

# 7. 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.

Bezeichnung:

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

# Flieskommazahlensysteme

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

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

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

In Basis- $\beta$ -Darstellung:

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

# Fließkommazahlensysteme

Beispiel

■  $\beta = 10$

Darstellungen der Dezimalzahl 0.1

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

# Normalisierte Darstellung

Normalisierte Zahl:

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

## Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

## Bemerkung 2

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

# Menge der normalisierten Zahlen

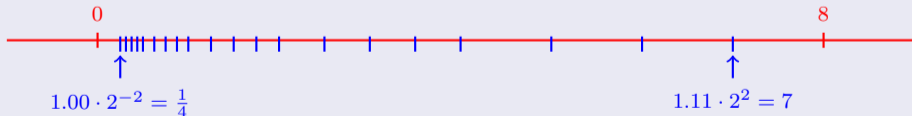
$$F^*(\beta, p, e_{\min}, e_{\max})$$

# Normalisierte Darstellung

Beispiel  $F^*(2, 3, -2, 2)$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
$1.00_2$	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)
- Literale und Eingaben haben  $\beta = 10$   
(dezimales System)
- Eingaben müssen umgerechnet werden!



# Umrechnung dezimal $\rightarrow$ binär

Angenommen,  $0 < x < 2$ .

- Also:  $x' = b_{-1}b_{-2}b_{-3}b_{-4}\dots = 2 \cdot (x - b_0)$
- Schritt 1 (für  $x$ ): Berechnen von  $b_0$ :

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für  $x$ ): Berechnen von  $b_{-1}, b_{-2}, \dots$ :  
Gehe zu Schritt 1 (für  $x' = 2 \cdot (x - b_0)$ )

# Binärdarstellung von 1.1

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

⇒  $1.000\overline{11}$ , periodisch, *nicht* endlich

# Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fließkommazahlensystem.
- `1.1f` und `0.1f` sind nicht `1.1` und `0.1`, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.
- In `diff.cpp`:  $1.1 - 1.0 \neq 0.1$

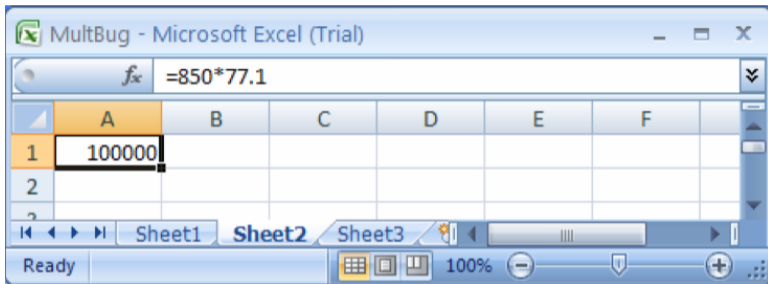
# Binärdarstellungen von 1.1 und 0.1

auf meinem Computer:

$$\begin{aligned} 1.1 &= \underline{1.100000000000000000}888178\dots \\ 1.1f &= \underline{1.1000000}238418\dots \end{aligned}$$

# Der Excel-2007-Bug

```
std::cout << 77.1 * 850; // 65535
```



- 77.1 hat keine endliche Binärdarstellung, wir erhalten  $65534.9999999999927 \dots$
- Für diese und genau 11 andere "seltene" Zahlen war die Ausgabe (und nur diese) fehlerhaft.

# Rechnen mit Fließkommazahlen

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \\ \hline = 1.001 \cdot 2^0 \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl 2. Binäre Addition der Signifikanden 3. Renormalisierung 4. Runden auf  $p$  signifikante Stellen, falls nötig

# Der IEEE Standard 754

- legt Fließkommazahlensysteme und deren Rundungsverhalten fest
- wird fast überall benutzt
- Single precision (`float`) Zahlen:

$$F^*(2, 24, -126, 127) \quad \text{plus } 0, \infty, \dots$$

- Double precision (`double`) Zahlen:

$$F^*(2, 53, -1022, 1023) \quad \text{plus } 0, \infty, \dots$$

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

# Der IEEE Standard 754

Warum

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

- 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)(254 mögliche Exponenten, 2 Spezialwerte: 0,  $\infty$ , ...)

