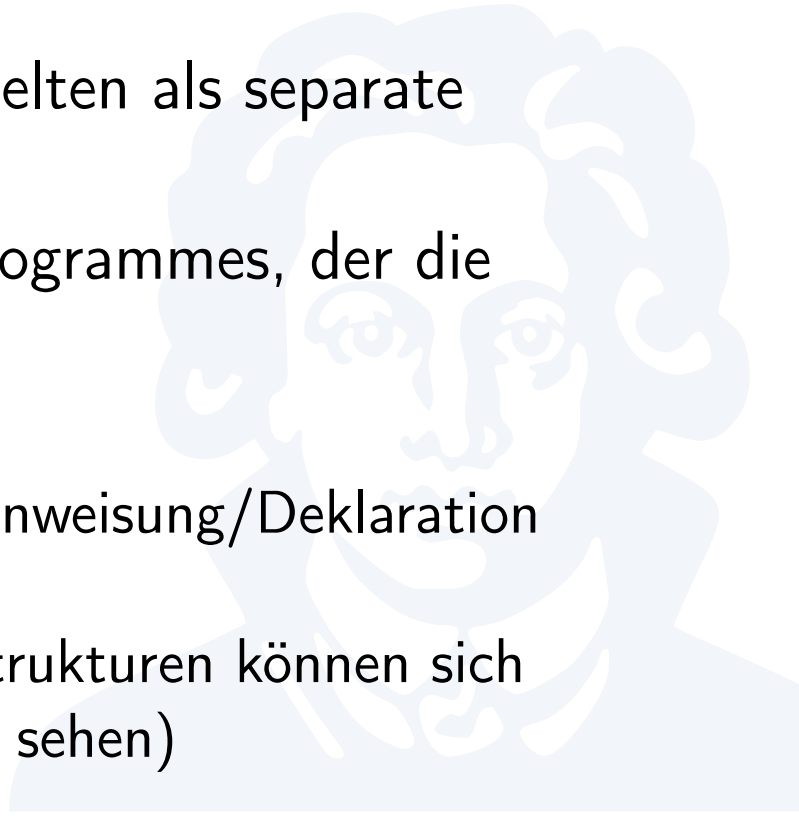


Variablen-Deklarationen (2/2)

Mehrere Variablen können zusammen deklariert werden:

```
short a, b;
int c = 2, d = c+1;
```

- Komma-getrennte Deklarationen gelten als separate Deklarationen: `d` kann `c` sehen
- *Gültigkeitsbereich*: Der Teil des Programmes, der die Variable sehen kann
- In C fast immer:
  - Gültigkeitsbereich ab nächster Anweisung/Deklaration
  - Bis Ende des aktuellen Blocks
  - (Ausnahmen: Funktionen und Strukturen können sich selbst während ihrer Deklaration sehen)



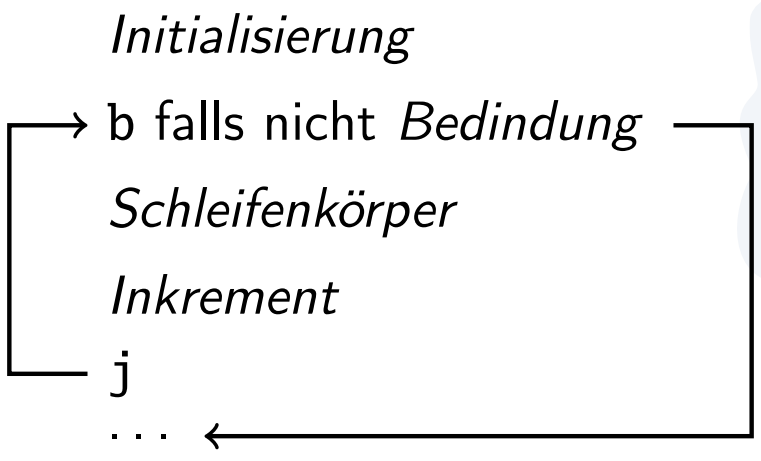
Iterative Berechnungen mit der `for`-Schleife

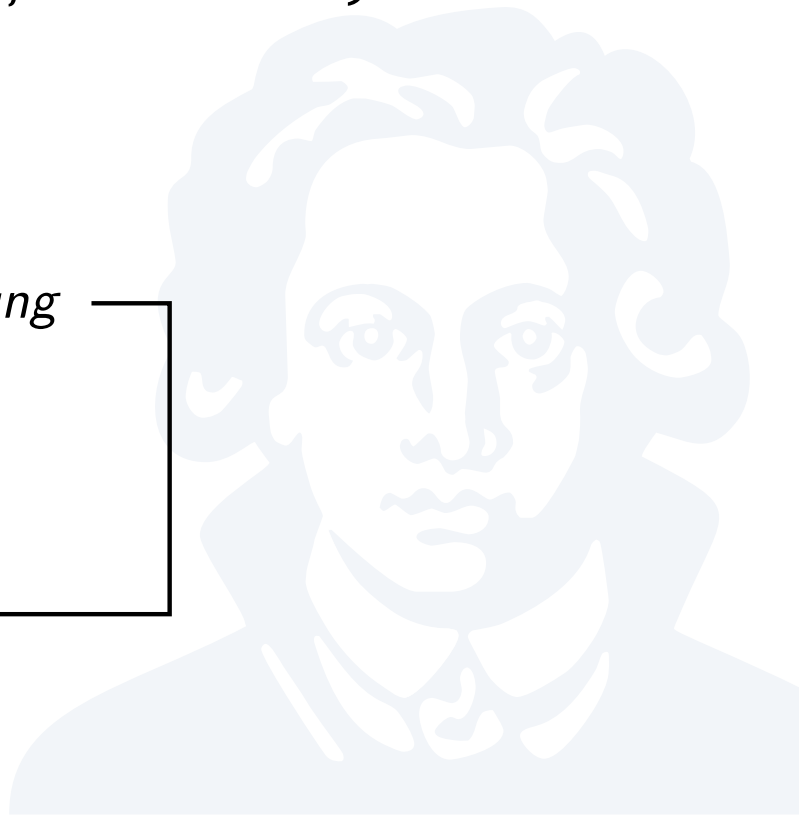
```
main()
{
  int wert = 1;
  for (int i = 1;// Initialisierung: Deklaration/Anweisung
           i < 7;          // Bedingung: Ausdruck
           i = i + 1) {  // Inkrement: Anweisung
           // Schleifenkörper: eine Anweisung
      wert *= i; // Kurz für:  wert = wert * i
    }
}
```

- *Initialisierung* wird vor Schleifenbeginn ausgeführt
- *Schleifenbedingung* wird vor jeder Ausführung des Schleifenkörpers geprüft
- *Inkrement* wird immer nach dem Schleifenkörper ausgeführt

Die `for`-Schleife in C

```
main()
{
  for (Initialisierung; Bedingung; Inkrement)
    Schleifenkörper
```





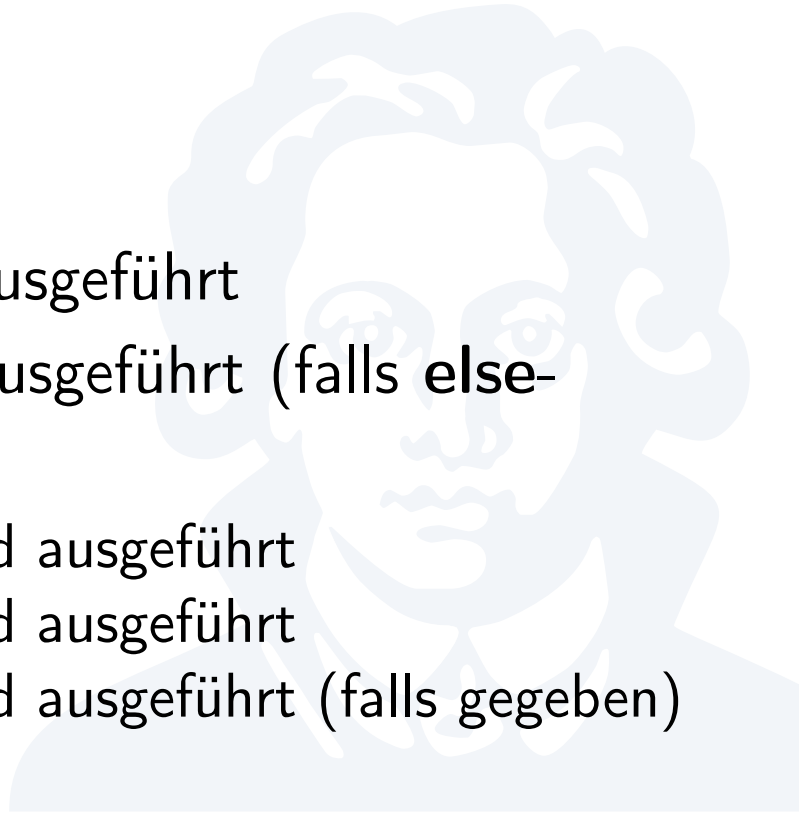
Die `if`-Anweisung

```
if (Ausdruck) // Bedingung
  Anweisung-1
else                          } optional
  Anweisung-0
```

Ausführung: *Ausdruck* wird ausgewertet:

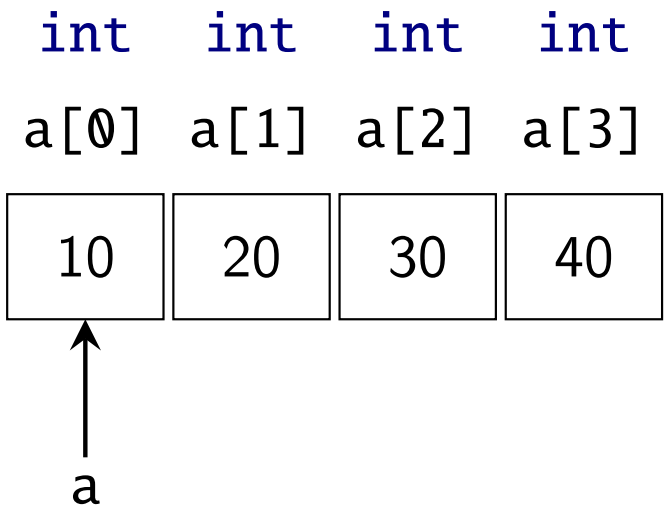
- Ergebnis $\neq 0$: *Anweisung-1* wird ausgeführt
- Ergebnis $= 0$: *Anweisung-0* wird ausgeführt (falls **else**-Zweig vorhanden)

