

Systemnahe Programmierung in C (SPiC)

7 Operatoren und Ausdrücke

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2019

http://www4.cs.fau.de/Lehre/SS19/V_SPiC



- Stehen für alle Ganzzahl- und Fließkommatypen zur Verfügung

+ Addition

- Subtraktion

* Multiplikation

/ Division

unäres - negatives Vorzeichen (z. B. $-a$) \rightsquigarrow Multiplikation mit -1

unäres + positives Vorzeichen (z. B. $+3$) \rightsquigarrow kein Effekt

- Zusätzlich nur für Ganzzahltypen:

% Modulo (Rest bei Division)



- Stehen für Ganzzahltypen und Zeigertypen zur Verfügung

++	Inkrement (Erhöhung um 1)
--	Dekrement (Verminderung um 1)

- Linksseitiger Operator (Präfix) ++x bzw. --x

- Erst wird der Inhalt von x verändert
- Dann wird der (neue) Inhalt von x als Ergebnis geliefert

- Rechtsseitiger Operator (Postfix) x++ bzw. x--

- Erst wird der (alte) Inhalt von x als Ergebnis geliefert
- Dann wird der Inhalt von x verändert

- Beispiele

```
a = 10;  
b = a++; // b: 10, a: 11  
c = ++a; // c: 12, a: 12
```



■ Vergleichen von zwei Ausdrücken

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich (zwei Gleichheitszeichen!)
!=	ungleich

■ **Beachte:** Ergebnis ist vom Typ `int`

[≠Java]

- Ergebnis: *falsch* ↦ 0
wahr ↦ 1
- Man kann mit dem Ergebnis rechnen

■ Beispiele

```
if (a >= 3) {···}
if (a == 3) {···}
return a * (a > 0); // return 0 if a is negative
```



- Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

&&	„und“	<i>wahr</i> && <i>wahr</i> \rightarrow <i>wahr</i>
	(Konjunktion)	<i>wahr</i> && <i>falsch</i> \rightarrow <i>falsch</i>
		<i>falsch</i> && <i>falsch</i> \rightarrow <i>falsch</i>

	„oder“	<i>wahr</i> <i>wahr</i> \rightarrow <i>wahr</i>
	(Disjunktion)	<i>wahr</i> <i>falsch</i> \rightarrow <i>wahr</i>
		<i>falsch</i> <i>falsch</i> \rightarrow <i>falsch</i>

!	„nicht“	! <i>wahr</i> \rightarrow <i>falsch</i>
	(Negation, unär)	! <i>falsch</i> \rightarrow <i>wahr</i>

- **Beachte:** Operanden und Ergebnis sind vom Typ `int` [\neq Java]

- Operanden (Eingangparameter): $0 \mapsto$ *falsch*
 $\neq 0 \mapsto$ *wahr*

- Ergebnis: *falsch* \mapsto 0
wahr \mapsto 1



- Die Auswertung eines logischen Ausdrucks wird **abgebrochen**, sobald das Ergebnis feststeht

■ Sei `int a = 5;` `int b = 3;` `int c = 7;`

$\underbrace{a > b}_{1} \ || \ \underbrace{a > c}_{?}$ ← wird nicht ausgewertet, da der erste Term bereits *wahr* ergibt

$\underbrace{a > c}_{0} \ \&\& \ \underbrace{a > b}_{?}$ ← wird nicht ausgewertet, da der erste Term bereits *falsch* ergibt

- Kann **überraschend** sein, wenn Teilausdrücke **Nebeneffekte** haben

```
int a = 5; int b = 3; int c = 7;
if ( a > c && !func(b) ) {···} // func() will not be called
```



- Allgemeiner Zuweisungsoperator (=)
 - Zuweisung eines Wertes an eine Variable
 - Beispiel: `a = b + 23`
- Arithmetische Zuweisungsoperatoren (`+=`, `-=`, ...)
 - Abgekürzte Schreibweise zur Modifikation des Variablenwerts
 - Beispiel: `a += 23` ist äquivalent zu `a = a + 23`
 - Allgemein: `a op= b` ist äquivalent zu `a = a op b`
für $op \in \{ +, -, *, \%, \ll, \gg, \&, ^, | \}$
- Beispiele

```
int a = 8;  
a += 8;    // a: 16  
a %= 3;   // a: 1
```



Zuweisungen sind Ausdrücke!

