Einführung in die Programmierung mit C++

Numerische Daten

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Inhalt

Überblick

- ganze Zahlen
  - char, short, int, long
  - ASCII code
  - signed vs. unsigned, size_t
- Gleitkommazahlen
  - float, double, long double

Essentiell: Verstehe die Grenzen der Computerarithmetik!

Inhalt

- Ganzen Zahlen sind als Binärzahlen kodiert, z.B. **int** i=8 als 00000000 00000000 00000000 00001000.
- Ganzzahlige Datentypen sind **char** (1 Byte), **short** (2 Bytes), **int** (4 Bytes) und **long** (8 Bytes) mit und ohne (**unsigned**) Vorzeichen.
- Negative ganze Zahlen werden als Zweierkomplement ihres Betrages dargestellt, d.h. Einerkomplement bilden durch $0 \rightarrow 1$, $1 \rightarrow 0$ und 1 hinzuaddieren, um zur negativen Zahl gleichen Betrages zu kommen, z.B. **int** i=−8 wird als

  11111111 11111111 11111111 11111000

  dargestellt.

→ Live: Experimente mit to_bin()

Ganzzahlige Variablen vom Datentyp **char** belegen ein Byte im Speicher. Sie werden bei Ausgabe (z.B. auf den Bildschirm) als darstellbare Zeichen oder mit Sonderzeichen assoziierte Aktionen (z.B. *backspace*) interpretiert.

Die entsprechenden Zeichen sind nach dem ASCII Standard[1] kodiert, z.B. gibt

```
1 int main() {
2   for (char c=33;c<127;c++) std::cout << c << " ";
3   return 0;
4 }
```

mithilfe einer → for-Schleife alle darstellbaren Zeichen auf den Bildschirm aus:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ `
a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Sonderzeichen (z.B. ein Leerzeichen für c==32 oder *backspace* für c==8) sind durch c==0...32,127 kodiert.

[1] American Standard Code for Information Interchange

AUFWACHEN!

Was wird ausgegeben?

```
1  #include <iostream>
2
3  int main() {
4      using namespace std;
5      char c=8;
6      cout << "1" << c << "1" << endl;
7      return 0;
8  }
```

Wach?

```
1  #include <iostream>
2
3  int main() {
4      using namespace std;
5      char c=8;
6      cout << "1" << c << "1" << endl;
7      return 0;
8  }
```

1

```
1  // naumann@stce.rwth-aachen.de
2
3  #include <iostream>
4
5  int main() {
6    using namespace std;
7    char c; cin >> c;
8    cout << c << " (" << static_cast<short>(c) << ")" << endl;
9    short s; cin >> s;
10   cout << static_cast<char>(s) << endl;
11   return 0;
12 }
```

→ Live: Experimente

AUFWACHEN!

Was wird ausgegeben?

```
1  #include<iostream>
2  #include "to_bin.hpp"
3
4  int main() {
5      char c=48;
6      to_bin(c);
7      std::cout << c << std::endl;
8      return 0;
9  }
```

Wach?

```
1  #include<iostream>
2  #include "to_bin.hpp"
3
4  int main() {
5     char c=48;
6     to_bin(c);
7     std::cout << c << std::endl;
8     return 0;
9  }
```

00110000
0

16+32=48

Ganze Zahlen

<limits>

Das Kapitel <limits> der Standardbibliothek stellt eine Reihe von Operationen zur Analyse von und Arbeit mit numerische Datentypen zur Verfügung, z.B.

```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<limits>
4
5  int main() {
6    using namespace std; using T=int;
7    cout << numeric_limits<T>::is_exact << endl; // 1
8    cout << numeric_limits<T>::max() << endl; // 2147483647
9    cout << numeric_limits<T>::min() << endl; // -2147483648
10   cout << numeric_limits<T>::digits << endl; // 31
11   cout << numeric_limits<T>::digits10 << endl; // 9
12   return 0;
13 }
```

Das Auftreten von Werten, die kleiner als -2147483648 oder größer als 2147483647 sind, nennt man *overflow*. Fehler werden erzeugt.

