SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++ Programmierung

Grundlagen: Einführung und Ausdrücke

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Hello, world: K&R C


```
main()
{
  printf("hello, world\n");
}
```

- Aus: „The C Programming Language“
- K&R: Brian W. Kernighan und Dennis M. Ritchie

Hello, world: K&R C


```
main()
{
  printf("hello, world\n");
}
```

- Aus: „The C Programming Language“
- K&R: Brian W. Kernighan und Dennis M. Ritchie
- Heute kein gültiges C mehr

Hello, world: ANSI C89

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>

main()
{
  printf("hello, world\n");
}
```

- Aus: „The C Programming Language (Second Edition)“

Hello, world: ANSI C89

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>

main()
{
  printf("hello, world\n");
}
```

- Aus: „The C Programming Language (Second Edition)“
- Immernoch kein gültiges C

Hello, world: ISO C99

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

- Offiziell: ISO/IEC 9899:1999 (E)

Hello, world: Erweitert

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

Hello, world: Erweitert


```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven

Hello, world: Erweitert


```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen

Hello, world: Erweitert


```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen
- Kommentare

Hello, world: Erweitert

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen
- Kommentare
- Anweisungen

Hello, world: Erweitert

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen
- Kommentare
- Anweisungen
- Ausdrücke

Hello, world: Erweitert

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen
- Kommentare
- Anweisungen
- Ausdrücke
- Literale

Hello, world: Erweitert

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
#include <stdio.h>
int main(int argc, char** argv) {
  /* Greet the world or the user, if argument given.*/
  if (argc > 1) {
    char const* name = argv[1];
    printf("hello, %s\n", name);
  } else {
    printf("hello, world\n");
  }
  return 0;
}
```

- Präprozessordirektiven
- Funktions- und Variablendeklarationen
- Kommentare
- Anweisungen
- Ausdrücke
- Literale
- Kommunikation mit der Umgebung

Programme übersetzen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Das Programm übersetzen:

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

oder für Fortgeschrittene: (später)

```
%make hello
```

und starten:

```
%./hello
hello, world
%./hello Christoph
hello, Christoph
```

Kommentare

§6.4.9

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
/* Dies ist ein mehrzeiliger Kommentar. Er beginnt
 * bei /* und endet bei */
// Einzeilige Kommentare enden mit dem Zeilenende
```

- Mehrzeilige Kommentare nicht schachtelbar, d.h. zweites `/*` oben wird ignoriert
- Einzeilige Kommentare ab C99 oder C++

Deklarationen

§6.7

- Vereinbaren Variablen und Funktionen
- Jede Variable/Funktion hat einen Typ
- später mehr: (Funktions-)Zeiger, (mehrdimensionale) Reihungen

Deklarationen

§6.7

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Vereinbaren Variablen und Funktionen
- Jede Variable/Funktion hat einen Typ
- später mehr: (Funktions-)Zeiger, (mehrdimensionale) Reihungen

```
char (*(*x())[])()
```
[1]

[1] Aus der „Ode to C“

Typspezifikationssymbole

§6.7.2

```
type-specifier: one of
  void
  char     short   int   long
  float    double
  signed   unsigned
  _Bool
  _Complex   _Imaginary
  struct-or-union-specifier  enum-specifier  typedef-name
```

- Ganzzahltypen: `char`, `short`, `int`, `long`, `long long` (C99)
- Ganzzahltypen sind normalerweise vorzeichenbehaftet (außer `char`, dazu später) → mit `signed`/`unsigned` kombinierbar
- Gleitkommatypen: `float`, `double`, `long double`
- Komplexe Zahlen: `_Complex`, `_Imaginary` sind Modifikatoren für Gleitkommatypen
- Boolescher Typ: `_Bool` (C99)
- Typgrößen und Darstellung ist implementierungsabhängig, Norm gibt nur Mindestgrößen vor → später mehr

Beispiele für Deklarationen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int          i; // Vorzeichenbehafteter int
signed int   j; // Dasselbe wie oben
unsigned     k; /* Vorzeichenlos,
                 * dasselbe wie "unsigned int" */
unsigned long long int l;

int a = 23;     // Mit Initialisierung
int b, c = 42; /* Mehrere Namen in einer Deklaration,
                * nur c ist mit 42 initialisiert */

/* Eine Funktion, die keine Parameter hat und nichts
 * zurueckgibt */
void f(void);
/* Eine Funktion, die einen double und einen int als
 * Parameter hat und einen double zurueckgibt */
double g(double, int);

double const pi = 3.14159; // Eine Konstante (spaeter)
char* p;                   // Zeiger auf char (spaeter)
```

