§2 Arithm. und log. Ausdrücke – Zahlen

Leitidee: *Die Darstellung von Zahlen durch eine feste Zahl von Bits erfordert eine Reihe von Kompromissen*

- Ganzzahl- oder Gleitpunktarithmetik?
- Dual- und Hexadezimalzahlsystem
- Arithmetische Datentypen in C++ - Überblick
- Binärdarstellungen für vorzeichenbehaftete ganze Zahlen
- Zahlbereiche und Zahlkonstanten für vorzeichenbehaftetete ganze Zahlen
- Vorzeichenlose ganze Zahlen
- Gleitpunktzahlen und wichtige Kennzahlen der Maschinenarithmetik
- IEEE-Arithmetik als standardisierte Gleitpunktarithmetik

Ganzzahl- oder Gleitpunktarithmetik? (I)

Ganzzahlarithmetik

Bsp.: 1078 (ganze Dezimalzahl)
Idee: Darstellung als Dualzahl (mit fest vorgeg. Bitanzahl)

- – Stark eingeschränkter Zahlbereich.
  Mit z.B. 32 Bits lässt sich nur ein Zahlbereich von ca. $-2^{31}..2^{31}$, d.h $-2.1 \cdot 10^9 \ldots 2.1 \cdot 10^9$ lückenlos darstellen
- \+ Exakte Ergebnisse bei Addition, Subtraktion und Multiplikation, sofern Zahlbereich nicht verlassen wird.
- – Umdefinition der Division erforderlich, sofern immer ganzzahlige Ergebnisse gewünscht (“Ganzzahldivision“)

Ganzzahl- oder Gleitpunktarithmetik? (II)

Gleitpunktarithmetik

Idee: $0.8868177 \cdot 10^{31} \rightarrow \underbrace{0.4372349}_{\textit{Mantisse}} \cdot 2^{\underbrace{104}_{\textit{Exp.}}}$

(Dualdarst. von Mantisse und Exp. getrennt speichern)

+ Großer Zahlbereich.
  Mit 32 Bits lässt sich ein positiver Zahlbereich von ca. $10^{-38} \ldots 10^{38}$ bei einer Genauigkeit von 6 - 7 Dezimalstellen darstellen.
− Normalerweise Rundung erforderlich, dadurch Genauigkeitsverlust.
+ Erreichbar: Grundrechenarten bis auf Rundung korrekt, sofern Zahlbereich nicht verlassen wird.
− Auch innerhalb des Zahlbereichs nicht immer: $x + 1 \neq x$
  (Schlecht für Schleifenvariable und Adressrechnung)
− Bei gleicher Bitzahl: Ganze Zahlen $\not\subset$ Gleitpunktzahlen

Dual- und Hexadezimalsystem

Sei $B \in \mathbb{N}$, $B \geq 2$. Für jedes $z \in \mathbb{Z}$ existieren $v \in \{-1, 1\}$, $N \in \mathbb{N}_0$ und $z_0, \ldots, z_N \in \{0, 1, \ldots, B-1\}$, so dass

$$z = v \cdot \sum_{i=0}^{N} B^i z_i.$$

B heißt Basis des Zahlsystems, $z_0, \ldots, z_N$ Ziffern der Zahl z.

B	*Bezeichnung*	*Zifferndarstellung*
2	Dualsystem	0 1
8	Oktalsystem	0 1 2 3 4 5 6 7
10	Dezimalsystem	0 1 2 3 4 5 6 7 8 9
16	Hexadezimalsystem	0 1 2 3 4 5 6 7 8 9 a b c d e f

- Speicherung von Zahlen (Daten) mit Dualziffern (Bits)
- Bei Ein/Ausgabe ggf. Umrechnung aus/in Dezimalsystem
- Hexadezimalzahlen zur Darstellung von Dualzahlen (und Bitmustern) übersichtlicher!

Bsp.: `a` `3` `b` → `1010` `0011` `1011` [dezimal: `2619`]

Arithmetische Datentypen in C++ - Überblick

- In C++ gibt es Datentypen für *ganze Zahlen* und für *Gleitpunktzahlen*.
- Die interne Darstellung ist für ganze und für Gleitpunktzahlen unterschiedlich; allerdings ist sie in C++ nicht standardisiert.
- Arithmetische Operationen unterscheiden sich zum Teil in ihrer Wirkung, das gilt insbesondere für die Division.
- Ganzzahl- und Gleitpunktdatentypen haben stark unterschiedliche Zahlbereiche.
- Sowohl für Ganzzahl- als auch für Gleitpunktdatentypen gibt es Ausprägungen mit unterschiedlicher Datenlänge (Bitzahl) und unterschiedlich großem Zahlbereich.
- Bei ganzzahligen Datentypen gibt es zu jedem *vorzeichenbehafteten* Datentyp einen korrespondierenden *vorzeichenlosen* Datentyp.

Binärdarstellungen ganzer Zahlen mit *n* Dualziffern

Darstellung mit Vorzeichen (ungebräuchlich)

$$z = v \cdot \sum_{i=0}^{n-2} 2^i z_i \quad \text{mit } z_i \in \{0,1\},\ v = (-1)^s \text{ wobei } s \in \{0,1\}$$

Bitmuster:

s	z_{n-2}	z_{n-3}	.....	z_1	z_0

Bsp.: $n = 16$

Darstellung		*Wert*
0000000000010011		19
1000000000010011		-19
0111111111111111	$2^{15} - 1 =$	32767
1111111111111111	$-2^{15} + 1 = -$	32767

- Zahlbereich: $\{-2^{n-1} + 1, \ldots, 2^{n-1} - 1\}$
- 0 besitzt zwei Darstellungen, Zahlbereich symmetrisch
- Gesonderte Behandlung des Vorzeichenbits erforderlich

Binärdarstellungen ganzer Zahlen - Fortsetzung

[$*$] **Zweierkomplementdarstellung** (üblich)

$z \in \{-2^{n-1}, \ldots, 2^{n-1}-1\}$ wird dargestellt durch die Dualziffern von $\tilde{z} := \begin{cases} z & \text{für } 0 \leq z \leq 2^{n-1}-1 \\ z+2^n & \text{für } -2^{n-1} \leq z < 0 \end{cases}$

Bitmuster: $\boxed{\tilde{z}_{n-1}}\boxed{\tilde{z}_{n-2}} \ldots\ldots \boxed{\tilde{z}_1}\boxed{\tilde{z}_0}$

Bsp.: $n = 16$

Darstellung		*Wert*
0000000000010011		19
1111111111101101		-19
0000000000000000		0
0000000000000001		1
1111111111111111		-1
0111111111111111	$2^{15}-1 =$	32767
1000000000000001	$-2^{15}+1 =$	-32767
1000000000000000	$-2^{15} =$	-32768

Binärdarstellungen ganzer Zahlen - Fortsetzung II

[∗] Zweierkomplementdarstellung - Fortsetzung

- Zahlbereich: $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$
- 0 besitzt genau eine Darstellung, Zahlbereich ist unsymmetrisch.
- Höchstes Bit kann als Vorzeichenbit interpretiert werden.
- Entscheidend ist die einfache Berechenbarkeit der Negation positiver Zahlen: Umklappen aller Bits („1-Komplement“) und Addition von 1

$$\left[z \in \{1, \ldots, 2^{n-1} - 1\} : \; z = \sum_{i=0}^{n-1} 2^i z_i \text{ mit } z_i \in \{0, 1\}, z_{n-1} = 0 \right.$$

$$\left. \widetilde{-z} = 2^n + (-z) = 2^n - 1 - z + 1 = \sum_{i=0}^{n-1} 2^i (1 - z_i) + 1 \right]$$

- Vorteilhaft: Rückführung der Subtraktion auf Addition und Komplementbildung
- Oft werden die Addition, Subtraktion und Multiplikation modulo 2^n durchgeführt (keine Überlaufbehandlung)