→ Live: Experimente mit **char**,**short**,**long**

Ganze Zahlen

<limits> und to_bin()


```
1  // naumann@stce.rwth-aachen.de
2  #include "to_bin.hpp"
3  #include<iostream>
4  #include<limits>
5
6  int main() {
7    using namespace std;
8    cout << numeric_limits<int>::min() << endl;
9    to_bin(numeric_limits<int>::min());
10   cout << numeric_limits<int>::max() << endl;
11   to_bin(numeric_limits<int>::max());
12   return 0;
13 }
```

generiert folgende Ausgabe:

```
1  -2147483648
2  10000000 00000000 00000000 00000000
3  2147483647
4  01111111 11111111 11111111 11111111
```

Vorzeichenlose (**unsigned**) ganze Zahlen sind immer >=0. Z.B. terminiert folgendes Programm nie:

```
1  #include<iostream>
2
3  int main() {
4    using namespace std;
5    size_t i;
6    for (i=0;i<10;i++) cout << i << endl;
7    for (;i>=0;i--) cout << i << endl;
8    return 0;
9  }
```

Der Compiler weiß das und kann eine entsprechende Warnung generieren:

```
In function ‘int main()’:
warning: comparison of unsigned expression >= 0 is always true [-Wtype-limits]
    7 |   for (;i>=0;i--) cout << i << endl;
```

Inhalt

Reelle Zahlen $x \in \boldsymbol{R}$ werden intern als Gleitkommazahlen (GKZ) mit Basis β, Genauigkeit t und Exponentenbereich $[L, U]$ wie folgt dargestellt:

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}} \right) \beta^e$$

wobei $0 \leq d_i \leq \beta - 1$ for $i = 0, \ldots, t-1$ and $L \leq e \leq U$. Die Basis-β Ziffernsequenz $m = d_0 d_1 \ldots d_{t-1}$ wird als Mantisse und e als Exponent bezeichnet.

Ein Gleitkommazahlensystem heißt normalisiert wenn $d_0 \neq 0$ falls nicht $x = 0$, d.h. $1 \leq m < \beta$.

Das Auftreten von Werten, die kleiner als der kleinste darstellbare Wert sind, nennt man *underflow*. Sie werden als 0 dargestellt. Damit kann in Gleitkommaarithmetik die Division durch die Differenz zweier nicht (aber fast) gleicher Zahlen eine $\rightarrow$ Ausnahme generieren. *overflow* is analog definiert.

Es sei $\beta = 10$, $t = 3$ und $[L, U] = [-2, 2]$ (eine Ziffer im Exponenten) normalisiert.

$$0.0127 = \left(1 + \frac{2}{10} + \frac{7}{10^2}\right) \cdot 10^{-2} = 1.27 \cdot 10^{-2}$$

$$98.1 = \left(9 + \frac{8}{10} + \frac{1}{10^2}\right) \cdot 10^{1} = 9.81 \cdot 10$$

$$1 = \left(1 + \frac{0}{10} + \frac{0}{10^2}\right) \cdot 10^{0} = 1.00 \cdot 1$$

betragsmäßig kleinste Zahl: $0.01 = 1.00 \cdot 10^{-2}$ =1e−2 $\Rightarrow$ *underflow*

betragsmäßig größte Zahl: $999 = 9.99 \cdot 10^{2}$ =1e2 $\Rightarrow$ *overflow*

