

GANZE ZAHLEN

Die Typen `int`, `unsigned int`;
Auswertung arithmetischer Ausdrücke, arithmetische Operatoren

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;
}
```

15 degrees Celsius are 59 degrees Fahrenheit.

9 * celsius / 5 + 32

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

Wie ist der Ausdruck geklammert?

Assoziativität und Präzedenz

Regel 1: Punkt- vor Strichrechnung

9 * celsius / 5 + 32

bedeutet

(9 * celsius / 5) + 32

Assoziativität und Präzedenz

Regel 2: Von links nach rechts

(9 * celsius / 5) + 32

bedeutet

((9 * celsius) / 5) + 32

Assoziativität und Präzedenz

Regel 1: Multiplikative Operatoren (`*`, `/`, `%`) haben höhere **Präzedenz** ("binden stärker") als additive Operatoren (`+`, `-`)

Regel 2: Arithmetische Operatoren (`*`, `/`, `%`, `+`, `-`) sind **linksassoziativ** (bei gleicher Präzedenz erfolgt Auswertung von links nach rechts)

Assoziativität und Präzedenz

Regel 3: Unäre +,- vor binären +,-

bedeutet $-3 - 4$

$(-3) - 4$

Assoziativität und Präzedenz

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 * celsius / 5 + 32

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

9 * celsius / 5 + 32

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

9 * celsius / 5 + 32

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

9 * celsius / 5 + 32

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

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

9 * celsius

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

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

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

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

9 * celsius

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

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

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

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

9 * celsius

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

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

Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:

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

9 * celsius

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

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

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

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

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

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

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

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

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:

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

9 * celsius

(9 * celsius) / 5

((9 * celsius) / 5) + 32

Auswertungsreihenfolge


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

Kinder

Knoten

Auswertungsreihenfolge

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



Kinder


Knoten

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

- o "Guter" Ausdruck: jede gültige Reihenfolge führt zum gleichen Ergebnis

Auswertungsreihenfolge

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



Kinder

Knoten

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

- o "Schlechter" Ausdruck: Beispiel in der Uebungsserie (sollte man vermeiden).

Arithmetische Operatoren

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

Arithmetische Operatoren

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

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

Division und Modulus

- o Operator / realisiert *ganzzahlige* Division:

`5 / 2 hat Wert 2`
- o in **fahrenheit.C**:

`9 * celsius / 5 + 32:`
`15 degrees Celsius are 59 degrees Fahrenheit`

`9 / 5 * celsius + 32:`
`15 degrees Celsius are 47 degrees Fahrenheit`

Division und Modulus

- o Operator / realisiert *ganzzahlige* Division:

`5 / 2 hat Wert 2`
- o in **fahrenheit.C**:

`9 * celsius / 5 + 32:`
`15 degrees Celsius are 59 degrees Fahrenheit`

`9 / 5 * celsius + 32:`
`15 degrees Celsius are 47 degrees Fahrenheit`

Division und Modulus

- Operator / realisiert *ganzzahlige* Division:
 - 5 / 2 hat Wert 2
- in **fahrenheit.C**:
 - 9 * celsius / 5 + 32:
15 degrees Celsius are 59 degrees Fahrenheit
 - 1 * celsius + 32:
15 degrees Celsius are 47 degrees Fahrenheit

Division und Modulus

- Operator / realisiert *ganzzahlige* Division:
 - 5 / 2 hat Wert 2
- in **fahrenheit.C**:
 - 9 * celsius / 5 + 32:
15 degrees Celsius are 59 degrees Fahrenheit
 - 1 * 15 + 32:
15 degrees Celsius are 47 degrees Fahrenheit

Division und Modulus

- Operator / realisiert *ganzzahlige* Division:
 - 5 / 2 hat Wert 2
- in **fahrenheit.C**:
 - 9 * celsius / 5 + 32:
15 degrees Celsius are 59 degrees Fahrenheit
 - 47:
15 degrees Celsius are 47 degrees Fahrenheit

Division und Modulus

- Modulus-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

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 (Effekte!)

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

	Gebrauch	Effekt / Wert
Post-Inkrement	<code>expr++</code>	Wert von <code>expr</code> wird um 1 erhöht, der <i>alte</i> Wert von <code>expr</code> wird (als R-Wert) zurückgegeben
Prä-Inkrement	<code>++expr</code>	Wert von <code>expr</code> wird um 1 erhöht, der <i>neue</i> Wert von <code>expr</code> wird (als L-Wert) zurückgegeben
Post-Dekrement	<code>expr--</code>	Wert von <code>expr</code> wird um 1 erniedrigt, der <i>alte</i> Wert von <code>expr</code> wird (als R-Wert) zurückgegeben
Prä-Dekrement	<code>--expr</code>	Wert von <code>expr</code> wird um 1 erniedrigt, der <i>neue</i> Wert von <code>expr</code> wird (als L-Wert) zurückgegeben

In-/Dekrement-Operatoren

Beispiel:

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

In-/Dekrement-Operatoren

Ist die Anweisung

`++expr;`

Wir bevorzugen diese Variante.

äquivalent zu

`expr++;` ?

o Ja, aber...

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

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
<code>+=</code>	<code>expr1 += expr2</code>	<code>expr1 = expr1 + expr2</code>
<code>-=</code>	<code>expr1 -= expr2</code>	<code>expr1 = expr1 - expr2</code>
<code>*=</code>	<code>expr1 *= expr2</code>	<code>expr1 = expr1 * expr2</code>
<code>/=</code>	<code>expr1 /= expr2</code>	<code>expr1 = expr1 / expr2</code>
<code>%=</code>	<code>expr1 %= expr2</code>	<code>expr1 = expr1 % expr2</code>

Arithmetische Zuweisungen werten `expr1` nur einmal aus.

Wertebereich des Typs `int`

- o b -Bit-Repräsentierung: Wertebereich umfasst die ganzen Zahlen

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

- o Auf den meisten Plattformen: $b = 32$

Wertebereich des Typs `int`

```
// Program: limits.cpp
// 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;
}
```

z.B. Minimum int value is -2147483648
Maximum int value is 2147483647

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!

Der Typ `unsigned int`

- Wertebereich
 - $\{0, 1, \dots, 2^b - 1\}$
- Alle arithmetischen Operatoren gibt es auch für `unsigned int`
- Literale: `1u`, `17u`,...

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

<code>int</code> Wert	Vorzeichen	<code>unsigned int</code> Wert
x	≥ 0	x
x	< 0	$x + 2^b$

Bei Zweierkomplement-Darstellung passiert dabei intern gar nichts!

Konversion "andersherum"

- Die Deklaration
 - `int a = 3u`
 konvertiert `3u` nach `int` (Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig)