§2 Logische Ausdrücke, Bitoperationen

Leitideen: *In C++ gibt es im Unterschied zu C einen logischen Datentyp (bool).*

In C gilt ein Ausdruck mit Wert $\neq 0$ *als wahr und mit Wert* 0 *als falsch. Dieses Verhalten wird in C++ durch implizite Typumwandlungen zwischen bool und den eingebauten Zahltypen erreicht.*

Bitoperationen sind von logischen Operationen zu unterscheiden, sie werden jeweils paarweise auf alle Bits ganzzahliger Operanden angewandt.

- Weshalb logische Operationen und Bitoperationen?
- Logische Ausdrücke
- Bitoperationen
- Operatorenvorrang

Weshalb logische Operationen und Bitoperationen? (I)

Aussagenlogik

“und“

$\wedge$	f	w
f	f	f
w	f	w

“oder“

$\vee$	f	w
f	f	w
w	w	w

“nicht“

$\neg$	f	w
	w	f

Weshalb w$\hat{=}$1 **und** f$\hat{=}$0 **?**

- Multiplikation und Addition entsprechen dann (fast) den logischen Operationen

$\cdot$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	2

- Auch der Vorrang von $\cdot$ vor $+$ entspricht dem von $\wedge$ vor $\vee$.

  Bsp.: $a \cdot b + c \hat{=} (a \cdot b) + c$ $\quad$ $\mathcal{A} \wedge \mathcal{B} \vee \mathcal{C} \hat{=} (\mathcal{A} \wedge \mathcal{B}) \vee \mathcal{C}$

Weshalb log. Operationen und Bitoperationen? (II)

- Generell würde 1 Bit genügen, um logische Werte darzustellen.
- In C++ keine Unterstützung von 1-Bit-Datentypen (weil auch von Prozessoren üblicherweise nicht unterstützt) Mindestlänge für Ganzzahloperationen ist Länge von `int` bzw. `unsigned int` (Integererweiterung!)
- Deshalb Beschreitung zweier verschiedener Wege:
  1. In C++ Einführung des Datentyp `bool` mit Werten `true` und `false` und entsprechenden logischen Operatoren. `bool` gilt als Ganzzahltyp und unterliegt ggf. der Integererweiterung. Dabei und bei Umwandlungen in andere Zahltypen `true`$\rightarrow$ 1 und `false`$\rightarrow$ 0.
  2. Definition von Bitoperatoren, die paarweise auf alle Bits der Ganzzahloperanden angewendet werden.
- Implizite Umwandlung Zahltyp$\rightarrow$`bool` mit $\neq 0$ $\rightarrow$`true` und 0 $\rightarrow$`false` wegen C-Kompatibilität. Fehlerträchtig, deshalb möglichst vermeiden. Kenntnis jedoch zum Verständnis vieler C++-Programme notwendig.

Logische Operatoren

Op.	*Typ*	*Beispiele*	*Bedeutung*
`!`	unär, präfix	`! a`	Negation
`&&`	binär, infix	`a && b`	und
`||`	binär, infix	`a || b`	oder

Die Operatoren sind entsprechend ihrem *Vorrang* absteigend geordnet, d.h der unäre Operator `!` hat den höchsten Rang.

- Logische Operatoren haben Operanden vom Datentyp `bool` und liefern `bool`.
- "Short circuit evaluation" – Auswertung von links nach rechts *nur solange*, bis Ergebnis feststeht.
- Auch die Vergleichsoperatoren `>`, `<`, `>=`, `<=`, `==` und `!=` liefern als Ergebnis den Datentyp `bool`.
- Bedingungsausdrücke in if- und Wiederholungsanweisungen sind vom Datentyp `bool`.
- Wegen der impliziten Typumwandlung Zahltyp$\rightarrow$`bool` mit $\neq 0 \rightarrow$`true` und $0 \rightarrow$`false` sind dort arithmet. Ausdrücke zulässig.

