8. Fliesskommazahlen II

Fliesskommazahlensysteme; IEEE Standard; Grenzen der Fliesskommaarithmetik; Fliesskomma-Richtlinien; Harmonische Zahlen

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

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

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

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

Bezeichnung:

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

Fliesskommazahlensysteme

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

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

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

Fliesskommazahlensysteme

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

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

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

In Basis-β-Darstellung:

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

Fliesskommazahlensysteme

Darstellungen der Dezimalzahl 0.1 (mit $\beta = 10$):

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Unterschiedliche Darstellungsmöglichkeiten durch Wahl des Exponenten

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 2

Die Zahl 0, sowie alle Zahlen kleiner als $\beta^{e_{\min}}$, haben keine normalisierte Darstellung (greifen wir später wieder auf)

Menge der normalisierten Zahlen

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

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
$\mathbf{1.00_2}$	0.25	0.5	1	2	4
$\mathbf{1.01_2}$	0.3125	0.625	1.25	2.5	5
$\mathbf{1.10_2}$	0.375	0.75	1.5	3	6
$\mathbf{1.11_2}$	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_{0\bullet}d_1d_2$	$\boldsymbol{e = -2}$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

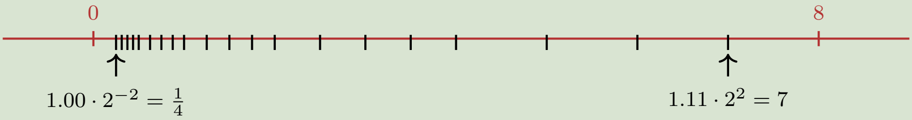
Beispiel $F^*(2,3,-2,\color{red}{2})$ (nur positive Zahlen)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$\boldsymbol{e=2}$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$ (binäres System)

Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$ (binäres System)
- Literale und Eingaben haben $\beta = 10$ (dezimales System)

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

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

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$x = b_0{}_\bullet b_1 b_2 b_3 \ldots$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_{0\bullet}b_1b_2b_3\ldots \\ &= b_0 + 0_\bullet b_1b_2b_3\ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0{\bullet}b_1b_2b_3\ldots \\ &= b_0 + 0{\bullet}b_1b_2b_3\ldots \\ &\Longrightarrow \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0{\bullet}b_1 b_2 b_3 \ldots \\ &= b_0 + 0{\bullet}b_1 b_2 b_3 \ldots \\ &\Longrightarrow \\ (x - b_0) &= 0{\bullet}b_1 b_2 b_3 b_4 \ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$
\begin{aligned}
x &= b_0{}_\bullet b_1 b_2 b_3 \ldots \\
&= b_0 + 0_\bullet b_1 b_2 b_3 \ldots \\
&\Longrightarrow \\
2 \cdot (x - b_0) &= b_1{}_\bullet b_2 b_3 b_4 \ldots
\end{aligned}
$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$
\begin{aligned}
x &= b_0{\bullet}b_1b_2b_3\ldots \\
&= b_0 + 0{\bullet}b_1b_2b_3\ldots \\
&\Longrightarrow \\
2\cdot(x - b_0) &= b_1{\bullet}b_2b_3b_4\ldots
\end{aligned}
$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$
\begin{aligned}
x &= b_0{\bullet}b_1 b_2 b_3 \ldots \\
&= b_0 + 0{\bullet}b_1 b_2 b_3 \ldots \\
&\Longrightarrow \\
2 \cdot (x - b_0) &= b_1{\bullet}b_2 b_3 b_4 \ldots
\end{aligned}
$$

```
for (int b_0; x != 0; x = 2 * (x - b_0)) {
  b_0 = (x >= 1);
  std::cout << b_0;
}
```

Beispiel (binär)

$$\begin{aligned} x &= 1_\bullet 01011 \\ &= 1 + 0_\bullet 01011 \\ &\Longrightarrow \\ 2 \cdot (x - 1) &= 0_\bullet 1011 \end{aligned}$$

Beispiel (binär)

$$\begin{aligned} x &= 1_\bullet 01011 \\ &= 1 + 0_\bullet 01011 \\ &\Longrightarrow \\ 2 \cdot (x-1) &= 0_\bullet 1011 \end{aligned}$$

Beispiel (binär)

$$
\begin{aligned}
x &= 0{\bullet}1011 \\
&= 0 + 0{\bullet}1011 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= 1{\bullet}011
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 0{\bullet}1011 \\
&= 0 + 0{\bullet}1011 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= 1{\bullet}011
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1_\bullet 011 \\
&= 1 + 0_\bullet 011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0_\bullet 11
\end{aligned}
$$

Beispiel (binär)

$$\begin{aligned} x &= 1_\bullet 011 \\ &= 1 + 0_\bullet 011 \\ &\Longrightarrow \\ 2 \cdot (x-1) &= 0_\bullet 11 \end{aligned}$$