`if (2 > 1) ...`:	Anweisung-1 wird ausgeführt
`if (1) ...`:	Anweisung-1 wird ausgeführt
`if (17 - 17) ...`:	Anweisung-0 wird ausgeführt (falls gegeben)



Arrays (leicht vereinfacht)

```
int a[4] = {10, 20, 30, 40};
```



- a zeigt auf Speicher mit 4 `int`-Werten
- Lesezugriff: `a[1]` ist `20`, `a[3]` ist `40`
- Schreibzugriff: z.B.: `a[1] = 60`

Verschachtelte Schleifen

```
int a[4] = {10, 20, 30, 40};
int b[12] = {1, 7, 20, 13, 54, 12, 30, 100 , 4, 2, 9, 43};
main() {
  int gleich = 0;
  for (int ib = 0; ib < 12; ib += 1)
    for (int ia = 0; ia < 4; ia += 1)
      if (a[ia] == b[ib])// Ist 'ia'ter Eintrag in 'a'
                         // gleich 'ib'tem Eintrag in 'b'?
        // bedingte Ausführung:
        gleich += 1;
  printf("%d", gleich);
}
```

- `int a[4]`: Typdeklaration für Array (Feld) mit 4 `int`-Einträgen
- `... = {10, 20, 30, 40}`: Arrayinitialisierung

Übersetzung nach MIPS (gcc -O3 für MIPS32)

```
        addiu   $sp, $sp, -24
        la      $t1, a
        la      $t2, b
        sw      $ra, 20($sp)
        move    $a1, $zero
        move    $t0, $zero
        sll     $v0, $t0, 2
$L9:    addu    $v0, $t2, $v0
        lw      $a3, 0($v0)
        move    $a2, $zero
$L4:    sll     $v0, $a2, 2
        addu    $v0, $t1, $v0
        lw      $v1, 0($v0)
        addiu   $a2, $a2, 1
        xor     $v1, $v1, $a3
        addiu   $v0, $a1, 1
        slt     $a0, $a2, 4
        bne     $a0, $zero, $L4
        movz    $a1, $v0, $v1
        addiu   $t0, $t0, 1
        slt     $v0, $t0, 12
        bne     $v0, $zero, $L9
        sll     $v0, $t0, 2

        lui     $a0, %hi($LC0)
        jal     printf
        addiu   $a0, $a0, %lo($LC0)

        lw      $ra, 20($sp)
        move    $v0, $zero
        j       $ra
        addiu   $sp, $sp, 24
```

$t1	a
$t2	b
$a2	ia
$t0	ib
$a1	gleich

Übersetzung nach x86_64 (gcc -O2)

```
    xorl    %edi, %edi
    xorl    %esi, %esi
.L2:
    movslq  %edi, %rax
    movl    b(,%rax,4), %ecx
    xorl    %eax, %eax
.L4:
    movslq  %eax, %rdx
    cmpl    %ecx, a(,%rdx,4)
    sete    %dl
    addl    $1, %eax
    movzbl  %dl, %edx
    addl    %edx, %esi
    cmpl    $4, %eax
    jne     .L4
    addl    $1, %edi
    cmpl    $12, %edi
    jne     .L2
```

```
    subq    $8, %rsp
    movl    $.LC0, %edi
    xorl    %eax, %eax
    call    printf
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
```

Kontrollstrukturen: Übersicht

Bedingte Anweisungen:

```
if (Ausdruck) // Bedingung
  Anweisung
```

mit Alternative:

```
if (Ausdruck) // Bedingung
  Anweisung
else
  Anweisung
```

Fallunterscheidung:

```
switch (Ausdruck)
{case Wert: Anweisungen
...default: Anweisungen }
```

Schrittweise Wiederholung:

```
for (Anweisung; // Init.
     Ausdruck;  // Bedingung
     Anweisung) // Fortschritt
  Anweisung
```

Wiederholen solange Bedingung wahr:

```
while (Ausdruck) // Bedingung
Anweisung
```

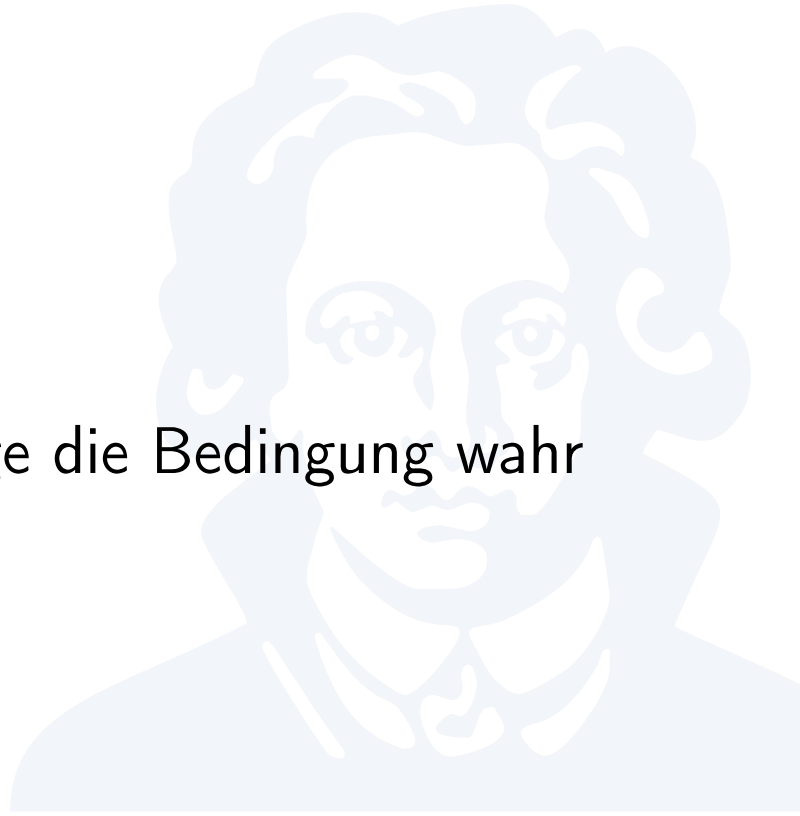
mindestens 1× ausführen:

```
do Anweisung while (Ausdruck)
// Bedingung
```

Kontrollstrukturen: `while`

```
int summe = 0;
int i = 0;
while (summe < 1000) {
  i += 1;
  summe += i * i;
}
```

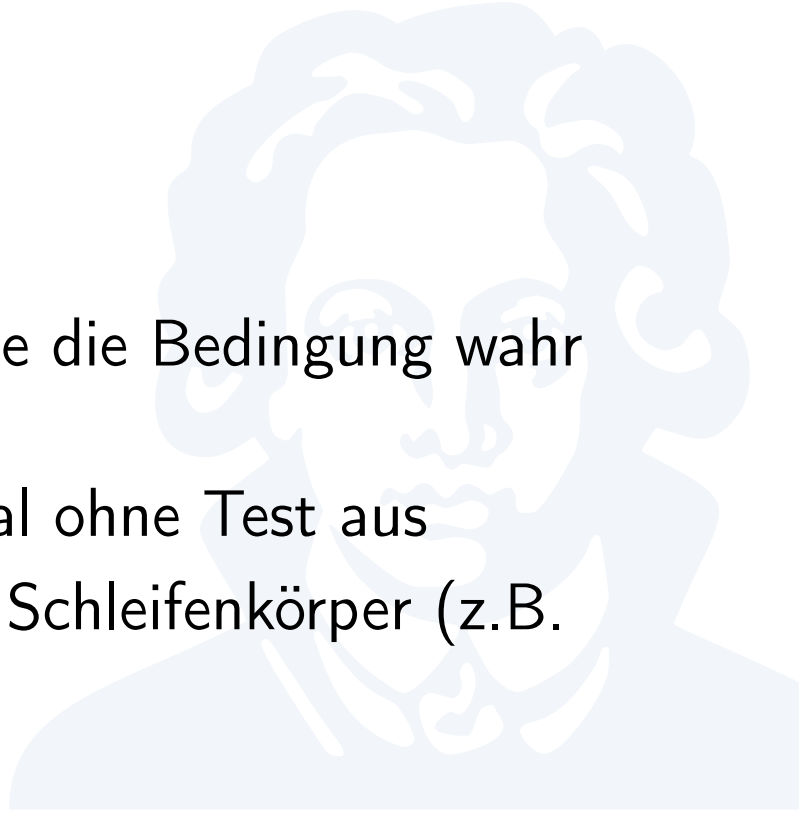
- Wiederholt Schleifenkörper, solange die Bedingung wahr ($\neq 0$) ist



Kontrollstrukturen: `do ... while`

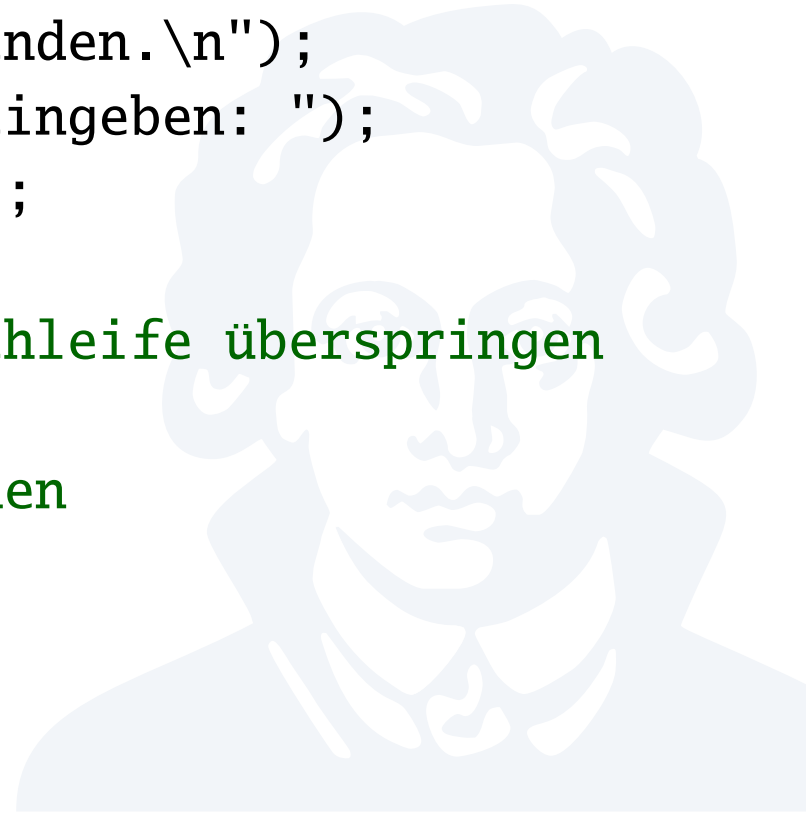
```
int summe = 0, eingabe;
do {
  int dummy = 0;
  eingabe = lies_eingabe();
  summe += eingabe;
} while (eingabe > 0);
```