Ausdrücke[2] (Expressions)

§6.5

Operator	Associativity
() [] -> .	left to right
! ~ ++ -- + - (type) * & sizeof new delete	right to left
->* .*	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= <<= >>= &= ^= |= throw	right to left
?: (C++, third operand)	right to left
,	left to right

[2] Aus: `man operator`

Ausdrücke (Expressions)

§6.5

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Jeder Ausdruck hat einen Ergebnistyp
  $\longrightarrow$ Wird meist aus (beiden) Operanden berechnet
- Operanden werden (fast immer) in mindestens `int` umgewandelt
- Auswertungsreihenfolge großteils unspezifiziert
  $\longrightarrow$ außer bei Funktionsaufruf (teilweise), `&&`, `||`, `?:` und `,`
- Können Seiteneffekte beinhalten
  Achtung: Ein Ausdruck darf nicht mehrfach das selbe Objekt verändern

Primärausdrücke

§6.5.1

```
primary-expression:
  identifier
  constant
  string-literal
  ( expression )
```

- Bezeichner: `i`, `name`, `printf`
- Konstanten: `23`, `3.14`, `'A'`
- Zeichenkettenliterale: `"Hello, world"`
- Geklammerter Ausdruck, z.B. `(a + b) * c`

Bezeichner

§6.4.2

```
identifier:
  [A-Za-z_][A-Za-z_0-9]*
```

- Bezeichner beginnen mit Buchstaben oder Unterstrich
- Danach beliebig Buchstaben, Unterstriche und Ziffern
- Nur Buchstaben A bis Z (oder klein), keine Umlaute etc.
- Es wird zwischen Groß- und Kleinschreibung, wie überall in C, unterschieden: `bla` ist ein anderer Bezeichner als `BLA`
- Keine Schlüsselwörter wie z.B. `if`
- Reservierte Namen vermeiden, z.B. `printf`
  Generell vermeiden: `_[A-Z_].*`
- Implementierungen dürfen weitere Zeichen in Bezeichnern akzeptieren, z.B. GCC erlaubt $

Konstanten

§6.4.4

```
constant:
  integer-constant
  floating-constant
  enumeration-constant
  character-constant
```

- Ganzzahlige Konstanten
- Gleitkommazahlen
- Aufzählungskonstanten (später)
- Zeichenkonstanten

Zahlenkonstanten: Dezimal

§6.4.4.1

```
integer-constant:
  [1-9][0-9]* integer-suffix?
```

- Basis 10
- Typ abhängig vom Betrag der Zahl, mindestens `int`, normalerweise immer `signed`
- Suffixe: `integer-suffix`
  - `U` (oder `u`): `unsigned`
  - `L` (oder `l`): mindestens `long`
  - `LL` (oder `ll`): mindestens `long long` (C99)
  - `U` mit `L`/`LL` kombinierbar
  - Beispiele: `23`, `23U`, `23L`, `23UL`, `23llU`
- Achtung: Unerwarteter Typ kann zu subtilen Fehlern führen
- Auffälligkeit: es gibt keine negativen Zahlenkonstanten

Zahlenkonstanten: Dezimal

§6.4.4.1

```
integer-constant:
  [1-9][0-9]* integer-suffix?
```

- Basis 10
- Typ abhängig vom Betrag der Zahl, mindestens `int`, normalerweise immer `signed`
- Suffixe: `integer-suffix`
  - `U` (oder `u`): `unsigned`
  - `L` (oder `l`): mindestens `long`
  - `LL` (oder `ll`): mindestens `long long` (C99)
  - `U` mit `L`/`LL` kombinierbar
  - Beispiele: `23`, `23U`, `23L`, `23UL`, `23llU`
- Achtung: Unerwarteter Typ kann zu subtilen Fehlern führen
- Auffälligkeit: es gibt keine negativen Zahlenkonstanten
- Auffälligkeit: 0 ist nicht darstellbar!

