

§2 ARITHMETISCHE AUSDRÜCKE

Leitideen: Ausdrücke haben in der Regel einen Wert und können zusätzlich Seiteneffekte bewirken.

Die Auswertung erfolgt durch Rückführung auf unäre oder binäre Operationen mit Operanden gleichen Typs.

Diese Zerlegung ist durch Operatorenvorrang, Assoziativität und Typangleichung bestimmt.

- Übersicht über die arithmetischen Operatoren
- Seiteneffekte und Assoziativität an Beispielen
- Division und Rest
- Zuweisungen
- Typumwandlungsregeln
- Mathematische Standardfunktionen
- Fehlerbehandlung mit `errno`

Arithmetische Operatoren - Übersicht

- ▶ Ausdrücke in C++ haben in der Regel einen *Wert*.
- ▶ Bei der Auswertung von Ausdrücken können *Seiteneffekte* auftreten, z.B. Änderung des Werts einer Variablen oder Ein/Ausgabeoperationen.

<i>Op.</i>	<i>Typ</i>	<i>Beispiele</i>	<i>Wert des Ausdr.</i>	<i>Seiteneffekte</i>
++ --	unär, postfix	i++ i--	ursprgl. Var.wert	Inkr./Dekr.
++ --	unär, präfix	++i --i	geänd. Var.wert	Inkr./Dekr.
+ -	unär, präfix	+a -a	unveränd./Neg.	keine
* / %	binär, infix	a*b a/b k%n	Mult./Div./Rest	keine
+ -	binär, infix	a+b a-b	Add./Subtr.	keine
=	binär, infix	v=a	rechte Seite	Zuweisung

- ▶ Abnehmende Bindungswirkung von oben nach unten , d.h. *unäres* “-“ bindet stärker als *binäres* “*” und das stärker als *binäres* “+“, z.B.

$$-a*b+c \hat{=} ((-a)*b)+c$$

Arithmetische Operatoren - Fortsetzung

- ▶ Bsp.: `i=5; k=i++;`

<code>i=5</code>	Wert des Ausdrucks:	5	Seiteneffekt:	$i = 5$
------------------	---------------------	---	---------------	---------

`k=i++` $\hat{=}$ `k=(i++)`

<code>i++</code>	Wert des Teilausdr:	5	Seiteneffekt:	$i = 6$
------------------	---------------------	---	---------------	---------

<code>k=i++</code>	Wert des Ausdrucks:	5	Seiteneffekt:	$k = 5$
--------------------	---------------------	---	---------------	---------

- ▶ Bsp.: `j=5; k=++j;`

<code>j=5</code>	Wert des Ausdrucks:	5	Seiteneffekt:	$j = 5$
------------------	---------------------	---	---------------	---------

`k=++j` $\hat{=}$ `k=(++j)`

<code>++j</code>	Wert des Teilausdr:	6	Seiteneffekt:	$j = 6$
------------------	---------------------	---	---------------	---------

<code>k=++j</code>	Wert des Ausdrucks:	6	Seiteneffekt:	$k = 6$
--------------------	---------------------	---	---------------	---------

Arithmetische Operatoren - Assoziativität I

- ▶ Bsp.: $a+b+c \hat{=} (a+b)+c$ (binäres “+“ linksassoziativ)
- ▶ $(a+b)+c$ bedeutet *nicht*, dass $a+b$ vor c berechnet wird!
- ▶ Gilt auch für andere linksassoz. Operatoren, z.B.:
`cout << a << b << c`
- ▶ Compiler darf Assoziativ- und Kommutativgesetze zu Optimierungszwecken *nur* anwenden, wenn Ergebnis unverändert.
- ▶ Gleitpunktarithmetik erfüllt i. allg. das Assoziativitätsgesetz *nicht*:

```
double eps=numeric_limits<double>::eps();
```

$$\begin{aligned} 1.0+eps/2-1.0 &\hat{=} (1.0+eps/2)-1.0 \\ &\hat{=} 1.0-1.0 \hat{=} 0 \end{aligned}$$

$$1.0-1.0+eps/2 \hat{=} (1.0-1.0)+eps/2 \hat{=} eps/2$$

Arithmetische Operatoren - Assoziativität II

- ▶ $k = i * -j \hat{=} k = (i * (-j)) \quad i = 5, j = 2 \rightarrow k = -10$
- ▶ $k = --i \hat{=} k = (--i) \quad i = 5 \rightarrow i = 4, k = 4$
- ▶ $k = - -j \hat{=} k = (-(-j)) \hat{=} k = j$
- ▶ $k = -i++ \hat{=} k = -(i++) \quad i = 4 \rightarrow i = 5, k = -4$
- ▶ $k = -+i+-++j \hat{=} k = ((- (+i)) + (- (++j)))$
 $i = 5, j = 2 \rightarrow i = 5, j = 3, k = -(+5) + -(3) = -8$
- ▶ $k = i+++++j \hat{=} k = (((i++)++) + j) \quad \text{syntakt. unzulässig}$
 $k = i++++ ++j \hat{=} k = ((i++) + (++j)) \quad \text{zulässig}$
 $i = 5, j = 3 \rightarrow i = 6, j = 4, k = 5 + 4 = 9$

Caveat

- ▶ $a/b * c \hat{=} (a/b) * c$, d.h. $\frac{a}{b}c$ und nicht $\frac{a}{bc}$, falls a, b, c Gleitpunktzahlen.

