Operatoren und Ausdrücke

Einführung in die Programmierung

Michael Felderer

Institut für Informatik, Universität Innsbruck

Operatoren

- Über Operatoren erhält man die Möglichkeit, z.B. Variablen mit Inhalten zu belegen oder die in ihnen gespeicherten Werte zu verändern.
- Abhängig von der Anzahl der Operanden unterscheidet man:
  - Unäre Operatoren (besitzen einen Operanden)
  - Binäre Operatoren (besitzen zwei Operanden)
  - Ternäre Operatoren (besitzen drei Operanden)
- Abhängig von der Position unterscheidet man:
  - Präfix-Form: Operator steht vor seinem bzw. seinen Operanden.
  - Postfix-Form: Operator steht hinter seinem bzw. seinen Operanden.
  - Infix-Form: Operator steht zwischen seinen Operanden (binäre Operatoren)
- Abhängig von der Auswertungsreihenfolge unterscheidet man:
  - Linksassoziativ: Auswertung von links nach rechts.
  - Rechtsassoziativ: Auswertung von rechts nach links.

Zuweisungsoperator

- Der Zuweisungsoperator = ist ein binärer Operator.
  - Beispiele

```
int a;
a = 1;
int b = 3;
```

- Der Zuweisungsoperator arbeitet von rechts nach links (rechtsassoziativ) und sein Wert muss nicht konstant sein!
  - Beispiel

```
int a = 1;
int b = a;
```

  - Komplexeres Beispiel

```
int a, b;
a = b = 1;
```

- Die Zuweisungsoperation `x = x + 1;` macht auch Sinn!
  - Der Wert aus x wird um 1 erhöht, der neue Wert wird in x gespeichert.

Arithmetische Operatoren

- Arithmetische Operatoren werden für Berechnungen verwendet und sind linksassoziativ.
- Die wichtigsten arithmetischen Operatoren:

Name	Verwendung	Operandentyp	Operator liefert
Minus (unär)	- Op1	ganzzahlig, Gleitpunkttyp	Vorzeichenwechsel
Plus	Op1 + Op2	ganzzahlig, Gleitpunkttyp	Summe
Minus (binär)	Op1 - Op2	ganzzahlig, Gleitpunkttyp	Differenz
Multiplikation	Op1 * Op2	ganzzahlig, Gleitpunkttyp	Produkt
Division	Op1 / Op2	ganzzahlig, Gleitpunkttyp	Ganzzahlige Division bei ganzen Zahlen (Nachkommaanteil wird abgeschnitten), Quotient
Modulo	Op1 % Op2	ganzzahlig	Rest bei ganzzahliger Division

Beispiel (Arithmetische Operatoren)

```c
#include <stdio.h>
#include <stdlib.h>
static const double PI = 3.141592654;

int main(void) {
    int a = 5, b = 2, c = 7;
    float x = 4.1f, y = 3.5f, z = 2.0f;
    float sum = x + y + z;
    printf("%d %d\n", a, b);
    printf("%f\n", sum);
    printf("%f\n", sum + 2.1);
    printf("%d\n", a + b + c);
    printf("%d\n", a / b);
    printf("%f\n", x / z);
    printf("%d\n", c % a);
    printf("%f\n", a * PI);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
5 2
9.600000
11.700000
14
2
2.050000
2
15.707963
```

Erweiterte Darstellung arithmetischer Operatoren

- Die arithmetischen Operatoren lassen sich in einer kürzeren Schreibweise verwenden:

Operation	**Bezeichnung**	**entspricht**
Op1 += Op2	Additionszuweisung	Op1 = Op1 + Op2
Op1 -= Op2	Subtraktionszuweisung	Op1 = Op1 - Op2
Op1 *= Op2	Multiplikationszuweisung	Op1 = Op1 * Op2
Op1 /= Op2	Divisionszuweisung	Op1 = Op1 / Op2
Op1 %= Op2	Modulozuweisung	Op1 = Op1 % Op2

Interaktive Aufgabe

- Was gibt das folgende C-Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int a = 5, b = 7, c = 8, d=6, e=4;

  a+=b;
  printf("%d %d\n", a, b);
  b-=(c%b);
  printf("%d %d\n", b, c);
  c*=(d+e);
  printf("%d %d %d\n", c, d, e);
  d/=e;
  printf("%d %d\n", d, e);
  e%=e;
  printf("%d\n", e);

  return EXIT_SUCCESS;
}
```

Overflow

- Was passiert, wenn nach einer arithmetischen Operation bei Ganzzahlen das Ergebnis nicht in den Wertebereich passt?
  - Es tritt ein Überlauf ein.
    - Meist undefiniertes Verhalten bei Zahlen mit Vorzeichen (abhängig von der Umgebung)!
    - Beginnt wieder bei 0 bei Zahlen ohne Vorzeichen.
    - Sollte immer vermieden werden!
- Beispiel (short mit 2 Bytes, zufällig gleiches Verhalten wie bei unsigned)

