

2. Ganze Zahlen

Auswertung arithmetischer Ausdrücke, Assoziativität und Präzedenz, arithmetische Operatoren, Wertebereich der Typen `int`, `unsigned int`

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.
#include <iostream>

int main() {
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

91

92

`9 * celsius / 5 + 32`

- Arithmetischer Ausdruck,
- enthält drei Literale, eine Variable, drei Operatorsymbole

Wie ist der Ausdruck geklammert?

Präzedenz

Punkt vor Strichrechnung

`9 * celsius / 5 + 32`

bedeutet

`(9 * celsius / 5) + 32`

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz („binden stärker“) als additive Operatoren (`+`, `-`)

93

94

Assoziativität

Von links nach rechts

$9 * \text{celsius} / 5 + 32$

bedeutet

$((9 * \text{celsius}) / 5) + 32$

Regel 2: Assoziativität

Arithmetische Operatoren ($*$, $/$, $\%$, $+$, $-$) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Stelligkeit

Regel 3: Stelligkeit

Unäre Operatoren $+$, $-$ vor binären $+$, $-$.

$-3 - 4$

bedeutet

$(-3) - 4$

95

96

Klammerung

Jeder Ausdruck kann mit Hilfe der

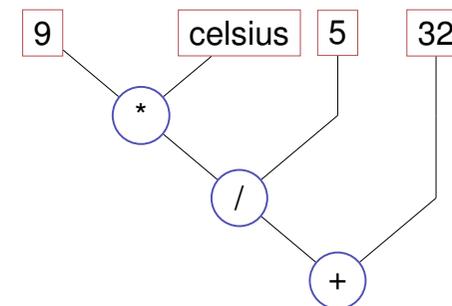
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

der beteiligten Operatoren eindeutig geklammert werden (Details im Skript).

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

$((9 * \text{celsius}) / 5) + 32$



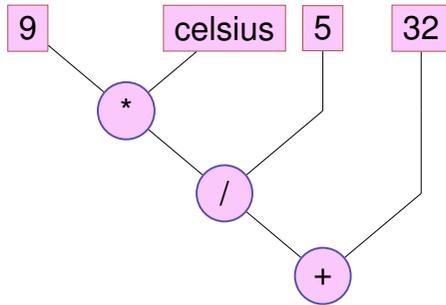
97

98

Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

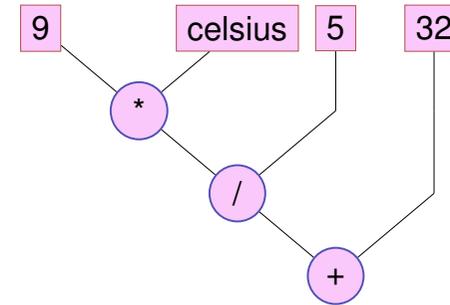
9 * celsius / 5 + 32



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

9 * celsius / 5 + 32



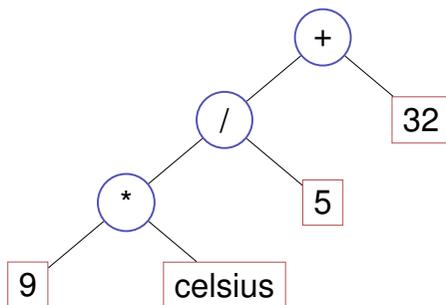
99

100

Ausdrucksbäume – Notation

Üblichere Notation: Wurzel oben

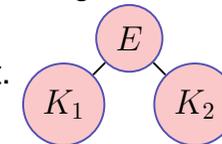
9 * celsius / 5 + 32



Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern

ausgewertet.



In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- „Guter Ausdruck“: jede gültige Reihenfolge führt zum gleichen Ergebnis.
- Beispiel eines „schlechten Ausdrucks“: $a*(a=2)$

101

102

Auswertungsreihenfolge

Richtlinie

Vermeide das Verändern von Variablen, welche im selben Ausdruck noch einmal verwendet werden!