- Wiederholt Schleifenkörper, solange die Bedingung wahr ($\neq 0$) ist
- Führt Schleifenkörper das erste Mal ohne Test aus
- Bedingung kann Deklarationen *im* Schleifenkörper (z.B. `dummy`, hier) nicht sehen



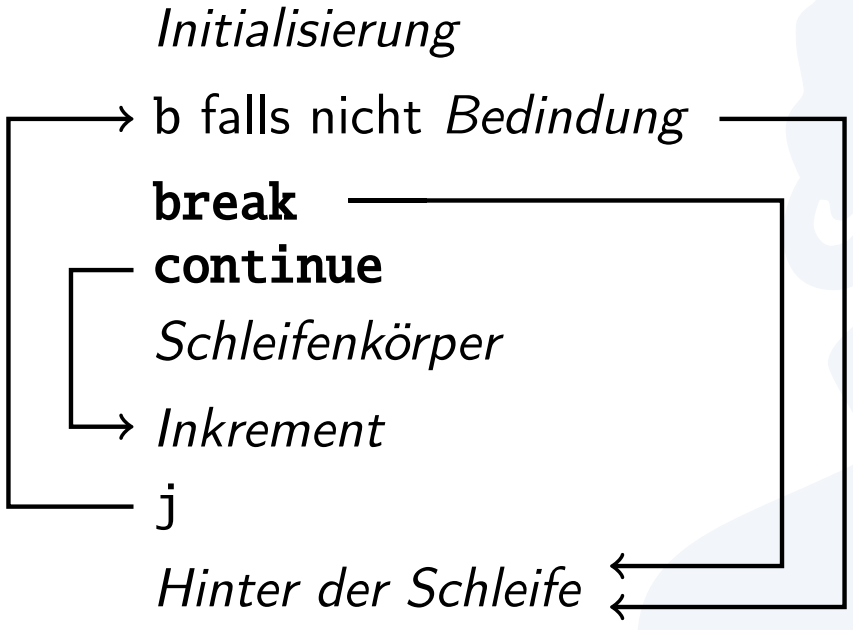
Kontrollfluß: `break` und `continue`

```
int summe;
while (1) {  // Keine Abbruchbedingung
    printf("Summe = %d.\n");
    printf("0 eingeben, um zu beenden.\n");
    printf("Sonst positive Zahl eingeben: ");
    int eingabe = zahl_einlesen();
    if (eingabe < 0)
        continue;  // Rest der Schleife überspringen
    if (eingabe == 0)
        break;  // Schleife beenden
    summe += eingabe;
}
```



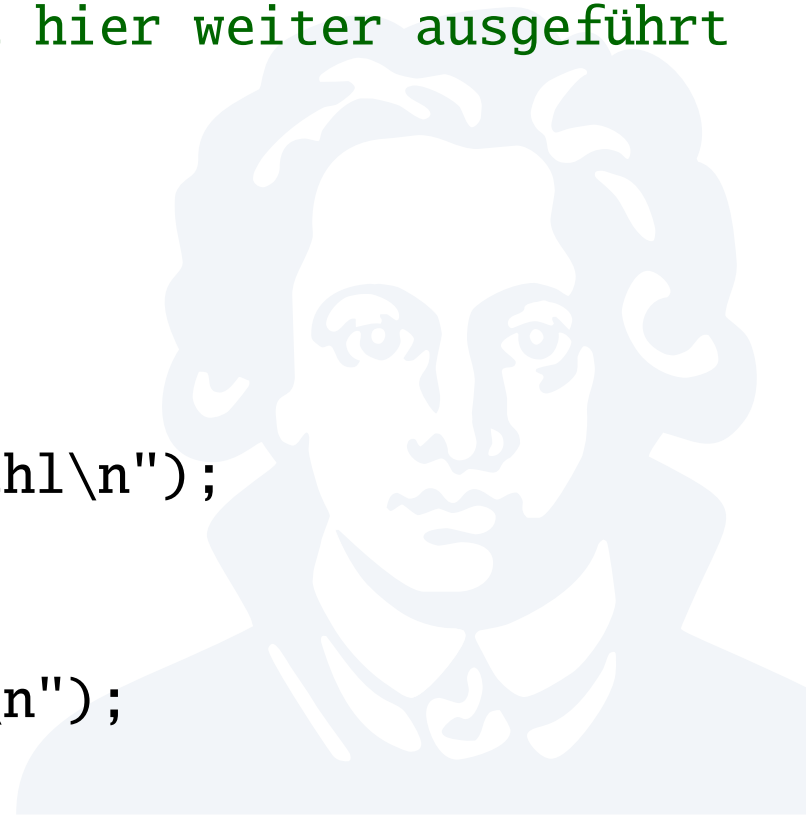
Die `for`-Schleife in C

```
main()
{
  for (Initialisierung; Bedingung; Inkrement)
    Schleifenkörper
  Hinter der Schleife
```



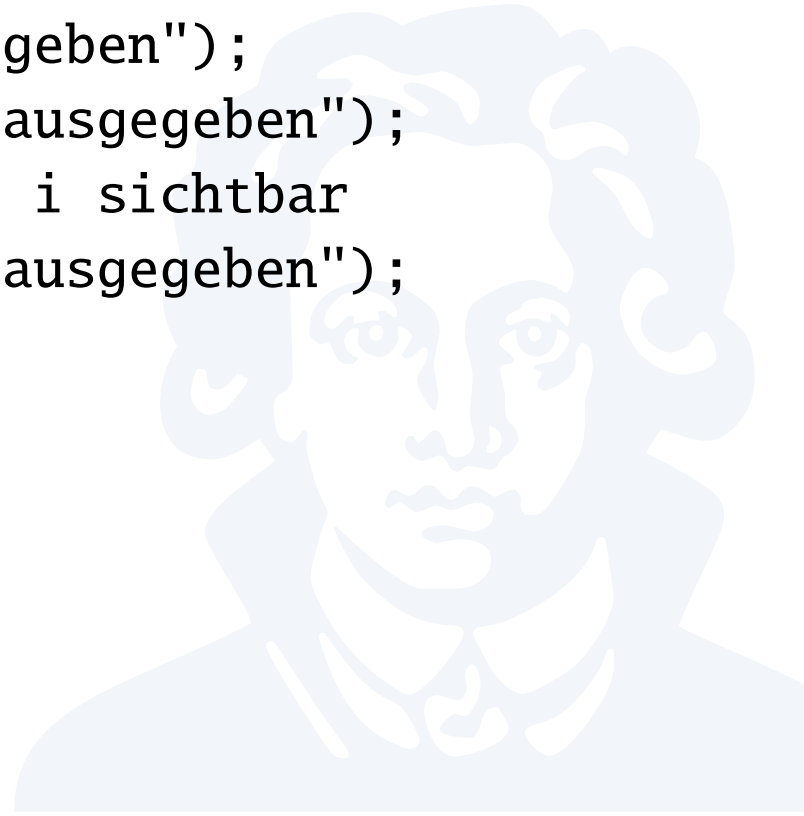
Kontrollstrukturen: switch

```
int mips_opcode = lies_opcode();
switch (mips_opcode) {
  case 0x02:
    printf("j\n");
    break;  // ohne break würde hier weiter ausgeführt
  case 0x03:
    printf("jal\n");
    break;
  case 0x00:
  case 0x18:
    printf("arithmetischer Befehl\n");
    break;
  default:
    printf("unbekannter Befehl\n");
}
```



Kontrollstrukturen: Blöcke

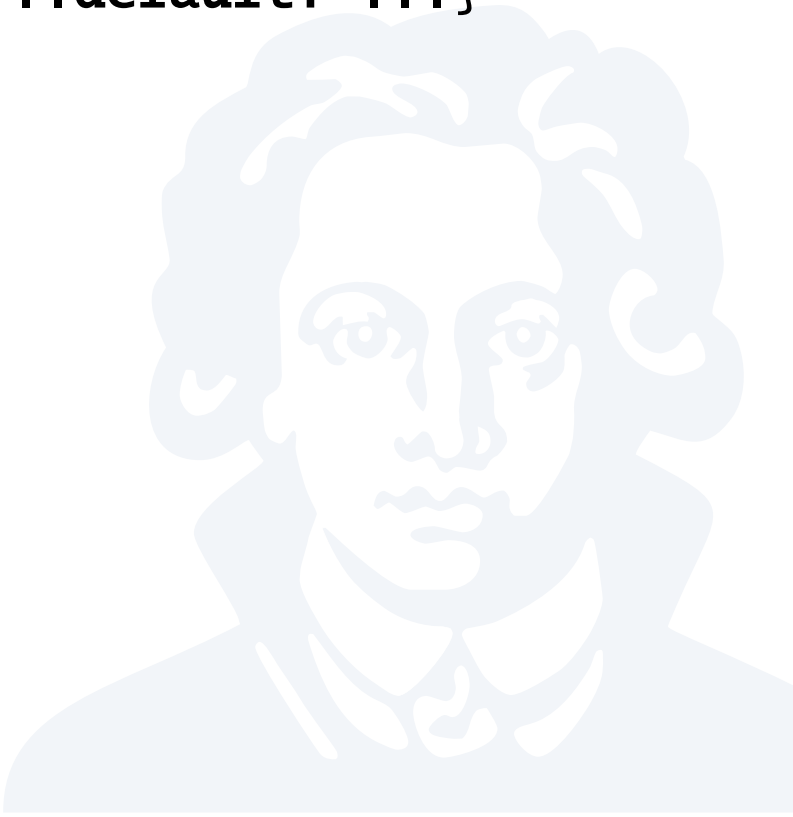
```
{  // Block
    printf("Wird zuerst ausgegeben");
    printf("Wird als zweites ausgegeben");
    int i = 1; // Ab hier ist i sichtbar
    printf("Wird als drittes ausgegeben");
}
```