```
short a = 30000, b = 30000, c;
c = a + b;
printf("%d\n", c);
c += b;
printf("%d\n", c);
c += b;
printf("%d\n", c);
```

Ausgabe:
```
-5536
24464
-11072
```

Inkrement- und Dekrementoperatoren (1)

- Mit dem Inkrementoperator (++) bzw. dem Dekrementoperator (--) wird der Wert einer Variable um 1 erhöht bzw. erniedrigt.
  - `i++` entspricht `i += 1` entspricht `i = i + 1.`
- Kann als Prä- oder Postfix-Operator verwendet werden:

Operator	Benennung	Beispiel	Erklärung
++	Präinkrement	++a	a wird vor seiner weiteren Verwendung um 1 erhöht
++	Postinkrement	a++	a wird nach seiner weiteren Verwendung um 1 erhöht
--	Prädekrement	--b	b wird vor seiner weiteren Verwendung um 1 erniedrigt
--	Postdekrement	b--	b wird nach seiner weiteren Verwendung um 1 erniedrigt

Inkrement- und Dekrementoperatoren (2)

- Beispiel

```
int a, b, c;
a = 3;
b = ++a * 3;
c = a++ * 3;
```

Nach der letzten Zeile sind:

```
a = 5
b = 12
c = 12
```

- Erklärung
  - b bekommt den Wert von a um 1 erhöht und multipliziert mit 3 (= 4 * 3) zugewiesen.
  - Nach der Zuweisung hat a für den nachfolgenden Befehl den Wert 4.
  - c bekommt daher den Wert von a multipliziert mit 3 (= 4 * 3) zugewiesen.
    - Das Inkrement wird erst danach ausgeführt, d.h. erst für den nächsten Befehl hat a den Wert 5!

Interaktive Aufgabe

- Was gibt das folgende C-Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main (void) {

   int i = 1;

   printf("%d \n", i--);
   printf("%d \n", ++i);
   printf("%d \n", i++ * 2);
   printf("%d \n", --i * 2);

   return EXIT_SUCCESS;
}
```

Unterschied zwischen Ausdruck und Anweisung

- Ausdruck (expression)
  - Ein Ausdruck ist eine Kombination von Operatoren und Operanden.
  - Ein Ausdruck kann damit Teil eines größeren Ausdrucks sein.
  - Zum Beispiel hat der Ausdruck 3 + 4 den Wert 7.
- Anweisung (statement)
  - Durch Anhängen eines Semikolons wird aus einem Ausdruck eine Anweisung (einfache Anweisung).
  - Anweisungen können keinen Wert haben.
    - Sie können nicht Teil eines größeren Ausdrucks sein.
  - Beispiel
    - `3 + 4;` möglich, aber nicht sinnvoll.
      - Sollte zugewiesen werden.
    - `i++;` möglich und sinnvoll, da ++ einen sogenannten Seiteneffekt hat (i wird um 1 erhöht)!

Vergleichsoperatoren

- ANSI-C (C89) hat keinen Datentyp zur Darstellung boolescher Werte.
  - Es wird der Wert 0 als **`false`** interpretiert und Werte ungleich 0 als **`true`**.
  - Vergleichende Operatoren haben den Wert 0, wenn die entsprechende Aussage falsch ist, andernfalls haben sie den Wert 1.
- Vergleichsoperatoren:

C - Notation	Math. Notation
`a < b`	a < b
`a > b`	a > b
`a <= b`	a ≤ b
`a >= b`	a ≥ b
`a == b`	a = b
`a != b`	a ≠ b

Logische Operatoren

- Mit Hilfe logischer Operatoren können Ausdrücke (arithmetische, vergleichende und zuweisende) als Aussagen miteinander verknüpft werden.
  - Der entstehende Ausdruck hat den Wert 1, wenn er als Aussage wahr ist, ansonsten hat er den Wert 0.
- Logische Operatoren
  - **a && b** = logisches Und
  - **a || b** = logisches Oder
  - **!a** = logische Negation
- Die Operatoren werden in C mittels **short circuit evaluation** ausgewertet.
  - Die Auswertung eines Ausdrucks bricht ab, sobald der endgültige Wert dieses Ausdrucks unabänderlich feststeht.
- Sowohl Vergleichsoperatoren als auch logische Operatoren sind linksassoziativ.

Beispiel (Logische Operatoren)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a = 1, b = 1, z = 0;
    float c = -0.5f;
    printf("%d\n", a);
    printf("%d\n", !a);
    printf("%d\n", !(a - b));
    printf("%d\n", !(a < b));
    printf("%d\n", b < (c < a));
    printf("%d\n", (a > c) && b);
    printf("%d\n", (a == 1) || (z = 5));
    printf("%d\n", z);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
1
0
1
1
0
1
1
0
```

Interaktive Aufgabe

- Was gibt das folgende C-Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int a = 1, b = 2, c = 0;
  double d = 1.5;

  printf("%d\n", b);
  printf("%d\n", !b);
  printf("%d\n", !(a - b + 1));
  printf("%d\n", b > (d > a));
  printf("%d\n", !(a <= b));
  printf("%d\n", (a > d) || c);
  printf("%d\n", (a == 1) && (c = 1));
  printf("%d\n", c);

  return EXIT_SUCCESS;
}
```

Bedingter Ausdruck

- In C gibt es nur einen ternären Operator, den sogenannten bedingten Ausdruck.
- Dieser rechtsassoziative Operator hat die Form:
  - `A ? B : C`
  - Der bedingte Ausdruck entspricht B, falls A als Aussage wahr ist, ansonsten entspricht er C.
- Beispiele

```
even = (a % 2 == 0) ? 1 : 0;
```

  - even erhält genau dann den Wert 1 wenn der Wert von a gerade ist.

```
maxab = (a > b) ? a : b;
```

  - `maxab` erhält das Maximum der Variablen a und b.

Auswertungsreihenfolge

- Wie in der Mathematik spielt es auch in C eine wichtige Rolle, in welcher Reihenfolge ein Ausdruck berechnet wird.
  - Beispiel: `5 + 2 * 3` ergibt 11 und nicht 21.
- Operatoren haben unterschiedliche **Prioritäten** (**Präzedenz**):
  - Ähnlich zu „Punktrechnung vor Strichrechnung“.
  - Regelt die implizite Klammerung bei Operatoren auf verschiedenen Stufen.
- **Assoziativität**
  - Regelt die implizite Klammerung bei Operatoren auf gleicher Stufe.

Operatoren (Priorität, Assoziativität)

Operator (höchste bis niedrigste Priorität)	Assoziativität
`(), [], {}, ->, ., ++, --`	links
`!, ~, ++, --, +, -, *, &, sizeof`	rechts
`(type)`	rechts
`*, /, %`	links
`+, -`	links
`<<, >>`	links
`<, <=, >, >=`	links
`==, !=`	links
`&`	links
`^`	links
`|`	links
`&&`	links
`||`	links
`?:`	rechts
`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=`	rechts
`,`	links

Reihenfolge der Auswertung

- **Achtung**: Die Reihenfolge der Verknüpfung hat nichts mit der Reihenfolge der Auswertung der Operanden zu tun!
- A * B + C * D wird von links nach rechts bearbeitet (linksassoziativ), aber der Compiler kann selbst bestimmen, welcher Operator für + (also A * B oder C * D) zuerst ausgewertet wird.
  - Man sollte sich beim Programmieren **nie** auf eine bestimmte Reihenfolge verlassen!
  - Beispiele folgen noch!

Bitoperationen

- Mit diesen Operatoren wird ein direkter Zugriff auf die binäre Darstellung von Ganzzahlen unterstützt.
- C besitzt vier logische Bit-Operatoren
  - **&** = UND-Operator für Bits
  - **|** = ODER-Operator für Bits
  - **^** = Exklusive-ODER-Operator für Bits (XOR)
  - **~** = Negationsoperator für Bits
- C kennt zwei Shift-Operatoren
  - **>>** Rechtsshift-Operator
    - Bits werden nach rechts verschoben.
    - Dabei gehen die niederwertigsten Bits verloren.
  - **<<** Linksshift-Operator
    - Bits werden nach links verschoben.
    - Dabei gehen die höherwertigen Bits verloren.
- Für Bitoperatoren gibt es die Zuweisungsoperatoren **&=**, **|=**, **^ =**, **<<=** und **>>=**

Beispiel (Bitoperatoren)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a = 14;
    short b = 32;
    short c = 47;
    printf("%d\n", a & 0); // 1110 & 0000 = 0000
    printf("%d\n", a | 1); // 1110 | 0001 = 1111
    printf("%d\n", a ^ 5); // 1110 ^ 0101 = 1011
    /* Negation der gesamten Zahl b, d.h.
     * ... 0000 0010 0000 = ... 1111 1101 1111 */
    printf("%hd\n", ~b);
    printf("%hd\n", b << 2);// Multiplikation mit 4
    printf("%hd\n", b >> 3);// Division durch 8
    printf("%hd\n", c >> 4);// Division durch 16
    return EXIT_SUCCESS;
}
```

Ausgabe:
```
0
15
11
-33
128
4
2
```

```
B      0000 0000 0010 0000
<< 2:  0000 0000 1000 0000
>> 3:  0000 0000 0000 0100
```

```
C      0000 0000 0010 1111
>> 4:  0000 0000 0000 0010
```

Interaktive Aufgabe

- Was gibt das folgende C-Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char a = 15;
  char b = 32;
  char c = 17;
  char d = 254;
  printf("%d\n", a & 0);
  printf("%d\n", b | 1);
  printf("%d\n", c ^ 3);
  printf("%d\n", ~d);
  printf("%d\n", b << 2);
  printf("%d\n", b >> 1);
  printf("%d\n", b >> 4);
  return EXIT_SUCCESS;
}
```

sizeof-Operator (1)

