

## 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;
}
```

# 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

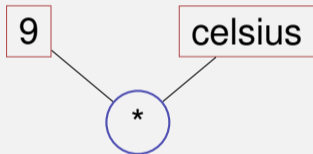
Klammerung ergibt Ausdrucksbaum

`9 * celsius / 5 + 32`

# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

`(9 * celsius) / 5 + 32`

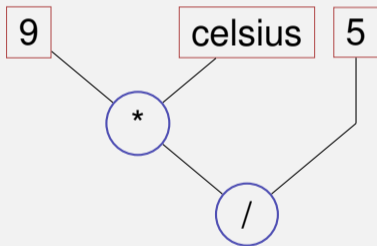




# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

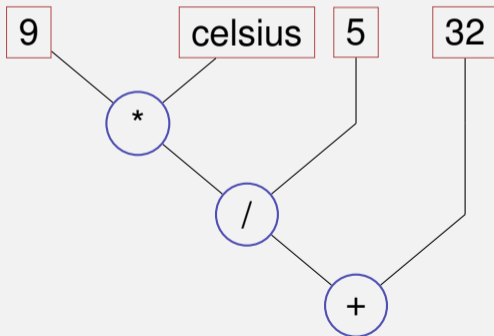
`((9 * celsius) / 5) + 32`



# Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

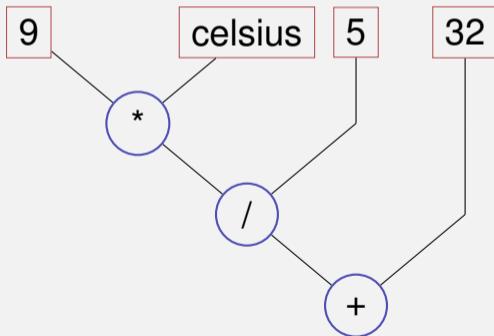
`((9 * celsius) / 5) + 32)`



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

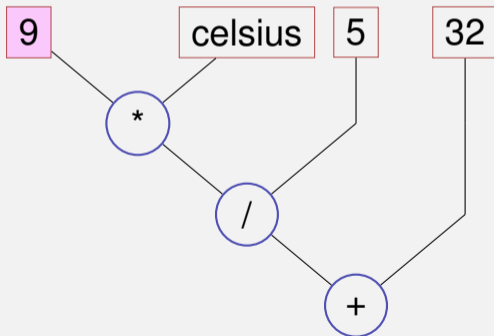
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

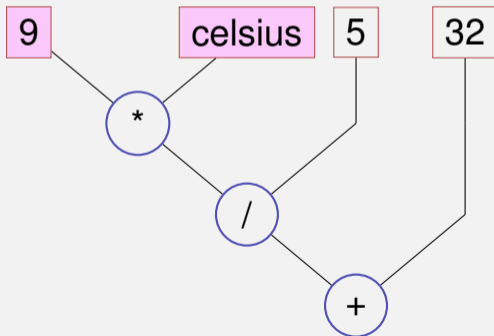
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

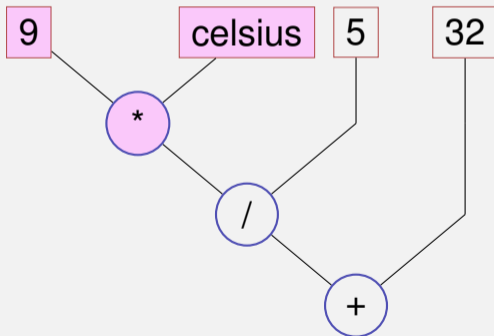
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

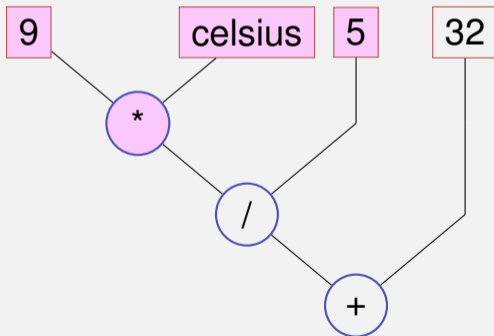
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

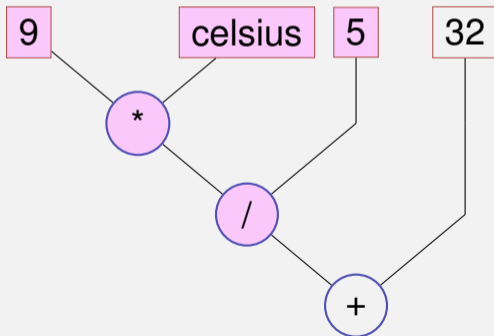
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

9 \* celsius / 5 + 32

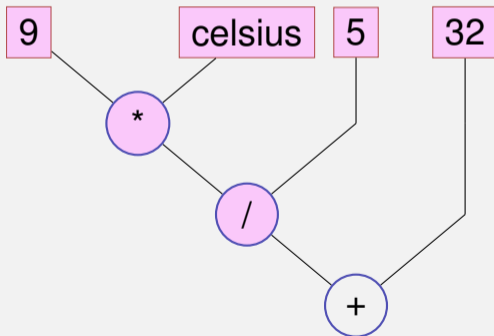




# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

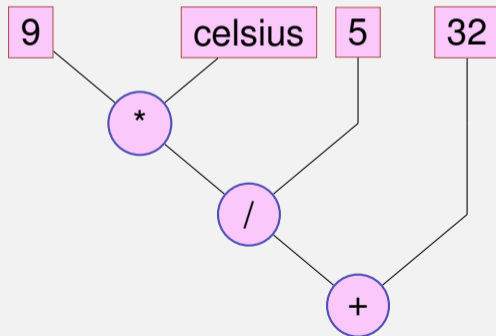
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

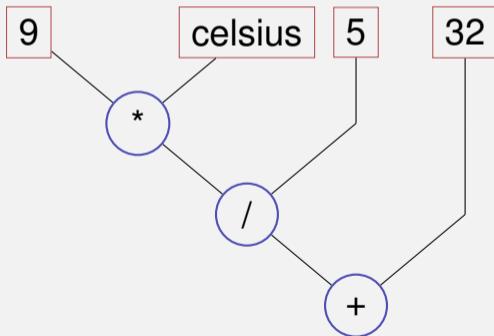
9 \* celsius / 5 + 32



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

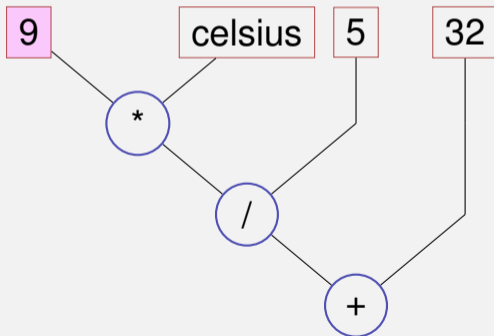
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

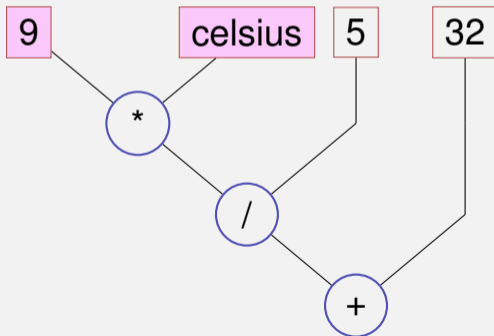