Kontrollstrukturen: Zusammenfassung

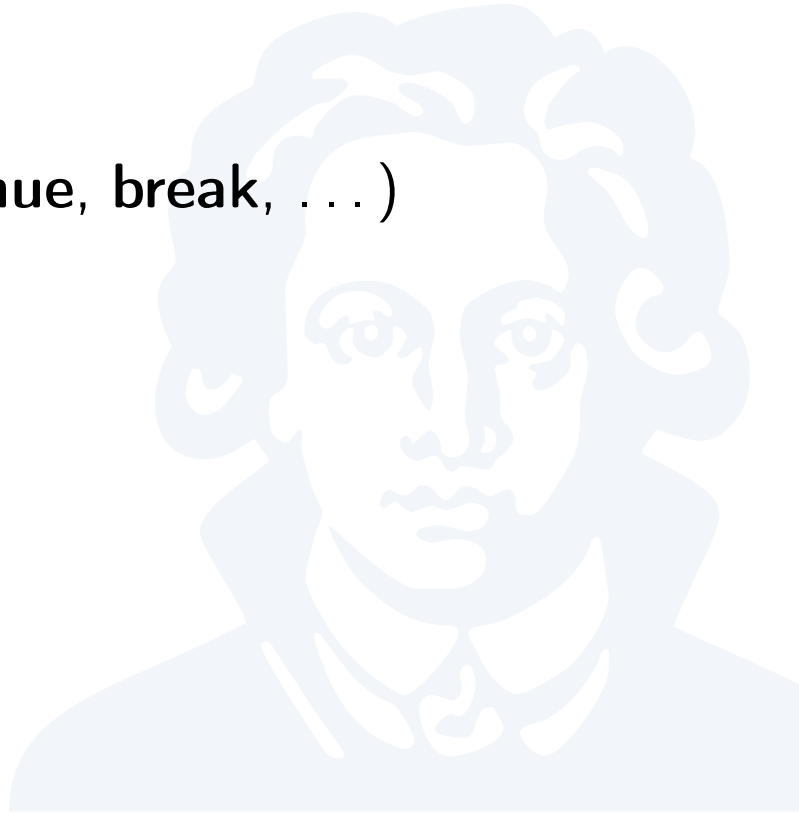
- Bedingte Ausführung:
  - `if ...else ...`
- Fallunterscheidung:
  - `if (...) ...else ...`
  - `switch (...) { case ...: ...default: ...}`
- Wiederholung:
  - `for (...; ...; ...) ...`
  - `while (...) ...`
  - `do ...while (...);`
  - ...oder Rekursion (später!)
- Sequenz:

```
{  // Block
    Anweisung-0;
    ...
    Anweisung-n;
}
```



Anweisungen

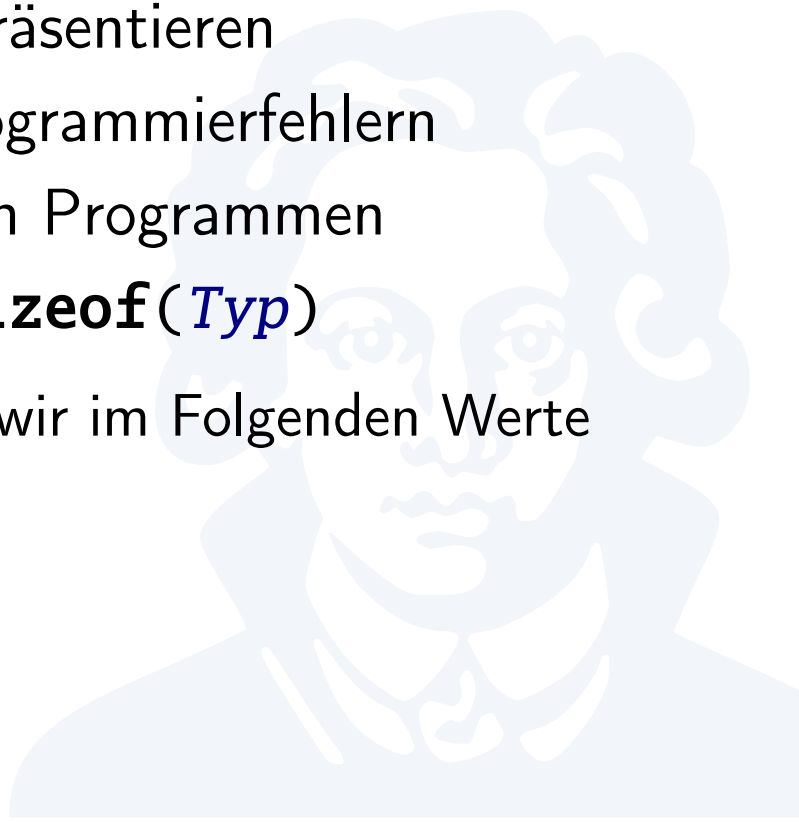
- Anweisungen werden fast immer durch ein Semikolon (;) abgeschlossen.
- Anweisungen nehmen folgende Formen an:
  - Kontrollstrukturen
  - Blöcke
  - Kontrollflußanweisungen (**continue**, **break**, . . .)
  - Deklarationen (seit C99)
  - Ausdrücke:
    - Werte
    - Berechnungen
    - Konvertierungen
    - Zuweisungen (=, +=, *=, . . .)
    - Bitoperationen



Werte und Typen

- Jeder Wert und jeder Ausdruck hat einen *Typ*
- Typen helfen beim effizienten Repräsentieren
- Typen helfen beim Finden von Programmierfehlern
- Typen helfen beim Konzipieren von Programmen
- Jeder Typ hat eine Byte-Größe: **sizeof**(*Typ*)

Da jeder Wert einen Typ hat, betrachten wir im Folgenden Werte anhand ihrer Typen



Ganzzahlen-Typen

Typ	Seit	sizeof gemäß C99	sizeof x86/gcc, MIPS	Direktnotation
`char`		1 oder mehr	1	
`short int`		2 oder mehr	2	
`int`		2 oder mehr	4	42, 0x2a, 0
`long int`		4 oder mehr	4	42l 0x2al, 0L
`long long int`	**C99**	8 oder mehr	8	42ll, 0x2all, 0LL

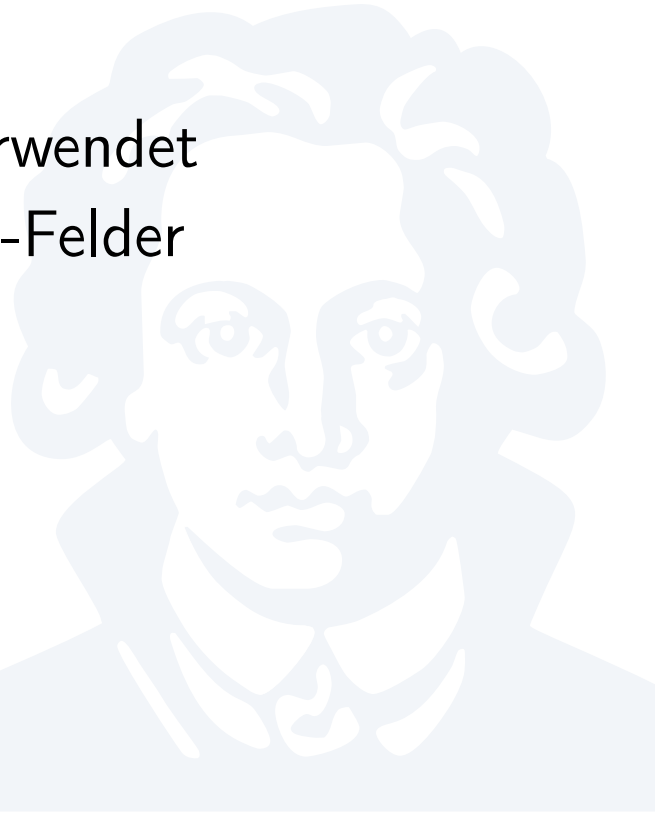
- Alle vorzeichenbehaftet, wenn kein `unsigned` vorangestellt wird
- Direktnotation für `unsigned`: `42u`, `42ul`, `42lu`, `17U` ...
- `int` kann weggelassen werden, wenn der Typname noch andere Komponenten hat (z.B. `unsigned long`)

Zeichen (char)

```
char zeichen0 = 'A';
char zeichen1 = 65;  // == zeichen0
char zeichen2 = zeichenkette[0]; // 'A'
char zeichen3 = zeichenkette[3]; // 0
```

- char wird auch für einzelne Zeichen verwendet
- Zeichenketten sind nullterminierte char-Felder

Beliebiges Zeichen	0x*n* '\x*n*'
Zeilenumbruch	0x0A '\n'
Tabulator	0x08 '\t'
Zeichenketten-Terminator	0x00 '\0'
'	0x25 '\''
"	0x22 '\"'
\	0x5C '\\'



Fließkommazahlen

Typ	Spezifikation	Beispielwerte
`float`	IEEE-754 **binary32**	`1.3f`, `1.2e7f`, `1.2e-7F`
`double`	IEEE-754 **binary64**	`1.3`, `22.0e+23`, `0x2a.e7p8`
`long double`	**binary64** oder besser	`1.3l`, `22E23l`, `.5L`

Vorsicht: Fließkommazahlen können u.U. nicht exakt repräsentiert werden

Konvertierung: der *cast*-Operator

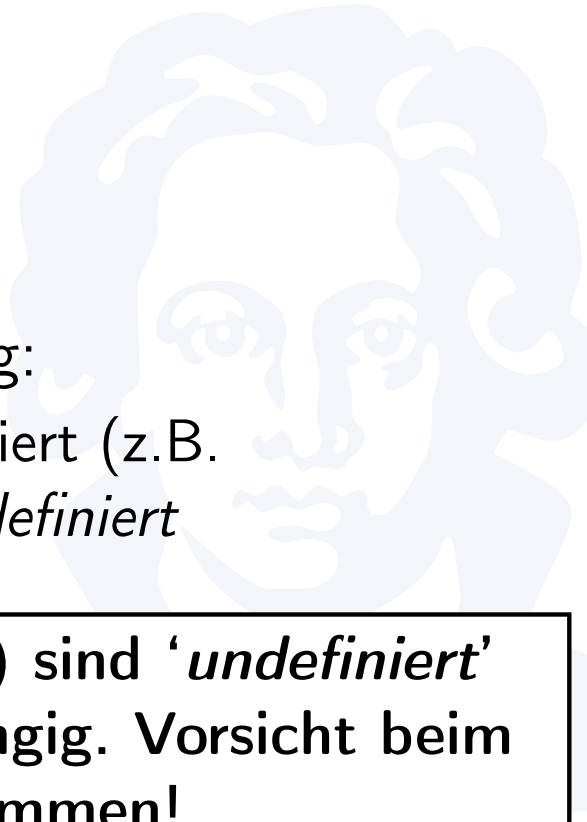
- Konvertierungen eines Wertes `w` zu einem Typen `t`:

```
(t) w
```

