Fliesskommazahlen

"Richtig" Rechnen

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

#include <iostream>

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

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

```
28 degrees Celsius are 82 degrees Fahrenheit.
```

Richtig wäre: `82.4`

"Richtig" Rechnen

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

  82.4 = 0000082.400

- 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
- 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

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

  $0.0824 = 824 \times 10^{-4}$
- Zahl ist $Significand \times 10^{Exponent}$

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fliesskommazahlen
- approximieren den Körper (**R**, +, ×) 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...

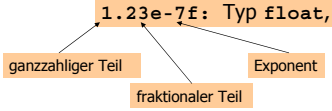
- Divisionsoperator / modelliert "echte" (reelle, nicht ganzzahlige) Division
- keine Modulo-Operatoren `%` und `%=`

Literale

Beispiele:

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

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



ganzzahliger Teil kann leer sein:

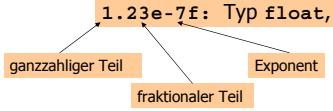
.23e-7 (0.23e-7)

Literale

Beispiele:

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

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



fraktionaler Teil kann leer sein:

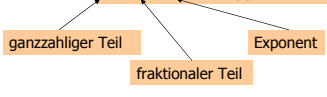
1.e-7 (1.0e-7)

Literale

Beispiele:

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

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



...aber nicht *beide* :

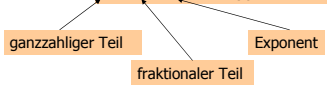
.e-7 (ungültig)

Literale

Beispiele:

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

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



Exponent kann leer sein (bedeutet 10^0)

1.23

Literale

Beispiele:

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

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

ganzzahliger Teil

fraktionaler Teil

Exponent

Dezimalpunkt kann fehlen, aber nur wenn ein Exponent da ist: 123e-7

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.C
// 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+...+1 / (i-1)! ⟶ 1+...+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 integrale 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 integrale Typen.
- in gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
9 * 28.0f / 5 + 32
```

Gemischte Ausdrücke, Konversion

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

```
252.0f / 5 + 32
```

Gemischte Ausdrücke, Konversion

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

```
50.4 + 32
```

Gemischte Ausdrücke, Konversion

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

```
82.4
```

Konversionsregeln

- ganze Zahl zu Fliesskommazahl:
  - nächste darstellbare Zahl

  5 wird zu 5.0
- Fliesskommazahl zu ganzer Zahl:
  - fraktionaler Teil wird abgeschnitten

```
int i = -1.6f; // initialisiert i mit -1
```

- `float` zu `double`: exakt

Wertebereich

Integrale Typen:

- Über- und Unterlauf häufig, aber...
- Wertebereich ist zusammenhängend (keine "Löcher"): **Z** ist "diskret".

Fliesskommatypen:

- Über- und Unterlauf selten, aber...
- es gibt Löcher: **R** ist "kontinuierlich".

Löcher im Wertebereich

```
// Program: diff.C
// 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.C
// 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:

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

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

Fliesskommazahlensysteme

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

enthält die Zahlen

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

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

Fliesskommazahlensysteme

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

enthält die Zahlen (Basis-β-Darstellung)

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

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

Fliesskommazahlensysteme

Beispiel:

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1:

1.0×10^{-1}, 0.1×10^{0}, 0.01×10^{1},...

Normalisierte Darstellung

Normalisierte Zahl:

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

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

Normalisierte Darstellung

Normalisierte Zahl:

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

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:

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

Die Menge der normalisierten Zahlen ist

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

Binäre und dezimale Systeme

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

Umrechung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$x = \Sigma_{i=-\infty,\ldots,0}\, b_i 2^i = b_0 . b_{-1}\, b_{-2}\, b_{-3} \ldots$

Umrechung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$x = \Sigma_{i=-\infty,\ldots,0}\, b_i 2^i = b_0 . b_{-1}\, b_{-2}\, b_{-3} \ldots$

$= b_0 + \Sigma_{i=-\infty,\ldots,-1}\, b_i 2^i$

$= b_0 + \Sigma_{i=-\infty,\ldots,0}\, b_{i-1} 2^{i-1}$

$= b_0 + (\Sigma_{i=-\infty,\ldots,0}\, b_{i-1} 2^i) / 2$

$x' = b_{-1} . b_{-2}\, b_{-3}\, b_{-4} \ldots$

Umrechung dezimal -> binär

Angenommen, $0 < x < 2$.

Binärziffern (x):

b_0 , Binärziffern $(b_{-1} . b_{-2}\, b_{-3}\, b_{-4} \ldots)$

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

Binärdarstellung von 1.1

$x - b_i$	$x' = 2(x - b_i)$	$\longrightarrow x$	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 ...
  sondern (auf meiner Plattform)
  1.100000023841857910156 25.

Der Excel-2007-Bug

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

Microsoft Excel 2007:

77.1 × 850 = 100000 (anstatt 65535)

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

Microsoft Excel 2007:

1 × 65535 = 65535 (Glück gehabt...)

Der Excel-2007-Bug

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

Microsoft Excel 2007:

77.1 × 850 = 100000 (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

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

Microsoft Excel 2007:

77.1 × 850 = 100000 (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 ($\beta = 2$, $p = 4$):

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

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

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

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

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

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

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

Schritt 2: Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rl} & 1.111 \times 2^{-2} \\ + & 10.110 \times 2^{-2} \\ \hline & 100.101 \times 2^{-2} \end{array}$$

Schritt 2: Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{r} 1.111 \times 2^{-2} \\ +10.110 \times 2^{-2} \\ \hline 100.101 \times 2^{-2} \end{array}$$

Schritt 3: Renormalisierung

Rechnen mit Fliesskommazahlen

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

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

Schritt 4: Runden auf p signifikante Stellen, falls notwendig

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{r} 1.111 \times 2^{-2} \\ +10.110 \times 2^{-2} \\ \hline 1.001 \times 2^{0} \end{array}$$

Schritt 4: Runden auf p signifikante Stellen, falls notwendig

Der IEEE-Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt
- single precision (`float`) Zahlen:
  $\mathcal{F}^*(2, 24, -126, 127)$ plus 0, ∞, ...
- double precision (`double`) Zahlen:
  $\mathcal{F}^*(2, 53, -1022, 1023)$ plus 0, ∞, ...

Der IEEE-Standard 754

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

Der IEEE-Standard 754

Warum $\mathcal{F}^*(2, 24, -126, 127)$?

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

insgesamt 32 Bit

Der IEEE-Standard 754

Warum $\mathcal{F}^*(2, 53, -1022, 1023)$?

- 1 Bit für das Vorzeichen
- 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 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!

```
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!

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

1.000×2^4

$+ \ 1.000 \times 2^0$ $= \cancel{1.0001 \times 2^4}$

$= 1.000 \times 2^4$

Rundung auf 4 Stellen!

Richtlinien fürs Rechnen mit Fliesskommazahlen

Regel 2:

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

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

1.000×2^4

$+ \ 1.000 \times 2^0$ $= 1.000 \times 2^4$

Addition von 1 hat keinen Effekt!

Beispiel für Regel 2: Harmonische Zahlen

n-te Harmonische Zahl:

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$$
$$= 1/n + 1/(n-1) + \ldots + 1$$

Summe kann vorwärts oder rückwärts berechnet werden.

Beispiel für Regel 2: Harmonische Zahlen

```
// Program: harmonic.C
// 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

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.
- Bei 1 + 1/2 + 1/3 + ... + 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) + ... + 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!

Auslöschungsproblematik, siehe Skript.