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

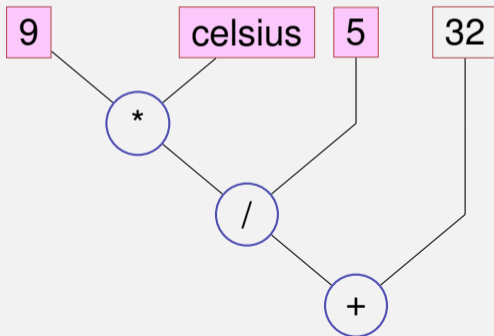
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

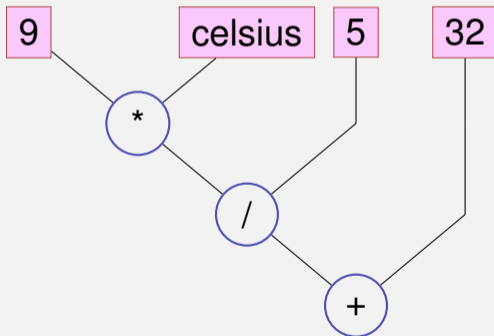
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

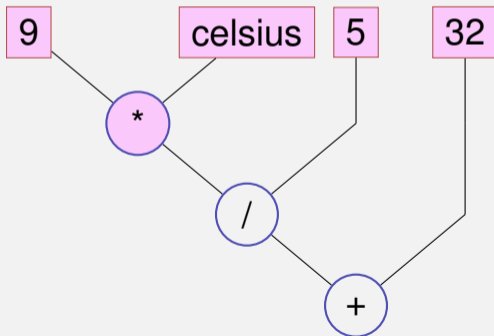
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

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

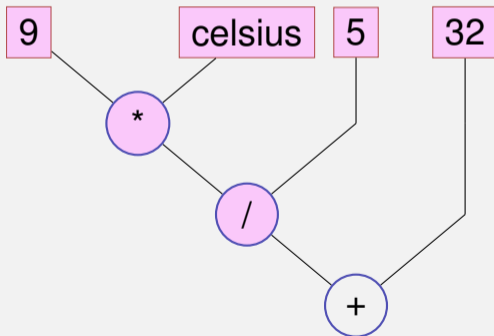




# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

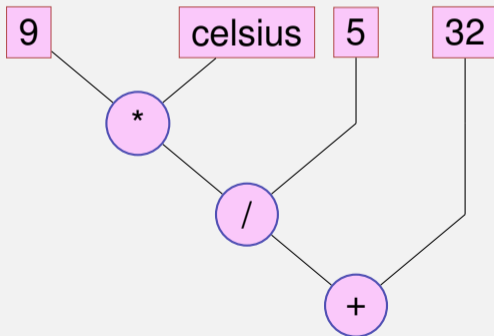
$$9 * \text{celsius} / 5 + 32$$



# Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

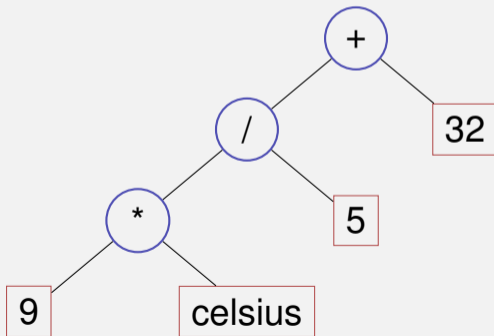
$$9 * \text{celsius} / 5 + 32$$



# Ausdrucksbäume – Notation

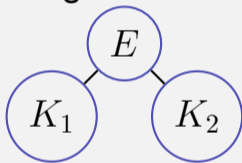
Üblichere Notation: Wurzel oben

9 \* celsius / 5 + 32



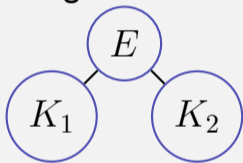
# Auswertungsreihenfolge – formaler

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



# Auswertungsreihenfolge – formaler

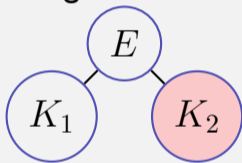
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

# Auswertungsreihenfolge – formaler

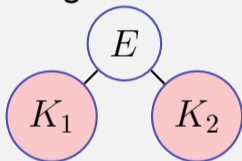
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

# Auswertungsreihenfolge – formaler

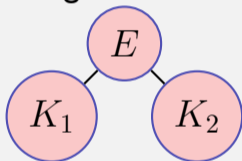
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

# Auswertungsreihenfolge – formaler

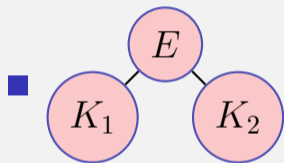
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.



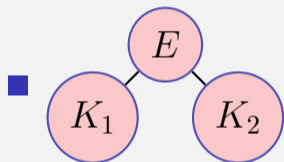
# Auswertungsreihenfolge – formaler



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