- Der Operator **`sizeof`** dient zur Ermittlung der Größe von Datentypen und Datenobjekten im Hauptspeicher.
- Der Operator kann sowohl auf einen Ausdruck als auch auf einen Typbezeichner angewendet werden.
  - Ein Ausdruck wird nicht ausgewertet, es wird nur der Typ und dessen Größe bestimmt.
- Beispiele (am zid-gpl mit %zu):

```
printf("%zu\n", sizeof(char));
printf("%zu\n", sizeof(short));
printf("%zu\n", sizeof(int));
printf("%zu\n", sizeof(long));
printf("%zu\n", sizeof(long long));
printf("%zu\n", sizeof(float));
printf("%zu\n", sizeof(double));
printf("%zu\n", sizeof(long double));
printf("%zu\n", sizeof(123 + 456));
```

Ausgabe:

```
1
2
4
8
8
4
8
16
4
```

sizeof-Operator (2)

- Klammerung
  - Klammern nach `sizeof` können bei einem Ausdruck weggelassen werden.
    - Sie werden aber üblicherweise immer gesetzt.
  - ACHTUNG: `sizeof` hat hohe Präzedenz
    - `sizeof x + y` wird vom Compiler akzeptiert, ermittelt aber zuerst die Größe von x und zählt dann y dazu!
    - Besser: `sizeof(x) + y`
  - Bei einem Datentyp müssen die Klammern gesetzt werden.
- Rückgabetyp
  - `sizeof` liefert ein Ergebnis vom Typ `unsigned long` oder `unsigned int`!
    - Ergebnis ist vom Typ `size_t` (definiert in `stddef.h`).
    - Architekturabhängig (32-Bit, 64-Bit)!
    - Muss bei Berechnungen bzw. Vergleichen berücksichtigt werden!

Typumwandlung

- Wenn man in C einen Datentyp in einen anderen konvertiert (umwandelt), führt man ein sogenanntes **Type-Casting** durch.
- Es gibt zwei Möglichkeiten, den Datentyp zu ändern:
  - Implizite Datentypumwandlung
    - Der Compiler nimmt eine automatische Konvertierung von einem zum anderen Datentyp vor.
    - Dies geschieht, wenn z. B. einem `int`-Wert ein `float`-Wert zugewiesen wird.
  - Explizite Datentypumwandlung
    - Der Programmierer kann die Konvertierung des Datentyps durch eine explizite Typumwandlung erzwingen.
- Bei beiden Vorgehensweisen wird vorausgesetzt, dass der Compiler eine Typumwandlung auch wirklich unterstützt.

Implizite Typumwandlung

- Es ist nicht notwendig, dass die Operanden eines Ausdrucks vom selben Typ sind.
- Auch bei einer Zuweisung muss der Typ der Operanden nicht übereinstimmen.
- In solchen Fällen (und noch anderen, die später besprochen werden) kann der Compiler implizite (automatische) Typumwandlungen durchführen.
  - Diese Umwandlungen laufen nach bestimmten Regeln ab.
  - Diese Umwandlungen erfolgen nur zwischen verträglichen Typen.

Regel 1

- Gerechnet wird mit nichts Kleinerem als mit `int`.
  - `char` und `short` werden implizit und ohne Einwirkung des Programmierers nach `int` umgewandelt.
  - Reicht der Datentyp `int` nicht aus, um einen Wert aufzunehmen, wird mit `unsigned int` gerechnet.
- Anmerkung
  - Ob ein reiner char-Wert als vorzeichenbehaftet behandelt wird oder nicht, ist von C nicht festgelegt und daher abhängig vom Compiler.

Beispiel (Regel 1)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
	char zeich;
	short short_ganz;
	int int_ganz;
	unsigned unsigned_ganz;
	int wert;
	printf("-----------------------------------\n");
	printf("    zeich: %zu Bytes\n", sizeof(zeich));
	printf("      'x': %zu Bytes\n", sizeof('x'));
	printf("zeich+'a': %zu Bytes\n", sizeof(zeich + 'a'));
	printf("-----------------------------------\n");
	printf("   short_ganz: %zu Bytes\n", sizeof(short_ganz));
	printf("          235: %zu Bytes\n", sizeof(235));
	printf("         -235: %zu Bytes\n", sizeof(-235));
	printf("short_ganz+12: %zu Bytes\n", sizeof(short_ganz + 12));
	printf("-----------------------------------\n");
	printf("   int_ganz: %zu Bytes\n", sizeof(int_ganz));
	printf("        235: %zu Bytes\n", sizeof(235));
	printf("       -235: %zu Bytes\n", sizeof(-235));
	printf("int_ganz+12: %zu Bytes\n", sizeof(int_ganz + 12));
	printf("-----------------------------------\n");
	printf("   unsigned_ganz: %zu Bytes\n", sizeof(unsigned_ganz));
	printf("             235: %zu Bytes\n", sizeof(235));
	printf("unsigned_ganz+12: %zu Bytes\n", sizeof(unsigned_ganz + 12));
	printf("-----------------------------------\n");
	wert = (1000000000 + 2000000000) / 2;
	printf("(1000000000+2000000000)/2 = %d\n", wert);
	printf("-----------------------------------\n");
	wert = (1000000000 + 3000000000) / 2;
	printf("(1000000000+3000000000)/2 = %d\n", wert);
	printf("-----------------------------------\n");
	return EXIT_SUCCESS;
}
```

Ausgabe:

```
-----------------------------------
    zeich: 1 Bytes
      'x': 4 Bytes
zeich+'a': 4 Bytes
-----------------------------------
   short_ganz: 2 Bytes
          235: 4 Bytes
         -235: 4 Bytes
short_ganz+12: 4 Bytes
-----------------------------------
   int_ganz: 4 Bytes
        235: 4 Bytes
       -235: 4 Bytes
int_ganz+12: 4 Bytes
-----------------------------------
   unsigned_ganz: 4 Bytes
             235: 4 Bytes
unsigned_ganz+12: 4 Bytes
-----------------------------------
(1000000000+2000000000)/2 = -647483648
-----------------------------------
(1000000000+3000000000)/2 = 2000000000
-----------------------------------
```

Hier gibt es eine Warnung!

Beispiel (Regel 1)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    unsigned char u_zeich = '\x84';
    signed char s_zeich = '\x84';
    int i1, i2;
    i1 = u_zeich;
    i2 = s_zeich;
    printf("'%c' :  %d  %#x\n", i1, i1, i1);
    printf("'%c' :  %d  %#x\n", i2, i2, i2);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
'▒' :  132  0x84
'▒' :  -124  0xffffff84
```

Regel 2 (signed und unsigned)

- Umwandlung unsigned-Typ in breiteren signed- oder unsigned-Typ:
  - Ursprünglicher Wert bleibt unverändert.
- Umwandlung signed-Typ in einen gleich breiten oder breiteren unsigned-Typ:
  - Erhält die interne Bitdarstellung
  - Vorzeichenbit wird bei längerem unsigned-Typ nach vorne verlängert.
- Umwandlung unsigned-Typ in einen gleich breiten signed-Typ:
  - Bitmuster wird im signed-Typ abgespeichert.
  - Liegt der Wertebereich des unsigned-Typs über dem des signed-Typs, dann schreibt der ANSI-C-Standard **nicht** vor, was passieren soll!
- Umwandlung signed- bzw. unsigned-Typ in einen kleineren signed- bzw. unsigned-Typ
  - Ergebnis hängt vom Compiler ab!
  - Typischerweise Bits abschneiden!

Beispiel (Regel 2)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
     unsigned char u_char;
     unsigned int u_int;
     unsigned long int u_l_int;
     signed long int s_l_int;
     int int1, int2;
     unsigned int u_int1, u_int2;
     unsigned long int u_l_int1, u_l_int2;
     u_char = 'a';
     u_int = 234;
     u_l_int = u_char;
     printf(" %c (unsigned char) ---> %lu (unsigned long int)\n", u_char, u_l_int);
     u_l_int = u_int;
     printf(" %u (unsigned int) ---> %lu (unsigned long int)\n", u_int, u_l_int);
     s_l_int = u_char;
     printf(" %c (unsigned char) ---> %ld (signed long int)\n", u_char, s_l_int);
     s_l_int = u_int;
     printf(" %d (unsigned int) ---> %ld (signed long int)\n\n", u_int, s_l_int);
     int1 = 0xfff;
     int2 = -3;
     u_int1 = int1;
     printf(" %d (int) ---> %u (unsigned int) = %#x\n", int1, u_int1, u_int1);
     u_int2 = int2;
     printf(" %d (int) ---> %u (unsigned int) = %#x\n", int2, u_int2, u_int2);
     u_l_int1 = int1;
```

Beispiel (Regel 2, 2. Teil)

