

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

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 Beschreibung 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` → 1 und `false` → 0.
 2. Definition von Bitoperatoren, die paarweise auf alle Bits der Ganzzahloperanden angewendet werden.
- ▶ Implizite Umwandlung Zahltyp → `bool` mit $\neq 0 \rightarrow \text{true}$ und $0 \rightarrow \text{false}$ wegen C-Kompatibilität. Fehlerträchtig, deshalb möglichst vermeiden. Kenntnis jedoch zum Verständnis vieler C++-Programme notwendig.

Logische Operatoren

<i>Op.</i>	<i>Typ</i>	<i>Beispiele</i>	<i>Bedeutung</i>
!	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 $\neq 0 \rightarrow \text{true}$ und $0 \rightarrow \text{false}$ 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 !!
$1 \leq x \ \&\& \ x \leq 2$	wahr, falls $1 \leq x \leq 2$
$1 \leq x \leq 2$	immer wahr, weil $(1 \leq x) \leq 2$ auf $0 \leq 2$ bzw. $1 \leq 2$ 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.: $a > b \quad ? \quad a \quad : \quad b \quad /* \max(a, b) */$
 $a \geq 0 \quad ? \quad a \quad : \quad -a \quad /* \text{abs}(a) \quad */$

- ▶ Aufgrund der Vorrangregeln ist im Ausdruck

$c = a > b \quad ? \quad a \quad : \quad b$

keine Klammerung nötig

Bitoperatoren

- ▶ Die Bitoperatoren \sim $\&$ $|$ \wedge wirken bitweise, sie dürfen *nicht* mit den logischen Operatoren $!$ $\&\&$ $||$ verwechselt werden.
- ▶ Bei den Shiftoperatoren, z.B. $a \ll n$ (Linksshift um n Bits), muss $0 \leq n < \text{Bitzahl (des Datentyps von } a)}$ gelten. Nullen werden nachgeschoben (bei vorzeichenlosen und nichtnegativen ganzen Zahlen)
- ▶ Seiteneffekte treten *nur* bei den Bitzuweisungsoperatoren $\ll=$ $\gg=$ $\&=$ $|=$ $\wedge=$ auf, *nicht* aber bei den Shiftoperatoren \ll \gg .
- ▶ 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

unsigned v	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>b_{29}</td><td>...</td><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	b_{29}	...	b_4	b_3	b_2	b_1	b_0	
b_{31}	b_{30}	b_{29}	...	b_4	b_3	b_2	b_1	b_0			
$v \gg 2$	<table border="1"><tr><td>0</td><td>0</td><td>b_{31}</td><td>...</td><td>b_6</td><td>b_5</td><td>b_4</td><td>b_3</td><td>b_2</td></tr></table>	0	0	b_{31}	...	b_6	b_5	b_4	b_3	b_2	v unverändert!
0	0	b_{31}	...	b_6	b_5	b_4	b_3	b_2			
1u	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>...</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	...	0	0	0	0	1	
0	0	0	...	0	0	0	0	1			
$(v \gg 2) \& 1u$	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>...</td><td>0</td><td>0</td><td>0</td><td>0</td><td>b_2</td></tr></table>	0	0	0	...	0	0	0	0	b_2	v unverändert!
0	0	0	...	0	0	0	0	b_2			

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	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>\dots</td><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	\dots	b_4	b_3	b_2	b_1	b_0	
b_{31}	b_{30}	\dots	b_4	b_3	b_2	b_1	b_0			
$1u \ll 4$	<table border="1"><tr><td>0</td><td>0</td><td>\dots</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	\dots	1	0	0	0	0	
0	0	\dots	1	0	0	0	0			
$v \mid 1u \ll 4$	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>\dots</td><td>1</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	\dots	1	b_3	b_2	b_1	b_0	v unverändert!
b_{31}	b_{30}	\dots	1	b_3	b_2	b_1	b_0			
$v = v \mid 1u \ll 4$	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>\dots</td><td>1</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	\dots	1	b_3	b_2	b_1	b_0	v geändert
b_{31}	b_{30}	\dots	1	b_3	b_2	b_1	b_0			
$v \mid= 1u \ll 4$	<i>gleiche Bedeutung wie vorige Zeile</i>									

Bitoperatoren - Fortsetzung III

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

v	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>...</td><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	...	b_4	b_3	b_2	b_1	b_0	
b_{31}	b_{30}	...	b_4	b_3	b_2	b_1	b_0			
$1u \ll 3$	<table border="1"><tr><td>0</td><td>0</td><td>...</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	...	0	1	0	0	0	
0	0	...	0	1	0	0	0			
$\sim(1u \ll 3)$	<table border="1"><tr><td>1</td><td>1</td><td>...</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	...	1	0	1	1	1	
1	1	...	1	0	1	1	1			
$v \& \sim(1u \ll 3)$	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>...</td><td>b_4</td><td>0</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	...	b_4	0	b_2	b_1	b_0	v unverändert!
b_{31}	b_{30}	...	b_4	0	b_2	b_1	b_0			
$v = v \& \sim(1u \ll 3)$	<table border="1"><tr><td>b_{31}</td><td>b_{30}</td><td>...</td><td>b_4</td><td>0</td><td>b_2</td><td>b_1</td><td>b_0</td></tr></table>	b_{31}	b_{30}	...	b_4	0	b_2	b_1	b_0	v geändert!
b_{31}	b_{30}	...	b_4	0	b_2	b_1	b_0			
$v \&= \sim(1u \ll 3)$	<i>gleiche Bedeutung wie vorige Zeile</i>									

Vorrang der Operatoren

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

- 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.