

## 2. Ganze Zahlen

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

# Beispiel: power8.cpp

```
int a; // Input
int r; // Result

std::cout << "Compute a^8 for a = ?";
std::cin >> a;

r = a * a; // r = a^2
r = r * r; // r = a^4

std::cout << "a^8 = " << r*r << '\n';
```

# 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;
}
```

# 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;
}
```

9 \* celsius / 5 + 32

- Arithmetischer Ausdruck,

9 \* celsius / 5 + 32

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

9 \* celsius / 5 + 32

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

9 \* celsius / 5 + 32

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



9 \* celsius / 5 + 32

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

Wie ist der Ausdruck geklammert?

# Präzedenz

## Punkt vor Strichrechnung

`9 * celsius / 5 + 32`

bedeutet

`(9 * celsius / 5) + 32`

# Präzedenz

## Regel 1: Präzedenz

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

# Assoziativität

Von links nach rechts

`9 * celsius / 5 + 32`

bedeutet

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

# Assoziativität

## 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

$$(-3) - 4$$

# Klammerung

Jeder Ausdruck kann mit Hilfe der

- Assoziativitäten
- Präzedenzen
- Stelligkeiten

der beteiligten Operatoren eindeutig geklammert werden.

# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

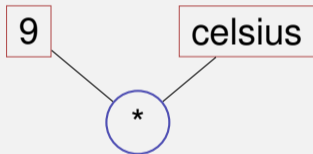
`9 * celsius / 5 + 32`



# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

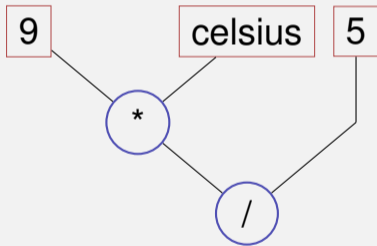
`(9 * celsius) / 5 + 32`



# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

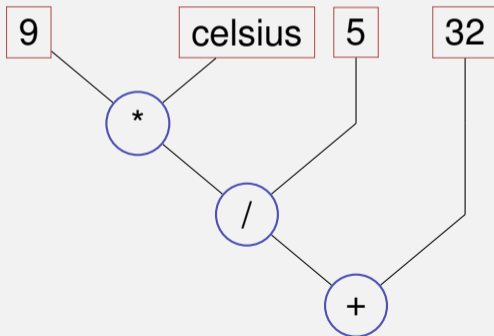
`((9 * celsius) / 5) + 32`



# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

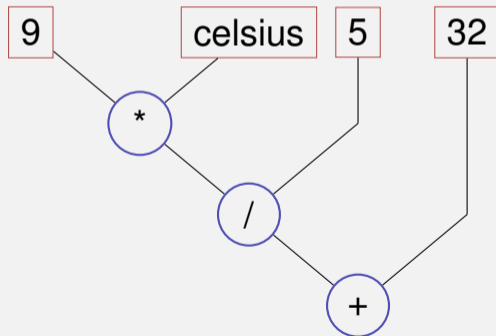
`((9 * celsius) / 5) + 32)`



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

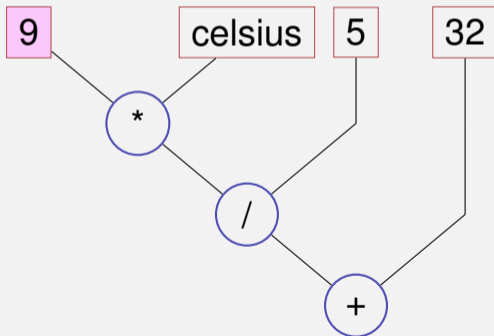
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

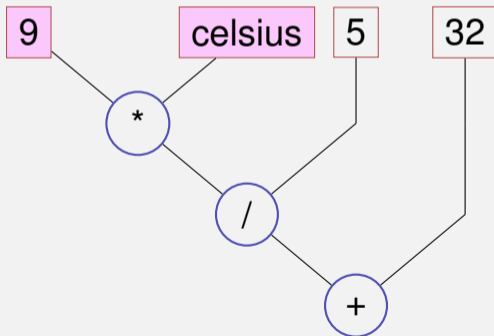
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