- Zuweisungen können in komplexere Ausdrücke geschachtelt werden
 - Das Ergebnis eines Zuweisungsausdrucks ist der zugewiesene Wert

```
int a, b, c;  
a = b = c = 1; // c: 1, b: 1, a: 1
```

- Die Verwendung von Zuweisungen in beliebigen Ausdrücken führt zu **Nebenwirkungen**, die nicht immer offensichtlich sind

```
a += b += c; // Value of a and b?
```



Zuweisungen sind Ausdrücke!

- Zuweisungen können in komplexere Ausdrücke geschachtelt werden
 - Das Ergebnis eines Zuweisungsausdrucks ist der zugewiesene Wert

```
int a, b, c;  
a = b = c = 1; // c: 1, b: 1, a: 1
```

- Die Verwendung von Zuweisungen in beliebigen Ausdrücken führt zu **Nebeneffekten**, die nicht immer offensichtlich sind

```
a += b += c; // Value of a and b?
```

Besonders gefährlich: Verwendung von = statt ==

In C sind Wahrheitswerte Integers: 0 \mapsto falsch, \emptyset \mapsto wahr

- Typischer „Anfängerfehler“ in Kontrollstrukturen:

```
if (a = 6) {...} else {...} // BUG: if-branch is always taken!!!
```

- Compiler beanstandet das Konstrukt nicht, es handelt sich um einen gültigen Ausdruck! \rightsquigarrow Fehler wird leicht übersehen!



■ Bitweise Verknüpfung von Ganzzahltypen, kommutativ

&	bitweises „Und“ (Bit-Schnittmenge)	$1 \& 1 \rightarrow 1$
		$1 \& 0 \rightarrow 0$
		$0 \& 0 \rightarrow 0$

	bitweises „Oder“ (Bit-Vereinigungsmenge)	$1 1 \rightarrow 1$
		$1 0 \rightarrow 1$
		$0 0 \rightarrow 0$

\wedge	bitweises „Exklusiv-Oder“ (Bit-Antivalenz)	$1 \wedge 1 \rightarrow 0$
		$1 \wedge 0 \rightarrow 1$
		$0 \wedge 0 \rightarrow 0$

~	bitweise Inversion (Einerkomplement, unär)	$\sim 1 \rightarrow 0$
		$\sim 0 \rightarrow 1$



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<< bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)

>> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ `uint8_t`)

Bit# 7 6 5 4 3 2 1 0

x=156

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

 0x9c



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<< bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)

>> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ uint8_t)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<< bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
 >> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ uint8_t)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63
7	0	0	0	0	0	1	1	1	0x07
x 7	1	0	0	1	1	1	1	1	0x9f



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ
 - << bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
 - >> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ uint8_t)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63
7	0	0	0	0	0	1	1	1	0x07
x 7	1	0	0	1	1	1	1	1	0x9f
x & 7	0	0	0	0	0	1	0	0	0x04



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

\ll bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
 \gg bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ `uint8_t`)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63
7	0	0	0	0	0	1	1	1	0x07
x 7	1	0	0	1	1	1	1	1	0x9f
x & 7	0	0	0	0	0	1	0	0	0x04
x ^ 7	1	0	0	1	1	0	1	1	0x9B



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<< bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
 >> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ uint8_t)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63
7	0	0	0	0	0	1	1	1	0x07
x 7	1	0	0	1	1	1	1	1	0x9f
x & 7	0	0	0	0	0	1	0	0	0x04
x ^ 7	1	0	0	1	1	0	1	1	0x9B
x << 2	0	1	1	1	0	0	0	0	0x70



- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<< bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
 >> bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ uint8_t)

Bit#	7	6	5	4	3	2	1	0	
x=156	1	0	0	1	1	1	0	0	0x9c
~x	0	1	1	0	0	0	1	1	0x63
7	0	0	0	0	0	1	1	1	0x07
x 7	1	0	0	1	1	1	1	1	0x9f
x & 7	0	0	0	0	0	1	0	0	0x04
x ^ 7	1	0	0	1	1	0	1	1	0x9B
x << 2	0	1	1	1	0	0	0	0	0x70
x >> 1	0	1	0	0	1	1	1	0	0x4e



Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit#

7 6 5 4 3 2 1 0

PORTD

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!



Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit# 7 6 5 4 3 2 1 0
PORTD

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!

0x80

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

PORTD |= 0x80

1	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist



Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit# 7 6 5 4 3 2 1 0
PORTD [? ? ? ? ? ? ? ?]

Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!

0x80 [1 0 0 0 0 0 0 0]
PORTD |= 0x80 [1 ? ? ? ? ? ? ?]

Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist

~0x80 [0 1 1 1 1 1 1 1]
PORTD &= ~0x80 [0 ? ? ? ? ? ? ?]

Löschen eines Bits durch **Ver-unden** mit Maske, in der nur das Zielbit 0 ist



Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit# 7 6 5 4 3 2 1 0
PORTD

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!

0x80

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

PORTD |= 0x80

1	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist

~0x80

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

PORTD &= ~0x80

0	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Löschen eines Bits durch **Ver-unden** mit Maske, in der nur das Zielbit 0 ist

0x08

0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

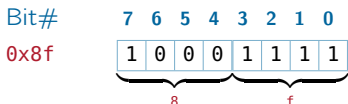
PORTD ^= 0x08

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

Invertieren eines Bits durch **Ver-xodern** mit Maske, in der nur das Zielbit 1 ist



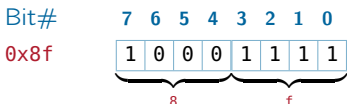
- Bitmasken werden gerne als Hexadezimal-Literale angegeben



Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) \rightsquigarrow Verständlichkeit



- Bitmasken werden gerne als Hexadezimal-Literale angegeben



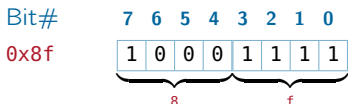
Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) ~ Verständlichkeit

- Für „Dezimal-Denker“ bietet sich die Linksschiebe-Operation an

```
PORTD |= (1<<7);     // set bit 7: 1<<7 --> 10000000
PORTD &= ~(1<<7);   // mask bit 7: ~(1<<7) --> 01111111
```



- Bitmasken werden gerne als Hexadezimal-Literale angegeben



Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) \rightsquigarrow Verständlichkeit

- Für „Dezimal-Denker“ bietet sich die Linksschiebe-Operation an

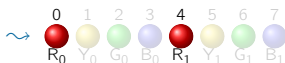
```
PORTD |= (1<<7);      // set bit 7: 1<<7 --> 10000000
PORTD &= ~(1<<7);    // mask bit 7: ~(1<<7) --> 01111111
```

- Zusammen mit der Oder-Operation auch für komplexere Masken

```
#include <led.h>
void main() {
    uint8_t mask = (1<<RED0) | (1<<RED1);

    sb_led_set_all_leds (mask);

    while(1) ;
}
```



- Formulierung von Bedingungen in Ausdrücken

$Ausdruck_1 ? Ausdruck_2 : Ausdruck_3$

- Zunächst wird $Ausdruck_1$ ausgewertet
 - $Ausdruck_1 \neq 0$ (*wahr*) \rightsquigarrow Ergebnis ist $Ausdruck_2$
 - $Ausdruck_1 = 0$ (*falsch*) \rightsquigarrow Ergebnis ist $Ausdruck_3$
- $?$: ist der einzige ternäre (dreistellige) Operator in C

- Beispiel

```
int abs(int a) {  
    // if (a<0) return -a; else return a;  
    return (a<0) ? -a : a;  
}
```