Logische Operatoren - Beispiele

`x>0 && log(x)>y` $\ln(x)$ wird nur für $x > 0$ berechnet

Vorsicht:

`i==1`	wahr, falls i den Wert 1 hat
`i=1`	immer wahr, weil der Ausdruck `i=1` den Wert 1 $\rightarrow$ `true` hat !!
`1<=x && x<=2`	wahr, falls $1 \leq x \leq 2$
`1<=x<=2`	immer wahr, weil `(1<=x)<=2` auf `0<=2` bzw. `1<=2` führt
`!(x>0)`	wahr, falls $x \leq 0$
`!x>0`	wahr, falls $x = 0$ wegen `(!x)>0` $\Leftrightarrow$ `(!x)==1` $\Leftrightarrow$ `x==0`

Ternärer Operator `?` `:`

Syntax

Bedingung `?` *Ausdruck1* `:` *Ausdruck2*

Wirkung

Falls Bedingung wahr:
Wert des dreigliedrigen Ausdrucks = Wert von Ausdruck1
Falls Bedingung falsch:
Wert des dreigliedrigen Ausdrucks = Wert von Ausdruck2

Bsp.:
```
a>b  ? a :  b  /* max(a,b) */
a>=0 ? a : -a  /* abs(a)   */
```

- Aufgrund der Vorrangregeln ist im Ausdruck
  ```
  c = a>b ? a : b
  ```
  keine Klammerung nötig

Bitoperatoren

- Die Bitoperatoren `~ & | ^` wirken bitweise, sie dürfen *nicht* mit den logischen Operatoren `! && ||` verwechselt werden.
- Bei den Shiftoperatoren, z.B. `a<<n` (Linksshift um n Bits), muss $0 \leq n \leq$ Bitzahl (des Datenyps von a) gelten. Nullen werden nachgeschoben (bei vorzeichenlosen und nichtnegativen ganzen Zahlen)
- Seiteneffekte treten *nur* bei den Bitzuweisungsoperatoren `<<= >>= &= |= ^=` auf, *nicht* aber bei den Shiftoperatoren `<< >>`.
- Bitoperationen in Vergleichen klammern, weil Vergleiche stärker binden als binäre Bitoperatoren, z.B. `if ( (a&b) == 1 ) {...}`
- Bitoperatoren sollten möglichst nur auf *vorzeichenlose* ganzzahlige Datentypen angewandt werden. (implementierungsabhängig: Vorzeichen bei Rechtsshift!)

