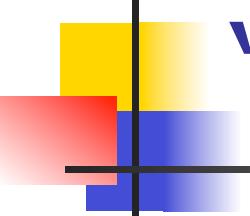


Fliesskommazahlen

Typen **float** und **double**;
Fliesskommazahlensysteme,
Löcher im Wertebereich, IEEE
Standard, Fliesskomma-Richtlinien



“Richtig” Rechnen

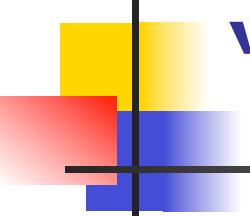
```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#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;
}
```

28 degrees Celsius are 82 degrees Fahrenheit.



“Richtig” Rechnen

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

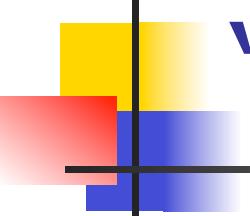
#include <iostream>

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;
}
```

28 degrees Celsius are 82 degrees Fahrenheit.

Richtig wäre: 82.4



“Richtig” Rechnen

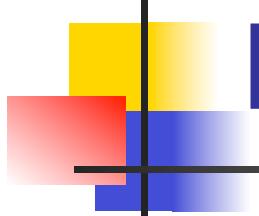
```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

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

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

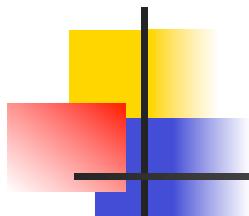
28 degrees Celsius are 82.4 degrees Fahrenheit.



Repräsentierung von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen):

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

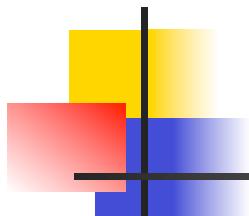


Repräsentierung von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen):

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

$$82.4 = 000082.400$$



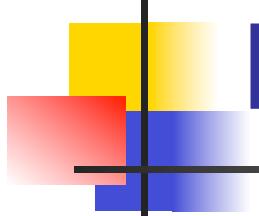
Repräsentierung von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen):

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

$$82.4 = 000082.400$$

- Nachteil 1:
 - Wertebereich wird *noch* kleiner als bei ganzen Zahlen.



Repräsentierung von Dezimalzahlen (z.B. 82.4)

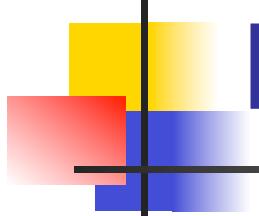
Fixkommazahlen (z.B. mit 10 Stellen):

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

$$0.0824 = 0000000.082$$

dritte Stelle abgeschnitten!

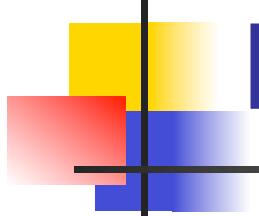
- Nachteil 2:
 - Repräsentierbarkeit hängt stark davon ab, wo das Komma ist.



Repräsentierung von Dezimalzahlen (z.B. 82.4)

Fliesskommazahlen (z.B. mit 10 Stellen):

- *feste* Anzahl signifikanter Stellen (10)
- *plus* Position des Kommas



Repräsentierung von Dezimalzahlen (z.B. 82.4)

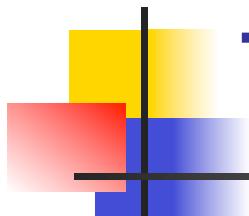
Fliesskommazahlen (z.B. mit 10 Stellen):

- o *feste* Anzahl signifikanter Stellen (10)
- o *plus* Position des Kommas

$$82.4 = 824 \times 10^{-1}$$

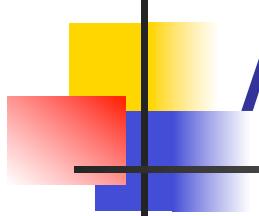
$$0.0824 = 824 \times 10^{-4}$$

- o Zahl ist *Signifikand* $\times 10^{\text{Exponent}}$



Typen `float` und `double`

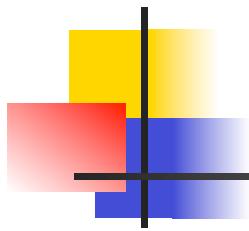
- sind die fundamentalen C++ Typen für Fließkommazahlen
- approximieren den Körper (\mathbb{R} , $+$, \times) in der Mathematik (reelle Zahlen)
- haben grossen Wertebereich, ausreichend für viele Anwendungen (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell



Arithmetische Operatoren

Wie bei `int`, aber...

- Divisionsoperator / modelliert “echte” (reelle, nicht ganzzahlige) Division
- keine Modulo-Operatoren `%` und `%=`

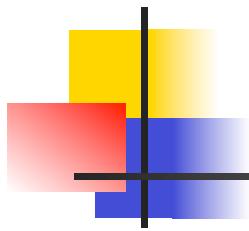


Literale

Beispiele:

`1.23e-7` : Typ **double**, Wert 1.23×10^{-7}

`1.23e-7f`: Typ **float**, Wert 1.23×10^{-7}



Literale

Beispiele:

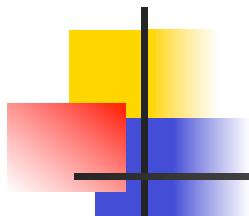
`1.23e-7` : Typ **double**, Wert 1.23×10^{-7}

`1.23e-7f` : Typ **float**, Wert 1.23×10^{-7}

ganzzahliger Teil

Exponent

fraktionaler Teil

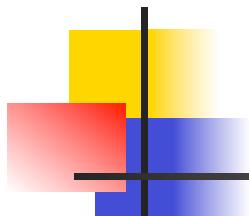


Rechnen mit `float`: Beispiel

Approximation der Euler-Konstante

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

mittels der ersten 10 Terme.