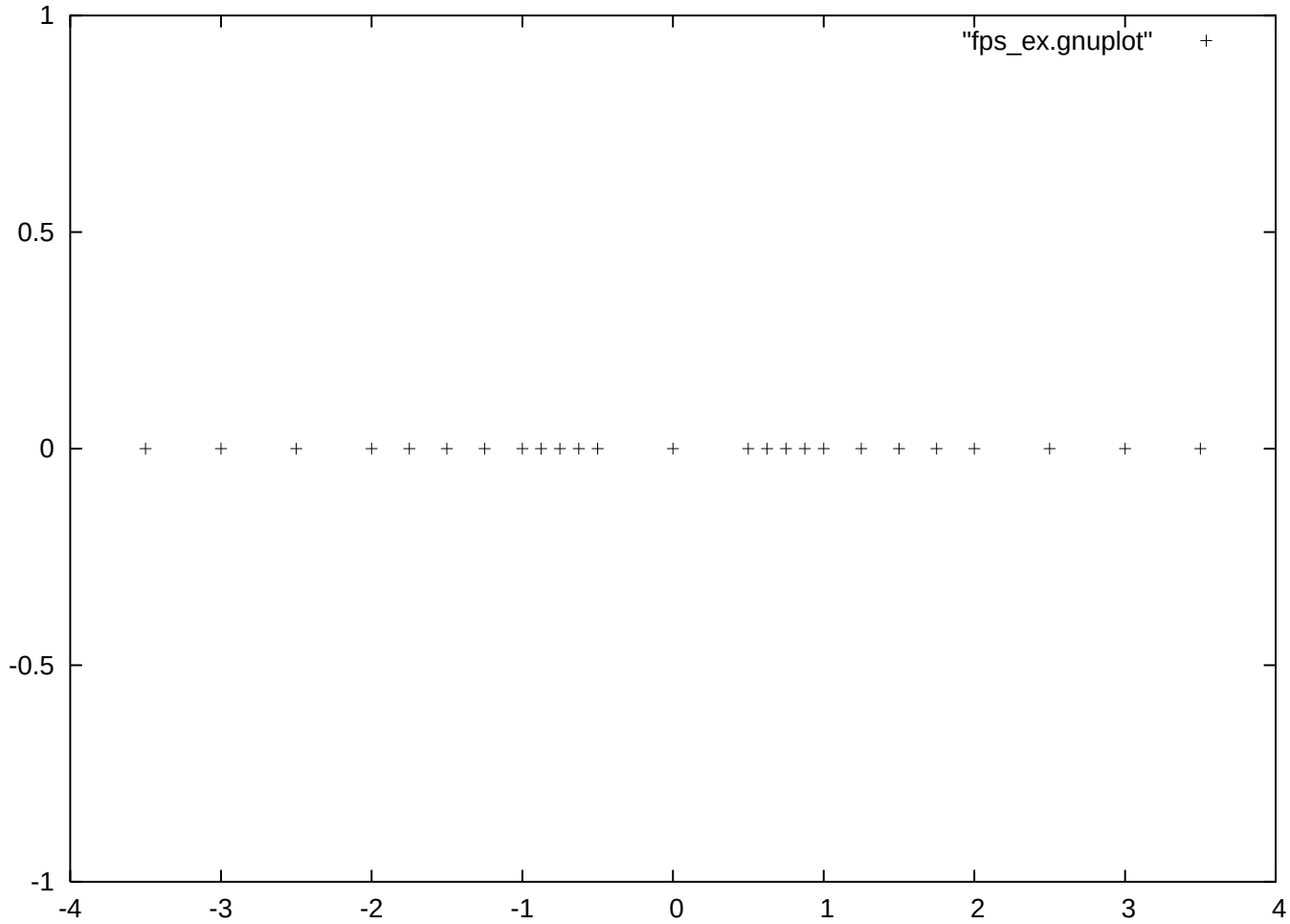
Beachte: Exponentenschreibweise in C++ Programmen, z.B. 20.3e−1==2.03.

Es sei $\beta = 2$, $t = 3$ und $[L, U] = [-1, 1]$. Das normalisierte Gleitkommazahlensystem besteht aus den folgenden 25 Elementen:

$$
\begin{aligned}
&0 \\
&\pm 1.00_2 * 2^{-1} = \pm 0.5_{10}, \quad \pm 1.01_2 * 2^{-1} = \pm 0.625_{10} \\
&\pm 1.10_2 * 2^{-1} = \pm 0.75_{10}, \quad \pm 1.11_2 * 2^{-1} = \pm 0.875_{10} \\
&\pm 1.00_2 * 2^{0} = \pm 1_{10}, \quad \pm 1.01_2 * 2^{0} = \pm 1.25_{10} \\
&\pm 1.10_2 * 2^{0} = \pm 1.5_{10}, \quad \pm 1.11_2 * 2^{0} = \pm 1.75_{10} \\
&\pm 1.00_2 * 2^{1} = \pm 2_{10}, \quad \pm 1.01_2 * 2^{1} = \pm 2.5_{10} \\
&\pm 1.10_2 * 2^{1} = \pm 3_{10}, \quad \pm 1.11_2 * 2^{1} = \pm 3.5_{10} \quad .
\end{aligned}
$$

Gleitkommazahlen

Beispiel: $\beta = 2, t = 3, [L, U] = [-1, 1]$



... verwendet 32 Bit:

- 23 Bit für die Mantisse (24. Bit gleich 1 wg. Normalisierung)
- 8 bit für den Exponent
- 1 Vorzeichenbit

und damit 6 signifikante Ziffern in der Dezimaldarstellung mit Minimalwert 1.17549e-38 und Maximalwert 3.40282e+38.