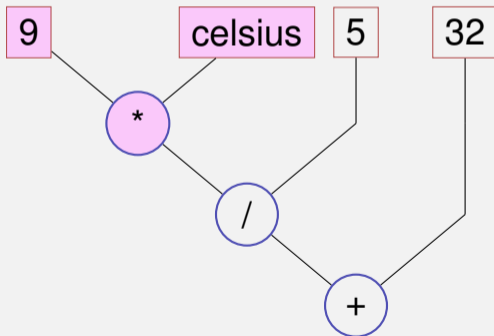
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

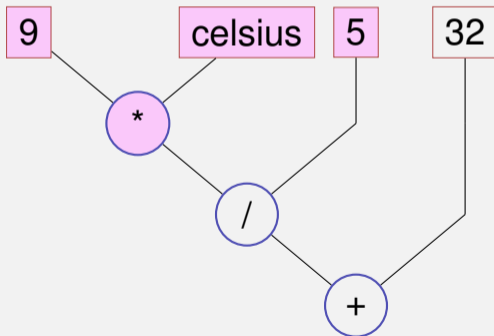
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

9 \* celsius / 5 + 32

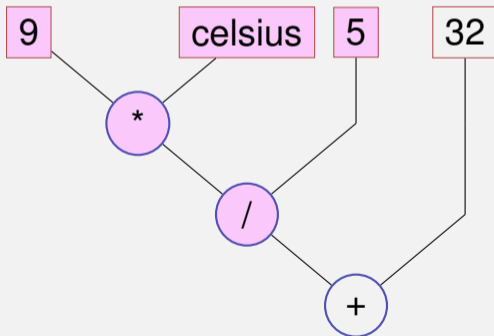




# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

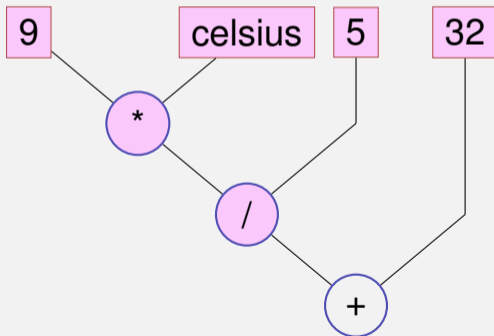
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

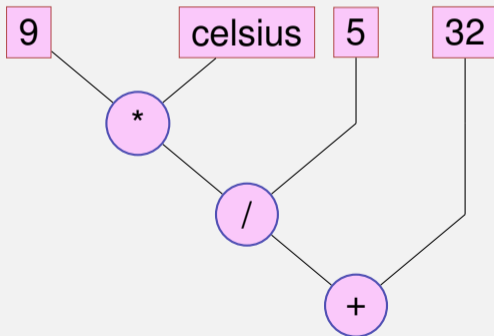
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

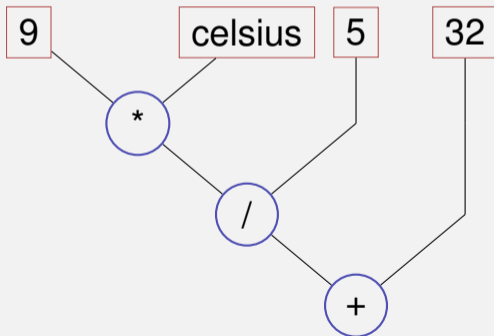
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

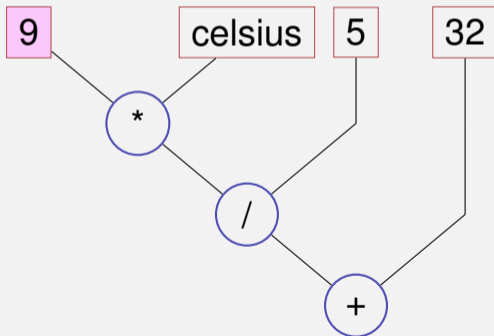
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

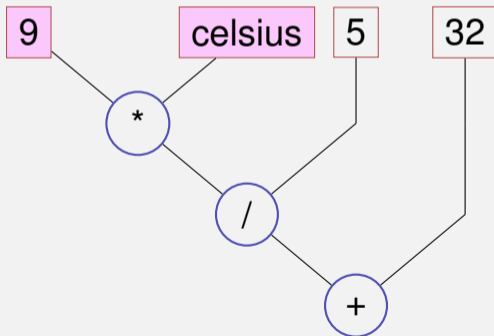
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