<b>Negation</b>	-	<b>1</b>	<b>16</b>	<b>rechts</b>
Multiplikation	*	2	14	links
Division	/	2	14	links
Modulo	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

# Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
<b>Negation</b>	-	<b>1</b>	<b>16</b>	<b>rechts</b>
Multiplikation	*	2	14	links
Division	<b>-a : R-Wert → R-Wert</b>			links
Modulo	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

# 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
<b>Addition</b>	<b>+</b>	<b>2</b>	<b>13</b>	<b>links</b>
Subtraktion	-	2	13	links

# 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
<b>Addition</b>	<b>+</b>	<b>2</b>	<b>13</b>	<b>links</b>
Subtraktion	-	2	13	links

$$a+b : \text{R-Wert} \times \text{R-Wert} \rightarrow \text{R-Wert}$$

# Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	-	1	16	rechts
<b>-a+b+c</b>				
Addition	+	2	13	links
Subtraktion	-	2	13	links



# Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
<b>Negation</b>	-	<b>1</b>	<b>16</b>	<b>rechts</b>
<div style="border: 1px solid black; background-color: #e0e0e0; padding: 10px; display: inline-block;"><math>-a+b+c</math></div>				
<b>Addition</b>	+	<b>2</b>	<b>13</b>	<b>links</b>
Subtraktion	-	2	13	links

# Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	-	1	16	rechts
$-a+b+c = ((-a) + b) + c$ $\text{R-Wert} \times \text{R-Wert} \times \text{R-Wert} \rightarrow \text{R-Wert}$				
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

$a = b = c$



$a = (b = c)$

# 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  $a$ .

# 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
```