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	-	1	16	rechts
Multiplikation	*	2	14	links
Division	/	2	14	links
Modulo	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

Alle Operatoren: [R-Wert ×] R-Wert → R-Wert

103

104

Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt: $a = b$ bedeutet Zuweisung von b (R-Wert) an a (L-Wert). Rückgabe: L-Wert
- Was bedeutet $a = b = c$?
- Antwort: Zuweisung rechtsassoziativ, also

$a = b = c \iff a = (b = c)$

Beispiel Mehrfachzuweisung:

$a = b = 0 \implies b=0; a=0$

105

Division

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematisch äquivalent... aber nicht in C++!

```
9 / 5 * celsius + 32
```

15 degrees Celsius are 47 degrees Fahrenheit

106

Präzisionsverlust

Richtlinie

- Auf möglichen Präzisionsverlust achten
- Potentiell verlustbehaftete Operationen möglichst spät durchführen um „Fehlereskalation“ zu vermeiden

107

Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

$5 / 2$ hat Wert 2, $5 \% 2$ hat Wert 1.

- Es gilt immer:

$(a / b) * b + a \% b$ hat den Wert von a .

108

Inkrement und Dekrement

- Erhöhen / Erniedrigen einer Zahl um 1 ist eine häufige Operation
- geht für einen L-Wert so:

$expr = expr + 1.$

Nachteile

- relativ lang
- $expr$ wird zweimal ausgewertet
 - Später: L-wertige Ausdrücke deren Auswertung „teuer“ ist
 - $expr$ könnte einen Effekt haben (aber sollte nicht, siehe Richtlinie)

109

In-/Dekrement Operatoren

Post-Inkrement

$expr++$

Wert von $expr$ wird um 1 erhöht, der *alte* Wert von $expr$ wird (als R-Wert) zurückgegeben

Prä-Inkrement

$++expr$

Wert von $expr$ wird um 1 erhöht, der *neue* Wert von $expr$ wird (als L-Wert) zurückgegeben

Post-Dekrement

$expr--$

Wert von $expr$ wird um 1 verringert, der *alte* Wert von $expr$ wird (als R-Wert) zurückgegeben

Prä-Dekrement

$--expr$

Wert von $expr$ wird um 1 verringert, der *neue* Wert von $expr$ wird (als L-Wert) zurückgegeben

110

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Präz	Assoz.	L/R-Werte
Post-Inkrement	<code>expr++</code>	1	17	links	L-Wert → R-Wert
Prä-Inkrement	<code>++expr</code>	1	16	rechts	L-Wert → L-Wert
Post-Dekrement	<code>expr--</code>	1	17	links	L-Wert → R-Wert
Prä-Dekrement	<code>--expr</code>	1	16	rechts	L-Wert → L-Wert