⇒ insgesamt 32 Bit.



# Der IEEE Standard 754

Warum

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

- 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:  $0, \infty, \dots$ )

⇒ insgesamt 64 Bit.

## 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!

## Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

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

“=”  $1.000 \cdot 2^5$  (Rundung auf 4 Stellen)

Addition von 1 hat keinen Effekt!

- 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.

---

```
// 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;
}
```

---

Ergebnisse:



```
Compute H_n for n =? 100000000  
Forward sum = 15.4037  
Backward sum = 16.686
```



```
Compute H_n for n =? 1000000000  
Forward sum = 15.4037  
Backward sum = 18.8079
```

Beobachtung:

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

Erklärung:

- Bei  $1 + 1/2 + 1/3 + \dots$  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

# 8. Funktionen I

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

# Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm: Unterteilung in kleine Teilaufgaben, jede davon durch eine Funktion realisiert

⇒ Prozedurales Programmieren, Prozedur: anderes Wort für Funktion.

# Beispiel: 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 << " = " << resultpow(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;
}
```

# Programm zur Potenzberechnung

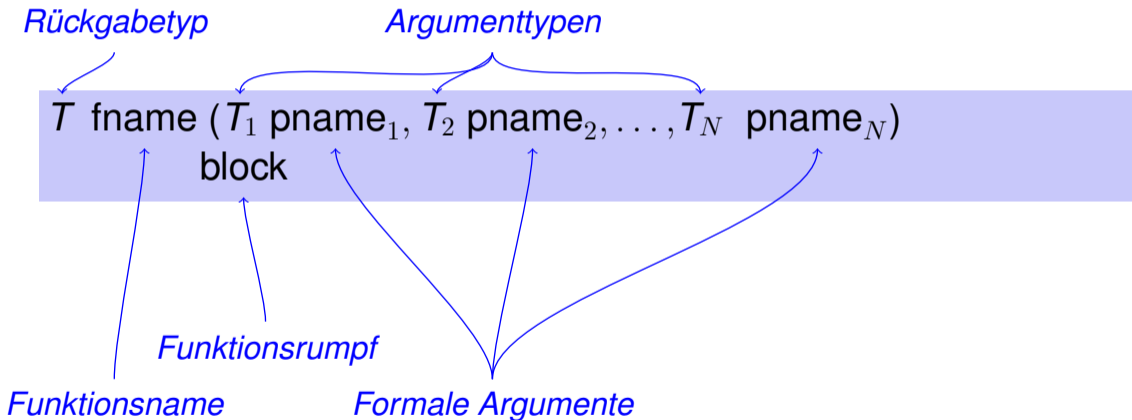
```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>
```

```
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



# Funktionsdefinitionen

- dürfen nicht *lokal* auftreten, also nicht in Blocks, nicht in anderen Funktionen und nicht in Kontrollanweisungen
- können im Programm ohne Trennsymbole aufeinander folgen

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

int main ()
{
    ...
}
```



# Beispiel: Xor

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

# Beispiel: 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;
}
```

# Beispiel: min

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

# Funktionsaufrufe

$fname ( expression_1, expression_2, \dots, expression_N )$

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabety. Wert und Effekt wie in der Nachbedingung der Funktion *fname* angegeben.

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

# Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente und Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert.

*f*name: R-Wert  $\times$  R-Wert  $\times \dots \times$  R-Wert  $\longrightarrow$  R-Wert

# Auswertung eines Funktionsaufrufes

- Auswertung der Aufrufargumente
- Initialisierung der formalen Argumente mit den resultierenden Werten
- Ausführung des Funktionsrumpfes: formale Argumente verhalten sich dabei wie lokale Variablen
- Ausführung endet mit `return expression;`

Rückgabewert ergibt den Wert des Funktionsaufrufes.