- Beispiel: `(int) 2.3f`
- Kann auch implizit geschehen, z.B.

```
double d = 2.3;
int i = d;
```

- Sucht den besten passenden Wert
- Bei Ganzzahl-zu-Ganzzahl-Konvertierung:
  - Wenn kein *exakt* passender Wert existiert (z.B. `(unsigned) -1`) ist das Ergebnis *undefiniert*

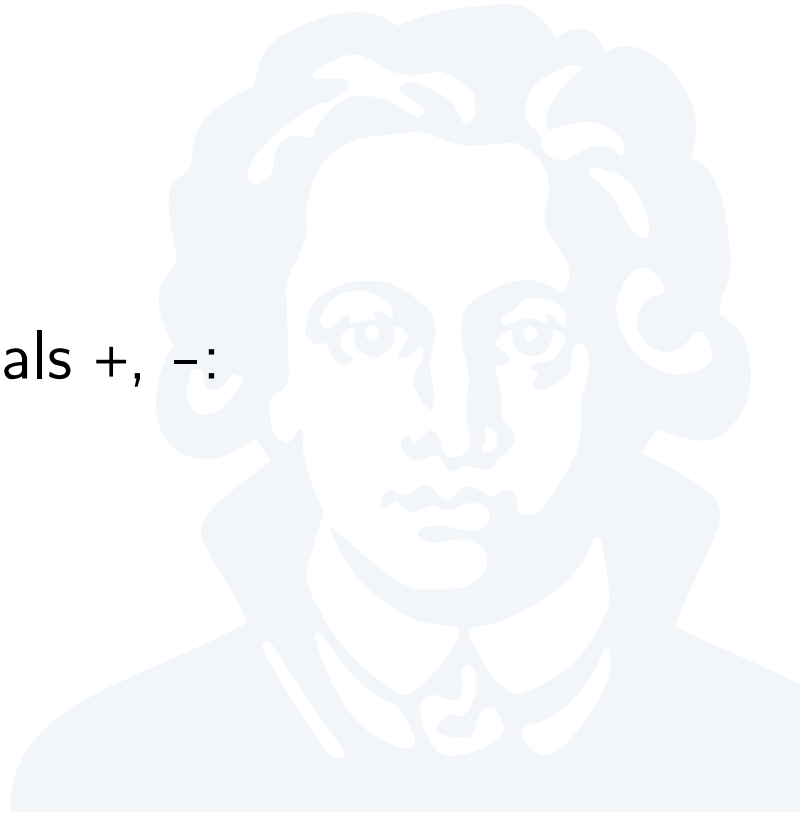


Viele Dinge in C (und noch mehr in C++!) sind '*undefiniert*' und damit Maschinen- und Übersetzerabhängig. Vorsicht beim Schreiben von portablen Programmen!

Arithmetische Operationen

C definiert arithmetische Infix-Operationen für ganze und Fließkommazahlen:

- +, -, *, /
- %: Modulus bzw. Divisionsrest
  `7 % 3` ist `1`
- *Linksassoziativ*:
  ```
      1 - 2 - 3
  =   (1 - 2) - 3
  ```
- Operatoren *, /, % binden stärker als +, -:
  ```
      1 + 2 * 3 / 4 + 5
  =   (1 + ((2 * 3) / 4)) + 5
  ```
- Präfix - negiert:
  ```
  int i = -2;
  int i2 = -i;  // 2
  ```



Boolesche Operationen und Vergleiche

Boolesche Operationen:

- `a && b`: Logisches ‘und’
- `a || b`: Logisches ‘oder’
- `!a`: Logische Negation

Vergleiche:

- == (gleich), != (ungleich)
- <, <=, >=, >

Der „ternäre Operator“ / `if`-Ausdruck:

- *Bedingung* `?` $Ausdruck_1$ `:` $Ausdruck_0$
- Wenn *Bedingung* wahr, dann $Ausdruck_1$, sonst $Ausdruck_0$

Beispiel:

```
int abs_wert = (wert < 0)? -wert : wert;
```

Bit-Manipulation mit den Bitoperatoren

- Bit-Und (&):

```
      0x1813
  &   0x2801
-------------
  =   0x0801
```

- Bit-Oder (|):

```
      0x1813
  |   0x2801
-------------
  =   0x3813
```

- Bit-Exklusiv-Oder (^):

```
      0x1813
  ^   0x2801
-------------
  =   0x3012
```

- Links schieben (<<):

```
          0x1813
  <<           4
-----------------
  =   0x18130
```

- Rechts schieben (>>):

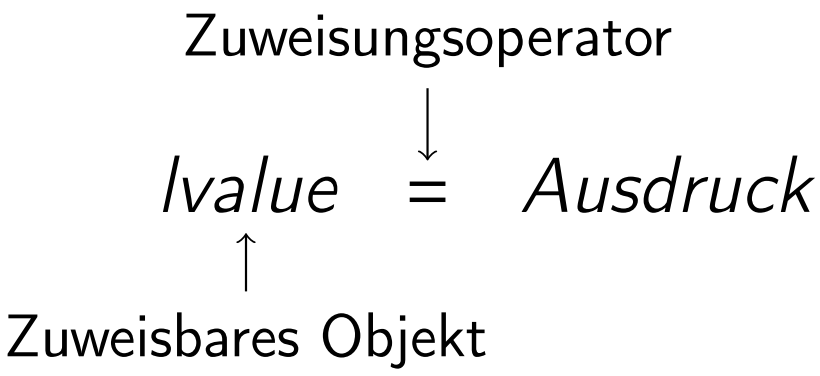
```
          0x1813
  >>           4
-----------------
  =    0x181
```

- Bit-Negation (~):

```
  ~        0x1813
-----------------
  =   0xffffe7ec
```

Zuweisungen und lvalues

Format von Zuweisungen:



- *Ausdruck* und *lvalue* müssen den gleichen Typ haben
- Jeder *lvalue* ist auch ein Ausdruck
- Ein *lvalue* repräsentiert ein zuweisbares Objekt (Register, Speicherzelle(n)), z.B.:
  - Variable (`ib`)
  - Zelle in einem Array (`b[ib]`)

Zuweisungen sind Ausdrücke

- Direkt: =
- Arithmetisch: +=, -=, *=, /=, %=
- Bitweise: <<=, >>=, &=, ^=, |=
- Preinkrement / Predekrement:

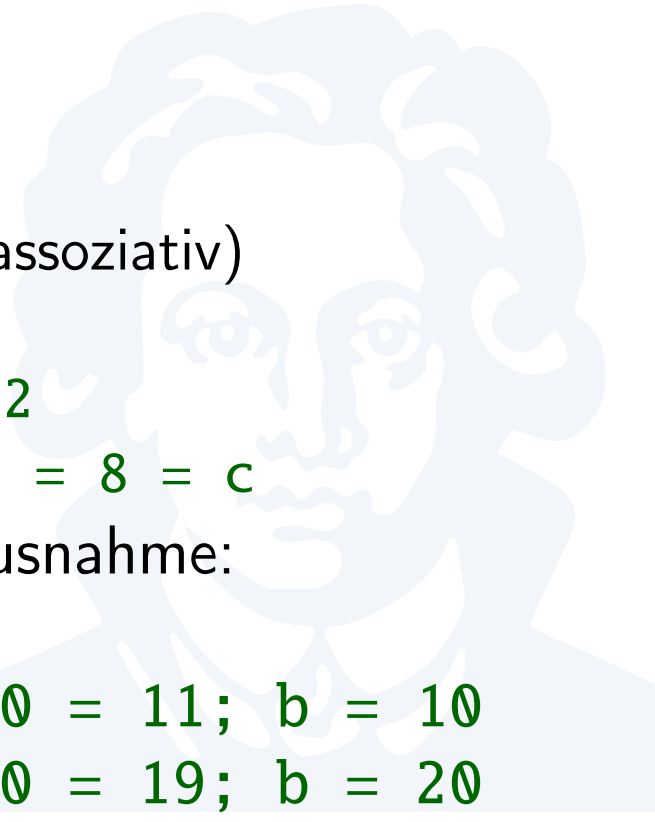
  `++a` gleichbedeutend zu: `a += 1`

  `--a` gleichbedeutend zu: `a -= 1`
- Jede Zuweisung ist ein Ausdruck:
  - Wert ist berechnetes Ergebnis (rechtsassoziativ)

```
a = 0;
b = a += 2;          // b = a = 2
c = b += a *= 3;     // a = 6, b = 8 = c
```

- Postinkrement / Postdekrement sind Ausnahme:

```
int a10 = 10, a20 = 20;
b = a10++                    // a10 = 11; b = 10
b = a20-                     // a20 = 19; b = 20
```



Operatoren: Zusammenfassung

Operatoren (stärker bindende oben)	Arg.	Assoz.
(*Ausdruck*) [...] -> .	2	links
! ~ ++ - + - (typ) * & **sizeof**	1	rechts
* / %	2	links
+ -	2	links
<< >>	2	links
< <= >= >	2	links
== !=	2	links
&	2	links
^	2	links
|	2	links
&&	2	links
||	2	links
... ? ... : ...	3	rechts
= += -= *= /= %= <<= >>= |= &= ^=	2	rechts
,	2	links

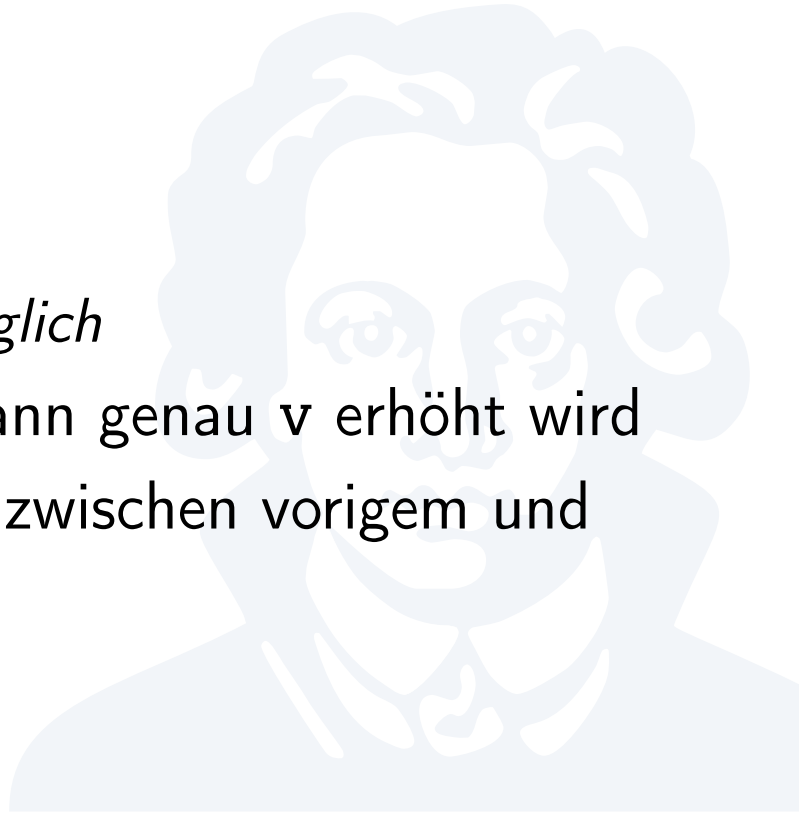
Operatoren verknüpfen Ausdrücke zu anderen Ausdrücken