Rechnen mit float: Beispiel

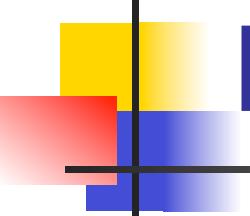
```
// Program: euler.cpp
// Approximate Euler's constant e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;    // 1/i!
    float e = 1.0f;    // i-th approximation of e

    std::cout << "Approximating the Euler constant... \n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i;    // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }

    return 0;
}
```



Rechnen mit float: Beispiel

```
// Program: euler.cpp
// Approximate Euler's constant e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;    // 1/i!
    float e = 1.0f;    // i-th approximation of e

    std::cout << "Approximating the Euler constant...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i;    // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }

    return 0;
}
```

Zuweisungen sind rechtsassoziativ: **e += (t /= i);**

Rechnen mit float: Beispiel

```
// Program: euler.cpp
// Approximate Euler's constant e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;    // 1/i!
    float e = 1.0f;    // i-th approximation of e

    std::cout << "Approximating the Euler constant...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i;    // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }
    return 0;
}
```

$$t: 1 / (i-1)! \longrightarrow 1 / i!$$

Zuweisungen sind rechtsassoziativ: **e += (t /= i);**

Rechnen mit float: Beispiel

```
// Program: euler.cpp
// Approximate Euler's constant e.

#include <iostream>

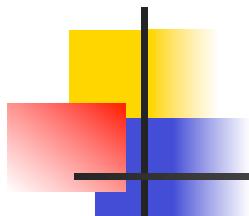
int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f;    // 1/i!
    float e = 1.0f;    // i-th approximation of e

    std::cout << "Approximating the Euler constant...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i;      // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }
    return 0;
}
```

$e: 1 + \dots + 1 / (i-1)! \longrightarrow 1 + \dots + 1 / i!$

↓

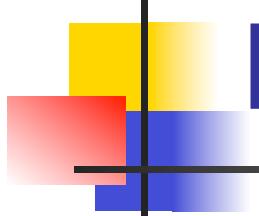
Zuweisungen sind rechtsassoziativ: $e += (t /= i);$



Rechnen mit float: Beispiel

Ausgabe:

```
Approximating the Euler constant...
Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828
```



Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

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

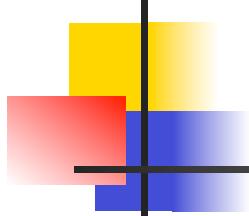
Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

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



Typ: `float`; Wert: 28



Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

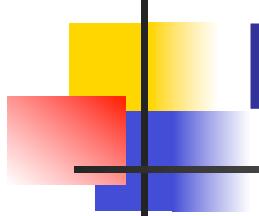
$9 * 28.0f / 5 + 32$

Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

9 * 28.0f / 5 + 32

wird nach `float` konvertiert: 9.0f



Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

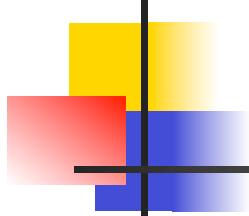
$$252.0 \text{f} \ / \ 5 + 32$$

Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

252.0f / 5 + 32

wird nach `float` konvertiert: 5.0f



Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

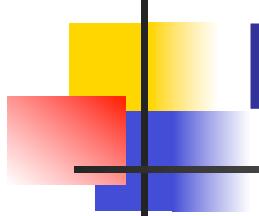
$$50.4 + 32$$

Gemischte Ausdrücke, Konversion

- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

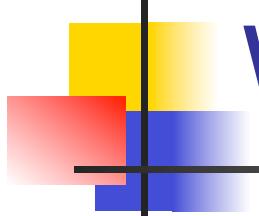
50.4 + 32

wird nach `float` konvertiert: 32.0f



Gemischte Ausdrücke, Konversion

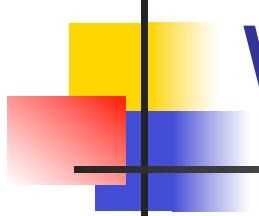
- Fliesskommatypen sind allgemeiner als ganzzahlige Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.



Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber...
- Wertebereich ist zusammenhängend (keine “Löcher”): \mathbb{Z} ist “diskret”.



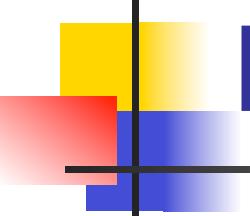
Wertebereich

Ganzzahlige Typen:

- o Über- und Unterlauf häufig, aber...
- o Wertebereich ist zusammenhängend (keine “Löcher”): \mathbb{Z} ist “diskret”.

Fliesskommatypen:

- o Über- und Unterlauf selten, aber...
- o es gibt Löcher: \mathbb{R} ist “kontinuierlich”.