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n"; // 9
```

111

112

In-/Dekrement Operatoren

Ist die Anweisung

`++expr;` ← wir bevorzugen dies

äquivalent zu

`expr++;`?

Ja, aber

- Prä-Inkrement ist effizienter (alter Wert muss nicht gespeichert werden)
- Post-In/Dekrement sind die einzigen linksassoziativen unären Operatoren (nicht sehr intuitiv)

113

C++ vs. ++C

Eigentlich sollte unsere Sprache ++C heißen, denn

- sie ist eine Weiterentwicklung der Sprache C,
- während C++ ja immer noch das alte C liefert.

114

Arithmetische Zuweisungen

$a += b$
 \Leftrightarrow
 $a = a + b$

Analog für $-$, $*$, $/$ und $\%$

Arithmetische Zuweisungen

Gebrauch	Bedeutung
$+=$ $expr1 += expr2$	$expr1 = expr1 + expr2$
$-=$ $expr1 -= expr2$	$expr1 = expr1 - expr2$
$*=$ $expr1 *= expr2$	$expr1 = expr1 * expr2$
$/=$ $expr1 /= expr2$	$expr1 = expr1 / expr2$
$\%=$ $expr1 \% expr2$	$expr1 = expr1 \% expr2$

Arithmetische Zuweisungen werten $expr1$ nur einmal aus.
Zuweisungen haben Präzedenz 4 und sind rechtsassoziativ

115

116

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$b_n b_{n-1} \dots b_1 b_0$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Beispiel: **101011** entspricht 43.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

117

Rechentricks

■ Abschätzung der Grössenordnung von Zweierpotenzen²:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3$.
 $2^{20} = 1\text{Mi} \approx 10^6$,
 $2^{30} = 1\text{Gi} \approx 10^9$,
 $2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}$.
 $2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}$.

²Dezimal vs. Binäre Einheiten: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) - mega (M, Mi) - giga (G, Gi) - tera (T, Ti) - peta (P, Pi) - exa (E, Ei)

118

Hexadezimale Zahlen

Zahlen zur Basis 16. Darstellung

$$h_n h_{n-1} \dots h_1 h_0$$

entspricht der Zahl

$$h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0.$$

Schreibweise in C++: vorangestelltes 0x

Beispiel: 0xff entspricht 255.

Hex Nibbles		
hex	bin	dec
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

119

Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits. Die Zahlen 1, 2, 4 und 8 repräsentieren Bits 0, 1, 2 und 3.
- „Kompakte Darstellung von Binärzahlen“.

120

Wozu Hexadezimalzahlen?

„Für Programmierer und Techniker“ (Auszug aus der Bedienungsanleitung des Schachcomputers *Mephisto II*, 1981)

Beispiele:

a) Anzeige **8200**
MEPHISTO ist mit genau 2 Bauern-Einheiten im Vorteil.

b) Anzeige **7F00**
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in **hexadezimaler Schreibweise**. Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F (A = 10, B = 11, ..., F = 15).
Für mathematisch vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (A \times 16^3) + (B \times 16^2) + (C \times 16^1) + (D \times 16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.
Eine Bauereinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten.
Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

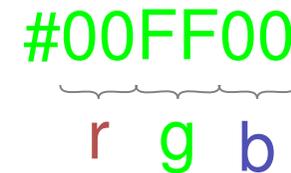
Beispiele:

c) Anzeige **805E**
(E=14) Umrechnung nach folgendem Verfahren:
 $(14 \times 16^3) + (5 \times 16^2) + (0 \times 16^1) + (0 \times 16^0) = 14 \times 4096 + 0 + 0 = +94 \text{ Punkte.}$

d) Anzeige **7F80**
(7=-1; F=15) Umrechnung wie folgt:
 $(0 \times 16^3) + (8 \times 16^2) + (15 \times 16^1) - (1 \times 16^0) = 0 + 128 + 3840 - 4096 =$

http://www.zaachetta.net/default.aspx?Kategorie=ECHIQUERS&Page=documentations

Beispiel: Hex-Farben



122

Wertebereich des Typs int

```
// Output the smallest and the largest value of type int.
#include <iostream>
#include <limits>

int main() {
    std::cout << "Minimum int value is "
              << std::numeric_limits<int>::min() << ".\n"
              << "Maximum int value is "
              << std::numeric_limits<int>::max() << ".\n";
    return 0;
}
```

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Woher kommen diese Zahlen?

123

Wertebereich des Typs int

- Repräsentation mit B Bits. Wertebereich umfasst die 2^B ganzen Zahlen:

$$\{-2^{B-1}, -2^{B-1} + 1, \dots, -1, 0, 1, \dots, 2^{B-1} - 2, 2^{B-1} - 1\}$$

Woher kommt gerade diese Aufteilung?

- Auf den meisten Plattformen $B = 32$
- Für den Typ `int` garantiert C++ $B \geq 16$
- Hintergrund: Abschnitt 2.2.8 (Binary Representation) im Skript

124

Überlauf und Unterlauf

- Arithmetische Operationen (+, -, *) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

```
power8.cpp:  $15^8 = -1732076671$ 
```

```
power20.cpp:  $3^{20} = -808182895$ 
```

- Es gibt *keine Fehlermeldung!*

125

Der Typ unsigned int

- Wertebereich

$$\{0, 1, \dots, 2^B - 1\}$$

- Alle arithmetischen Operationen gibt es auch für `unsigned int`.
- Literale: `1u`, `17u` ...

126

Gemischte Ausdrücke

- Operatoren können Operanden verschiedener Typen haben (z.B. `int` und `unsigned int`).