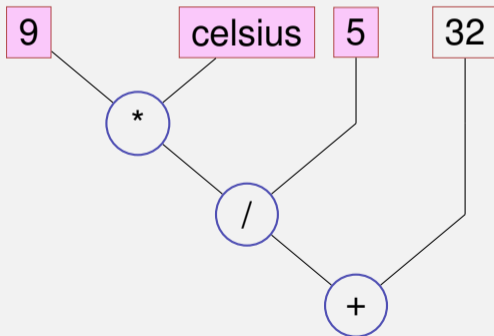
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

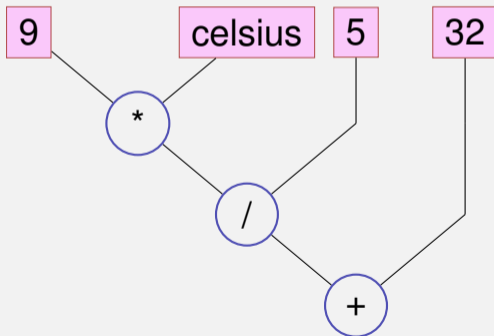
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

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

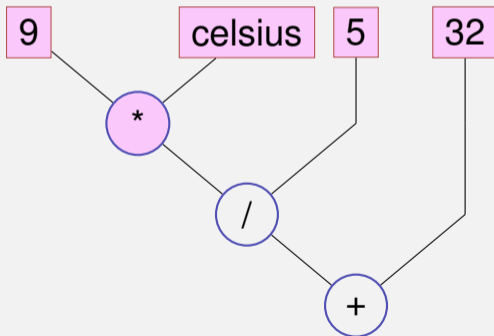




# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

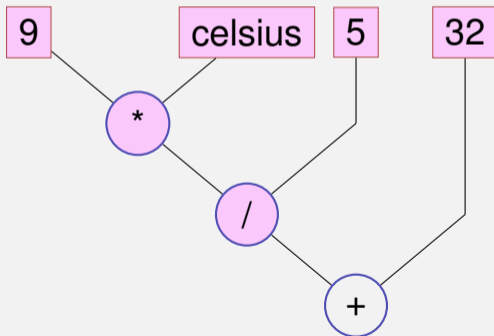
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

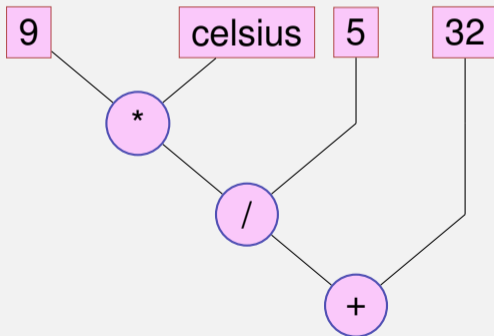
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

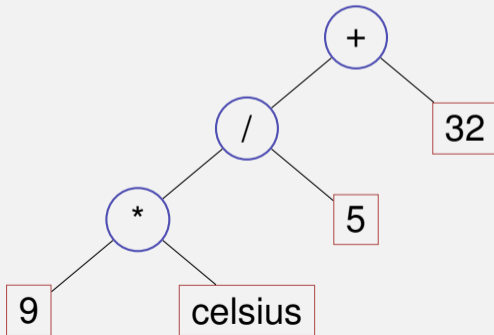
$$9 * \text{celsius} / 5 + 32$$



# Ausdrucksbäume – Notation

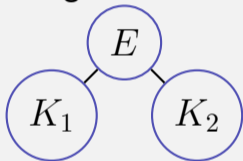
Üblichere Notation: Wurzel oben

9 \* celsius / 5 + 32



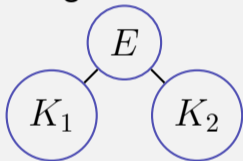
# Auswertungsreihenfolge – formaler

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



# Auswertungsreihenfolge – formaler

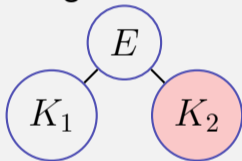
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

# Auswertungsreihenfolge – formaler

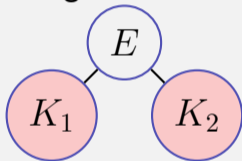
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

# Auswertungsreihenfolge – formaler

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

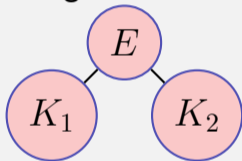


C++: anzuwendende gültige Reihenfolge nicht spezifiziert.



