Zahlendarstellung im Computer

Karsten Brodmann

Februar 2017

Sowohl im Alltag als insbesondere auch in der Mathematik sind wir es gewohnt, Zahlen als unendlich und präzise zu betrachten. Unsere digitalen Computer können dagegen nur eine begrenzte Anzahl von Zahlen darstellen. Diese Einschränkung ist offensichtlich, wenn wir uns klarmachen, zur Speicherung einer Zahl nur eine bestimmte Anzahl von Bits zur Verfügung zu haben. Weniger offensichtlich ist die nicht zwingend vorhandene Präzision der Zahlendarstellung im Computer.

Wie Zahlen im digitalen Computer dargestellt werden und wie präzise sie sind oder auch nicht, beleuchtet und erläutert dieser Artikel.

1 Zahl ist nicht gleich Zahl

Betrachten wir das folgende Pascal-Programm:

```
1  program SteinDesAnstosses;
2  { .. liest einen Single ein und gibt ihn "falsch" aus }
3  var
4    s: Single; // Gleitkommazahl (einfache Genauigkeit)
5  begin
6    write('Bitte eine Gleitkommazahl: ');
7    readLn(s);
8    // Ausgabe auf 10 Nachkommastellen genau
9    writeLn('Ich habe im Speicher: ', s:0:10)
10 end.
```

Listing 1: Zahl ist nicht gleich Zahl

Ein Testlauf des Programms führt zu folgendem Resultat:

```
$ ./stein
Bitte eine Gleitkommazahl: 0.3
Ich habe im Speicher: 0.3000000119
$ _
```

Das Ergebnis ist einigermaßen ernüchternd. Die vermeintliche Präzision unseres Computers ist angesichts dieses Ergebnisses schwer in Frage gestellt.

Dieser Artikel soll die Hintergründe für dieses Verhalten unseres Computers aufklären, so dass wir verstehen, warum eine vermeintlich „handelsübliche“ Zahl für unseren Computer nicht zwingend ebenso handelsüblich sein muss wie für uns.

Um die folgenden Erläuterungen verständlich zu halten, gehe ich zuerst auf die Codierung ganzer Zahlen ohne Vorzeichen ein. Sodann folgt die Codierung ganzer Zahlen mit Vorzeichen. Zum Abschluss folgen dann die Gleitkommazahlen, also diejenigen Zahlen, die gerade eben als Illustrationsobjekt haben herhalten müssen und die Ursache unseres Unbehagens sind.

Damit mir niemand vorwerfen kann, ich würde leere Behauptungen in den Raum stellen und/oder lediglich theoretisieren, mache ich das Ganze konkret. Ich untermauere meine Aussagen mit kleinen Programmen, die Sie gerne am eigenen Computer nachvollziehen können. Ich verwende hier Free Pascal als Programmiersprache. Die folgenden Sachverhalte sind aber nicht spezifisch für Free Pascal, sie gelten auch für andere Programmiersprachen. Free Pascal ist jedoch für nahezu jede Plattform verfügbar und identisch zu verwenden. Auf diese Weise ist sichergestellt, dass Sie die vorgestellten Beispielcodes 1:1 am eigenen Rechner ausprobieren können.

Ich habe alle vorgestellten Programm mit Free Pascal 3.0.2 unter macOS 10.12.4 (Sierra) übersetzt und ausgeführt. Sie können aber gerne auch Microsoft Windows oder irgendeine Linux-Distribution verwenden, um die Programm nachzuvollziehen.

2 Vorzeichenlose Ganzzahlen

2.1 Von der binären Darstellung zum dezimalen Wert

Vorzeichenlose Ganzzahlen sind positive ganze Zahlen, also so etwas wie 0, 1, 2, 3, viele. Mathematisch ausgedrückt handelt es sich um die Zahlen der Menge $\mathbb{N}_0$.

Betrachten wir die folgende Zeichenfolge: 123. So, wie wir in der Regel konditioniert sind, interpretieren wir diese Zeichenfolge sofort als die Zahl „einhundertdreiundzwanzig“. Wie kommen wir aber dazu? – Blitzschnell, ohne uns dessen bewusst zu werden, führen wir die folgende Rechnung aus, decodieren die Zeichenfolge und ermitteln deren Wert:

$$3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 = 3 \cdot 1 + 2 \cdot 10 + 1 \cdot 100 = 123$$

Das heißt, wir wissen um den Stellenwert, der mit jeder Position der einzelnen Ziffern in der Zeichenfolge verbunden ist, die wir als Zahl interpretieren. Das 10er-System (Dezimalsystem, Basis 10) ist das System, in dem wir üblicherweise Zahlen notieren und welches wir im Sprachgebrauch verwenden.