Binärdarstellungen ganzer Zahlen - Fortsetzung III

[$*$] Zweierkomplementdarstellung - Math. Hintergrund

- $\phi : \mathbb{Z} \to {}^{\mathbb{Z}}/_{2^n\mathbb{Z}}$, $z \to [z]_{2^n}$ surjektiver Ringhomomorphismus
- Die Zweierkomplementdarstellung wird erhalten, wenn als Repräsentant der Äquivalenzkl. $[z]_{2^n} \quad \tilde{z} \in \{0, \ldots, 2^n - 1\}$ gewählt wird, sofern $z \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$
- $\phi|_{\{-2^{n-1},\ldots,2^{n-1}-1\}}$ ist bijektiv $(*)$
- Aus $\phi(z) + \phi(z') = \phi(z + z')$ und der Bijektivität $(*)$ lässt sich ablesen, dass die Addition der Zweierkompl.darst. modulo 2^n die Zweierkompl.darst. der Summe liefert, wenn $z, z', z + z' \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$
- Entsprechend für die Subtraktion und die Multiplikation.

Ganzzahlige Datentypen mit Vorzeichen in C++

Datentyp	*Mindestbitzahl*	*g++-4.3 (IA/32)*	*g++-7.5 (AMD/64)*
`short`	16	16	16
`int`	16	32	32
`long`	32	32	64

- Hauptsächlich verwendet: `int`
- Bei 64-Bit-Betriebssystemen ist `long` meistens 64 Bit lang.
- Bei DOS-Compilern (z.B. Turbo C++) war `int` nur 16 Bit lang.
- Addition, Subtraktion und Multiplikation liefern math. exakte Ergebnisse, wenn die Operanden *und* das math. korrekte Ergebnis im Zahlbereich des Datentyps liegen.
- In der Regel gibt es *keine* Überlaufbehandlung (d.h. keine Fehlermeldungen bei Überschreitungen des Zahlbereichs).
- Der Zahlbereich von `short` ist eine *echte* Teilmenge des Zahlbereichs von `long`.

Zahlbereiche von ganzen Zahlen mit Vorz. in C++

Typ	*g++-4.3 (IA/32)*	*g++-7.5 (AMD/64)*
`short`	$-32768 \ldots 32767$	$-32768 \ldots 32767$
`int`	$-2.1 \cdot 10^9 \ldots 2.1 \cdot 10^9$	$-2.1 \cdot 10^9 \ldots 2.1 \cdot 10^9$
`long`	$-2.1 \cdot 10^9 \ldots 2.1 \cdot 10^9$	$-9.2 \cdot 10^{18} \ldots 9.2 \cdot 10^{18}$

- Die Zahlbereiche sind in `limits` definiert, z.B. für `int`
  `std::numeric_limits<int>::min()`
  `std::numeric_limits<int>::max()`
- Für ihre Benutzung im Programm ist daher `#include <limits>` erforderlich!
- Namespace-Anw. `using namespace std;` verringert Schreibaufwand!

Vorzeichenlose ganze Zahlen

Typ	*g++-4.3 (IA/32)*	*g++-7.5 (AMD/64)*
`unsigned short`	$0 \ldots 65535$	$0 \ldots 65535$
`unsigned int`	$0 \ldots 4.3 \cdot 10^9$	$0 \ldots 4.3 \cdot 10^9$
`unsigned long`	$0 \ldots 4.3 \cdot 10^9$	$0 \ldots 1.8 \cdot 10^{19}$

- Die Darstellung vorzeichenloser Zahlen ist laut C++-Standard die Dualdarstellung.
- Addition, Subtraktion und Multiplikation werden modulo 2^n durchgeführt. Das Ergebnis ist immer nichtnegativ.
- Die Zweierkomplementdarstellung bildet den vorzeichenbehafteten Zahlbereich bijektiv auf den entsprechenden vorzeichenlosen Zahlbereich ab. Der Rest modulo 2^n bleibt dabei unverändert. Daher kann die *vorzeichenlose* Ganzzahlarithmetik auch für die *vorzeichenbehaftete* Addition, Subtraktion und Multiplikation benutzt werden. *Voraussetzung:* Operanden und Ergebnis liegen bei mathematisch exakter Rechnung im Zahlbereich des vorzeichenbehafteten Datentyps.