Zahlenkonstanten: Oktal

§6.4.4.1

```
integer-constant:
  0[0-7]* integer-suffix?
```

- Basis 8, gekennzeichnet durch führende `0`
- Typ abhängig vom Betrag der Zahl, `signed` oder `unsigned`
- Suffixe wie bei dezimal
- Früher häufiger verwendet, einige Systeme verwendeten 12 Bit, 18 Bit oder 36 Bit breite Wörter
- Heute fast nur noch für UNIX-Rechte (`-rwxr-xr-x` $\longrightarrow$ `0755`)

Zahlenkonstanten: Oktal

§6.4.4.1

```
integer-constant:
  0[0-7]* integer-suffix?
```

- Basis 8, gekennzeichnet durch führende `0`
- Typ abhängig vom Betrag der Zahl, `signed` oder `unsigned`
- Suffixe wie bei dezimal
- Früher häufiger verwendet, einige Systeme verwendeten 12 Bit, 18 Bit oder 36 Bit breite Wörter
- Heute fast nur noch für UNIX-Rechte (`-rwxr-xr-x` $\longrightarrow$ `0755`)
- Kuriosum: 0 ist eine Oktalkonstante

Zahlenkonstanten: Hexadezimal

§6.4.4.1

How many people can read hex if only you and dead people can read hex?

```
integer-constant:
  0[xX][0-9A-Fa-f]+ integer-suffix?
```

- Basis 16, gekennzeichnet durch vorangestelltes `0x` (oder `0X`)
- Typ und Suffixe wie bei oktal
- Heute häufiger als oktal, da Bytes üblicherweise 8 Bit breit
- Oft anzutreffen bei Bitgefummel

Zahlenkonstanten: Hexadezimal

§6.4.4.1

How many people can read hex if only you and dead people can read hex?

```
integer-constant:
  0[xX][0-9A-Fa-f]+ integer-suffix?
```

- Basis 16, gekennzeichnet durch vorangestelltes `0x` (oder `0X`)
- Typ und Suffixe wie bei oktal
- Heute häufiger als oktal, da Bytes üblicherweise 8 Bit breit
- Oft anzutreffen bei Bitgefummel
- Antwort: 57006

Gleitkommazahlen

§6.4.4.2

```
floating-constant:
  decimal-floating-constant
  hexadecimal-floating-constant

decimal-floating-constant:
  fractional-constant exponent-part? [FfLl]?

fractional-constant:
  [0-9]* . [0-9]+
  [0-9]+ .

exponent-part:
  [Ee] [+-]? [0-9]+
```

- Beispiel: `1.337e+3` ist $1{,}337 * 10^3$, also 1337
- Weitere Beispiele: `.5`, `1.e-0f`
- Typ abhängig vom Suffix, ohne: `double`, `f` oder `F`: `float`, `l` oder `L`: `long double`
- Auch zur Basis 16 möglich (hier nicht)

Zahlenkonstanten: Zeichenkonstante

§6.4.4.4

```
character-constant:
  ' c-char '
  ' \ [0-7] [0-7]? [0-7]? '
  ' \ x [0-9A-Fa-f]+ '
  ' simple-escape-sequence '

simple-escape-sequence: one of
  \' \" \? \\ \a \b \f \n \r \t \v
```

- `c-char`: Jedes Zeichen außer `'`, `\` oder Zeilenumbruch
- Entweder einfaches Zeichen: `'A'`
- Achtung: Wert ist implementierungsabhängig
- Oder bis zu dreiziffrige Oktalsequenz: `'\101'`
- Oder Hexadezimalsequenz: `'\x41'`
- Bei ASCII haben alle drei Beispiele den selben Wert
- Typ: `int` (in C), `char` (in C++)
- Werden wie normale Zahlen gehandhabt: `'0' + 3 == '3'`
- Häufig verwendete Oktalsequenz ist das NUL-Zeichen: `'\0'`

Zeichenkonstanten: Funktionssequenzen

§6.4.4.4

```
'\'' // Einfaches Anfuehrungszeichen
'\"' // Doppeltes Anfuehrungszeichen
'\?' // Fragezeichen
'\\' // Umgekehrter Schraegstrich (backslash)
'\a' // Piepton (alert)
'\b' // Rueckschritt (backspace)
'\f' // Seitenvorschib (form feed)
'\n' // Zeilenumbruch (newline)
'\r' // Wagenruecklauf (carriage return)
'\t' // Horizontaler Tabulator
'\v' // Vertikaler Tabulator
```