Der digitale Computer benutzt – das sollte uns nicht fremd sein – ein duales System (Basis 2) zur Codierung von Zahlen. Wir notieren diese mit Nullen und Einsen. Betrachten wir: 110. Das könnte der Polizeinotruf sein, als Dualzahl im Computer repräsentiert diese Ziffernfolge aber die dezimale Zahl 6.

Wie kommen wir auf 6? – Nun, wir machen genau das, was wir vorher bereits mit unserer 123 getan haben. Allerdings führen wir die Berechnung nun mit der Basis 2 aus.

$$0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 = 6$$

Sei w der dezimale Wert einer dualen Zeichenfolge, die als positive Ganzzahl ausgewertet werden soll. d_i seien die einzelnen Digits, also die Nullen und Einsen, aus denen die Folge besteht. Dann können wir die Wertermittlung wie folgt formulieren:

$$w = \sum_{i=0}^{n-1} d_i \cdot 2^i$$

Wir wissen nun, wie wir von der Binärdarstellung einer positiven ganzen Zahl zu ihrem dezimalen Wert gelangen.

Tabelle 1: Vorzeichenlose Ganzzahldatentypen in Free Pascal

Datentyp	Größe	Wertebereich
`Byte`	1 Byte	0..255
`Word`	2 Byte	0..65.535
`LongWord`	4 Byte	0..42.94.967.295
`QWord`	8 Byte	0..18.446.744.073.709.551.615

Tabelle 1 zeigt einige der von Free Pascal unterstützten ganzzahligen, vorzeichenlosen Datentypen. Ein weiterer Datentyp, `Cardinal`, wird auf `Word` oder `LongWord` abgebildet, je nach Compiler und Compilermodus. Viele Programmiersprachen besitzen analoge Datentypen mit ähnlichen Namen.

Mit dem Wissen, das ein Byte acht Bit besitzt, können wir mittels der oben genannten Formel leicht die Wertebereiche ausrechnen. Ich habe das hier bereits getan. Übrigens: Die Punkte sollen, im Gegensatz zum sonstigen Text, in dem ich die amerikanische Zahlendarstellung mit dem Dezimalpunkt verwende, die bei uns in Deutschland üblichen Tausenderpunkte darstellen. – Das erleichtert die Lesbarkeit und hat sich der geneigte Leser sicherlich schon gedacht.

2.2 Vom Dezimalwert zur Binärdarstellung

So wie der Computer Zahlen und die Ergebnisse arithmetischer Operationen in Dezimaldarstellung am Bildschirm präsentiert, also die interne Codierung in eine uns genehme Darstellung transformiert, so ist er auch so freundlich, eine von uns eingegebene Dezimalzahl eigenständig in sein internes Format zu wandeln. Niemand gibt 110 ein, wenn er eine 6 meint.

Wir wissen, dass wir unsere Dezimalzahl irgendwie in eine Summe von 2er-Potenzen zerlegen müssen. Kommt eine bestimmte 2er-Potenz vor, notieren wir eine 1, ansonsten eine 0. Herumprobieren kann ziemlich nervenaufreibend sein. Versuchen wir es also systematischer!

Teilen wir eine uns gegebene Dezimalzahl ganzzahlig durch 2, dann ist sie entweder ohne Rest teilbar oder es verbleibt ein Rest von 1. Ist der Rest 1, so schreiben wir für die Binärdarstellung eine 1, sonst eine 0. Mit dem ganzzahligen Teiler, den wir bei der Restwertdivision erhalten haben, führen wir das Spielchen fort, bis wir den Restwert auf 0 transformiert haben. Auf diese Weise ermitteln wir die einzelnen Digits von rechts nach links, errechnen also erst die Wertigkeit 2^0, dann 2^1 und so fort.

Während die dezimale Wertermittlung einer Dualzahl noch halbwegs im Kopf zu bewerkstelligen ist, ist die umgekehrte Richtung schon weniger bequem. Es liegt also nahe, auch wenn das nicht das Hauptthema dieses Beitrags ist, ein Programm dafür zu schreiben. Damit können wir dann ganz einfach die eine oder andere Zahl durchspielen und uns deren Binärdarstellung ansehen. Ich habe das Ganze mal für den Datentyp `QWord` implementiert, dem größten Datentyp in Free Pascal zur Darstellung vorzeichenloser Ganzzahlen.

```
1  program Dez2Dual;
2  { ... wandelt eine positive Ganzzahl im Intervall
3        [0..18446744073709551615]
4    in die Binärdarstellung einer Dualzahl um.
5  }
6  const
7    BITS=64;          // QWord hat 64 Bit
8  var
9    z: QWord;         // einzulesende, zu wandelnde Zahl
10   i: Integer=0;     // Laufvariable/Zähler
11   s: String[BITS]; // Ausgabe - Stringdarstellung
12 begin
13   write('Bitte eine positive Ganzzahl: ');
14   readLn(z);
15
16   i:=0;                          // Initialisierung
17   fillChar(s,SizeOf(s),'0');
18   setLength(s,BITS);
19
20   while(z>0) do begin
21     if (z mod 2)=1 then          // 1 oder 0 ermitteln und
22       s[length(s)-i]:='1'        // s von rechts nach links
23     else                         // auffüllen
24       s[length(s)-i]:='0';
25     z:=z div 2;                  // noch zu verarbeiten
26     inc(i)                       // Stelle/Zähler erhöhen
27   end;
28   writeLn(s)                     // Ergebnisausgabe
29 end.
```

Listing 2: Binäre Darstellung vorzeichenloser Ganzzahlen

Free Pascal bietet dafür übrigens auch eine Funktion an, so dass wir uns die eigene Umrechnung hätten sparen können. Ich denke aber, die Implementierung verdeutlicht ganz gut, wie der Algorithmus zur Umrechnung vom Dezimal- in das Dualsystem funktioniert.

3 Vorzeichenbehaftete Ganzzahlen

Neben den Ganzzahlen ohne Vorzeichen interessieren uns nun die, die eines besitzen. Wir erweitern also unseren Zahlenhorizont von $\mathbb{N}_0$ auf $\mathbb{Z}$. Unsere erste Überlegung gilt dem Vorzeichen. Wo und wie wird es gespeichert? – Das Vorzeichen findet seinen Platz in dem jeweils ganz links stehenden Bit einer Dualzahl. Bei einem Datentyp, der nur ein Byte Speicherplatz beansprucht, ist dies das Bit Nummer 7, wenn wir von rechts nach links, mit 0 beginnend,

zählen. Ist das Bit gesetzt, ist es also 1, so signalisiert dies ein negatives Vorzeichen. Ist das Bit dagegen 0, haben wir es mit einer positiven Ganzzahl zu tun. Da dieses höchstwertige Bit zur Codierung des Vorzeichens verwendet wird, steht es uns nicht mehr zur Codierung der Ziffern einer Zahl zur Verfügung – logisch. Damit ergeben sich folgende Wertebereiche für die in Free Pascal standardmäßig definierten ganzzahligen Datentypen mit Vorzeichen[1].

Tabelle 2: Vorzeichenbehaftete Ganzzahldatentypen in Free Pascal

Datentyp	Größe	Wertebereich
`ShortInt`	1 Byte	−128..127
`SmallInt`	2 Byte	−32.768..32.767
`LongInt`	4 Byte	−2.147.483.648..2.147.483.647
`Int64`	8 Byte	−9.223.372.036.854.775.808..9.223.372.036.854.775.807

Es gibt noch einen weiteren prominenten Ganzzahldatentypen. Der Datentyp `Integer` wird entweder auf `SmallInt` oder `LongInt` abgebildet, je Compiler und Compilermodus.

Um uns jetzt nicht die Finger wund zu schreiben, betrachten wir einen imaginären Datentyp, der uns vier Bit zur Darstellung vorzeichenbehafteter Ganzzahlen spendieren soll. Das höchste Bit signalisiert das Vorzeichen. Bei positiven Zahlen soll es 0 sein.

VZ	2^2	2^1	2^0	dezimal
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0

So weit, so schön. Wie codieren wir jetzt −1? – Eine Idee könnte sein, vorne eine 1 zu schreiben und die restlichen Ziffern genauso zu codieren, wie beim positiven Pendant. −1 ergäbe dann 1001. Offensichtlich liegt zwischen zwischen 1001 und 0000 noch 1000.

VZ	2^2	2^1	2^0	dezimal
0	0	0	1	1
0	0	0	0	0
1	0	0	0	−0
1	0	0	1	−1

Unserer bisherigen Logik folgend, hätten wir eine zweite, negative Null kreiert, was eine redundante Darstellung derselben bedeutet. Das ist alles andere als optimal. Ein paar schlaue

[1] Auch hier gilt wieder: Andere Programmiersprachen besitzen oftmals analoge und ähnlich benannte Datentypen.

Köpfe haben sich daher ausgedacht, den ganzen Kram, der sich über der Null befindet, einfach darunter zu kopieren. Das sieht dann so aus:

VZ	2^2	2^1	2^0	dezimal
...				
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Diese Darstellung nennt sich *2er-Komplement*. Um es zu ermitteln, müssen wir uns natürlich nicht irgendwelche Tabellen malen und via Copy & Paste ganze Zahlenblöcke kopieren und irgendwo ankleben. Das 2er-Komplement lässt sich einfach errechnen. Das Bitmuster einer negativen Ganzzahl, $-x$, errechnen wir, indem wir zuerst das Bit-Muster von x bilden, dieses Muster invertieren und dann 1 addieren. Beispiel:

	VZ	2^2	2^1	2^0	dezimal
Ausgangszahl	0	1	0	1	5
invertiert	1	0	1	0	-6
1 addieren – fertig	1	0	1	1	-5

Das 2er-Komplement besitzt einige herausragende Eigenschaften, die diese Darstellung unserer ersten Überlegung, die ja nicht optimal war, überlegen machen. Zum Einen vermeiden wir das Problem mit der doppelten Null, sie ist jetzt eindeutig. Zum Anderen können wir zwei Zahlen konsistent miteinander verknüpfen also problemlos auch so etwas wie $2 - 5$ rechnen. Das ist nichts anderes als $2 + (-5)$, wie wir aus dem Mathematikunterricht wissen. Überhaupt, so wissen wir, „gibt es die Subtraktion gar nicht“, sie ist lediglich die Addition mit negativen Zahlen. Da sich an den Regeln zur schriftlichen Addition gegenüber denen für Dezimalzahlen nichts ändert, wenn wir mit Dualzahlen rechnen, probieren wir es aus:

VZ	2^2	2^1	2^0	dezimal
0	0	1	0	2
1	0	1	1	-5
1	1	0	1	-3

Also: $1 + 0$ ergibt 1, $1 + 1$ ist 0 und einen im Sinn, $0 + 0$ plus einen aus dem Sinn ist 1 sowie 1 und 0 ergibt 1, was denn auch das korrekte Vorzeichen ergibt. Vergleichen wir das Ergebnis mit der vorigen Tabelle haben wir tatsächlich -3 ausgerechnet.

4 Überlauf

Was passiert, wenn wir 3 und 5 addieren wollen? – Nach allgemeiner Lehrmeinung ist das kein Problem. Auch unser Taschenrechner für weniger als 10 € liefert auch brav das erwartete Ergebnis: 8.

Probieren wir aus, dieses Ergebnis manuell zu errechnen:

VZ	2^2	2^1	2^0	dezimal
0	0	1	1	3
0	1	0	1	5
1	0	0	0	−8

−8 ist ganz offensichtlich nicht das, was uns jahrelange Erfahrung und viele Stunden Mathematikunterricht gelehrt haben. In unserem beschränkten System können wir nur positive Zahlen bis maximal 7 darstellen. Für die 8, die nach Adam Riese[2] hätte herauskommen sollen, ist kein Platz, um sie zu codieren. Wir haben zu wenig Bits zur Verfügung. So etwas bezeichne wir in der Informatik als *Überlauf*.

Das Problem gilt auch für größere Datentypen, die mehr Bits besitzen, analog. Dumm, wenn man nun mühsam rechnet und nicht weiß, ob das Ergebnis der Bemühungen auch valide ist. Mit einem kleinen Trick können wir uns jedoch helfen, um herauszubekommen, ob wir dem Ergebnis unserer intellektuellen Schufterei vertrauen können. Hierzu verdoppeln wir temporär das Vorzeichenbit. Sind nach Abschluss unserer Verknüpfung die beiden Vorzeichenbits, die wir (gedacht und temporär) haben, identisch, ist die Welt in Ordnung und wir können dem Ergebnis unserer Berechnung trauen, andernfalls nicht.

Betrachten wir das am vorigen Beispiel:

VZ	VZ	2^2	2^1	2^0	dezimal
0	0	0	1	1	3
0	0	1	0	1	5
0	**1**	0	0	0	−8

Die verschiedenen Vorzeichenbits, 0 und 1 signalisieren ganz klar einen Fehler, einen Überlauf.

Dieses Prinzip lässt sich nun auf die tatsächlich in Free Pascal definierten Ganzzahldatentypen mit Vorzeichen übertragen. Stellt sich allerdings die Frage, ob Free Pascal einen Überlauf prüft? – Nein, tut es nicht. Und andere Programmiersprachen sind da auch nicht besser.

Die folgende Schleife sollte, gemäß gesundem Menschenverstand, eine Endlos-Schleife sein, da x beim Verdoppeln je Schleifendurchlauf nur *eingeschränkte Chancen* hat, kleiner oder gleich Null zu werden – meint man. Tatsächlich fährt uns ein Überlauf in die Parade, der lediglich zu einem (möglicherweise) unerwarteten Resultat führt. Uns überrascht das natürlich nicht,

[2] Eigentlich hieß der große Rechenmeister Adam Ries. Er lebte von 1492 (oder 1493) bis 1559.

wissen wir doch um die Hintergründe. Das Programm läuft jedenfalls ansonsten einwandfrei und ermittelt, munter und gar nicht faul, für `i` den Wert -2147483648[3].

```
1  program Ueberlauf;
2  { ... demonstriert einen Überlauf, den FP nicht bemäkelt }
3  var
4    i: Integer;
5  begin
6    i:=1;                // initialisiere mit 1
7    while i>0 do         // solange i > 0
8      i:=i*2;            // verdoppele i
9    writeLn('i: ', i)    // gib i aus
10 end.
```

Listing 3: Provozierter Überlauf

Apropos, wenn wir uns die interne Bit-Darstellung unserer Ganzzahlen, mit oder ohne Vorzeichen, komfortabel berechnen und anzeigen lassen möchten, sei auf die Funktion `IntToBin` verwiesen, die in der Unit `StrUtils` zu finden ist. Das folgende Programm zeigt deren Gebrauch zur Anzeige der Codierungen von 16-Bit `Integer`-Werten, also ganzen Zahlen aus dem Intervall von -32768 bis 32767.

```
1  program Int2Bin;
2  { ... zeigt die Binärdarstellung von 16-Bit-Integer-Werten }
3  uses
4    StrUtils;
5  var
6    i: Integer;
7  begin
8    write('Bitte ein Integer: ');
9    readLn(i);
10   writeLn(i:6,': ', IntToBin(i,16,0))
11 end.
```

Listing 4: Binärdarstellung von `integer`-Werten

5 Gleitkommazahlen

Ganzzahlen waren noch ganz übersichtlich. Da Gleitkommazahlen etwas komplexer in ihrer Codierung sind, erlaube ich mir, erst einmal eine nicht ganz korrekte Darstellung zu präsentieren. Ich werde natürlich auch die korrekte Darstellung zeigen. Ungeachtet der Vereinfachungen, die ich gleich vornehmen werde, wird der sittliche Nährwert, der daraus resultiert, auch in Bezug auf die korrekte Darlegung der Verhältnisse, erhalten bleiben.

Gleitkommazahlen werden in Free Pascal in der Normaldarstellung oder im wissenschaftlichen Format notiert. 12.5 kann also auch als 1.25E1 geschrieben werden. Ingenieure kokettieren

[3]Zur Verteidigung von Free Pascal sei angemerkt, dass Ausdrücke, die zur Compile-Zeit ausgewertet werden können, zumindest eine Warnung hervorrufen, wenn ein Überlauf auftritt. Der Free-Pascal-Compiler gibt dann folgende Meldung aus: `Warning: range check error while evaluating constants`. Allerdings geschieht das auch nur, wenn die Check-Option des Compilers nicht abgeschaltet wurde.

gerne mit der zweiten Notation. Wie auch immer, entscheidend ist der Fakt, dass es Nachkommastellen gibt, die codiert werden wollen.

Die beiden grundlegenden Datentypen für Gleitkommaarithmetik in Free Pascal sind `Single` und`Double`. Der Typ `Real` wird auf einen der beiden anderen Datentypen abgebildet, je nach Plattform und Compilermodus. Auch hier wieder der Hinweis: Viele andere Programmiersprachen unterstützen analoge Datentypen.

Tabelle 3: Datentypen für Gleitkommazahlen in FP

Datentyp	Größe	Wertebereich	Genauigkeit
`Single`	4 Byte	$1.5 \cdot 10^{-45}..3.4 \cdot 10^{38}$	7-8 Stellen
`Double`	8 Byte	$5.0 \cdot 10^{-324}..1.7 \cdot 10^{308}$	15-16 Stellen

Wie man auf die Wertebereiche kommt, sehen wir, wenn wir die jeweilige Codierung kennen. Ebenso wissen wir dann, was es mit der Angabe der Genauigkeit auf sich hat. Im Gegensatz zu Operationen auf Ganzzahlen, mit denen absolut präzise gerechnet wird, sind Gleitkommazahlen nicht genau. Im Gegenzug reicht der Wertebereich einer `Double`-Variablen aber aus, um problemlos alle Atome in unserem Kosmos abbilden zu können. Schlaue Physiker haben nämlich errechnet, dass es 10^{78} sein sollen[4]. Uns bleibt also noch reichlich Luft für die Entdeckung weiterer Galaxien.