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

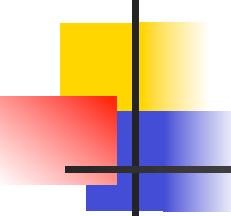
#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

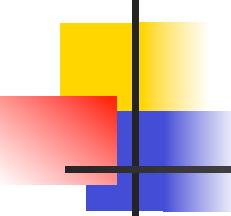
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```

input: 1.5

input: 1.0

input: 0.5



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

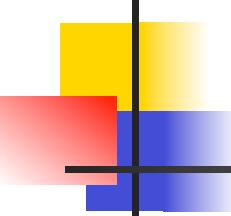
    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```

input: 1.5

input: 1.0

input: 0.5

output: 0



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

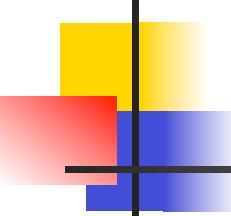
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```

input: 1.1

input: 1.0

input: 0.1



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

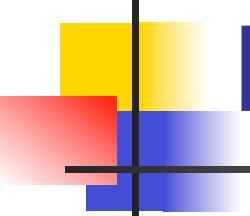
    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```

input: 1.1

input: 1.0

input: 0.1

output: 2.23517e-8



Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number      =? ";
    std::cin >> n1;

    float n2;
    std::cout << "Second number     =? ";
    std::cin >> n2;