```
    printf(" %d (int) ---> %lu (unsigned long int) = %#lx\n", int1, u_l_int1, u_l_int1);
    u_l_int2 = int2;
    printf(" %d (int) ---> %lu (unsigned long int) = %#lx = %ld (long int)\n\n",
    int2, u_l_int2, u_l_int2,  u_l_int2);
    s_l_int = -3;
    u_l_int = 0xabcdef;
    u_char = s_l_int;
    printf(" %#lx = %ld (signed long int) ---> %c (unsigned char) = %#x\n", s_l_int, s_l_int,
    u_char, u_char);
    u_char = u_l_int;
    printf(" %#lx = %lu (unsigned long int) ---> %c (unsigned char) = %#x\n", u_l_int,
    u_l_int, u_char, u_char);
    u_int = s_l_int;
    printf(" %#lx = %ld (signed long int) ---> %u (unsigned int) = %#x\n", s_l_int, s_l_int,
    u_int, u_int);
    u_int = u_l_int;
    printf(" %#lx = %lu (unsigned long int) ---> %u (unsigned int) = %#x\n\n", u_l_int,
    u_l_int, u_int, u_int);
    u_int = 123;
    int1 = u_int;
    printf(" %#x = %u (unsigned int) ---> %d (int) = %#x\n", u_int, u_int, int1, int1);
    u_int = 50000;
    int1 = u_int;
    printf(" %#x = %u (unsigned int) ---> %d (int) = %#x\n", u_int, u_int, int1, int1);
    return EXIT_SUCCESS;
}
```

Beispiel (Regel 2, 3. Teil)

Ausgabe:

a (unsigned char) ---> 97 (unsigned long int)
234 (unsigned int) ---> 234 (unsigned long int)
a (unsigned char) ---> 97 (signed long int)
234 (unsigned int) ---> 234 (signed long int)

4095 (int) ---> 4095 (unsigned int) = 0xfff
-3 (int) ---> 4294967293 (unsigned int) = 0xfffffffd
4095 (int) ---> 4095 (unsigned long int) = 0xfff
-3 (int) ---> 18446744073709551613 (unsigned long int) = 0xfffffffffffffffd = -3 (long int)

0xfffffffffffffffd = -3 (signed long int) ---> ▒ (unsigned char) = 0xfd
0xabcdef = 11259375 (unsigned long int) ---> ▒ (unsigned char) = 0xef
0xfffffffffffffffd = -3 (signed long int) ---> 4294967293 (unsigned int) = 0xfffffffd
0xabcdef = 11259375 (unsigned long int) ---> 11259375 (unsigned int) = 0xabcdef

0x7b = 123 (unsigned int) ---> 123 (int) = 0x7b
0xc350 = 50000 (unsigned int) ---> 50000 (int) = 0xc350

Regel 3 (Gleitkommazahlen und Ganzzahlen)

- Gleitkommazahl -> Ganzzahl
  - Der gebrochene Anteil der Gleitpunktzahl wird abgeschnitten.
  - Liegt der ganzzahlige Anteil außerhalb des Wertebereichs des Ganzzahltyps, dann ist das Verhalten undefiniert!
- Ganzzahl -> Gleitkommazahl
  - Umwandlung erfolgt
  - Aber: Liegt der Wert der Ganzzahl im Wertebereich der darstellbaren Gleitkommazahl, ist aber nicht darstellbar, dann ist das Ergebnis entweder der nächsthöhere oder der nächstniedrigere darstellbare Wert.
    - Ergebnis ist implementierungsabhängig und nicht vom ANSI-C-Standard vorgeschrieben.
  - Liegt der Wert außerhalb des darstellbaren Bereichs, dann liegt undefiniertes Verhalten vor.

Beispiel (Regel 3)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
      float gleit;
      int ganz;
      unsigned int u_ganz;
      long int lang;
      gleit = 7.33;
      ganz = gleit;
      printf("%f (float) ---> %d (int) = %#x\n", gleit, ganz, ganz);
      gleit = -3.7;
      ganz = gleit;
      printf("%f (float) ---> %d (int) = %#x\n", gleit, ganz, ganz);
      gleit = 7.33;
      u_ganz = gleit;
      printf("%f (float) ---> %u (unsigned int) = %#x\n", gleit, u_ganz, u_ganz);
      gleit = -3.7;
      u_ganz = gleit;
      printf("%f (float) ---> %u (unsigned int) = %#x\n", gleit, u_ganz, u_ganz);
      lang = 12345678;
      gleit = lang;
      printf("%ld (long int) ---> %.0f (float)\n", lang, gleit);
      lang = 1234567890;
      gleit = lang;
      printf("%ld (long int) ---> %.0f (float)\n", lang, gleit);
      return EXIT_SUCCESS;
}
```

Ausgabe:

```
7.330000 (float) ---> 7 (int) = 0x7
-3.700000 (float) ---> -3 (int) = 0xfffffffd
7.330000 (float) ---> 7 (unsigned int) = 0x7
-3.700000 (float) ---> 4294967293 (unsigned int) = 0xfffffffd
12345678 (long int) ---> 12345678 (float)
1234567890 (long int) ---> 1234567936 (float)
```

Regel 4 (float, double und long double)

- 1. Fall: `float` -> `double`, `float` -> `long double`, `double` -> `long double`
  - Ursprünglicher Wert bleibt unverändert.
- 2. Fall: `double` -> `float`, `long double` -> `float`, `long double` -> `double`
  - Falls der Wert größer ist als der Zieldatentyp, liegt undefiniertes Verhalten vor.
  - Ist der Wert nicht zu groß für den Zieldatentyp, kann aber dort nicht genau dargestellt werden, da dort weniger Bits zur Darstellung zur Verfügung stehen, dann ist das Ergebnis entweder der nächsthöhere oder nächstniedrigere darstellbare Wert (vom Standard nicht vorgeschrieben).
  - Ansonsten wird der Wert direkt übernommen.