5.1 Vereinfachte Darstellung

Wie ist nun eine Gleitkommazahl aufgebaut? – Sie besteht aus einem Vorzeichenbit, einem Exponenten und einer Mantisse. Beim Datentyp `Single` bekommen wir 8 Bit für den Exponenten und 23 für die Mantisse spendiert. Bei `Double` sind es 11 und 52.

Eine Gleitkommazahl wird zu ihrer Darstellung im Rechner in Exponent (e) und Mantisse (m) zerlegt, so dass sich der Wert (w) einer Gleitkommazahl mathematisch wie folgt darstellt.

$$w = (-1)^s \cdot 2^e \cdot m$$

Das Vorzeichenbit (s) ist 0 bei positiven Zahlen, ansonsten 1 – nichts Neues.

Um zu verstehen, wie das genau funktioniert, betrachten wir eine einfache Gleitkommazahl: 12.0. Das Vorzeichenbit ist, weil unsere Zahl positiv ist, 0. Schauen wir uns an, wie wir unsere 12.0 in Exponent und Mantisse zerlegen können. Dabei muss der Exponent positiv oder negativ ganzzahlig sein.

$$\begin{aligned} 12 &= 2^0 \cdot 12 \\ 12 &= 2^1 \cdot 6 \\ 12 &= 2^2 \cdot 3 \\ 12 &= 2^3 \cdot 1.5 \end{aligned}$$

[4] `https://elearning.physik.uni-frankfurt.dedata/FB13-PhysikOnlinelm_data/lm_282/auto/kap09/cd236b.htm`

Offenkundig ist das nicht eindeutig. Da wir jedoch eine eindeutige Darstellung benötigen, hat man sich darauf geeinigt, dass die Mantisse einen Wert m mit $1 <= m < 2$ haben muss. Hierbei wird jedoch nur der um 1 verminderte Wert gespeichert. Man nennt das die *reduzierte Mantisse*.

Anhand dieser Regel, wissen wir nun, dass wir für unsere 12.0 den Exponenten 3 und die Mantisse 0.5 speichern müssen. Dabei gelten für den Exponenten die bereits bekannten Regeln der vorzeichenbehafteten Ganzzahlen. Er wird also im 2er-Komplement codiert. Für die 3 ergibt sich dann, wir betrachten einen `Single`, das Bitmuster 00000011.

Kommen wir nun zur Mantisse. In ihr werden, gemäß Definition nur Werte zwischen 0 und näherungsweise 1 gespeichert. De facto ergibt das Werte zwischen 1 und fast 2. Wir erinnern uns: reduzierte Mantisse – Eins im Sinn. In der Binärdarstellung haben wir es demnach mit folgenden Wertigkeiten bezüglich der einzelnen Digits in der Mantisse zu tun (Darstellung verkürzt):

-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	...
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	...

Der Wert der Mantisse errechnet sich, wie wir leicht erkennen, nach dem gleichen Schema, wie bei den Ganzzahlen. Nur der Index hat sich geändert.

$$w = \sum_{i=-1}^{-k} d_i \cdot 2^i$$

Zur Findung des Bitmusters unserer Mantisse gehen wir im Prinzip genauso vor, wie wir das für positive Ganzzahlen getan haben. Nur dividieren wir jetzt nicht durch 2, vielmehr multiplizieren wir damit und schauen, ob der sich ergebende Wert größer 1 ist. Ist das der Fall, haben wir eine 1 für unser Bitmuster errechnet, ansonsten eine 0. War unser Wert größer oder gleich 1, subtrahieren wir 1 und rechnen weiter. Das treiben wir solange, bis wir 0 erreicht haben. So füllen wir unser Bitmuster von links nach rechts.

Für unser Beispiel geht das schnell: $0.5 \cdot 2 = 1.0$, was eine 1 ergibt. Subtrahieren wir hiervon 1, sind wir bei 0 angelangt und somit fertig.

Damit ergibt sich folgende Darstellung unserer Gleitkommazahl 12.0 als `Single`-Datentyp (die Mantisse ist gekürzt dargestellt).

0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	...

Der Datentyp `Double` funktioniert analog, nur hat er mehr Stellen für Exponent und Mantisse zur Verfügung.

Jetzt müssen wir nur noch wissen, wie die Grenzen definiert sind, wozu wir uns die größt- beziehungsweise kleinstmöglichen Bitmuster ansehen. Für einen `Double` hat der Exponent mit -1024 seinen kleinsten Wert. Das ist als 0 definiert. 1023 ist der größte Wert. Dieser markiert in Verbindung mit dem Vorzeichenbit minus respektive plus unendlich. Alle Werte

dazwischen sind „echte“ Zahlen, also $(-1)^s \cdot 2^e \cdot (1.m)$ für $-1024 < e < 1023$. s ist, wie gehabt, das Vorzeichenbit.

Damit liegt die größte darstellbare positive Zahl im vereinfachten `Double`-Format knapp unter

$$2 \cdot 2^{1022} = 2^{1023} \approx 10^{308}$$

Die kleinste darstellbare positive Zahl im vereinfachten `Double`-Format lautet

$$1.0 \cdot 2^{-1023} = 2^{-1023} \approx 10^{-308}$$

5.2 Gleitkommaarithmetik und Präzision

Im Sinne obiger Definitionen rechnet das folgende Beispielprogramm zwar korrekt, es kommt aber nicht das heraus, was uns im Mathematikunterricht in der vierten Klasse eingebimst wurde: „Die Division durch Null ist nicht erlaubt.“ – Scheinbar doch!

```
1 program Arithmetik;
2 { ... Arithemtik mit Gleitkommazahlen }
3 begin
4   writeLn('1.0 / 0.0 = ', 1.0/0.0);
5   writeLn('0.0 / 0.0 = ', 0.0/0.0);
6   writeLn('0.0 / 1.0 = ', 0.0/1.0)
7 end.
```

Listing 5: Seltsame Gleitkommaarithmetik

```
$ ./arithmetik
1.0 / 0.0 =              +Inf
0.0 / 0.0 =               Nan
0.0 / 1.0 =  0.000000000E+00
$ _
```

`+Inf` steht für unendlich, `Nan` für *Not a number*. Bei ganzzahliger Arithmetik würde das Programm bei den ersten beiden Operationen den Dienst verweigern. Für Ganzzahlen gilt immer noch die gute alte Schulmathematik. Das Verhalten der Gleitkommaarithmetik müssen wir jedoch kennen!

Noch etwas ist beachtlich. Nehmen wir uns die handelsübliche 0.3, die bereits in der Einleitung als schlechtes Beispiel hat herhalten müssen, und versuchen sie zu codieren. Offensichtlich sind Vorzeichen und Exponent 0. Kümmern wir uns daher um die reduzierte Mantisse. Wir benutzen den bereits bekannten Algorithmus.

$$\begin{aligned}
2 \cdot 0.3 &= 0.6 \Longrightarrow 0 \\
2 \cdot 0.6 &= 1.2 \Longrightarrow 1 \\
2 \cdot 0.2 &= 0.4 \Longrightarrow 0 \\
2 \cdot 0.4 &= 0.8 \Longrightarrow 0 \\
2 \cdot 0.8 &= 1.6 \Longrightarrow 1 \\
2 \cdot 0.6 &= 1.2 \Longrightarrow 0
\end{aligned}$$

Ich brauche hier nicht weiter zu machen. Da sich die 1.2 wiederholt, erkennen wir, in eine nicht endende Periode geraten zu sein. Erkenntniswert: Was uns wie eine handelsübliche Zahl erscheint, muss für den Rechner nicht zwingend auch handelsüblich sein! Es gibt im Dezimalsystem Zahlen, die sich dual nicht codieren lassen. Diese Schwäche ist übrigens jedem Zahlensystem innewohnend. Stellen wir uns ein Zahlensystem zur Basis 3 vor. Dann ist 0.1 eine ziemlich präzise und auch endliche Gleitkommadarstellung vom Dezimalbruch $\frac{1}{3}$, der als Gleitkommazahl im Dezimalsystem nur als unendliche Periode darstellbar ist.

Es sollte uns nun klar sein, weshalb das, was wir handschriftlich exakt berechnen können, von Computerprogrammen oft nur näherungsweise genau berechnet wird.

5.3 Korrekte Darstellung

Das folgende Programm gibt das im Computer gespeicherte Bitmuster zu einer einzugebenden `Single`-Zahl aus.

```
1  program Single2Bit;
2  { ... zeigt das Bitmuster eines Double-Wertes an }
3  uses
4    StrUtils;
5  var
6    zahl: record
7      case Boolean of
8        true: (d: Single);
9        false:(b: Longint)
10   end;
11 begin
12   write('Bitte ein Single: '); readLn(zahl.d);
13   writeLn('Single: ', zahl.d:0:10,'  Bitmuster: ',
14           IntToBin(zahl.b,32,0))
15 end.
```

Listing 6: Tatsächliche Bit-Darstellung von Gleitkommazahlen

Hier ein Testlauf mit der uns bekannten 12.0:

```
Bitte ein Single: 12.0
Single: 12.0000000000  Bitmuster: 01000001010000000000000000000000
```

Vergleichen wir dieses Bitmuster mit dem vermuteten Bitmuster, welches wir vorhin per Hand errechnet haben.

vermutet:	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	...
Fakt:	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	...