Arithmetische Operatoren - Division und Rest

- ▶ $i = 9/4 \hat{=} i = (9/4) \hat{=} i = 2$ ganzzahlige Division
- ▶ $i = 9\%4 \hat{=} i = (9\%4) \hat{=} i = 1$ Rest bei ganzzahl. Div.
- ▶ **Caveat:** Falls mindestens ein Operand bei / oder % negativ, Resultat implementierungsabhängig!
- ▶ $x = 9.0/4 \hat{=} x = (9.0/4) \hat{=} x = (9.0/4.0) \hat{=} x = 2.25$
reellwertige Division
In $9.0/4$ ist ein Operand vom Typ `double` und der andere vom Typ `int`.
Nach den impliziten Typumwandlungsregeln (*später!*) wird `int` in `double` umgewandelt. (`double` ist hier der längste beteiligte Gleitpunktooperand)
- ▶ $x = 9/4 \hat{=} x = (9/4) \hat{=} x = 2$ ganzzahlige Division
- ▶ $x = 1/2 * 0.5 \hat{=} x = ((1/2) * 0.5) \hat{=} x = (0 * 0.5) \hat{=} x = (0.0 * 0.5) \hat{=} x = 0.0$
- ▶ $x = 0.5 * 1/2 \hat{=} x = ((0.5 * 1) / 2) \hat{=} x = ((0.5 * 1.0) / 2) \hat{=} x = (0.5 / 2) \hat{=} x = (0.5 / 2.0) \hat{=} x = 0.25$

Arithmetische Operatoren - Zuweisungen

Zuweisungen sind *rechtsassoziativ* und haben untereinander den gleichen Vorrang.

Die Variable v habe vor Ausführung der Zuw. den Wert v_0 .

Der Wert des Zuweisungsausdrucks ist immer der Wert der linken Seite nach Ausführung.

Op.	Beispiele	Wert des Ausdrucks	Seiteneffekte
=	$v=a$	a	Zuweisung von a an v
+=	$v+=a$	$v_0 + a$	Addition von a zu v
-=	$v-=a$	$v_0 - a$	Subtraktion von a von v
=	$v=a$	$v_0 \cdot a$	Multiplikation von v mit a
/=	$v/=a$	v_0/a	Division von v durch a
%=	$v\%=n$	$v_0 \bmod n$	Zuw. des Div.rest an v

► $x=y=z=a \hat{=} x=(y=(z=a))$ bewirkt $x=a$; $y=a$; $z=a$;

► $x*=y+5 \hat{=} x*=(y+5) \hat{=} x=x*(y+5)$ *nicht* $x=x*y+5$

► $x*=y=5 \hat{=} x*=(y=5)$ bewirkt $y=5$ und $x=x*5$.

Typumwandlungsregeln (→ Inf.bl. 4/S.4, 5/S.1)

Motivation

- ▶ Reduktion der Anzahl der Operandentypen bei binären Operationen:
 1. Ziel: Gleiche Operandentypen, insbesondere gleich lang
⇒ Regeln für Typumwandlungen von ganzen in Gleitpunktzahlen und für Längenangleichungen
 2. Ziel: Nicht zu viele Sonderfälle für kurze Datentypen
⇒ Regeln für Integererweiterungen
(Anpassung kürzerer Datentypen auf `int`)
 3. Bei Zuweisungen Umwandlung in den Datentyp des linken Operanden, falls möglich

Explizite Typumwandlungen

- ▶ 4 „Cast“-Operatoren in C++ für unterschiedliche Situationen, 1 „Cast“-Operator in C für alle Situationen (auch in C++ vorhanden)

- ▶ *Bsp. für Casting:*

```
static_cast<double>(i)/j // Gleitpkt.division  
(double)i/j             // Gleitpkt.division
```


Typumwandlungsregeln II

Caveat - Beispiele

- ▶ Bestimmte implizite Typumwandlungen sind „gefährlich“, weil wertändernd

Bsp. `int` \rightarrow `double` ungefährlich
 `double` \rightarrow `int` gefährlich

- ▶ `int i; i=4.7 // i = 4 (Abschneiden)`
- ▶ `int i; double x; i=x // unzulässig für zu große |x|`
- ▶ *Kein* Syntaxfehler liegt vor, wenn für einen `int`-Parameter in einer Funktion eine `double`-Größe eingesetzt wird!

GNU-Compiler: `c++ -Wconversion`

Nicht in `-Wall` enthalten!