```
17 + 17u
```

- Solche gemischten Ausdrücke sind vom „allgemeineren“ Typ `unsigned int`.
- `int`-Operanden werden *konvertiert* nach `unsigned int`.

Konversion

int Wert	Vorzeichen	unsigned int Wert
x	≥ 0	x
x	< 0	$x + 2^B$

Bei Zweierkomplementdarstellung (folgt) passiert dabei intern gar nichts

127

128

Konversion “andersherum”

Die Deklaration

```
int a = 3u;
```

konvertiert `3u` nach `int`.

Der Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig.

Vorzeichenbehaftete Zahlendarstellung

- Soweit klar (hoffentlich): Binäre Zahlendarstellung ohne Vorzeichen, z.B.

$$[b_{31}b_{30} \dots b_0]_u \hat{=} b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \dots + b_0$$

- Nun offensichtlich notwendig: Verwende ein Bit für das Vorzeichen.
- Suche möglichst konsistente Lösung

Die Darstellung mit Vorzeichen sollte möglichst viel mit der vorzeichenlosen Lösung „gemein haben“. Positive Zahlen sollten sich in beiden Systemen algorithmisch möglichst gleich verhalten.

129

130

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

$$\begin{array}{r}
 2 \qquad 0010 \\
 +3 \qquad +0011 \\
 \hline
 5 \qquad 0101
 \end{array}$$

Einfache Subtraktion

$$\begin{array}{r}
 5 \qquad 0101 \\
 -3 \qquad -0011 \\
 \hline
 2 \qquad 0010
 \end{array}$$

131

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

$$\begin{array}{r}
 7 \qquad 0111 \\
 +9 \qquad +1001 \\
 \hline
 16 \qquad (1)0000
 \end{array}$$

Negative Zahlen?

$$\begin{array}{r}
 5 \qquad 0101 \\
 +(-5) \qquad \text{????} \\
 \hline
 0 \qquad (1)0000
 \end{array}$$

132

Rechnen mit Binärzahlen (4 Stellen)

Einfacher: -1

$$\begin{array}{r}
 1 \qquad 0001 \\
 +(-1) \qquad 1111 \\
 \hline
 0 \qquad (1)0000
 \end{array}$$

Nutzen das aus:

$$\begin{array}{r}
 3 \qquad 0011 \\
 +? \qquad +???? \\
 \hline
 -1 \qquad 1111
 \end{array}$$

133

Rechnen mit Binärzahlen (4 Stellen)

Invertieren!

$$\begin{array}{r}
 3 \qquad 0011 \\
 +(-4) \qquad +1100 \\
 \hline
 -1 \qquad 1111 \hat{=} 2^B - 1
 \end{array}$$

$$\begin{array}{r}
 a \qquad a \\
 +(-a-1) \qquad \bar{a} \\
 \hline
 -1 \qquad 1111 \hat{=} 2^B - 1
 \end{array}$$

134

Rechnen mit Binärzahlen (4 Stellen)

- Negation: Inversion und Addition von 1

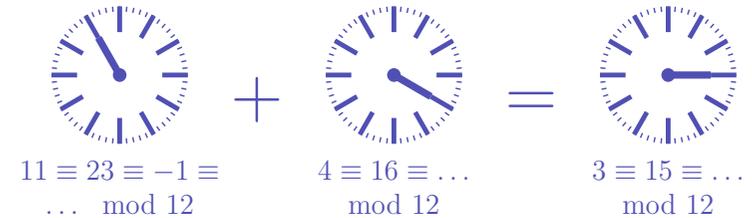
$$-a \hat{=} \bar{a} + 1$$

- Wrap-around Semantik (Rechnen modulo 2^B)

$$-a \hat{=} 2^B - a$$

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis³



³Die Arithmetik funktioniert auch mit Dezimalzahlen (und auch für die Multiplikation)

135

136

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100	100	-4
5	101		
6	110		
7	111		

Das höchste Bit entscheidet über das Vorzeichen *und* es trägt zum Zahlwert bei.

Zweierkomplement

- Negation durch bitweise Negation und Addition von 1.

$$-2 = -[0010] = [1101] + [0001] = [1110]$$

- Arithmetik der Addition und Subtraktion *identisch* zur vorzeichenlosen Arithmetik.

$$3 - 2 = 3 + (-2) = [0011] + [1110] = [0001]$$

- Intuitive „Wrap-Around“ Konversion negativer Zahlen.

$$-n \rightarrow 2^B - n$$

- Wertebereich: $-2^{B-1} \dots 2^{B-1} - 1$

137

138