    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
          << n1 - n2 - d << ".\n";
    return 0;
}
```

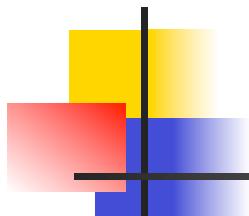
input: 1.1

input: 1.0

input: 0.1

Was ist
hier los?

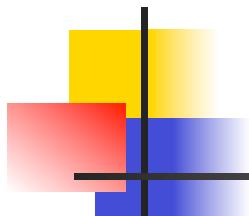
output: 2.23517e-8



Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- o $\beta \geq 2$, die *Basis*
- o $p \geq 1$, die *Präzision*
- o e_{min} , der *kleinste Exponent*
- o e_{max} , der *grösste Exponent*

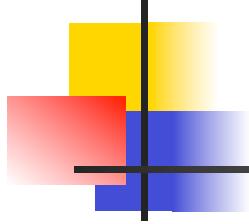


Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

$$F(\beta, p, e_{min}, e_{max})$$

- o $\beta \geq 2$, die *Basis*
- o $p \geq 1$, die *Präzision*
- o e_{min} , der *kleinste Exponent*
- o e_{max} , der *grösste Exponent*.



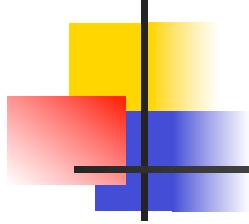
Fliesskommazahlensysteme

$$F(\beta, p, e_{min}, e_{max})$$

enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \times \beta^e,$$

$$d_i \in \{0, \dots, \beta - 1\}, \quad e \in \{e_{min}, \dots, e_{max}\}$$



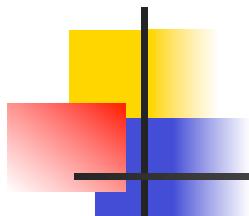
Fliesskommazahlensysteme

$$F(\beta, p, e_{min}, e_{max})$$

enthält die Zahlen (Basis- β -Darstellung)

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e,$$

$$d_i \in \{0, \dots, \beta - 1\}, \quad e \in \{e_{min}, \dots, e_{max}\}$$



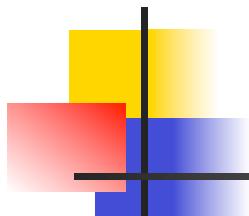
Fliesskommazahlensysteme

Beispiel:

- o $\beta = 10$

Darstellungen der Dezimalzahl 0.1:

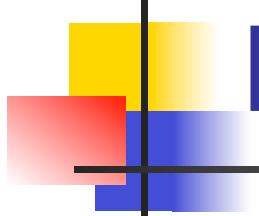
$$1.0 \times 10^{-1}, 0.1 \times 10^0, 0.01 \times 10^1, \dots$$



Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

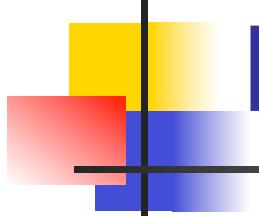


Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 1: Die normalisierte Darstellung einer Fliesskommazahl ist eindeutig und deshalb zu bevorzugen.

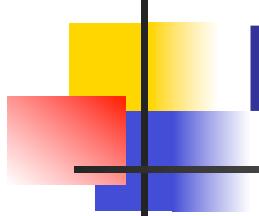


Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 2: Die Zahl 0 (und alle Zahlen kleiner als $\beta^{e_{min}}$) haben keine normalisierte Darstellung (werden wir später beheben)!



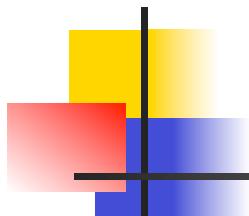
Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Die Menge der normalisierten Zahlen ist

$$F^*(\beta, p, e_{min}, e_{max})$$

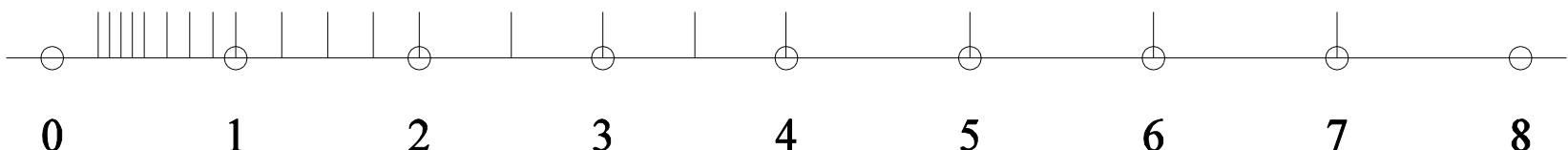


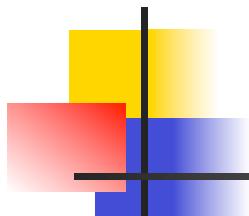
Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Beispiel: $F^*(2, 3, -2, 2)$ (Zahlen ≥ 0)



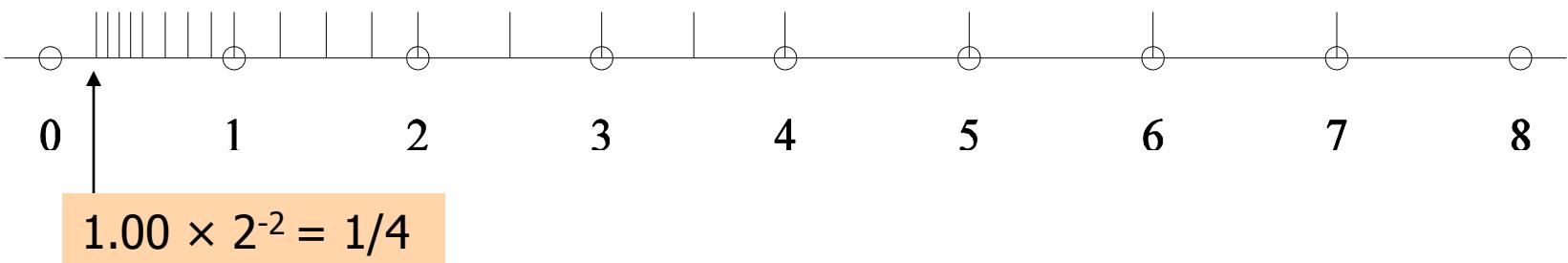


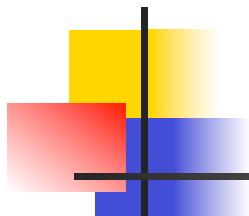
Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Beispiel: $F^*(2, 3, -2, 2)$ (Zahlen ≥ 0)



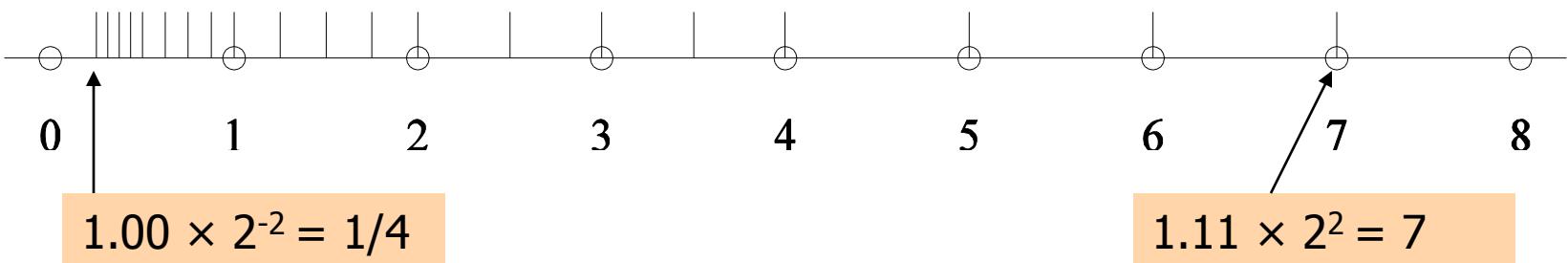


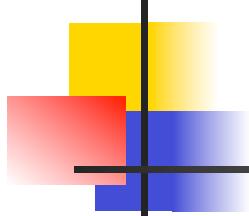
Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \cdot d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

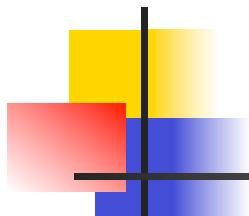
Beispiel: $F^*(2, 3, -2, 2)$ (Zahlen ≥ 0)





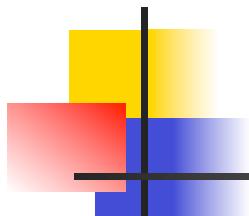
Binäre und dezimale Systeme

- o intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlen-
system)



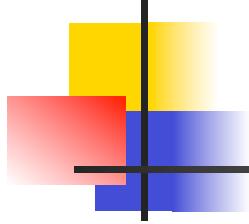
Binäre und dezimale Systeme

- o intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlensystem)
- o Literale und Eingaben haben $\beta = 10$ (dezimales Fliesskommazahlensystem)



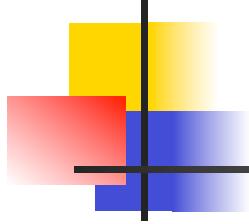
Binäre und dezimale Systeme

- o intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlensystem)
- o Literale und Eingaben haben $\beta = 10$ (dezimales Fliesskommazahlensystem)
- o Eingaben müssen umgerechnet werden!



Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

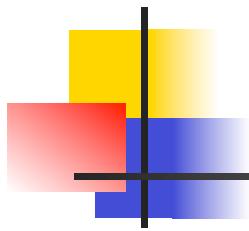


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty, \dots, 0} b_i 2^i$$

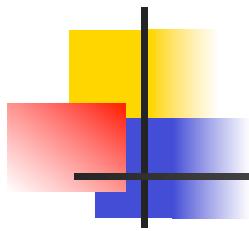


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty, \dots, 0} b_i 2^i = b_0 \cdot b_{-1} b_{-2} b_{-3} \dots$$

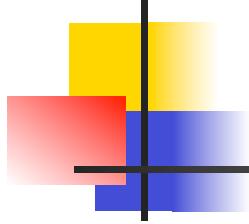


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$\begin{aligned}x &= \sum_{i=-\infty, \dots, 0} b_i 2^i = b_0 \cdot b_{-1} b_{-2} b_{-3} \dots \\&= b_0 + \sum_{i=-\infty, \dots, -1} b_i 2^i\end{aligned}$$

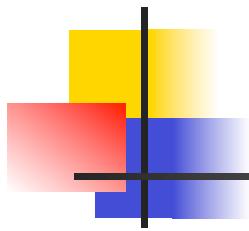


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$\begin{aligned}x &= \sum_{i=-\infty, \dots, 0} b_i 2^i = b_0 \cdot b_{-1} b_{-2} b_{-3} \dots \\&= b_0 + \sum_{i=-\infty, \dots, -1} b_i 2^i \\&= b_0 + \sum_{i=-\infty, \dots, 0} b_{i-1} 2^{i-1}\end{aligned}$$

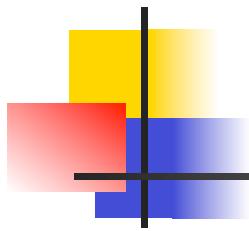


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$\begin{aligned}x &= \sum_{i=-\infty, \dots, 0} b_i 2^i = b_0 \cdot b_{-1} b_{-2} b_{-3} \dots \\&= b_0 + \sum_{i=-\infty, \dots, -1} b_i 2^i \\&= b_0 + \sum_{i=-\infty, \dots, 0} b_{i-1} 2^{i-1} \\&= b_0 + (\sum_{i=-\infty, \dots, 0} b_{i-1} 2^i) / 2\end{aligned}$$