# Beispiel: 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

Rückgabe

# Formale Funktionsargumente<sup>6</sup>

- Deklarative Region: Funktionsdefinition
- sind ausserhalb der Funktionsdefinition *nicht* sichtbar
- werden bei jedem Aufruf der Funktion neu angelegt (automatische Speicherdauer)
- Änderungen ihrer Werte haben keinen Einfluss auf die Werte der Aufrufargumente (Aufrufargumente sind R-Werte)

---

<sup>6</sup>oder „formale Parameter“



# 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`

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

```
// 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;
}
```

# void-Funktionen

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

Ausdruck vom Typ `void` (z.B. Aufruf einer Funktion mit Rückgabety `void`)

# Vor- und Nachbedingungen

- beschreiben (möglichst vollständig) was die Funktion „macht“
- dokumentieren die Funktion für Benutzer / Programmierer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert
- Vor- und Nachbedingungen machen – unter der Annahme ihrer Korrektheit – Aussagen über die Korrektheit eines Programmes möglich.

# Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsbereich* der Funktion.

$0^e$  ist für  $e < 0$  undefiniert

```
// PRE: e >= 0 || b != 0.0
```

# Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Funktionsaufruf?
- Spezifiziert *Wert* und *Effekt* des Funktionsaufrufes.

Hier nur Wert, kein Effekt.

```
// POST: return value is  $b^e$ 
```

# Vor- und Nachbedingungen

- sollten korrekt sein:
- *Wenn* die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Funktion `pow`: funktioniert für alle Basen  $b \neq 0$

# Vor- und Nachbedingungen

- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage.
- C++-Standard-Jargon: „Undefined behavior“.

Funktion `pow`: Division durch 0



# Vor- und Nachbedingungen

- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so *stark* wie möglich sein (möglichst detaillierte Aussage)

# Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be
```

ist formal inkorrekt:

- Überlauf, falls e oder b zu gross sind
- $b^e$  vielleicht nicht als double Wert darstellbar (Löcher im Wertebereich)

# Fromme Lügen... sind erlaubt.

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be
```

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr kompliziert. Wir abstrahieren und geben die mathematischen Bedingungen an.  $\Rightarrow$  Kompromiss zwischen formaler Korrektheit und lascher Praxis.

# Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.
- Wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

# ... mit Assertions

```
#include <cassert>
...
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e) {
    assert (e >= 0 || b != 0);
    double result = 1.0;
    ...
}
```

# Nachbedingungen mit Assertions

- Das Ergebnis “komplizierter” Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von `assert` für die Nachbedingung

```
// PRE: the discriminant  $p*p/4 - q$  is nonnegative
// POST: returns larger root of the polynomial  $x^2 + p x + q$ 
double root(double p, double q)
{
    assert( $p*p/4 \geq q$ ); // precondition
    double x1 = - p/2 + sqrt( $p*p/4 - q$ );
    assert( $\text{equals}(x1*x1+p*x1+q,0)$ ); // postcondition
    return x1;
}
```

# Assertions: Syntax und Semantik

```
assert ( expression )
```

- *expression*: Ausdruck, dessen Wert nach `bool` konvertierbar ist.
- Semantik:
  - Fall *expression* `== true`: kein Effekt
  - Falls *expression* `== false`: Programm wird mit entsprechender Fehlermeldung abgebrochen.

# Assertions – Empfehlung

- Assertions sind ein wichtiges Werkzeug zur Fehlervermeidung während der Programmentwicklung, wenn man sie *konsequent* und *oft* einsetzt.
- Der Laufzeit-Overhead ist vernachlässigbar. Darüber hinaus kann der Compiler angewiesen werden, die Assertions „hinauszukompilieren“, also zu ignorieren.



# Ausnahmen (Exception Handling)

- Assertions sind ein grober Hammer; falls eine Assertion fehlschlägt, wird das Programm hart abgebrochen.
- C++ bietet elegantere Mittel (Exceptions), um auf solche Fehlschläge situationsabhängig (und oft auch ohne Programmabbruch) zu reagieren.
- “Narrensichere” Programmen sollten nur im Notfall abbrechen und deshalb mit Exceptions arbeiten; für diese Vorlesung führt das aber zu weit.