Bitoperatoren - Fortsetzung

“und“

&	0	1
0	0	0
1	0	1

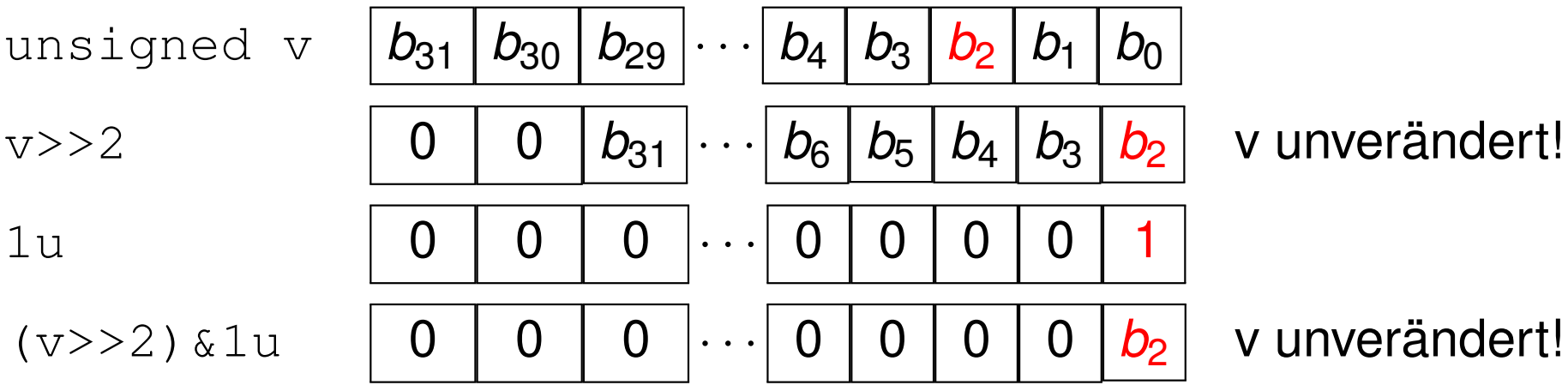
“inkl. oder“

|	0	1
0	0	1
1	1	1

“exkl. oder“

^	0	1
0	0	1
1	1	0

Bsp.: Bit 2 (von rechts her ab 0 gezählt) auslesen



Bem.:

- `1u` ist ein Literal vom Typ `unsigned int` mit Wert 1.

Bitoperatoren - Fortsetzung II

Bsp.: Bit 4 (von rechts her ab 0 gezählt) auf 1 setzen

`v`	b_{31} b_{30} $\cdots$ b_4 b_3 b_2 b_1 b_0	
`1u<<4`	0 0 $\cdots$ 1 0 0 0 0	
`v | 1u<<4`	b_{31} b_{30} $\cdots$ 1 b_3 b_2 b_1 b_0	v unverändert!
`v = v | 1u<<4`	b_{31} b_{30} $\cdots$ 1 b_3 b_2 b_1 b_0	v geändert
`v |= 1u<<4`	*gleiche Bedeutung wie vorige Zeile*	

Bitoperatoren - Fortsetzung III

Bsp.: Bit 3 (von rechts her ab 0 gezählt) auf 0 setzen

`v`	b_{31} b_{30} $\cdots$ b_4 b_3 b_2 b_1 b_0	
`1u<<3`	0 0 $\cdots$ 0 1 0 0 0	
`~(1u<<3)`	1 1 $\cdots$ 1 0 1 1 1	
`v & ~(1u<<3)`	b_{31} b_{30} $\cdots$ b_4 0 b_2 b_1 b_0	v unveränd!
`v = v & ~(1u<<3)`	b_{31} b_{30} $\cdots$ b_4 0 b_2 b_1 b_0	v geändert!
`v &= ~(1u<<3)`	*gleiche Bedeutung wie vorige Zeile*	

Vorrang der Operatoren

Namensauflösungsoperator	*höchster Vorrang*
Fkt.op.,Indexop.,Auswahlop.,Postfixop.	
Präfixoperatoren	*:*
binäre arithmetische Operatoren	
Vergleiche	
binäre Bitoperatoren	*:*
binäre logische Operatoren	
ternärer Operator `?` `:`	
Zuweisungen	*:*
Kommaoperator	*niedrigster Vorrang*

- Unäre (=einwertige) Operatoren haben Vorrang vor den meisten binären (=zweiwertigen) und ternären (=dreiwertigen) Operatoren.

  Problem: Kontraintuitiv, erwarten würde man eine Gruppierung nach Art der Operatoren und innerhalb jeder Gruppe den Vorrang unärer vor binären Operatoren.

Vorrang der Operatoren II

- Vergleichsoperatoren haben niedrigeren Vorrang als arithmetische Operatoren, aber höheren Vorrang als Bitoperatoren.

  Problem: Arithmetische Ausdrücke in Vergleichen müssen nicht geklammert werden, Bitausdrücke in Vergleichen dagegen schon.
- Trotz der Vorrangregeln empfiehlt sich Klammerung, vor allem bei seltener gebrauchten Operatoren.
- Die Vorrangregeln geben die Intention der Syntax wieder, in seltenen Fällen treffen sie nicht zu.