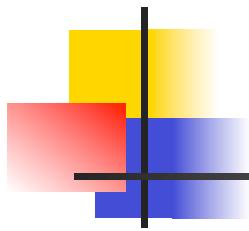


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$\begin{aligned}x &= \sum_{i=-\infty, \dots, 0} b_i 2^i = b_0 \cdot b_{-1} b_{-2} b_{-3} \dots \\&= b_0 + \sum_{i=-\infty, \dots, -1} b_i 2^i \\&= b_0 + \sum_{i=-\infty, \dots, 0} b_{i-1} 2^{i-1} \\&= b_0 + \underbrace{\left(\sum_{i=-\infty, \dots, 0} b_{i-1} 2^i \right)}_{x'} / 2 \\&\quad x' = b_{-1} \cdot b_{-2} b_{-3} b_{-4} \dots\end{aligned}$$

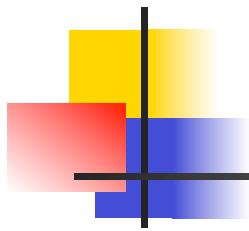


Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärziffern (x):

b_0 , Binärziffern ($b_{-1} \cdot b_{-2} b_{-3} b_{-4} \dots$)



Umrechnung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärziffern (x):

b_0 , Binärziffern $(b_{-1} \cdot b_{-2} b_{-3} b_{-4} \dots)$

$$x' = 2 (x - b_0)$$

Binärdarstellung von 1.1

Binärdarstellung von 1.1

$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$

Binärdarstellung von 1.1

$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$

Binärdarstellung von 1.1

$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$
0.4	0.8	0.8	$b_{-3} = 0$

Binärdarstellung von 1.1

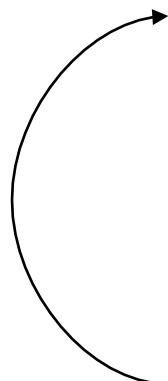
$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$
0.4	0.8	0.8	$b_{-3} = 0$
0.8	1.6	1.6	$b_{-4} = 1$

Binärdarstellung von 1.1

$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$
0.4	0.8	0.8	$b_{-3} = 0$
0.8	1.6	1.6	$b_{-4} = 1$
0.6	1.2	1.2	$b_{-5} = 1$

Binärdarstellung von 1.1

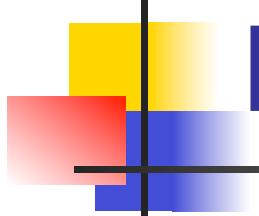
$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$
0.4	0.8	0.8	$b_{-3} = 0$
0.8	1.6	1.6	$b_{-4} = 1$
0.6	1.2	1.2	$b_{-5} = 1$
0.2	0.4	0.4	$b_{-6} = 0$



Binärdarstellung von 1.1

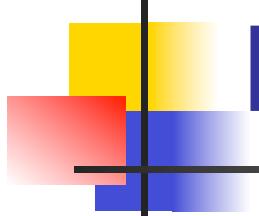
$x - b_i$	$x' = 2(x - b_i)$	$\rightarrow x$	b_i
		1.1	$b_0 = 1$
0.1	0.2	0.2	$b_{-1} = 0$
0.2	0.4	0.4	$b_{-2} = 0$
0.4	0.8	0.8	$b_{-3} = 0$
0.8	1.6	1.6	$b_{-4} = 1$
0.6	1.2	1.2	$b_{-5} = 1$
0.2	0.4	0.4	$b_{-6} = 0$

Binärdarstellung ist $1.0\overline{0011}$ (periodisch, *nicht* endlich)



Binärdarstellung von 1.1

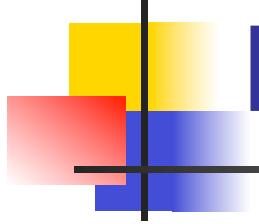
- ist nicht endlich, also gibt es
- Fehler bei der Konversion in ein binäres Fliesskommazahlensystem
- 1.1 ist für den Computer *nicht* 1.1 ...



Binärdarstellung von 1.1

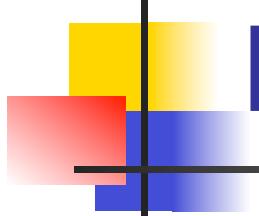
- ist nicht endlich, also gibt es
- Fehler bei der Konversion in ein binäres Fliesskommazahlensystem
- 1.1 ist für den Computer *nicht* 1.1 ... sondern (auf meiner Plattform)

1.1000002384185791015625.



Der Excel-2007-Bug

- Umrechnungsfehler sind *sehr* klein...

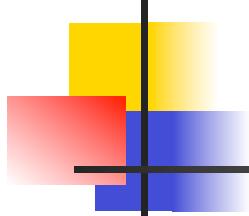


Der Excel-2007-Bug

- o Umrechnungsfehler sind *sehr* klein...
- o können aber *grosse* Auswirkungen haben!

Microsoft Excel 2007:

$77.1 \times 850 = 100000$ (anstatt 65535)



Der Excel-2007-Bug

- o Umrechnungsfehler sind *sehr* klein...
- o können aber *grosse* Auswirkungen haben!

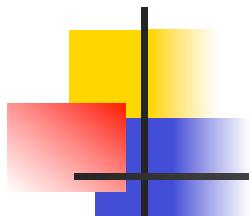
Microsoft Excel 2007:

$$77.1 \times 850 = 100000 \text{ (anstatt 65535)}$$

Microsoft: Resultat wird korrekt berechnet, "nur" falsch angezeigt.

Microsoft Excel 2007:

$$1 \times 65535 = 65535 \text{ (Glück gehabt...)}$$



Der Excel-2007-Bug

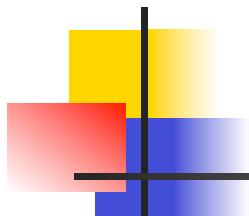
- o Umrechnungsfehler sind *sehr* klein...
- o können aber *grosse* Auswirkungen haben!

Microsoft Excel 2007:

$$77.1 \times 850 = 100000 \text{ (anstatt 65535)}$$

Microsoft: Resultat wird korrekt berechnet, "nur" falsch angezeigt.

stimmt nicht ganz; 77.1 hat keine endliche Binärexpansion, berechnet wird also eine Zahl λ *sehr nahe* an 65535.



Der Excel-2007-Bug

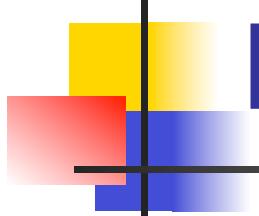
- o Umrechnungsfehler sind *sehr* klein...
- o können aber *grosse* Auswirkungen haben!

Microsoft Excel 2007:

$$77.1 \times 850 = 100000 \text{ (anstatt 65535)}$$

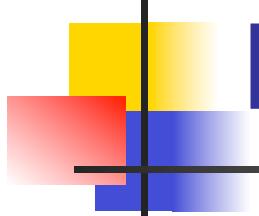
Microsoft: Resultat wird korrekt berechnet, "nur" falsch angezeigt.

λ ist eine von **nur zwölf** Fliesskommazahlen (lt. Microsoft), für die die Umwandlung ins Dezimalsystem fehlerhaft ist.



Rechnen mit Fliesskommazahlen

- fast so einfach wie mit ganzen Zahlen
- Beispiel ($\beta = 2, p = 4$):

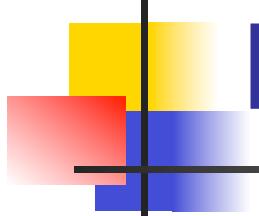


Rechnen mit Fliesskommazahlen

- fast so einfach wie mit ganzen Zahlen
- Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \times 2^{-2} \\ + 1.011 \times 2^{-1} \\ \hline \end{array}$$

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

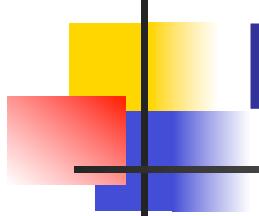


Rechnen mit Fliesskommazahlen

- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{r} 1.111 \times 2^{-2} \\ + 10.110 \times 2^{-2} \\ \hline \end{array}$$

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

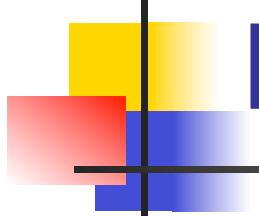


Rechnen mit Fliesskommazahlen

- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{r} 1.111 \times 2^{-2} \\ + 10.110 \times 2^{-2} \\ \hline \end{array}$$

Schritt 2: Binäre Addition der Signifikanden



Rechnen mit Fliesskommazahlen

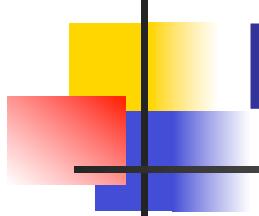
- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$1.111 \times 2^{-2}$$

$$+10.110 \times 2^{-2}$$

$$100.101 \times 2^{-2}$$

Schritt 2: Binäre Addition der Signifikanden



Rechnen mit Fliesskommazahlen

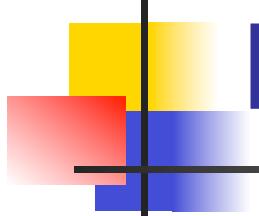
- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$1.111 \times 2^{-2}$$

$$+10.110 \times 2^{-2}$$

$$100.101 \times 2^{-2}$$

Schritt 3: Renormalisierung

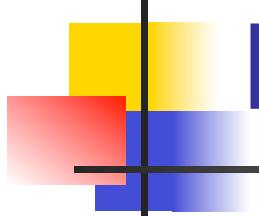


Rechnen mit Fliesskommazahlen

- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{r} 1.111 \times 2^{-2} \\ + 10.110 \times 2^{-2} \\ \hline 1.00101 \times 2^0 \end{array}$$

Schritt 3: Renormalisierung



Rechnen mit Fliesskommazahlen

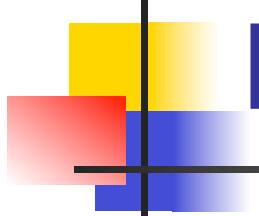
- fast so einfach wie mit ganzen Zahlen
- Beispiel ($\beta = 2$, $p = 4$):

$$1.111 \times 2^{-2}$$

$$+10.110 \times 2^{-2}$$

$$1.00101 \times 2^0$$

Schritt 4: Runden auf p signifikante Stellen, falls notwendig



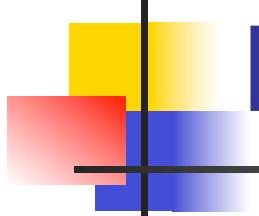
Rechnen mit Fliesskommazahlen

- o fast so einfach wie mit ganzen Zahlen
- o Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{r} 1.111 \times 2^{-2} \\ + 10.110 \times 2^{-2} \\ \hline \end{array}$$

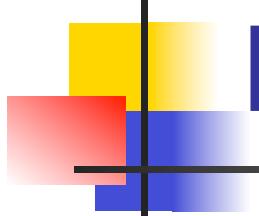
$$1.001 \times 2^0$$

Schritt 4: Runden auf p signifikante Stellen, falls notwendig



Der IEEE-Standard 754

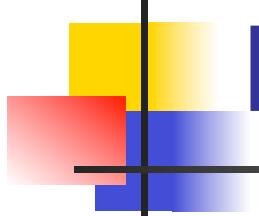
- legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt



Der IEEE-Standard 754

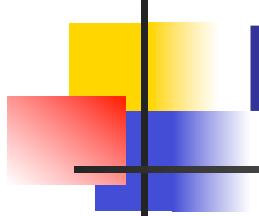
- o legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- o wird von vielen Plattformen unterstützt
- o single precision (`float`) Zahlen:

$$F^*(2, 24, -126, 127)$$



Der IEEE-Standard 754

- o legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- o wird von vielen Plattformen unterstützt
- o single precision (**float**) Zahlen:
$$F^*(2, 24, -126, 127)$$
- o double precision (**double**) Zahlen:
$$F^*(2, 53, -1022, 1023)$$



Der IEEE-Standard 754

- o legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- o wird von vielen Plattformen unterstützt
- o single precision (**float**) Zahlen:

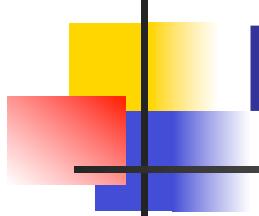
$$F^*(2, 24, -126, 127)$$

plus 0, ∞ , ...

- o double precision (**double**) Zahlen:

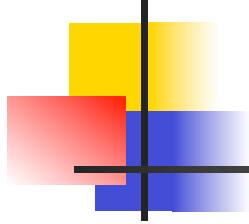
$$F^*(2, 53, -1022, 1023)$$

plus 0, ∞ , ...



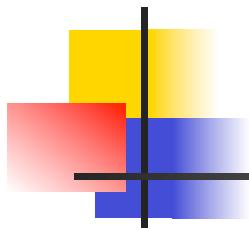
Der IEEE-Standard 754

- o legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- o wird von vielen Plattformen unterstützt
- o alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl



Der IEEE-Standard 754

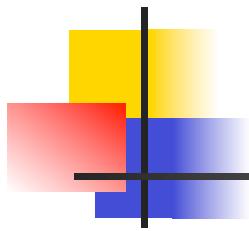
Warum $F^*(2, 24, -126, 127)$?



Der IEEE-Standard 754

Warum $F^*(2, 24, -126, 127)$?

- o 1 Bit für das Vorzeichen
- o 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- o 8 Bit für den Exponenten (256 mögliche Werte)

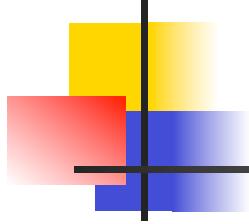


Der IEEE-Standard 754

Warum $F^*(2, 24, -126, 127)$?

- o 1 Bit für das Vorzeichen
- o 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- o 8 Bit für den Exponenten (256 mögliche Werte)

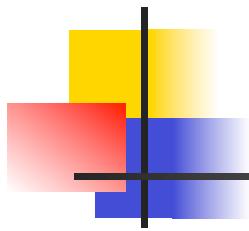
insgesamt 32 Bit



Der IEEE-Standard 754

Warum $F^*(2, 24, -126, 127)$?

- o 1 Bit für das Vorzeichen
- o 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- o 8 Bit für den Exponenten (254 mögliche Exponenten, 2 Spezialwerte: 0, ∞ , ...)

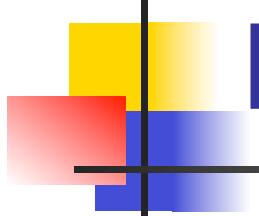


Der IEEE-Standard 754

Warum $F^*(2, 53, -1022, 1023)$?

- o 1 Bit für das Vorzeichen
- o 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- o 11 Bit für den Exponenten (**2046** mögliche Exponenten, **2** Spezialwerte)

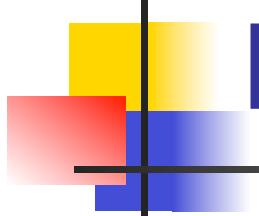
insgesamt 64 Bit



Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 1:

Teste keine zwei Fliesskommazahlen auf
Gleichheit, wenn mindestens eine das
Ergebnis einer Rundungsoperation ist!

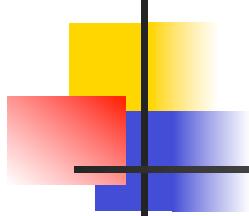


Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 1:

Teste keine zwei Fliesskommazahlen auf
Gleichheit, wenn mindestens eine das
Ergebnis einer Rundungsoperation ist!