Ausführungsreihenfolge

```
int v;
v=1; int w1 = v++ + v; // Wann wird v erhöht?
v=1; int w2 = v + v++;
v=1; int w3 = ++v + v;
v=1; int w4 = v + ++v;
```

Sowohl 2 als auch 3 sind als Ergebnis möglich

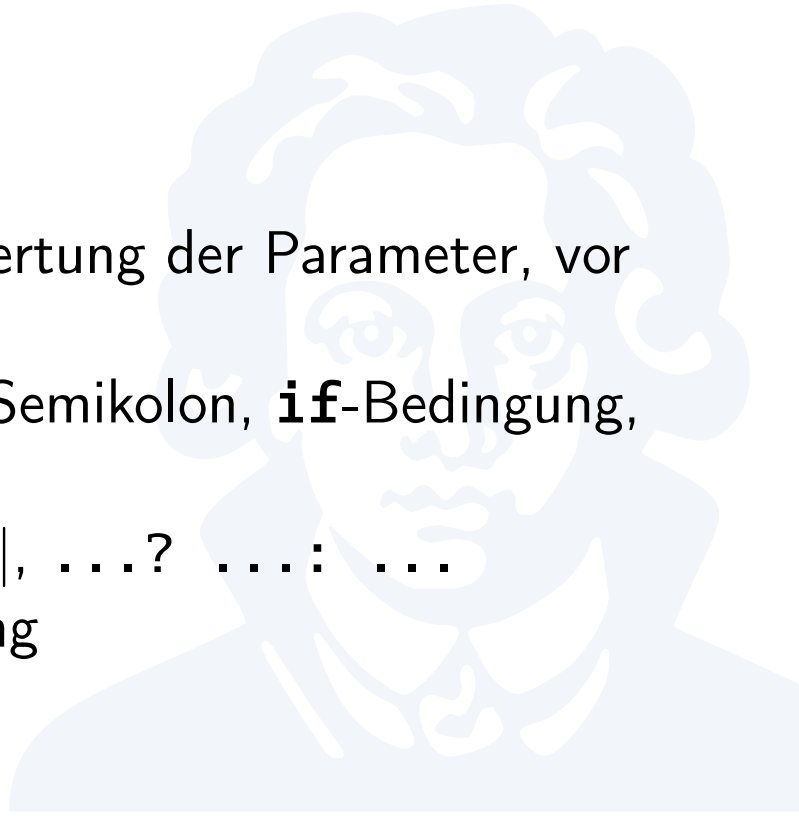
- Sprachdefinition legt nicht fest, wann genau v erhöht wird
- Einzige Garantie: Erhöhung findet zwischen vorigem und nächstem *Sequenzpunkt* statt



Sequenzpunkte

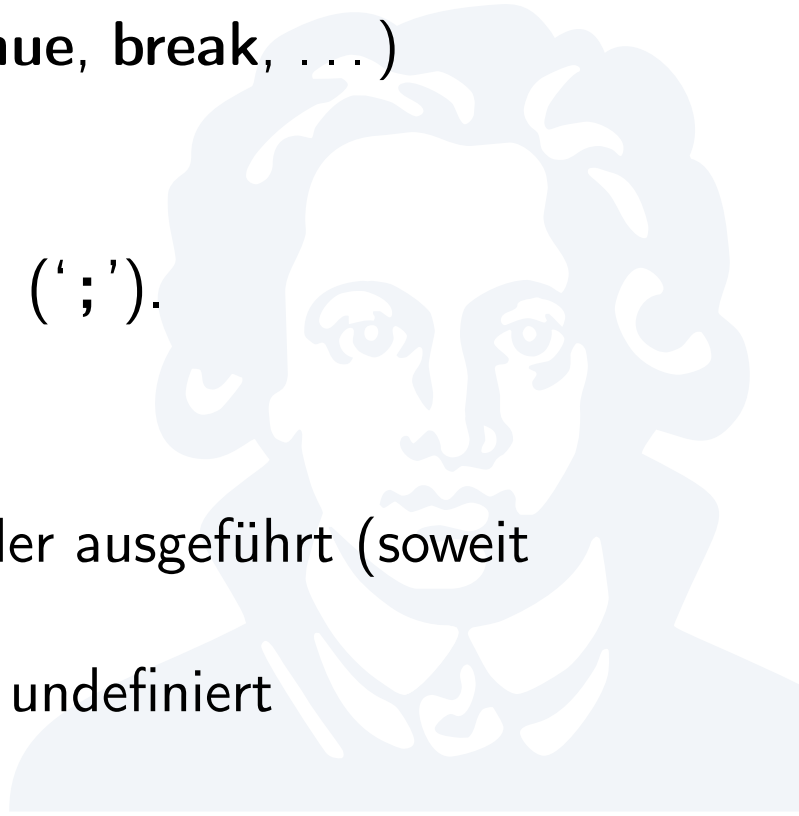
Sequenzpunkte (*sequence points*) sind Programmstellen, an denen Seiteneffekte abgeschlossen sein müssen.

- Seiteneffekte:
  - Speicherzugriffe
  - Zuweisungen
- Sequenzpunkte in C99:
  - Funktionsausdrücke: nach Auswertung der Parameter, vor Funktionsaufruf
  - Nach jedem „vollen Ausdruck“ (Semikolon, **`if`**-Bedingung, **`while`**-Bedingung, . . .)
  - Nach erstem Parameter zu &&, ||, `...? ...: ...`
  - Nach jeder Variableninitialisierung
  - Diverse Sonderfälle



Ausdrücke und Anweisungen: Zusammenfassung

- Ausdrücke beschreiben Berechnungen und Zuweisungen
- Anweisungen beinhalten:
  - Kontrollstrukturen (**if**, **while**, **do**...**while**, **switch**)
  - Blöcke { ... }
  - Kontrollflußanweisungen (**continue**, **break**, ...)
  - Ausdrücke
  - Deklarationen (seit C99)
- Anweisungen enden mit Semikolon ('**;**').
  Ausnahme: Blöcke.
- Ausführungsreihenfolge:
  - Anweisungen werden nacheinander ausgeführt (soweit beobachtbar)
  - Innerhalb von Ausdrücken meist undefiniert
  - Ausnahme: Sequenzpunkte



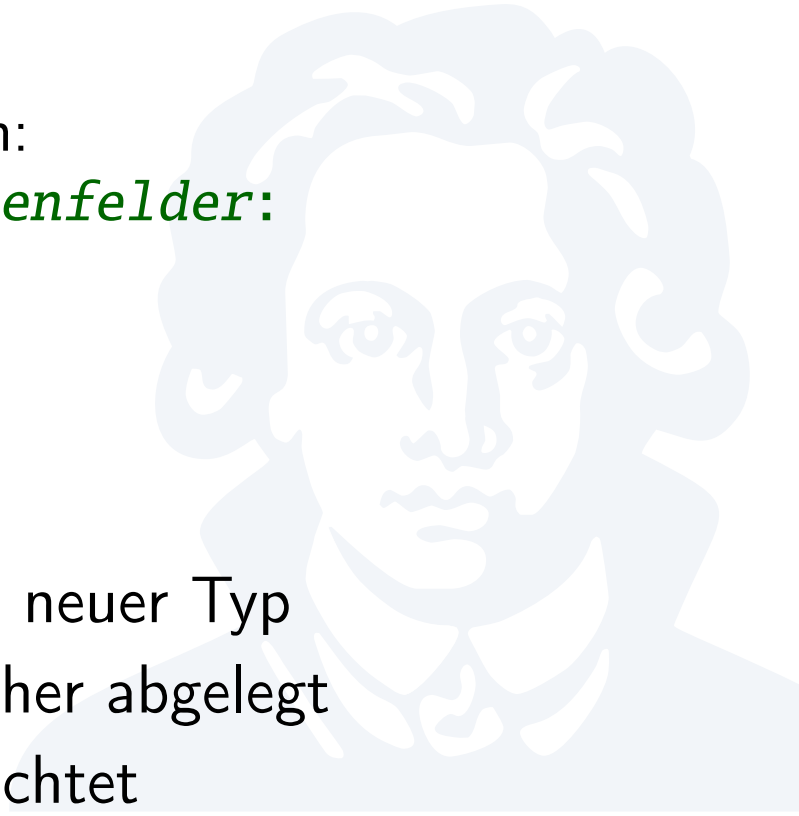
Strukturen

- Daten werden oft gruppiert
- *Beispiel*: Kundendaten:
  - `char name[32]`
  - `int kundennr`
  - `int bonuspunkte`

Wir können einen neuen Typen deklarieren:

```
struct kundenkonto { // Drei Datenfelder:
 char name[32];
 int kundennr;
 int bonuspunkte;
};
```

- `struct kundenkonto` ist nun ein neuer Typ
- Daten werden konsekutiv im Speicher abgelegt
- Felder werden automatisch ausgerichtet



