

Ganze Zahlen

Celsius to Fahrenheit

```
// Program: fahrenheit.C
// 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.

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

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

Wie ist der Ausdruck geklammert?

Assoziativität und Präzedenz

Regel 1: Punkt- vor Strichrechnung

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

bedeutet

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

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 `+`, `-`

`-3 - 4`
bedeutet

`(-3) - 4`

Assoziativität und Präzedenz

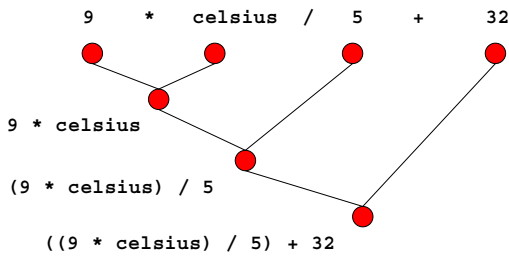
Jeder Ausdruck kann mit Hilfe der

- o Assoziativitäten
- o Präzedenzen
- o Stelligkeiten (Anzahl Operanden)

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

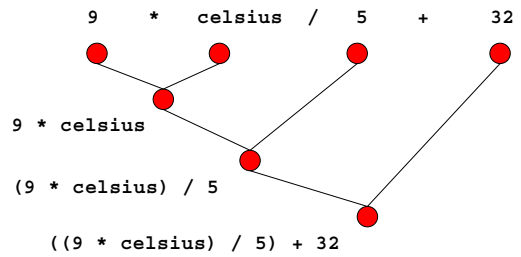
Auswertungsreihenfolge

Von oben nach unten im Ausdrucksbaum:



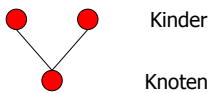
Auswertungsreihenfolge

Reihenfolge ist nicht eindeutig bestimmt:



Auswertungsreihenfolge

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



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

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

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 Modulus-Operator berechnet *Rest* der ganzzahligen Division

`5 / 2` hat Wert 2

- o Es gilt immer:

`(a / b) * b + a % b` hat den Wert von a

- o Falls **a** oder **b** negativ, so gibt es zwei Möglichkeiten:

`-5 / 2` hat Wert -2

`-5 % 2` hat Wert -1

Rundung gegen 0

Division und Modulus

- o Modulus-Operator berechnet *Rest* der ganzzahligen Division

`5 / 2` hat Wert 2

- o Es gilt immer:

`(a / b) * b + a % b` hat den Wert von a

- o Falls **a** oder **b** negativ, so gibt es zwei Möglichkeiten:

`-5 / 2` hat Wert -3

`-5 % 2` hat Wert 1

Rundung gegen unendlich

Division und Modulus

- o Modulus-Operator berechnet *Rest* der ganzzahligen Division

`5 / 2` hat Wert 2

`5 % 2` hat Wert 1

- o Es gilt immer:

`(a / b) * b + a % b` hat den Wert von a

- o Meistens: falls **a** oder **b** negativ, so wird gegen 0 gerundet;

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"; // outputs 8
std::cout << a++ << "\n"; // outputs 8
std::cout << a << "\n"; // outputs 9
```

In/Dekrement-Operatoren

Ist die Anweisung

```
++expr;
```

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
+=	<i>expr1</i> += <i>expr2</i>	<i>expr1</i> = <i>expr1</i> + <i>expr2</i>
-=	<i>expr1</i> -= <i>expr2</i>	<i>expr1</i> = <i>expr1</i> - <i>expr2</i>
*=	<i>expr1</i> *= <i>expr2</i>	<i>expr1</i> = <i>expr1</i> * <i>expr2</i>
/=	<i>expr1</i> /= <i>expr2</i>	<i>expr1</i> = <i>expr1</i> / <i>expr2</i>
%=	<i>expr1</i> %= <i>expr2</i>	<i>expr1</i> = <i>expr1</i> % <i>expr2</i>

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.C
// 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.c`: $15^8 = -1732076671$