Beispiel (Regel 4)

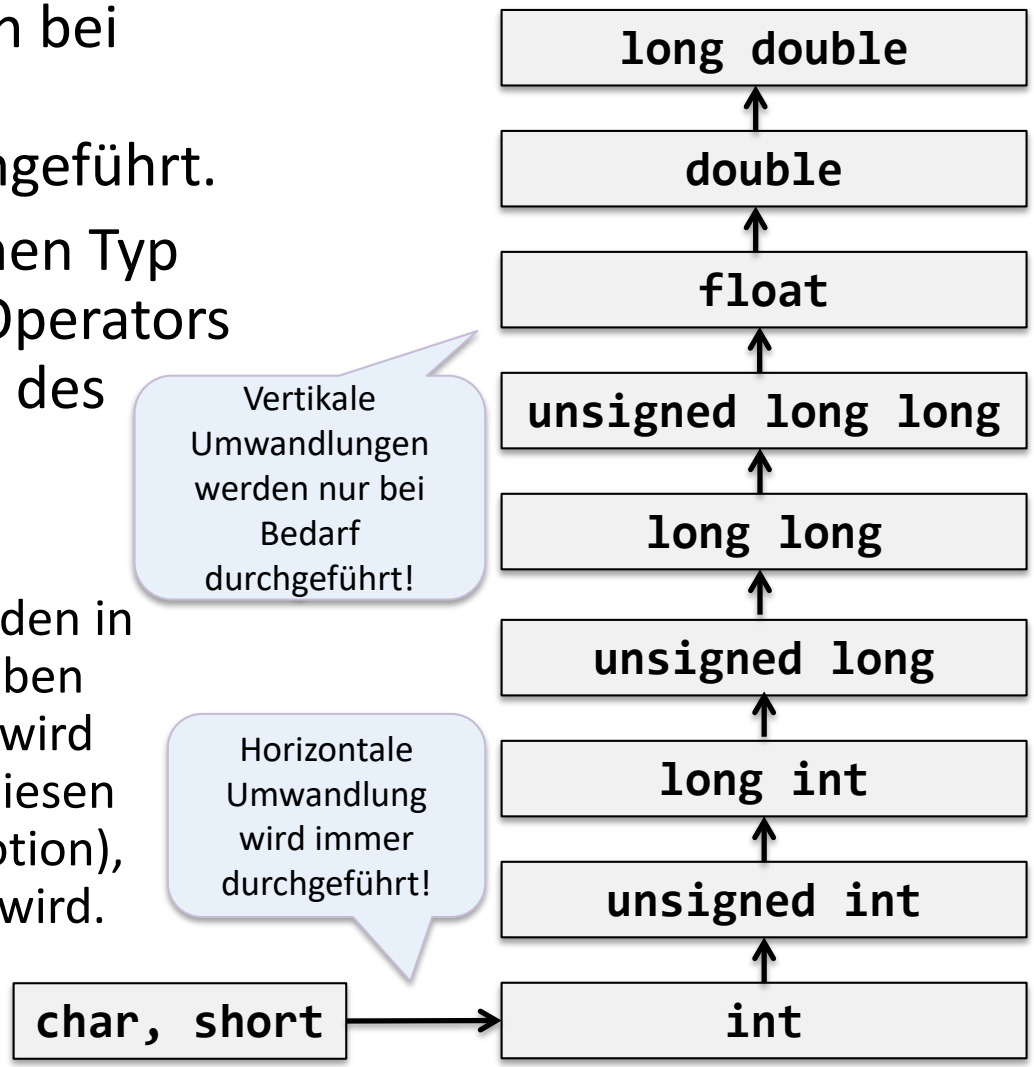
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
      float einfach;
      double doppelt;
      long double lang_doppelt;
      einfach = 4 * 1.23456;
      doppelt = einfach;
      printf("%f (float) ---> %f (double)\n", einfach, doppelt);
      lang_doppelt = einfach;
      printf("%f (float) ---> %Lf (long double)\n", einfach, lang_doppelt);
      doppelt = 4 * 1.23456;
      lang_doppelt = doppelt;
      printf("%lf (double) ---> %Lf (long double)\n", doppelt, lang_doppelt);
      doppelt = 4 * 1.23456;
      einfach = doppelt;
      printf("%.10lf (double) ---> %.10f (float)\n", doppelt, einfach);
      lang_doppelt = 4 * 1.23456 + 12.2345256266535;
      einfach = lang_doppelt;
      printf("%.10Lf (long double) ---> %.10f (float)\n", lang_doppelt, einfach);
      doppelt = lang_doppelt;
      printf("%.15Lf (long double) ---> %.15lf (double)\n", lang_doppelt, doppelt);
      lang_doppelt = 2.34e-103;
      einfach = lang_doppelt;
      printf("%.10Lg (long double) ---> %.10g (float)\n", lang_doppelt, einfach);
      return EXIT_SUCCESS;
}
```

Ausgabe:

```
4.938240 (float) ---> 4.938240 (double)
4.938240 (float) ---> 4.938240 (long double)
4.938240 (double) ---> 4.938240 (long double)
4.9382400000 (double) ---> 4.9382400513 (float)
17.1727656267 (long double) ---> 17.1727657318 (float)
17.172765626653501 (long double) ---> 17.172765626653501 (double)
2.34e-103 (long double) ---> 0 (float)
```

Regel 5 (Übliche arithmetische Umwandlungen)

- Diese Umwandlungen werden bei binären Operatoren (außer Zuweisung, && und ||) durchgeführt.
- Das Ziel ist, einen gemeinsamen Typ der Operanden des binären Operators zu erhalten, der auch der Typ des Ergebnisses ist.
- Vertikale Umwandlung:
  - Wenn einer der beiden Operanden in einem Ausdruck einen weiter oben stehenden Datentyp hat, dann wird zuerst der andere Operand in diesen Datentyp umgewandelt (promotion), bevor der Ausdruck berechnet wird.
- Horizontale Umwandlung:
  - Diese Umwandlung wird bei Berechnungen **immer** durchgeführt.



Beispiel (Regel 5)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a = 40;
    char b = 'A';
    char c = 'K';
    long d;
    float e = 1000.000;
    double sum;
    a = b + c;
    printf ("%d\n", a);
    d = a + e;
    printf ("%ld\n", d);
    sum = a + b + c + d + e;
    printf ("%f\n", sum);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
140
1140
2420.000000
```

Explizite Typumwandlung durch Casts

- Ein Cast ist ein in Klammern gesetzter Datentyp, der einem Ausdruck vorangestellt wird.
  - **`(Datentyp) Ausdruck`**
  - Damit wird der Ausdruck (der auch als Konstante oder in Form einer Variable vorliegen kann) zum angegebenen Datentyp konvertiert.
  - Dadurch können auch Informationen verloren gehen.
- Achtung!
  - Cast muss an der richtigen Stelle platziert werden (siehe nachfolgendes Beispiel).
  - Priorität der Operatoren legt den Auswertungszeitpunkt fest.

Beispiel (Casts)

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a = 40, b = 15;
    float x = 1000000.0;
    printf("%d\n", a / b);
    printf("%f\n", (float) a / b);
    printf("%f\n", a / (float) b);
    printf("%f\n", (float)(a / b));
    printf("%f\n", (float) a + b / 30);
    printf("%f\n", (float) a + (double) b);
    printf("%d\n", (short) x);
    return EXIT_SUCCESS;
}
```

Ausgabe:

```
2
2.666667
2.666667
2.000000
40.000000
55.000000
16960
```

Interaktive Aufgabe

- Was gibt das folgende C-Programm aus?

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int a = 10, b = 3;
  double c = 4.0;

  printf("%d\n", a / b);
  printf("%f\n", (double) a / b);
  printf("%f\n", (double)(a / b));
  printf("%f\n", a / c);
  printf("%d\n", a / (int) c);
  printf("%f\n", (double) (a + b) / 10);
  printf("%f\n", (double) a + b / 10);
  printf("%f\n", (float) a + (double) c);
  printf("%d\n", (int) c);
  printf("%d\n", a + b + (int) c);
  printf("%f\n", a + b + c);

  return EXIT_SUCCESS;
}
```

Sequenzpunkte allgemein

- Seiteneffekt
  - Ist eine Modifikation eines Datenobjekts.
  - Beispiel: `a++;` hat den Seiteneffekt, dass a um 1 erhöht wird.
- Sequenzpunkt (sequence point)
  - Punkt in der Programmausführung, bei dem alle bisherigen Seiteneffekte ausgewertet sein müssen.
    - Danach wird erst mit der Ausführung fortgefahren.
  - Zwischen zwei Sequenzpunkten darf nur ein Schreibzugriff auf ein und dasselbe Objekt (Variable) stattfinden.
    - Falls neben dem Schreibzugriff zusätzlich Lesezugriffe vorhanden sind, müssen diese der Ermittlung des neuen Wertes dieses Objektes dienen.
  - Bei ++ und -- finden jeweils ein Lese- und ein Schreibzugriff statt.

Sequenzpunkte in C

- Bisher besprochen
  - `;` (Abschluss einer Anweisung)
  - `,` (Komma-Operator)
  - `&&` und `||`
  - `?`
- Weitere Sequenzpunkte in den folgenden Foliensätzen
  - Funktionsaufrufe (call-Zeitpunkt)
  - Direkt vor der Rückkehr von Bibliotheksfunktionen
  - Das Ende vollständiger Ausdrücke
    - Bei `if`, `while`, `for`, `switch`