- Reihung von Ausdrücken
Ausdruck₁ , *Ausdruck₂*
 - Zunächst wird *Ausdruck₁* ausgewertet
 ↪ Nebeneffekte von *Ausdruck₁* werden sichtbar
 - Ergebnis ist der Wert von *Ausdruck₂*
- Verwendung des Komma-Operators ist selten erforderlich!
(Präprozessor-Makros mit Nebeneffekten)



	Klasse	Operatoren	Assoziativität
1	Funktionsaufruf, Feldzugriff Strukturzugriff Post-Inkrement/-Dekrement	x() x[] x.y x->y x++ x--	links → rechts
2	Prä-Inkrement/-Dekrement unäre Operatoren Adresse, Verweis (Zeiger) Typkonvertierung (cast) Typgröße	++x --x +x -x ~x !x & * (<Typ>)x sizeof(x)	rechts → links
3	Multiplikation, Division, Modulo	* / %	links → rechts
4	Addition, Subtraktion	+ -	links → rechts
5	Bitweises Schieben	>> <<	links → rechts
6	Relationaloperatoren	< <= > >=	links → rechts
7	Gleichheitsoperatoren	== !=	links → rechts
8	Bitweises UND	&	links → rechts
9	Bitweises OR		links → rechts
10	Bitweises XOR	^	links → rechts
11	Konjunktion	&&	links → rechts
12	Disjunktion		links → rechts
13	Bedingte Auswertung	?:=	rechts → links
14	Zuweisung	= op=	rechts → links
15	Sequenz	,	links → rechts



Typumwandlung in Ausdrücken

- Eine Operation wird *mindestens* mit `int`-Wortbreite berechnet
 - `short`- und `signed char`-Operanden werden implizit „aufgewertet“ (\hookrightarrow *Integer Promotion*)
 - Erst das Ergebnis wird auf den Zieldatentyp abgeschnitten/erweitert

```
int8_t a=100, b=3, c=4, res; // range: -128 --> +127  
  
res = a * b / c; // promotion to int: 300 fits in!  
int8_t: 75      int: 100  int: 3  int: 4  
                └───┬───┘  
                int: 300  
                └──────────┘  
                int: 75
```



- Generell wird die *größte* beteiligte Wortbreite verwendet

↔ 6-3

```
int8_t a=100, b=3, res; // range: -128 --> +127
int32_t c=4;           // range: -2147483648 --> +2147483647
```

```
res = a * b / c; // promotion to int32_t

int8_t: 75      int: 100   int: 3
               └───┬───┘
                   int: 300
               └───┬───┘
                   int32_t: 300
               └───┬───┘
                   int32_t: 75
```



Typumwandlung in Ausdrücken (Forts.)

- Fließkomma-Typen gelten dabei als „größer“ als Ganzzahl-Typen
- Alle Fließkomma-Operationen werden *mindestens* mit **double**-Wortbreite berechnet

```
int8_t a=100, b=3, res; // range: -128 --> +127

res = a * b / 4.0f ; // promotion to double
int8_t: 75      int: 100   int: 3      double 4.0
                └──────────┘
                int: 300
                └──────────┘
                double: 300.0
                └──────────────────┘
                double: 75.0
```



- `unsigned`-Typen gelten dabei als „größer“ als `signed`-Typen

```
int s = -1, res;           // range: -32768 --> +32767
unsigned u = 1;           // range: 0 --> 65535

res = s < u;              // promotion to unsigned: -1 --> 65535
int: 0      unsigned: 65535
           └──────────┬──────────┘
                   unsigned: 0
```

- ↪ Überraschende Ergebnisse bei negativen Werten!
- ↪ Mischung von `signed`- und `unsigned`-Operanden vermeiden!



Typumwandlung in Ausdrücken – Typ-Casts

- Durch den Typ-Cast-Operator kann man einen Ausdruck gezielt in einen anderen Typ konvertieren

(Typ) Ausdruck

```
int s = -1, res;           // range: -32768 --> +32767
unsigned u = 1;           // range: 0 --> 65535

res = s < (int) u;        // cast u to int
  {      {
int: 1   int: 1
  }      }
      {
int: 1
      }
```