# C++ **vs.** ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,

# 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.

# Arithmetische Zuweisungen

`a += b`

$\Leftrightarrow$

`a = a + b`



# Arithmetische Zuweisungen

`a += b`

$\Leftrightarrow$

`a = a + b`

Analog für `-`, `*`, `/` und `%`

# 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: **1**010**11** entspricht 43.

*Niedrigstes Bit, Least Significant Bit (LSB)*

*Höchstes Bit, Most Significant Bit (MSB)*

# Binäre Zahlen: Zahlen der Computer?

## Wahrheit: Computer rechnen mit Binärzahlen.

NEUE ZÜRCHER ZEITUNG

# TECHNIK

Mittwoch, 30. August 1950 Blatt 13  
Mittagsausgabe Nr. 1796 (50)

### Das programmgesteuerte Rechengertät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika wird in den Artikeln „Elektronische Rechenmaschinen“ (vgl. Nr. 2149 der „N. Z. Z.“ vom 13. Oktober 1948) und „Die neueste elektronische Rechenmaschine“ (vgl. Nr. 872 der „N. Z. Z.“ vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K-6, Neubibchen — die Rolle sein, welches im Juli dieses Jahres am Institut für angewandte Mathematik der Eidgenössischen Technischen Hochschule in Zürich, das unter der Leitung von Prof. Dr. F. Siefel steht, in Betrieb genommen wurde. Damit ist dieses Institut in der Lage, dem in der Schweiz immer stärker werdenden Bedarf nach einer leistungsfähigen Zentraltabelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden, und die Erfüllung vieler anderer Aufgaben ist vorbereitet.

#### Merkmale des Gerätes

Das Gerät ist ein Glied in dem progressiven Entwicklungsgang des Ingenieurs Konrad Zuse; es wurde im Auftrag des Instituts für angewandte Mathematik der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als „Modell E 4“ konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen des Vereinigten Staates. Es ist äusserst interessant festzustellen, wie für die meisten wichtigen funktionalen Probleme bedenkenswerte genau dieselbe Lösung gefunden wurde, wie aber andererseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigemessen wurde.

Eine kurze technische Charakterisierung lautet wie folgt: Elektronenmechanisches Gerät mit 2200 Relais, 21 Schaltkästchen und einem Speicher für 64 Zahlen, welcher mit neuartigen, mechanischen Schaltgliedern arbeitet; Vervollständigung des Dualsystems und der halblogischen Darstellung; Multiplikationszeit 2,5 Sekunden; Programmsteuerung mit Hilfe zweier Lochstreifen, auf die willkürlich umgeschaltet werden kann; Eingabe von Zahlen durch eine Tastatur oder durch einen Lochstreifen; Abgabe der Resultate durch Lampenfeld, Lochstreifen oder Druckwerk.