Zum vorzeichenbehafteten Exponent wird $2^7 - 1 = 127$ hinzuaddiert (*biased exponent*), womit Darstellung nach $1 \ldots 254$ verschoben wird und was den Vergleich zweier GKZ vereinfacht.

Die kleinste positive Gleitkommazahl einfacher Genauigkeit, deren gerundete Summe mit 1.0 größer als 1.0 ist, ist durch `std::numeric_limits<float>::epsilon()` gegeben, d.h. 1.0 + `std::numeric_limits<float>::epsilon()` > 1.0.

```
1 to_bin(static_cast<float>(1.0));
2 cout << pow(2,
3                 pow(2,0)+pow(2,1)+pow(2,2)+pow(2,3)+pow(2,4)
4                 +pow(2,5)+pow(2,6) // exponent + 2^7-1 (bias)
5                 -(pow(2,7)-1) // unbias
6         )
7         *1 // mantissa
8         << endl;
```

generiert Ausgabe

```
00111111 10000000 00000000 00000000
1
```

Beachte: Gleitkommakonstanten werden standardmäßig als **double** interpretiert, z.B. generiert to_bin(1.0) die Ausgabe

```
00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
```

Gleitkommazahlen

static_cast<float>(2.1)


```
1 to_bin(static_cast<float>(2.1));
2 cout << pow(2,
3               pow(2,7) // exponent + 2^7-1 (bias)
4               -(pow(2,7)-1) // unbias
5              )
6              *(1+pow(2,-5)+pow(2,-6)+pow(2,-9)+pow(2,-10)
7              +pow(2,-13)+pow(2,-14)+pow(2,-17)+pow(2,-18)
8              +pow(2,-21)+pow(2,-22)) // mantissa
9       << endl;
```

generiert Ausgabe

```
01000000 00000110 01100110 01100110
2.1
```

```
1 to_bin(static_cast<float>(0.0));
2 cout << pow(2,0 // exponent + 2^7-1 (bias)
3             -(pow(2,7)-1)// unbias
4           )
5             *1 // mantissa
6     << endl;
```

generiert Ausgabe

```
00000000 00000000 00000000 00000000
5.87747e-39
```

Alle GKZ sind aufgrund der Normalisierung ungleich Null. Daher ist eine spezielle Definition der Null (alle bits in GKZ verschwinden) notwendig.

... verwendet 64 Bit:

- 52 Bit für die Mantisse (53. Bit gleich 1 wg. Normalisierung)
- 11 bit für den Exponent
- 1 Vorzeichenbit

und damit 15 signifikante Ziffern in der Dezimaldarstellung mit Minimalwert 2.22507e-308 und Maximalwert 1.79769e+308.

Zum vorzeichenbehafteten Exponent wird $2^{10} - 1 = 1023$ hinzuaddiert (*biased exponent*), womit Darstellung nach $1 \ldots 2046$ verschoben wird.

Die kleinste positive Gleitkommazahl einfacher Genauigkeit, deren gerundete Summe mit 1.0 größer als 1.0 ist, ist durch std::numeric_limits<**double**>::epsilon() gegeben, d.h. 1.0 + std::numeric_limits<**double**>::epsilon() $>$ 1.0.

Im Vergleich zu float benötigt Arithmetik mit double mehr Speicher und ist langsamer (aber genauer).