```
for (float i = 0.1; i != 1.0; i += 0.1)  
    std::cout << i << "\n";
```



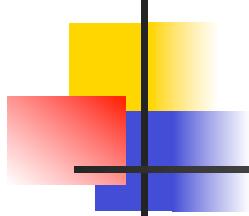
Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 1:

Teste keine zwei Fliesskommazahlen auf
Gleichheit, wenn mindestens eine das
Ergebnis einer Rundungsoperation ist!

```
for (float i = 0.1; i != 1.0; i += 0.1)  
    std::cout << i << "\n";
```

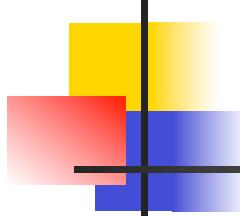
In der Praxis ist das eine Endlosschleife, weil `i` niemals exakt 1 ist!



Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!



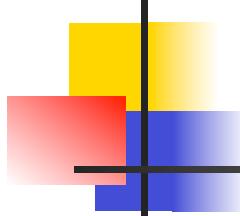
Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel ($\beta = 2, p = 4$):

$$\begin{aligned} & 1.000 \times 2^4 \\ & + 1.000 \times 2^0 \end{aligned}$$



Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.000 \times 2^4 \\ + 1.000 \times 2^0 = 1.0001 \times 2^4 \end{array}$$

Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.000 \times 2^4 \\ + 1.000 \times 2^0 \\ \hline \end{array} \quad \begin{array}{l} \del{= 1.0001 \times 2^4} \\ = 1.000 \times 2^4 \end{array}$$

Rundung auf 4 Stellen!

Richtlinien fürs Rechnen mit Fliesskommazahlen

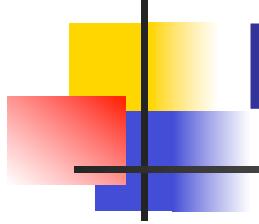
Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.000 \times 2^4 \\ + 1.000 \times 2^0 = 1.000 \times 2^4 \end{array}$$

Addition von 1 hat keinen Effekt!



Beispiel für Regel 2: Harmonische Zahlen

n -te Harmonische Zahl:

$$\begin{aligned}H_n &= 1 + 1/2 + 1/3 + \dots + 1/n \\&= 1/n + 1/(n-1) + \dots + 1\end{aligned}$$

Summe kann vorwärts oder rückwärts berechnet werden.

Beispiel für Regel 2: Harmonische Zahlen

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>

int main()
{
    // Input
    std::cout << "Compute H_n for n =? ";
    unsigned int n;
    std::cin >> n;

    // Forward sum
    float fs = 0;
    for (unsigned int i = 1; i <= n; ++i)
        fs += 1.0f / i;

    // Backward sum
    float bs = 0;
    for (unsigned int i = n; i >= 1; --i)
        bs += 1.0f / i;

    // Output
    std::cout << "Forward sum = " << fs << "\n"
           << "Backward sum = " << bs << "\n";
    return 0;
}
```

Beispiel für Regel 2: Harmonische Zahlen

Compute H_n for $n =? 10000000$

Forward sum = 15.4037

Backward sum = 16.686

Beispiel für Regel 2: Harmonische Zahlen

Compute H_n for $n =? 10000000$

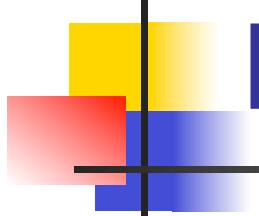
Forward sum = 15.4037

Backward sum = 16.686

Compute H_n for $n =? 100000000$

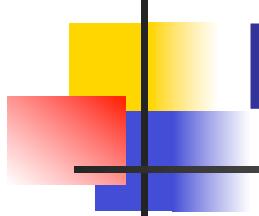
Forward sum = 15.4037

Backward sum = 18.8079



Beispiel für Regel 2: Harmonische Zahlen

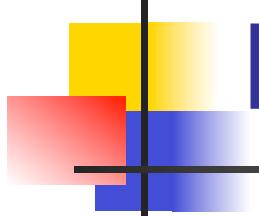
- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme ist eine gute Approximation von H_n .



Beispiel für Regel 2: Harmonische Zahlen

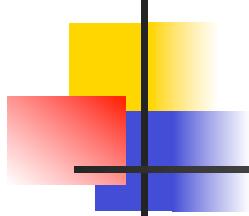
- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme ist eine gute Approximation von H_n .
- Bei $1 + 1/2 + 1/3 + \dots + 1/n$ sind späte Terme zu klein, um noch beizutragen.

wie bei $2^4 + 1 = 2^4$



Beispiel für Regel 2: Harmonische Zahlen

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme ist eine gute Approximation von H_n .
- Bei $1/n + 1/(n-1) + \dots + 1$ sind späte Terme vergleichsweise gross und gehen deshalb in die Gesamtsumme ein.



Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 3:

Vermeide die Subtraktion von Zahlen
sehr ähnlicher Grösse!

Auslösungsproblematik, siehe Skript.

Literatur

- David Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic* (1991)



© 1996 Randy Glasbergen.