Beispiel (binär)

$$\begin{aligned} x &= \color{red}{0}_\bullet 11 \\ &= \color{red}{0} + 0_\bullet 11 \\ &\Longrightarrow \\ 2 \cdot (x - \color{red}{0}) &= 1_\bullet 1 \end{aligned}$$

Beispiel (binär)

$$\begin{aligned} x &= 0{\bullet}11 \\ &= 0 + 0{\bullet}11 \\ &\Longrightarrow \\ 2 \cdot (x - 0) &= 1{\bullet}1 \end{aligned}$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1_\bullet 1 \\
&= 1 + 0_\bullet 1 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 1
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1_\bullet \textcolor{blue}{1} \\
&= 1 + 0_\bullet 1 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= \textcolor{blue}{1}
\end{aligned}
$$

Beispiel (binär)

$$\begin{aligned} x &= 1 \\ &= 1 + 0 \\ &\Longrightarrow \\ 2 \cdot (x - 1) &= 0 \end{aligned}$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1 \\
&= 1 + 0 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0
\end{aligned}
$$

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
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

$\Rightarrow$ $1.0\overline{0011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Binärdarstellungen von 1.1 und 0.1

auf meinem Computer:

$$
\begin{aligned}
\texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\
\texttt{1.1f} &= \underline{1.1000000}238418\ldots
\end{aligned}
$$

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen.

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

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \checkmark \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

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

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
= & 100.101 \cdot 2^{-2}
\end{array}
$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \checkmark \end{array}$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \end{array}$$

4. Runden auf p signifikante Stellen, falls nötig

Rechnen mit Fliesskommazahlen

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

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 1.001 \cdot 2^{0} \checkmark
\end{array}
$$

4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

legt Fliesskommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ (32 bit)

Der IEEE Standard 754

legt Fliesskommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (**`float`**) Zahlen:

  $F^*(2, 24, -126, 127)$ (32 bit)

- Double precision (**`double`**) Zahlen:

  $F^*(2, 53, -1022, 1023)$ (64 bit)

Der IEEE Standard 754

legt Fliesskommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (**`float`**) Zahlen:

  $F^*(2, 24, -126, 127)$ (32 bit)   plus 0, ∞, $\ldots$

- Double precision (**`double`**) Zahlen:

  $F^*(2, 53, -1022, 1023)$ (64 bit)   plus 0, ∞, $\ldots$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Beispiel: 32-bit Darstellung einer Fliesskommazahl

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$\pm$ Exponent Mantisse

$\pm$ $2^{-126}, \ldots, 2^{127}$ $1.00000000000000000000000$

$0, \infty, \ldots$ $\cdots$ $1.11111111111111111111111$

Fliesskomma-Richtlinien Regel 1

Regel 1

Teste keine gerundeten Fliesskommazahlen auf Gleichheit!

Regel 1

Teste keine gerundeten Fliesskommazahlen auf Gleichheit!

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

Regel 1

Teste keine gerundeten Fliesskommazahlen auf Gleichheit!

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

Endlosschleife, weil i niemals exakt 1 ist!

Fliesskomma-Richtlinien

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{array}{r} 1.000 \cdot 2^5 \\ +1.000 \cdot 2^0 \end{array}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$
\begin{aligned}
&1.000 \cdot 2^5 \\
&+1.000 \cdot 2^0 \\
&= 1.00001 \cdot 2^5
\end{aligned}
$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$
\begin{aligned}
& 1.000 \cdot 2^5 \\
+ & 1.000 \cdot 2^0 \\
= & \ 1.00001 \cdot 2^5 \\
\text{„}=\text{“} & \ 1.000 \cdot 2^5 \quad \text{(Rundung auf 4 Stellen)}
\end{aligned}
$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & +1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \\ „=“ \ & 1.000 \cdot 2^5 \quad \text{(Rundung auf 4 Stellen)} \end{aligned}$$

Addition von 1 hat keinen Effekt!

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

Harmonische Zahlen


```cpp
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

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

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Harmonische Zahlen

Regel 2

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

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

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Eingabe: **10000000**

Vorwärts: **15.4037**

Rückwärts: **16.686**

Harmonische Zahlen


```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

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

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Eingabe: **100000000**

Vorwärts: **15.4037**

Rückwärts: **18.8079**

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.

Harmonische Zahlen

Regel 2

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

Harmonische Zahlen — Regel 2

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.

Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^5 + 1$ „=“ 2^5

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

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

Randy Glasbergen, 1996

9. Funktionen I

Funktionsdefinitionen- und Aufrufe, Auswertung von Funktionsaufrufen, Der Typ **`void`**

Potenzberechnung

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;

std::cout << a << "^" << n << " = " << result << ".\n";
```

Potenzberechnung

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;

std::cout << a << "^" << n << " = " << result << ".\n";
```

Potenzberechnung

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;
```

"Funktion pow"

```
std::cout << a << "^" << n << " = " << pow(a,n) << ".\n";
```

Funktion zur Potenzberechnung

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Funktion zur Potenzberechnung

```
double pow(double b, int e){...}
```

Funktion zur Potenzberechnung

```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>

double pow(double b, int e){...}

int main()
{
  std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5, 2) << "\n"; // outputs 2.25
  std::cout << pow(-2.0, 9) << "\n"; // outputs -512

  return 0;
}
```

Funktionsdefinitionen

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
block

Funktionsname

Funktionsdefinitionen

Rückgabetyp

T fname $(T_1$ pname$_1, T_2$ pname$_2, \ldots, T_N$ pname$_N)$
block

Funktionsname

Funktionsdefinitionen

Rückgabetyp

$$T\ \ \text{fname}\ (T_1\ \text{pname}_1, T_2\ \text{pname}_2, \ldots, T_N\ \ \text{pname}_N)$$
$$\text{block}$$

Funktionsname

Formale Argumente

Funktionsdefinitionen

Rückgabetyp *Argumenttypen*

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
block

Funktionsname *Formale Argumente*

Funktionsdefinitionen

Rückgabetyp *Argumenttypen*

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
block

Funktionsrumpf

Funktionsname *Formale Argumente*

Xor

```
// post: returns l XOR r
bool Xor(bool l, bool r)
{
    return l && !r || !l && r;
}
```

Harmonic

```
// PRE: n >= 0
// POST: returns nth harmonic number
//       computed with backward sum
float Harmonic(int n)
{
    float res = 0;
    for (unsigned int i = n; i >= 1; --i)
        res += 1.0f / i;
    return res;
}
```

min

```
// POST: returns the minimum of a and b
int min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
}
```

Funktionsaufrufe

fname (*expression*$_1$, *expression*$_2$, $\ldots$, *expression*$_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.

Funktionsaufrufe

fname (*expression*$_1$, *expression*$_2$, $\ldots$, *expression*$_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyp.

Funktionsaufrufe

fname (*expression*$_1$, *expression*$_2$, …, *expression*$_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyp.

Beispiel: **`pow(a,n)`**: Ausdruck vom Typ **`double`**

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
  $\hookrightarrow$ *call-by-value* (auch *pass-by-value*), dazu gleich mehr
- Funktionsaufruf selbst ist R-Wert.

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
  $\hookrightarrow$ *call-by-value* (auch *pass-by-value*), dazu gleich mehr
- Funktionsaufruf selbst ist R-Wert.

$$\textit{fname}\text{: R-Wert} \times \text{R-Wert} \times \cdots \times \text{R-Wert} \longrightarrow \text{R-Wert}$$

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Aufruf von pow

Auswertung Funktionsaufruf

```
double pow(double b, int e){            b=2.0,e=-2
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){                b=2.0,e=-2
    assert (e >= 0 || b != 0);              // ok
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

result=1.0

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {            e == -2
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;  → b=0.5
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;                              e=2
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

i=0

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

i=0

result=0.5

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)          i=1
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)      i=1
        result * = b;                 result=0.25
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

i=2

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;          result=0.25
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

result=0.25

Rückgabe

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Rückgabe

Wert: 0.25

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Wert: 0.25

Gültigkeit formaler Argumente

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Gültigkeit formaler Argumente

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Gültigkeit formaler Argumente

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Nicht die formalen Argumente **b** und **e** von pow, sondern die hier definierten Variablen lokal zum Rumpf von **main**

Der Typ `void`

```
// POST: "(i, j)" has been written to standard output
???? print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

Der Typ `void`

```
// POST: "(i, j)" has been written to standard output
void print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich
- Verwendung als Rückgabetyp für Funktionen, die *nur* einen Effekt haben

`void`-Funktionen

- benötigen kein **`return`**.
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird oder
- **`return;`** erreicht wird

Funktionen und `return`

Falsch:

```
bool compare(float x, float y) {
  float delta = x - y;
  if (delta*delta < 0.001f) return true;
}
```

Funktionen und `return`

Das Verhalten einer Funktion mit Rückgabetyp ungleich `void` ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne `return` Anweisung erreicht wird.

Falsch:

```
bool compare(float x, float y) {
  float delta = x - y;
  if (delta*delta < 0.001f) return true;
}
```

Der Wert von `compare(10,20)` ist hier undefiniert

Funktionen und `return`

Das Verhalten einer Funktion mit Rückgabetyp ungleich `void` ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne `return` Anweisung erreicht wird.

Besser:

```
bool compare(float x, float y) {
   float delta = x - y;
   if (delta*delta < 0.001f)
    return true;
   else
    return false;
}
```

Alle Ausführungspfade erreichen ein `return`

Funktionen und `return`

Das Verhalten einer Funktion mit Rückgabetyp ungleich `void` ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne `return` Anweisung erreicht wird.

Noch besser (weil einfacher)

```
bool compare(float x, float y) {
   float delta = x - y;
   return delta*delta < 0.001f;
}
```