- Werte sind implementierungsabhängig
  z.B. ASCII: `'\t'` ist 9, `'\n'` ist 10
- Jedes andere Zeichen nach einem `\` ist ein Fehler

Zeichenkettenliterale

§6.4.5

```
string-literal:
  " s-char* "
```

- `s-char`: Funktionssequenz oder beliebiges Zeichen außer `"`, `\` und Zeilenumbruch
- Sind einfach eine Reihung von einzelnen Zeichen
- Also gleiche Regeln wie bei Zeichenkonstanten, nur mehrere Zeichen hintereinander und `"` statt `'`
- Zeichenkettenliteral wird automatisch am Ende mit einem NUL-Zeichen (`'\0'`) ergänzt
  → Wichtig für Verarbeitung von Zeichenketten in C (später)
- Beispiel: `"hello, world\n"`

Postfixoperatoren

§6.5.2

```
postfix-expression:
  primary-expression
  postfix-expression [ expression ]
  postfix-expression ( argument-expression-list? )
  postfix-expression . identifier
  postfix-expression -> identifier
  postfix-expression ++
  postfix-expression --
  ( type-name ) { initializer-list ,? }

argument-expression-list:
  assignment-expression
  argument-expression-list , assignment-expression
```

- Zugriff auf Elemente von Reihungen mittels `[]` (später)
- Funktionsaufrufe, auch Aufrufe über Funktionszeiger
- Zugriff auf Elemente von Strukturen mittels . und -> (später)
- Nachgestelltes ++/-- erhöht/verringert Variable um 1 und liefert vorherigen Wert
- Compound-literal (hier nicht)

Unäre Operatoren

§6.5.3

```
unary-expression:
  postfix-expression
  ++ unary-expression
  -- unary-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-name )

unary-operator: one of
  & * + - ~ !
```

- Vorangestelltes `++`/`--` erhöht/verringert Variable um 1 und liefert neuen Wert
- `&` und `*` für Operation mit Zeigern (später)
- `+` tut (fast) nichts, `-` liefert negierten Wert des Operanden
- Bitweise Negation: `~` liefert Einerkomplement des Operanden
- Logische Negation: `!` vergleicht Operand mit `0`, gibt `1` zurück falls gleich `0`, sonst `0`
- `sizeof` liefert Größe eines Ausdrucks/Typs in Byte (später)

Explizite Typumwandlung

§6.5.4

```
cast-expression:
  unary-expression
  ( type-name ) cast-expression
```

- Wandelt Operand in den gewünschten Typ um
- Umwandlung von Gleitkommazahlen in Ganzzahlen schneidet Nachkommastellen ab
  $\rightarrow$ Wenn Ergebnis nicht darstellbar, dann undefiniert
- Umwandlungen zwischen Zeigertypen möglich (später)
- C++ bietet weitere, sicherere Typumwandlungen (später)

Multiplikate Ausdrücke

§6.5.5

```
multiplicative-expression:
  cast-expression
  multiplicative-expression * cast-expression
  multiplicative-expression / cast-expression
  multiplicative-expression % cast-expression
```

- `*` multipliziert die Operanden
- `/` teilt linken durch rechten Operanden
- `%` liefert Divisionsrest
- Division durch `0` ist undefiniert
- Achtung: Ergebnistyp bestimmt durch Typ der Operanden:
  `5 / 2` ist `2`, nicht `2.5`

Additive Ausdrücke

§6.5.6

```
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression
```

- Addiert/subtrahiert Operanden
- Auch für Arithmetik mit Zeigern (später)

Bitschiebeoperationen

§6.5.7

```
shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
```

- `a << b` schiebt Bits von `a` um `b` Positionen nach links
- Ergebnistyp ist Typ des linken Operanden (mindestens `int`)
- `b` muss mindestens `0` und weniger als Breite von `a` sein, sonst Verhalten undefiniert
- Wenn Typ `unsigned`, dann Ergebnis wie $a * 2^b$ (bzw. $a/2^b$ bei >>)
- Wenn `signed`, Wert nicht negativ und Ergebnis darstellbar, dann wie oben
- Sonst bei >> implementierungsabhängig und << undefiniert

Relationale Operatoren

§6.5.8

```
relational-expression:
  shift-expression
  relational-expression <  shift-expression
  relational-expression >  shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
```