#### Das duale Zahlensystem

Allgemein wird programmgesteuertes Rechengertät häufig als duale Zahlensystem zugrunde gelegt, welches nur die zwei Zahlensymbole 0 und 1 verwendet, während das bekannte Dezimalsystem

lesen wir eine Dezimalzahl von rechts nach links, so erhält sich ein Glied in dem progressiven Entwicklungsgang des Ingenieurs Konrad Zuse; es wurde im Auftrag des Instituts für angewandte Mathematik der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als „Modell E 4“ konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen des Vereinigten Staates. Es ist äusserst interessant festzustellen, wie für die meisten wichtigen funktionalen Probleme bedenkenswerte genau dieselbe Lösung gefunden wurde, wie aber andererseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigemessen wurde.

$$a \cdot 10^3 + b \cdot 10^2 + c \cdot 10^1 + d \cdot 10^0 + e \cdot 10^{-1} + f \cdot 10^{-2} + g \cdot 10^{-3}$$

Die Zahl 1 wird in beiden Systemen gleich dargestellt. Um jedoch duale von dezimalen Zahlen deutlich zu trennen, schreiben wir die duale 1 als  $1_2$ . — Dagegen weicht schon die 2 ab, indem sie dual  $10_2$  lautet, denn dies bedeutet  $1 \cdot 10^1 + 0 \cdot 10^0 = 2$ . Wenn einer Zahl (ohne Stellen nach dem Komma) bereits eine Null zugefügt wird, so vergrössert sie sich um den Faktor 2 (und sieht, wie im Dualsystem, um den Faktor 10). Auf diese Weise kann aus  $10_2 = 2$  auf einfache Weise gebildet werden:  $100_2 = 4$ ,  $1000_2 = 8$ ,  $10000_2 = 16$ , usw.

Die Dualzahl  $10101_2$  bedeutet aus also:

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Ganz analog sind etwaige Stellen nach dem Komma zu interpretieren; so wird  $1,011_2$  wie folgt interpretiert:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + \frac{1}{4} + \frac{1}{8} = 1,375$$

Der große Vorteil, der das Dualsystem für Rechenautomaten so geeignet macht, nämlich die Reduktion der Anzahl der verwendeten Symbole auf nur zwei, wird allerdings durch einen Nachteil erkauft: Es braucht mehr Stellen, um eine bestimmte Zahl anzustellen. Die einstellige Zahl 8

Änderung des Maßstabes durchgeführt werden können.

Die beschriebene Darstellung bringt eine gewisse Komplikation der Rechenoperationen mit sich. So müssen vor einer Addition die beiden Summanden zunächst so verschoben werden, daß ihre Kommata untereinander zu liegen kommen, was am Hand eines Beispiels erläutert werden soll. Damit der Leser nicht durch das ausgeführte duale Zahlensystem verärrt wird, ist das Beispiel im Dezimalsystem durchgeführt; doch wird daran erinnert, daß das Gerät in Wirklichkeit mit dualen Zahlen rechnet.

Es soll also etwa addiert werden:  $2,345678 \times 10^3 + 9,876543 \times 10^1$  (Man beachte, daß die jeweilige Zahl stets zwischen 1 und 10 liegt, also das Komma nachher ersten Stelle ist). Nun müssen die beiden Summanden „ausgerichtet“ werden, d. h. die beiden Exponenten sind einander gleich zu machen, was erreicht wird, indem das kleinere Exponent den Wert des größeren, also 2. Die Zahlen unten nun, richtig untereinander geschrieben, sind so, wie folgt:

$$\begin{array}{r} 2,345678 \times 10^3 \\ 0,987654 \times 10^3 \\ \hline 2,355554 \times 10^3 \end{array}$$

Es ist ersichtlich, daß bei der kleineren der beiden Zahlen rechts einige Stellen abgeschrieben werden mußten; denn wenn die Summanden siebenstellig gegeben waren, so soll auch das Resultat nicht mehr als sieben Stellen enthalten.



Abb. 2. Der Schalter bei der Festlegung eines Rechnerplatzes. Die Ableser für den Lochstreifen sind deutlich sichtbar.

Befehle können „bellend“ gegeben werden, d. h. ihre Ausführung wird von der Natur eines errechneten Resultates abhängig gemacht. Erst dadurch werden die aufeinanderfolgenden Prozesse



# Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.