- ▶ (Teilweise) vorzeichenlose Arithmetik, falls in Ganzzahlausdrücken vorzeichenloser Operand

`int i; unsigned u; // i*u vorzeichenlos (≥ 0)`

Problem: Viele größenbestimmenden Operatoren und Funktionen liefern vorzeichenlose Ergebnisse (*später!*)

Mathematische Standardfunktionen I (Auszug)

Funktionen in cmath

#include <cmath> *erforderlich!*

x double, Ergebnistyp: double

Funktionen

Bedeutung

Bemerkungen

abs(x)

$|x|$

C: fabs(x) !

sqrt(x)

\sqrt{x}

besser als pow(x, 0.5) !

sin(x) cos(x)

trig. Fkt.

tan(x)

asin(x) acos(x)

trig. Umkehrfkt.

nicht arcsin usw.!

atan(x)

exp(x)

e^x

sinh(x) cosh(x)

hyperb. Fkt.

im allg. genauer als

tanh(x)

$0.5 * (\exp(x) - \exp(-x))$

log(x)

$\ln(x)$

nicht ln !

log10(x)

$\lg(x)$

floor(x)

$[x] = \max_{z \in \mathbb{Z}, z \leq x} z$

Ergebnis: double

ceil(x)

$-[-x] = \min_{z \in \mathbb{Z}, z \geq x} z$

Ergebnis: double

Mathematische Standardfunktionen II (Auszug)

Weitere Funktionen in cmath

`x, y: double, n: int, Ergebnis: double`

<i>Funktionen</i>	<i>Bedeutung</i>	<i>Bemerkungen</i>
<code>pow(x, y)</code>	x^y	C99: <code>pow(x, ±0.0) = 1</code>
<code>pow(x, n)</code>	x^n	$x < 0$ zulässig. C99: <code>pow(x, 0) = 1</code>
<code>atan2(y, x)</code>	Argument-funktion	y vor x wegen $\arctan(y/x)$ C99: <code>atan2(±0.0, -x) = ±π (x > 0)</code> <code>atan2(±0.0, +x) = ±0 (x > 0)</code> <code>atan2(±0.0, -0.0) = ±π</code> <code>atan2(±0.0, +0.0) = ±0</code>
<code>ldexp(x, n)</code>	$x \cdot 2^n$	$2^n = \text{ldexp}(1.0, n)$

- Zusätzlich gibt es jeweils gleichnamige Funktionen mit Parameter- und Ergebnistyp `float` bzw. `long double`.

Funktionen in cstdlib `#include <cstdlib>` erforderlich!

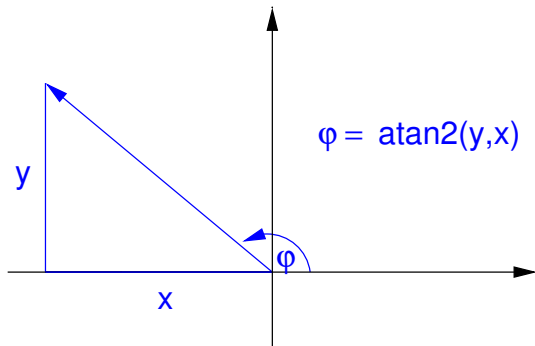
`n: int bzw. long Ergebnis: int bzw. long`

<i>Funktion</i>	<i>Bedeutung</i>
-----------------	------------------

<code>abs(n)</code>	$ n $
---------------------	-------

Mathematische Standardfunktionen III

Skizze zur Funktion atan2



- ▶ $\varphi = \text{atan2}(y, x) \in [-\pi, \pi]$
- ▶ C99: $\text{atan2}(+0.0, x) = \pi \quad (x < 0)$
 $\text{atan2}(-0.0, x) = -\pi \quad (x < 0)$

Fehlerbehandlung mit `errno`

- ▶ Viele Standardfunktionen setzen im Fehlerfall die globale, vordefinierte Variable `errno` (vom Datentyp `int`) auf einen Wert $\neq 0$. Dieser Wert ist fehlerspezifisch.
- ▶ Math. Standardfunktionen setzen nur zwei Fehlerwerte:
 `EDOM` Argument nicht im Definitionsbereich
 `ERANGE` Ergebnis nicht im Wertebereich darstellbar
`EDOM` und `ERANGE` sind konkrete ganze Zahlen, die in Include-Dateien mittels `#define` festgelegt sind.
- ▶ Durch `#include <cerrno>` wird `errno` im eigenen Programm deklariert, d.h. der Variablenname ist bekannt und damit benutzbar.
- ▶ `strerror(errno)` aus `<cstring>` liefert zur Fehlernummer *errno* einen fehlerspezifischen C-String.
- ▶ Damit andere Standardfunktionen das Ergebnis nicht verfälschen, sollte `errno` vor einem fraglichen Funktionsaufruf auf 0 gesetzt werden.