- Vergleicht Operaden, Ergebnis ist ein `int`
- `1` wenn wahr, `0` sonst
- Auch für Vergleiche von Zeigern

Gleichheitsoperatoren

§6.5.9

```
equality-expression:
  relational-expression
  equality-expression == relational-expression
  equality-expression != relational-expression
```

- `==` testet auf Gleichheit, `!=` testet auf Ungleichheit
- Ergebnis wiederrum Typ `int`, `1` wenn wahr, `0` sonst
- Auch für Vergleiche von Zeigern
- Falle: `==` nicht mit `=` (Zuweisung) verwechseln
- Falle: Haben niedrigere Präzedenz als relationale Operatoren:
  `a < b == c > d` verhält sich wie `(a < b) == (c > d)`

Bitweise Operatoren

§6.5.10-12

```
AND-expression:
  equality-expression
  AND-expression & equality-expression

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression ^ AND-expression

inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression | inclusive-OR-expression
```

- Bitweises Und, exklusives Oder und inklusives Oder
- Falle: Alle drei haben verschiedene Präzedenzen
- Falle: Haben niedrigere Präzedenz als Vergleiche
  `if (x & 4 != 0)` wird interpretiert als `if (x & (4 != 0))`

Logische Operatoren

§6.5.13+14

```
logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression && inclusive-OR-expression

logical-OR-expression:
  logical-AND-expression
  logical-OR-expression || logical-AND-expression
```

- Ergebnis ist `1`, wenn beide (`&&`)/mindestens ein (`||`) Operand `!= 0`, `0` sonst
- Kurzauswertung: Zuerst linken Operanden auswerten; steht das Gesamtergebnis dann schon fest, wird der rechte Operand nicht ausgewertet $\longrightarrow$ wichtig bei Seiteneffekten
- Falle: `&&` hat höhere Präzedenz als `||`
  `if (a || b && c)` äquivalent zu `if (a || (b && c))`

Ternärer Operator

§6.5.15

```
conditional-expression:
  logical-OR-expression
  logical-OR-expression ? expression :     (C)
    conditional-expression
  logical-OR-expression ? expression :     (C++)
    assignment-expression
```

- Auswahloperator, ternär → dreistellig
- `a ? b : c`: Wenn `a != 0`, dann ist Ergebnis `b`, sonst `c`
- Ergebnistyp wird nur aus Typen von `b` und `c` bestimmt
- Es wird zuerst `a`, danach entweder `b` oder `c` ausgewertet
- Grammatik in C und C++ leicht verschieden

```
a ? b = c : d = e
(a ? (b = c) : d) = e // Bedeutung in C, Syntaxfehler
a ? (b = c) : (d = e) // Bedeutung in C++
```

Zuweisungen

§6.5.16

```
assignment-expression:
  conditional-expression
  unary-expression op assignment-expression       (C)
  logical-OR-expression op assignment-expression (C++)

op: one of
  = *= /= %= += -= <<= >>= &= ^= |=
```

- `=` ist einfache Zuweisung
- `op=` ist Operation kombiniert mit Zuweisung
- `a op= b` fast äquivalent zu `a = a op (b)`
  `a` wird bei `op=` nur einmal ausgewertet, z.B.: `data[i++] += 1`
- Typ/Wert des Ausdrucks ist Typ/neuer Wert der linken Seite
- Grammatik erlaubt Mehrfachzuweisung und ist rechtsassoziativ: `x = y = z += 1` $\longrightarrow$ `x = (y = (z += 1))`
- Syntax leicht anders in C++, bietet mehr Möglichkeiten bei Operatorüberladung

Kommaoperator

§6.5.17

```
expression:
  assignment-expression
  expression , assignment-expression
```

- Zuerst wird zuerst linker Operand ausgewertet, dann rechter
- Wert des Gesamtausdrucks ist Wert des rechten Operanden
- Häufig verwendet in Schleifen und Präprozessormakros (später)
- Achtung: Nicht verwechseln mit Komma, das Funktionsargumente trennt:
  - `f(1, 2)`: kein Kommaoperator, Funktionsaufruf mit zwei Argumenten, Auswertungsreihenfolge unspezifiziert
  - `f((1, 2))`: Kommaoperator und Funktionsaufruf mit einem Argument