# Rechentricks

## ■ Abschätzung der Grössenordnung von Zweierpotenzen<sup>2</sup>:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

---

<sup>2</sup>Dezimal 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)

## ■ Abschätzung der Grössenordnung von Zweierpotenzen<sup>2</sup>:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

---

<sup>2</sup>Dezimal 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)

## ■ Abschätzung der Grössenordnung von Zweierpotenzen<sup>2</sup>:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

---

<sup>2</sup>Dezimal 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)

# 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
<b>1</b>	<b>0001</b>	<b>1</b>
<b>2</b>	<b>0010</b>	<b>2</b>
3	0011	3
<b>4</b>	<b>0100</b>	<b>4</b>
5	0101	5
6	0110	6
7	0111	7
<b>8</b>	<b>1000</b>	<b>8</b>
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.

# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.



# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen: 0x00000000 -- 0xffffffff .

0x80000000: höchstes Bit einer 32-bit Zahl gesetzt.

0xffffffff: alle Bits einer 32-bit Zahl gesetzt.

“0x8a20aaf0 ist eine Adresse in den oberen 2G des 32-bit Adressraumes”

# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen: 0x00000000 -- 0xffffffff .

0x80000000: höchstes Bit einer 32-bit Zahl gesetzt.

0xffffffff: alle Bits einer 32-bit Zahl gesetzt.

“0x8a20aaf0 ist eine Adresse in den oberen 2G des 32-bit Adressraumes”

# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen: 0x00000000 -- 0xffffffff .

0x80000000: höchstes Bit einer 32-bit Zahl gesetzt.

0xffffffff: alle Bits einer 32-bit Zahl gesetzt.

“0x8a20aaf0 ist eine Adresse in den oberen 2G des 32-bit Adressraumes”

# Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen: 0x00000000 -- 0xffffffff .

0x80000000: höchstes Bit einer 32-bit Zahl gesetzt.

0xffffffff: alle Bits einer 32-bit Zahl gesetzt.

“0x8a20aaf0 ist eine Adresse in den oberen 2G des 32-bit Adressraumes”

# Wozu Hexadezimalzahlen?

„Für Programmierer und Techniker“

(Bedienungsanleitung Schachcomputer *Mephisto II*, 1981)

Beispiele:

8200

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

7F00

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:

805E

c) Anzeige 805E  
(E=-14) Umrechnung nach folgendem Verfahren:  
 $(14 \times 16^0) + (5 \times 16^1) + (0 \times 16^2) + (0 \times 16^3) = 14 + 80 + 0 + 0 =$   
 $= +94 \text{ Punkte.}$

7F80

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

# Beispiel: Hex-Farben

#00FF00

r g b

# Beispiel: Hex-Farben

#FFFFFF00



r g b

# Beispiel: Hex-Farben

#808080



r g b



# Beispiel: Hex-Farben

#FF0050



r g b

# Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

4e 5a 5a

01001110 01011010 01011010

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg.

01001010 01010110 01001010

www.nzz.ch · Fr. 4.00 · €3.50



01000110 01101100 11111100

01100011 01101000 01101100 01101000 01100001 01101110 01100011 01100011 01100101 01101100 01100101 01101110 01100100 0000-  
0000 01101001 01101110 00100000 01010000 01100001 01101100 01101000 01100001 01100001 00001101 00000100 01000100 01101001  
01100101 00100000 01100111 01100101 01100100 01100100 01100100 01101001 01100110 01100101 01100001

01101000

01000010 01100101  
01110010 01101001

01100011 01101000 01110100 01100101

00100000 11111100 01100010  
01100101 01110010 00100000  
01101110 01100101 01110101 011-  
00101 0110011 00100000 010-  
01101 01100001 01110011

01110011 01100001

01100011 01100001 01100000 010-  
0001 0110110 0000000 0101001 0111-  
001 01110010 01100001 01100101 01101110  
0001010 0001010 0001010 0000010  
000101 0110110 0100101 0101010 010-  
00010 0100010 0001010 0100001 0100001  
01100011 01100000 01100000 0100001  
01100011 01100000 01100000 01000-  
001 01100100 01110010 0100000