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

- 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"

- o Die Deklaration

```
int a = 3u
```

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

Wahrheitswerte

Boole'sche Funktionen

- o Boole'sche Funktion:

$f: \{0,1\}^2 \rightarrow \{0,1\}$

- o 0 entspricht "falsch"
- o 1 entspricht "wahr"

x	y	AND (x,y)
0	0	0
0	1	0
1	0	0
1	1	1

Boole'sche Funktionen

- o Boole'sche Funktion:

$f: \{0,1\}^2 \rightarrow \{0,1\}$

- o 0 entspricht "falsch"
- o 1 entspricht "wahr"

x	y	OR (x,y)
0	0	0
0	1	1
1	0	1
1	1	1

Boole'sche Funktionen

- Boole'sche Funktion:

x	NOT (x)
0	1
1	0

$f: \{0,1\}^1 \rightarrow \{0,1\}$

- 0 entspricht "falsch"
- 1 entspricht "wahr"

Vollständigkeit

- AND, OR, NOT sind die in C++ verfügbaren Boole'schen Funktionen
- Alle anderen *binären* boole'schen Funktionen sind daraus erzeugbar

x	y	XOR (x,y)
0	0	0
0	1	1
1	0	1
1	1	0

$XOR(x,y) =$

$AND(OR(x,y), NOT(AND(x,y)))$

Vollständigkeit: Beweis

- Identifiziere binäre boole'sche Funktion mit ihrem *charakteristischem Vektor*:

x	y	XOR (x,y)
0	0	0
0	1	1
1	0	1
1	1	0

Charakteristischer Vektor: **0110**

$XOR = f_{0110}$

Vollständigkeit: Beweis

- Schritt 1: erzeuge die Funktionen f_{0001} , f_{0010} , f_{0100} , f_{1000}

$f_{0001} = AND(x, y)$

$f_{0010} = AND(x, NOT(y))$

$f_{0100} = AND(y, NOT(x))$

$f_{1000} = NOT(OR(x, y))$

Vollständigkeit: Beweis

- Schritt 2: erzeuge alle Funktionen

$$f_{1101} = \text{OR}(f_{1000}, \text{OR}(f_{0100}, f_{0001}))$$

$$f_{0000} = 0$$

Der Typ `bool`

- Repräsentiert Wahrheitswerte
- Literale `true` und `false`
- Wertebereich `{true, false}`

```
bool b = true; // Variable mit Wert true
```

Logische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Logisches Und (AND)	&&	2	6	links
Logisches Oder (OR)		2	5	links
Logisches Nicht (NOT)	!	1	16	rechts

```
bool (x bool) -> bool
```

```
R-Wert (x R-Wert) -> R-Wert
```

Relationale Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Kleiner	<	2	11	links
Grösser	>	2	11	links
Kleiner oder gleich	<=	2	11	links
Grösser oder gleich	>=	2	11	links
Gleich	==	2	10	links
Ungleich	!=	2	10	links

```
Zahlentyp x Zahlentyp -> bool
```

```
R-Wert x R-Wert -> R-Wert
```

Präzedenzen

Binäre arithmetische Operatoren
binden stärker als
Relationale Operatoren,
und diese binden stärker als
binäre logische Operatoren.

```
7 + x < y && y != 3 * z
```

Präzedenzen

Binäre arithmetische Operatoren
binden stärker als
Relationale Operatoren,
und diese binden stärker als
binäre logische Operatoren.

```
((7 + x) < y) && (y != (3 * z))
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken* Operanden zuerst aus
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet

```
x != 0 && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken* Operanden zuerst aus
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet

x hat Wert 1:

```
true && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken* Operanden zuerst aus
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet

x hat Wert 0:

```
false && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken* Operanden zuerst aus
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet

x hat Wert 0:

```
false
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken* Operanden zuerst aus
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet

```
x != 0 && z / x > y
```

keine Division durch Null!