Zugriff auf Strukturen

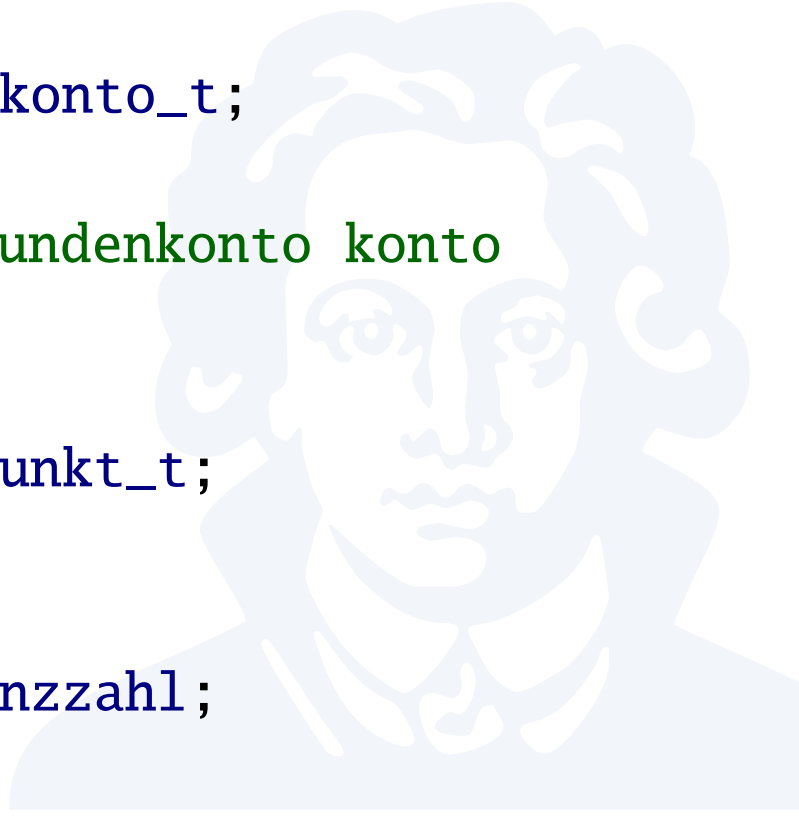
```
main() {
  struct kundenkonto konto = // Initialisierung:
      { .name="Hans", .kundennr=23, .bonuspunkte=0 };

  printf("Kunde %s hat Nummer %d\n",
         konto.name, konto.kundennr);
  konto.bonuspunkte += 1;
}
```

- Initialisierung (seit C99):
  `= { .feldname1 = wert1, .feldname2 = wert2 }`
- Werte (z.B. für Zuweisungen):
  `konto = (struct kundenkonto){ .name="Peter", .kundennr=17, .bonuspunkte=-1 }`
- Feldzugriff: `struktur.feldname // ist ein lvalue`

Typdefinitionen mit `typedef`

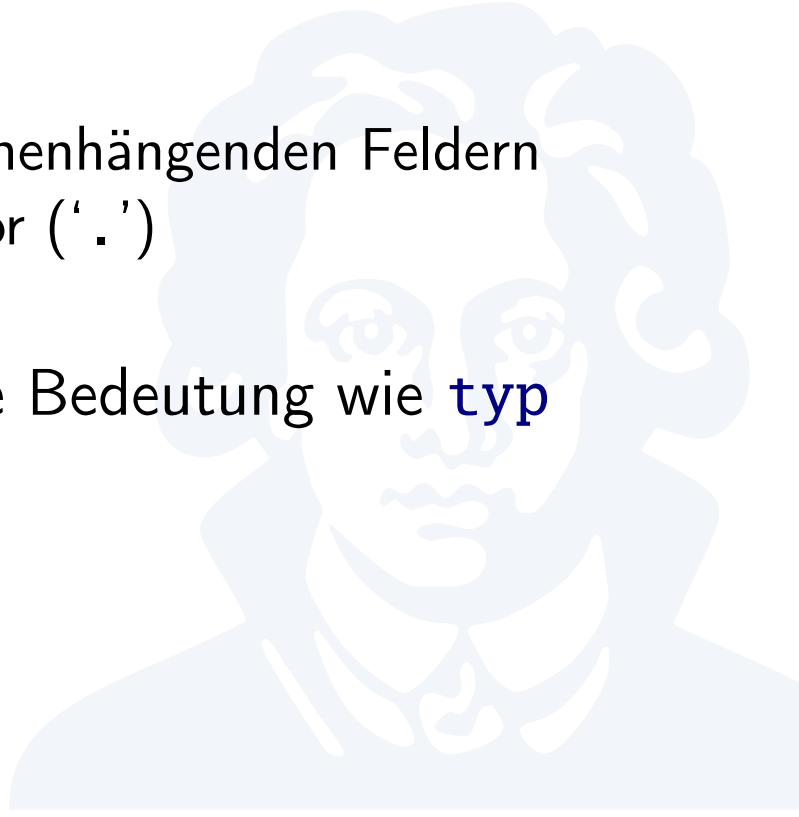
typedef erlaubt die Definition neuer Typen:

```
struct kundenkonto {
  char name[32];
  int kundennr;
  int bonuspunkte; };
typedef struct kundenkonto kundenkonto_t;

kundenkonto_t konto;  // struct kundenkonto konto

// Oder kurz:
typedef struct { int x, y, z; } punkt_t;

// Auch mit nicht-struct:
typedef unsigned long long int ganzzahl;
ganzzahl z = 1000ull;
```



Zusammenfassung: Datenstrukturen und Typdefinitionen

- **`struct`** definiert Datenstrukturen
  - Gruppierung von logisch zusammenhängenden Feldern
  - Feldzugriff durch Punkt-Operator (‘`.`’)
- **`typedef`** `typ name`:
  Definiert neuen Typnamen, gleiche Bedeutung wie `typ`



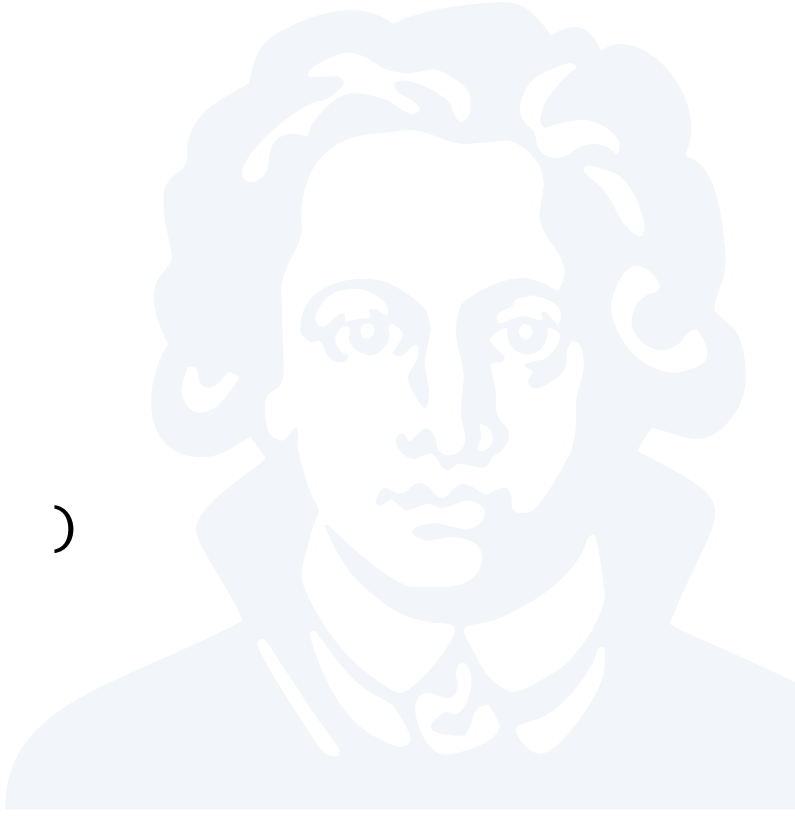
Funktionen

- Funktionaltät in C wird organisiert durch *Funktionen*
  - Äquivalent zu Subroutinen in Assembler
- Beispiele: `printf`, `main`
- Funktionen können:
  - *definiert* werden
  - *deklariert* werden
  - *aufgerufen* werden

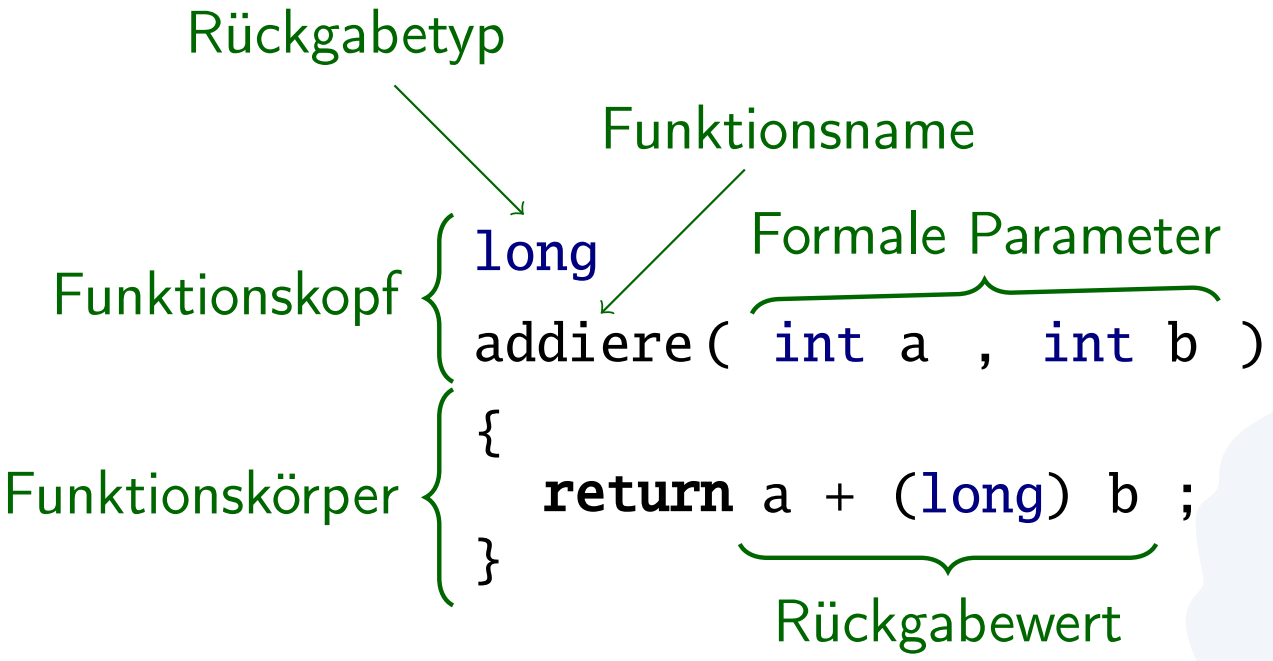
Beispiel für Aufruf:

```
printf ( "foo: %d", 10 )
```

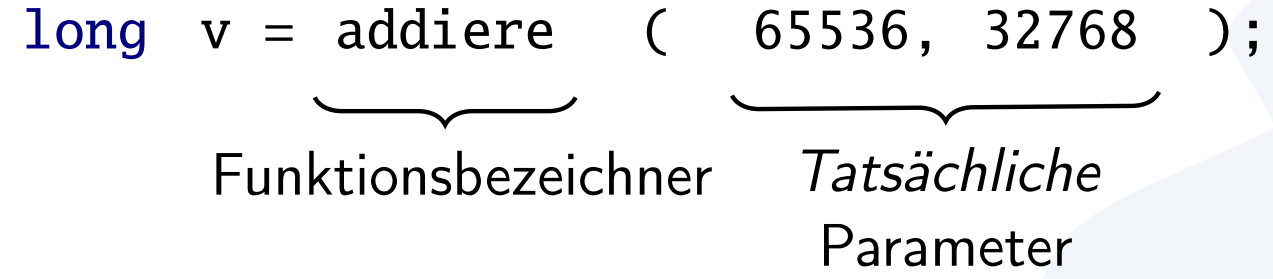
Funktionsbezeichner — Parameter



Funktionsdefinitionen



Ein Aufruf von `addiere` ist ein *Ausdruck* mit dem Typ `long`:



Funktionsdeklarationen

Im Sprachgebrauch der Programmiersprachen:

- *Deklaration*: Konzept wird angekündigt und beschrieben
- *Definition*: Konzept wird *vollständig* und unabänderbar beschrieben
- Jede Definitionen ist auch eine Deklaration

```
//Deklaration ohne Definition:
int ungerade(unsigned);

int gerade(unsigned x) {
  if (x == 0)
    return 1;
  else
    return ungerade(x - 1);
}
```

```
//Deklaration ohne Definition:
int gerade(unsigned);

int ungerade(unsigned x) {
  if (x == 0)
    return 0;
  else
    return gerade(x - 1);
}
```

Iteration vs. Rekursion

Rekursion:

```
int ungerade(unsigned);
int gerade(unsigned x) {
  if (x == 0)
    return 1;
  else
    return ungerade(x - 1);
}
int ungerade(unsigned x) {
  if (x == 0)
    return 0;
  else
    return gerade(x - 1);
}
```

Iteration:

```
int gerade(unsigned x) {
  while (x > 0) {
    if (x == 1)
      return 0;  // ungerade
    -x;
  }
  return 1; x == 0
}

int ungerade(unsigned x) {
  return !gerade(x);
}
```

Nur zur Illustration: dieses Problem ist auch in $O(1)$ lösbar!

Übersetzer für C generieren meist effizienteren Code für Iteration

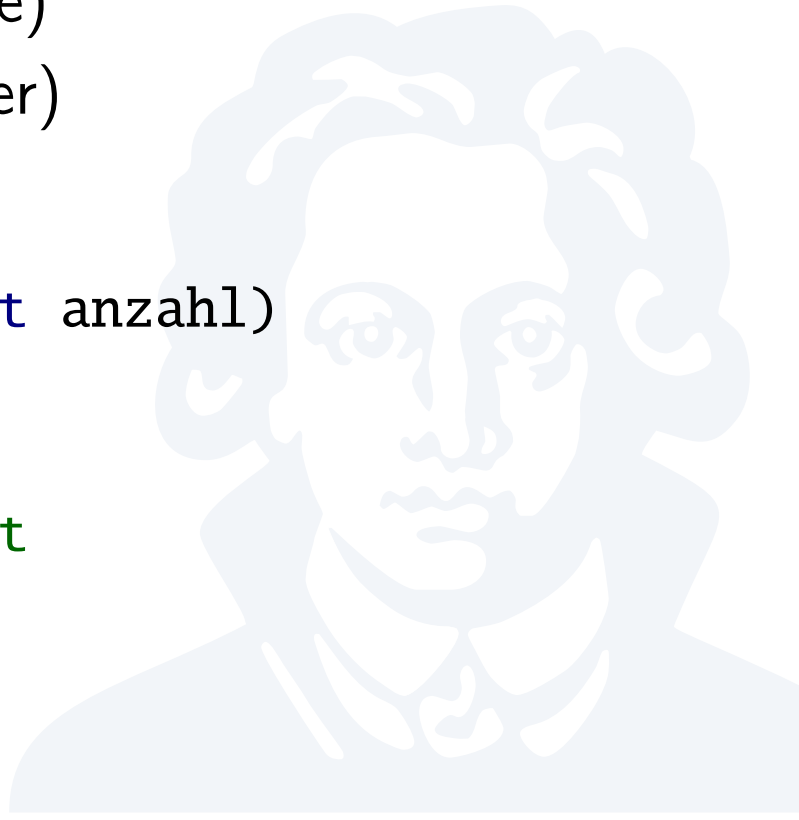
Der Typ `void`

- Manche Funktionen geben keine Werte zurück
- Beispiel: `exit(int)` beendet das Programm

`void`: Typ ohne Werte

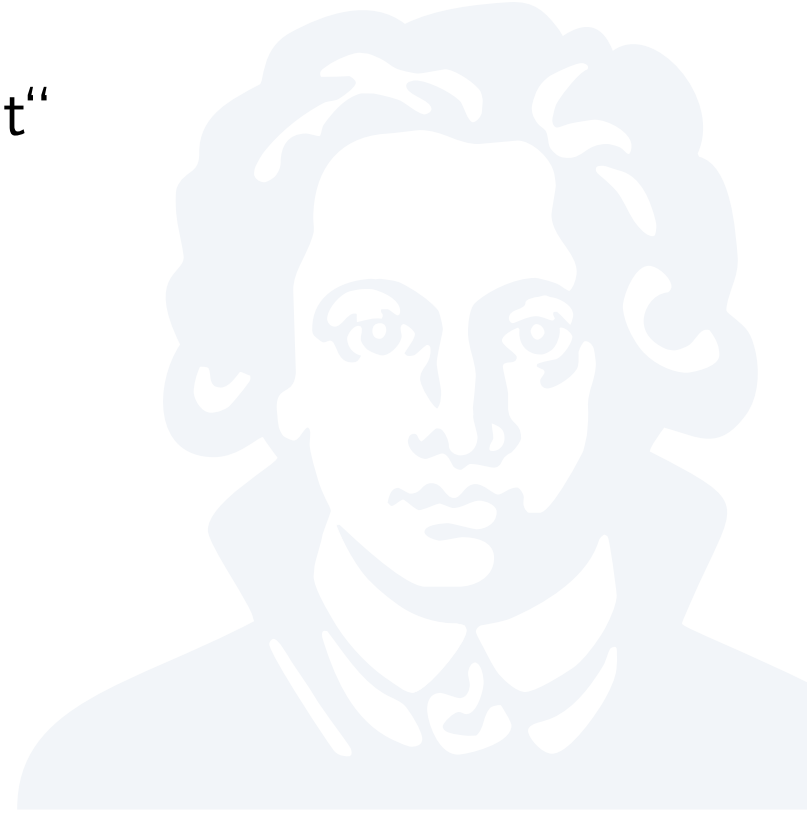
- `void exit(int)` (keine Rückgabe)
- `int rand(void)` (keine Parameter)

```
void
kopiere(char *nach, char *von, int anzahl)
{
  if (anzahl < 0)
    return;  // ohne Rueckgabewert
  while (--anzahl)
    *nach++ = *von++;
}
```



Funktionsdeklarationen mit `void`-Parameter

- `int rand();`
  Deklaration: „Parameter unbekannt“
- `int rand(void);`
  Deklaration: „keine Parameter“

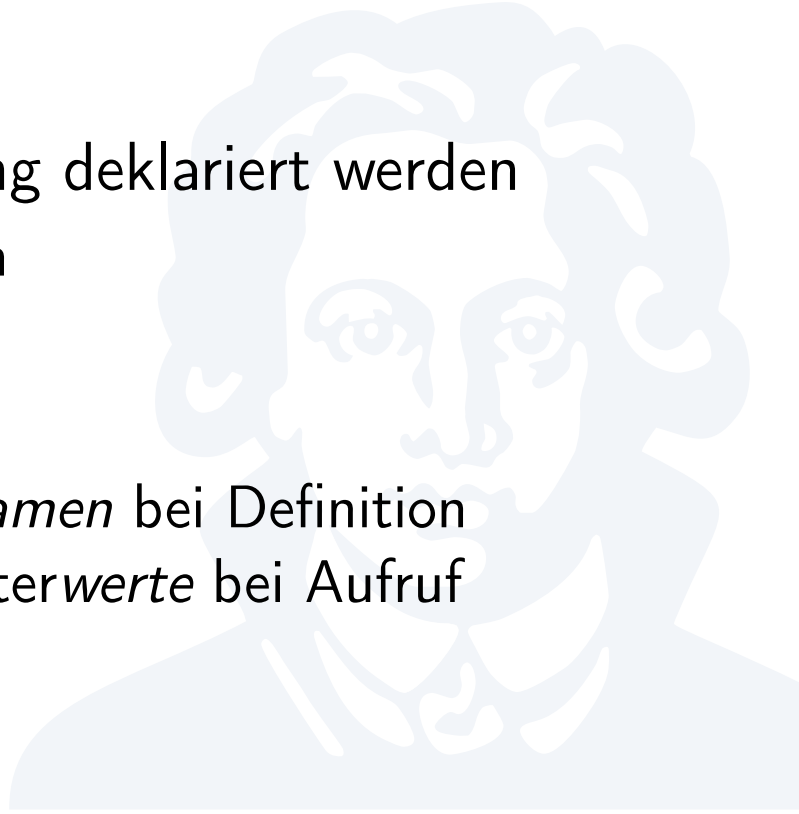


Die `main`-Funktion

- C erlaubt zwei Definitionen der `main`-Funktion:
  1. `int main(void)`
  2. `int main(int argc, char *argv[])`
- Rückgabewert: Programmstatus (Konvention)
  - `0`: Erfolgreiche Ausführung
  - `1`: Ausführung fehlgeschlagen
  - Andere Werte programmspezifisch
  - Werden von Kommandozeileninterpretern verwendet
- `argc` und `argv` kodieren Kommandozeilenparameter
  - `argv[0]`: Programmname
  - `argv[`i`]`: $i$ter Kommandozeilenparameter, $i <$ `argc`
  - `argv[argc]`: **`NULL`**

Zusammenfassung: Funktionen

- Funktionen erlauben:
  - Wiederverwertung von Funktionalität
  - Strukturierung des Programms
  - Rekursion
- Funktionen müssen *vor* Verwendung deklariert werden
  - Definition beinhaltet Deklaration
  - Separate Deklaration möglich
- Parameter:
  - *formale* Parameter: Parameter*namen* bei Definition
  - *tatsächliche* Parameter: Parameter*werte* bei Aufruf



Nächste Woche:

Das Laufzeitsystem der Sprache C