01100110 01100101

01100010 01101110 01100011 01100001 011-  
0000 01100001 01101100 0110100 010-  
0001 01101100 0000101 00000000 0000101  
00000000 0000100 01100001 0110101 011-  
10000 0000000 0000000 01100001 0111-  
0010 0110100 01100001 01100000 01110000

01100101 01110011 00100000 0100110 011-  
0001 0110011 0110011 0110011 0100001 0101-  
011 01100101 01100010 00100000 0110011  
0110000 01100001 01100000 0110000 011-  
0010 01000101 01100110 0110010 0110010  
0110010 01100101 01100110 0101010 001-  
0000 01000100 01100101 01100101 001-  
0000 01100100 01100101 01100101 01100101  
01100101 01110010 01100101 01100101  
01100111 00100000 01100101 01100001 011-  
0001 01101000 01100100 01100101 0000-  
000 01110001 01101100 01100101 01100101  
01100100 01100001 01100101 01100101 0111-  
0000 0100010 00000000 0000101 01000100  
01100101 01100101 01100101 01100111 011-  
10000 01100100 01100101 01100100 01100101  
01100101 01100101 00100000 01100000 011-  
00001 01100101 11111100 01110000

00100000 01110110

01100101 01110010 01100001 01101110 011-  
0100 01100111 0010111 0110010 01100100  
01100100 01100101 0100001 01101000 001-  
0110 0000101 00001010 0000101 000-  
0000 01000000 11111100 01100000 010001-  
11 00100000 01000000 01100101 01100101  
01100001 01100000 01101111 01100101 011-  
0010 01010100 00100000 00000000 010000-

# 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

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

Woher kommt gerade diese Aufteilung?

- Auf den meisten Plattformen  $B = 32$

# Wertebereich des Typs `int`

- Repräsentation mit  $B$  Bits. Wertebereich

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

Woher kommt gerade diese Aufteilung?

- 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:  $15^8 = -1732076671$ 
```

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

- 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

int Wert	Vorzeichen	unsigned int Wert
----------	------------	-------------------

---

$x$	$\geq 0$	$x$
-----	----------	-----

$x$	$< 0$	$x + 2^B$
-----	-------	-----------

# Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array}$$

$$\begin{array}{r} 0010 \\ +0011 \\ \hline 0101 \end{array}$$

# Rechnen mit Binärzahlen (4 Stellen)

Einfache Subtraktion

$$\begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 0101 \\ -0011 \\ \hline 0010 \end{array}$$

# Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

$$\begin{array}{r} 7 \\ +9 \\ \hline 16 \end{array}$$

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

# Rechnen mit Binärzahlen (4 Stellen)

Negative Zahlen?

$$\begin{array}{r} 5 \\ +(-5) \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0101 \\ \quad ??? \\ \hline (1)0000 \end{array}$$



# Rechnen mit Binärzahlen (4 Stellen)

Einfacher: -1

$$\begin{array}{r} 1 \\ +(-1) \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0001 \\ 1111 \\ \hline (1)0000 \end{array}$$

# Rechnen mit Binärzahlen (4 Stellen)

Nutzen das aus:

3

+?

---

-1

0011

+????

---

1111

# Rechnen mit Binärzahlen (4 Stellen)

Invertieren!

$$\begin{array}{r} 3 \\ +(-4) \\ \hline -1 \end{array}$$

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

# Rechnen mit Binärzahlen (4 Stellen)

$$\begin{array}{r} a \\ +(-a - 1) \\ \hline -1 \end{array}$$

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

# Rechnen mit Binärzahlen (4 Stellen)

- Negation: Inversion und Addition von 1

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


# Rechnen mit Binärzahlen (4 Stellen)

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

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

# Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis<sup>3</sup>



$11 \equiv 23 \equiv -1 \equiv \dots \pmod{12}$

$4 \equiv 16 \equiv \dots \pmod{12}$

$3 \equiv 15 \equiv \dots \pmod{12}$

<sup>3</sup>Die Arithmetik funktioniert auch mit Dezimalzahlen (und auch für die Multiplikation)

# Negative Zahlen (3 Stellen)

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



# Negative Zahlen (3 Stellen)

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

# Negative Zahlen (3 Stellen)

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

# Negative Zahlen (3 Stellen)

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

# Negative Zahlen (3 Stellen)

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

# Negative Zahlen (3 Stellen)

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

# 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		

# 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.