Ganzzahl-Literale

- Ausgeschriebene ganze Zahlen im Programmtext werden als *(Ganzzahl-)Literale* bezeichnet.
- Der Typ des Literals ergibt sich durch ein Suffix oder das Fehlen desselben.

  Bsp.: `5` hat den Datentyp `int`
  `5u` hat den Datentyp `unsigned int`
  `5ul` hat den Datentyp `unsigned long`

  Komplexere Regeln gelten, wenn die Zahl nicht in den Zahlbereich passt. (Möglichst vermeiden!)
  → *Ganzzahlliterale auf Inf.blatt 4, S.1*
- *Vorsicht:* Führende Null impliziert Oktalschreibweise! Vorangestelltes `0x` bewirkt Hexadezimalschreibweise.

```
int i1 =   15;  // i1: 15
int i2 =  015;  // i2: 13
int i3 = 0x15;  // i3: 21
```

Gleitpunktzahlen

- 3 Gleitpkt.datentypen: `float`, `double`, `long double`
  Hauptsächlich benutzt: `double`
  Bsp. für `double`-Literale: `12.73` `498.` `.2105`
  `7.2e9` ($= 7.2 \cdot 10^9$) `20e-30` ($= 2 \cdot 10^{-29}$)
  Suffixe für Literale: keiner $\rightarrow$ `double`
  `f` $\rightarrow$ `float` `l` $\rightarrow$ `long double`
- Darstellung von Gleitpunktzahlen in der Regel durch *normalisierte Maschinenzahlen (NMZ)* :
  Vorzeichenbit + Mantisse fester Länge + Exp. fester Länge
  Normalisierung: Erste Ziffer der Mantisse ist ungleich 0
  Bsp. (Basis B=10): $-42.73 = -0.4273 \cdot 10^2$
- Wichtige Kenngrößen der Gleitpunktarith. in `limits`:

`epsilon()`	$2 \cdot$ Maschinengenauigkeit
`min()`	kleinste positive NMZ
`max()`	größte NMZ

- Heute übliche Gleitpunktarithmetik: „IEEE-Arithmetik“

Einige Eigenschaften der IEEE-Arithmetik

- Darstellung für `float`: $x = (-1)^s \cdot (1.f)_2 \cdot 2^{e-127}$

  Bitmuster: | s | e | f | s: 1 Bit e: 8 Bit f: 23 Bit

  Bsp.: $-6 = (-1) \cdot 4 \cdot \frac{3}{2} = (-1)^1 \cdot 2^{129-127} \cdot (1.1)_2$

  Bitmuster: | 1 | 10000001 | 10000000000000000000000 |

  Hex.: 1100 0000 1100 0000 0000 0000 0000 0000

  c 0 c 0 0 0 0 0

  Bsp.: $\frac{1}{3} = 1 \cdot \frac{1}{4} \cdot \frac{4}{3} = (-1)^0 \cdot 2^{125-127} \cdot (1 + \sum_{k=1}^{\infty} \frac{1}{4^k})$

  Bitmuster: | 0 | 01111101 | 01010101010101010101011 |

  Hex.: 0011 1110 1010 1010 1010 1010 1010 1011

  3 e a a a a a b

  Letztes Bit ist 1 wegen Rundung!

Einige Eigenschaften der IEEE-Arithmetik II

- Zusätzlich zu normalisierten Maschinenzahlen:
  subnormale Maschinenzahlen
  $\pm\infty$ („Unendlich“)
  `NaN` („Not a number“)

  Bsp.: `1.0/0.0` $=\infty$ `1.0/(-0.0)` $=-\infty$,
  `log(0.0)` $=-\infty$ `sqrt(-1.0)` $=$ `NaN`
- Voreinstellung: Kein Abbruch wegen mathematisch nicht darstellbarer Ergebnisse, stattdessen `NaN`.

  Vorsicht: Gilt nicht für Ganzzahlarithmetik!
- Arithmetische Grundoperationen mit Gleitpunktoperanden liefern stets das gerundete exakte Ergebnis.