Vorzeichen und Mantisse sind scheinbar in Ordnung. Verwirrung stiftet aber der Exponent. Die tatsächliche Bitdarstellung sieht verdächtig nach einer 130 aus. Wenn wir nachrechnen bestätigt sich dieser Verdacht. Was ist da los?

Nun, ich habe ja gesagt, ich würde eine zuerst vereinfachte Darstellung der Gleitkommazahlen vorstellen. Der aufmerksame Leser wird auch bemerkt haben, dass sich die Wertebereiche in der vereinfachten Darstellung von denen unterscheiden, die ich in vorangehender Abbildung angegeben habe. Ich habe bei den Codierungsregeln für die vereinfachte Darstellung respektive `Double` wie folgt:

`Single`:

1. Wenn $0 < e < 255$, dann $w = (-1)^s \cdot 2^{e-127} \cdot (1.m)$.
2. Wenn $e = 0$ und $m <> 0$, dann $w = (-1)^s \cdot 2^{-126} \cdot (0.m)$.
3. Wenn $e = 0$ und $m = 0$, dann $w = (-1)^s \cdot 0$.
4. Wenn $e = 255$ und $m = 0$, dann $w = (-1)^s \cdot \text{Inf}$.
5. Wenn $e = 255$ und $m <> 0$, dann $w = \text{Nan}$.

`Double`:

1. Wenn $0 < e < 2047$, dann $w = (-1)^s \cdot 2^{e-1023} \cdot (1.m)$.
2. Wenn $e = 0$ und $m <> 0$, dann $w = (-1)^s \cdot 2^{-1022} \cdot (0.m)$.
3. Wenn $e = 0$ und $m = 0$, dann $w = (-1)^s \cdot 0$.
4. Wenn $e = 2047$ und $m = 0$, dann $w = (-1)^s \cdot \text{Inf}$.
5. Wenn $e = 2047$ und $m <> 0$, dann $w = \text{Nan}$.

Diese Regeln haben auch einen Namen. Es handelt sich um den so genannten *IEEE-754-Standard*, sehr schön nachzulesen unter `http://de.wikipedia.org/wiki/IEEE_754`. Hieraus ergeben sich dann auch die tatsächlichen Wertebereiche, die ich oben, bei der Vorstellung von `Single` und `Double`, angegeben habe.

Wenden wir den Standard für `Single`-Werte an (Regel 1), erkennen wir, der von unserem Programm ausgegebene Exponent mit dem Wert 130 ist vollkommen korrekt. In unserer Vereinfachung haben wir 3 als Exponent angenommen. Nach IEEE ist $3 + 127 = 130$ zu speichern. Unsere vereinfachte Darstellung der Gleitkommazahlen war also bei Weitem nicht richtig. Ich hoffe aber, sie hat das Verständnis derselben etwas erleichtert. Unsere gezogenen Schlussfolgerungen, die sich so leichter haben ableiten lassen, bleiben jedenfalls bestehen.

6 Fazit

Das Wissen über die Codierung von Zahlen im Computer ist kein Hexenwerk. Für den Informatiker ist es die Voraussetzung, um Datentypen für den gedachten Anwendungszweck korrekt zu wählen und die Ergebnisse von Berechnungen einschätzen zu können. Für alle anderen mag es die Grundlage sein, Nachsicht üben zu können, wenn bei komplexen Berechnungen mal nicht das herauskommt, was denn hätte herauskommen sollen. Die dabei wesentliche Erkenntnis ist aber wohl die, dass das Rechnen mit Gleitkommazahlen nur selten genau ist.

Was ich versucht habe bildhaft mit Programmbeispielen in Free Pascal zu untermalen, gilt im Übrigen für wirklich alle Programmiersprachen. Ganzzahlen sind präzise, Gleitkommazahlen nicht! Das ist kein Manko einer spezifischen Programmiersprache, es liegt schlicht am Aufbau unserer Computer, die im dualen System rechnen.

Findige Geister haben daher für Finanzberechnungen einen eigenen Datentyp geschaffen. Der nennt sich`Currency` und ist, obgleich er sich nach außen als Gleitkommazahl mit vier Nachkommastellen darstellt, eigentlich eine Ganzzahl. Er ist also im Rahmen der kaufmännischen Rundung präzise. Dumm ist nur, dass weder alle Programmiersprachen noch Excel & Co. diesen Datentyp unterstützen.

Ich hoffe, dieser Beitrag hat etwas Licht in die dunkle Welt der Zahlendarstellung im Computer gebracht.

Karsten Brodmann