Gleitkommazahlen

<limits>

```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<limits>
4
5  int main() {
6    using namespace std; using T=float;
7    cout << numeric_limits<T>::is_exact << endl; // 0
8    cout << numeric_limits<T>::epsilon() << endl; // 1.19209e-07
9    cout << numeric_limits<T>::max() << endl; // 3.40282e+38
10   cout << numeric_limits<T>::min() << endl; // 1.17549e-38
11   cout << numeric_limits<T>::lowest() << endl; // -3.40282e+38
12   cout << numeric_limits<T>::digits << endl; // 24
13   cout << numeric_limits<T>::min_exponent << endl; // -125
14   cout << numeric_limits<T>::max_exponent << endl; // 128
15   return 0;
16 }
```

Gleitkommazahlen

<limits> und to_bin()


```
1  // naumann@stce.rwth-aachen.de
2  #include "to_bin.hpp"
3  #include<iostream>
4  #include<cmath>
5  #include<limits>
6
7  int main() {
8    using namespace std; using T=float;
9    to_bin(numeric_limits<T>::epsilon());
10   to_bin(numeric_limits<T>::max());
11   to_bin(numeric_limits<T>::min());
12   to_bin(numeric_limits<T>::lowest());
13   return 0;
14 }
```

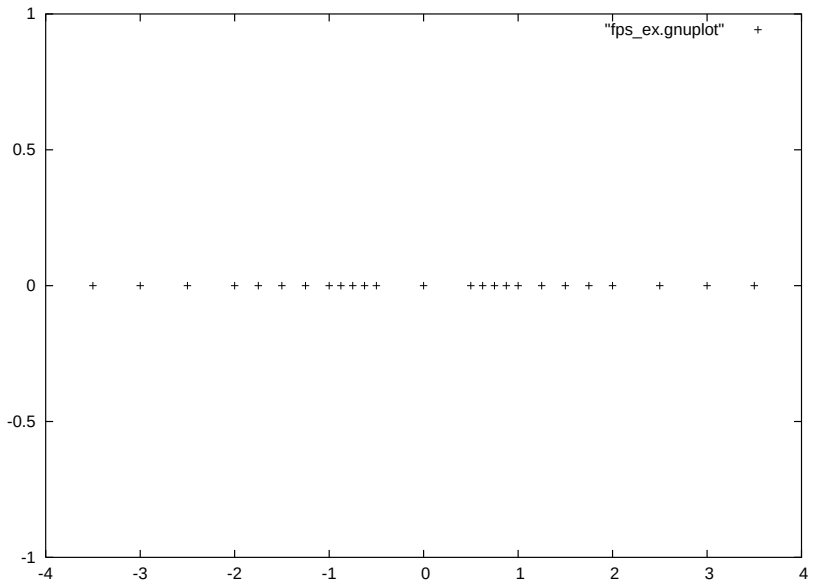
generiert folgende Ausgabe:

```
1  00110100 00000000 00000000 00000000
2  01111111 01111111 11111111 11111111
3  00000000 10000000 00000000 00000000
4  11111111 01111111 11111111 11111111
```

Nicht exakt darstellbare reelle Zahlen werden typischerweise zur nächstgelegenen darstellbaren Zahl gerundet. Uneindeutigkeit wird dabei durch Runden zum Nachbarn mit gerader Mantisse (letztes Bit gleich Null) aufgelöst.

Beispiel:

Im $(\beta = 2, t = 3, [L, U] = [-1, 1])$ Gleitkommazahlensystem ergibt sich $1.126 \approx 1.25$ und $1.125 \approx 1$.



```
1 double h=1e-13,a=1,b,c,d;
2 b=a+h; c=b-a; d=c/h;
3 std::cout << h << " " << a
4           << " " << b << " "
5           << c << " " << d
6           << std::endl;
```

generiert Ausgabe

```
1e-13 1 1 9.99201e-14 0.999201
```

`d` sollte eigentlich gleich 1 sein ...

Wenn zwei Gleitkommazahlen x und y bis auf die letzten k Ziffern der Mantisse identisch sind und wir $z = x - y$ berechnen, so hat das Ergebnis z nur k Ziffern Genauigkeit. Die Verwendung von z in weiteren Berechnungen kann somit einen negativen Einfluss auf das Gesamtresultat haben.

Die Approximation von Ableitungen mittels finiter Differenzen ist ein Paradebeispiel; siehe spätere Fallstudie.

Zusammenfassung

Überblick

Ganze Zahlen

- ASCII Code
- <limits>

Gleitkommazahlen

- **float**
- **double**
- Runden
- Auslöschung