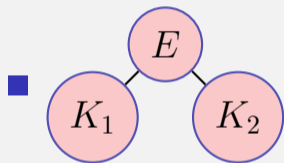
# Auswertungsreihenfolge – formaler

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



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

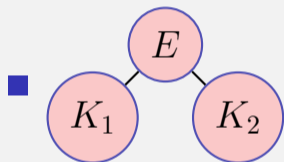
# Auswertungsreihenfolge – formaler



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

- „Guter Ausdruck“: jede gültige Reihenfolge führt zum gleichen Ergebnis.

# Auswertungsreihenfolge – formaler



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

- Beispiel eines „schlechten Ausdrucks“:  $a*(a=2)$

# 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

# Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt:  $a = b$  bedeutet  
Zuweisung von  $b$  (R-Wert) an  $a$  (L-Wert).  
Rückgabe: L-Wert

# Einschub: Zuweisungsausdruck – nun genauer

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

# 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

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



# Einschub: Zuweisungsausdruck – nun genauer

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

Beispiel Mehrfachzuweisung:

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

# Division

- Operator / realisiert ganzzahlige Division

5 / 2 hat Wert 2

# Division

- Operator / realisiert ganzzahlige Division

5 / 2 hat Wert 2

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

# Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

# Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematisch äquivalent...

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

# Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematisch äquivalent...

```
1 * celsius + 32
```

# Division

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
15 + 32
```

# Division

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
47
```



# Division

- 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
```

# Präzisionsverlust

## Richtlinie

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

# Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

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

# 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  $a$ .

# 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  $a$ .

- Daraus lässt sich herleiten, welche Ergebnisse Division und Modulo mit negativen Zahlen ergeben (müssen)

# Inkrement und Dekrement

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

```
expr = expr + 1.
```

# Inkrement und Dekrement

```
expr = expr + 1.
```

Nachteile

- relativ lang

# Inkrement und Dekrement

```
expr = expr + 1.
```

## Nachteile

- relativ lang
- `expr` wird zweimal ausgewertet
  - Später: L-wertige Ausdrücke deren Auswertung „teuer“ ist



# Inkrement und Dekrement

```
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)

# In-/Dekrement Operatoren

## Post-Inkrement

`expr++`

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

# In-/Dekrement Operatoren

## Prä-Inkrement

`++expr`

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

# In-/Dekrement Operatoren

## Post-Dekrement

```
expr--
```

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

# In-/Dekrement Operatoren

## Prä-Dekrement

--expr

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

# In-/Dekrement Operatoren

## Beispiel

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

# In-/Dekrement Operatoren

## Beispiel

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

# In-/Dekrement Operatoren

## Beispiel

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



# In-/Dekrement Operatoren

## Beispiel

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

# Binäre Zahlendarstellungen

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

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

entspricht der Zahl  $b_n \cdot 2^n + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

# Binäre Zahlendarstellungen

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

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

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

# 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

# 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  $32+8+2+1$ .

# 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.

# 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)*

# 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;
}
```



# 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.
```

# 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?

# Wertebereich des Typs `int`

- Repräsentation mit  $B$  Bits. Wertebereich

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

# Wertebereich des Typs `int`

- Repräsentation mit  $B$  Bits. Wertebereich

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

- Auf den meisten Plattformen  $B = 32$

# Wertebereich des Typs `int`

- Repräsentation mit  $B$  Bits. Wertebereich

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

- Für den Typ `int` garantiert C++  $B \geq 16$

# Überlauf und Unterlauf

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

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

- Es gibt *keine Fehlermeldung!*

# Der Typ `unsigned int`

- Wertebereich

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

- Alle arithmetischen Operationen 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
-----------------------	------------	--------------------------------

$x$	$\geq 0$	$x$
$x$	$< 0$	$x + 2^B$

# Konversion

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

Dank cleverer Repräsentation (Zweierkomplement – behandeln wir nicht) muss intern gar nicht addiert werden