Einführung in die Informatik

`07 Numbers`

Prof. Dr. Carsten Link

Stand: 6. November 2024

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, Fertigkeiten zur Problemlösung):

Sie können zwischen verschiedenen Darstellungsarten von positiven ganzen Zahlen umwandeln und kennen Ansätze, reale Zahlen abzubilden.
Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- erläutern, auf welche Art Zahlen im Speicher in Form von Bytes dargestellt werden
- im Speicher abgelegte Zahlen anhand der dort zu findenden Bytes berechnen
- verschiedene Darstellungen von Zahlen in Form von Bytes ineinander umwandeln
- berechnen, wie viele Ziffern zur Darstellung einer Zahl in einem bestimmten Stellenwertsystem nötig sind
- die Probleme erläutern, die mit den Darstellungsweisen von Zahlen einhergehen
- Ansätze erläutern, mit den Problemen umzugehen

2 Konzepte

2.1 Historische Entwicklung

2.2 Ganze positive Zahlen

Mikroprozessoren (CPUs) verfügen über so genannte Register. In diesen können Werte gespeichert werden und Rechenoperationen können darauf ausgeführt werden. Die Register haben eine feste nicht änderbare Breite. Demnach haben ganze Zahlen in diesen Registern immer eine feste Anzahl an Hexadezimalziffern. Die Dezimalzahl `254` sieht in verschiedenen Breiten so aus:

- 8 Bit: `0xFE`
- 16 Bit: `0x00FE`
- 32 Bit: `0x000000FE`
- 64 Bit: `0x00000000000000FE`

Die höherwertigen Stellen werden bei Bedarf mit Nullen gefüllt.
Da eine Speicher mit n Bits 2^n verschiedene Zustände haben kann und einer davon für die Darstellung der Null benötigt wird, ist die größte Zahl, die darin gespeichert werden kann 2^{n-1}
Datentypen und Wertebereiche von vorzeichenlosen Ganzzahltypen in verschiedenen Programmiersprachen [2]:

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen

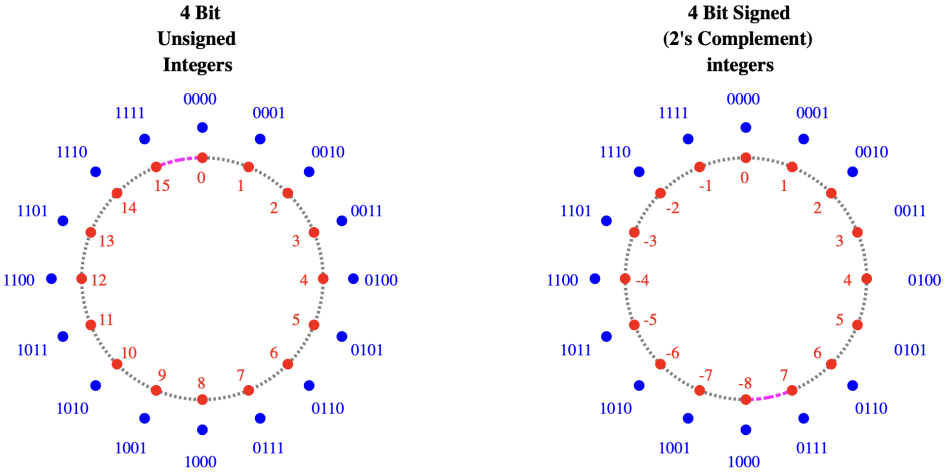
[2] Bei C/C++ werden im Quellcode meist die Typen `unsigned char`, `unsigned int`, `unsigned long`, und `unsigned long long` verwendet, die vom Compiler auf einen der unten genannten Typen abgebildet werden

Anzahl Bits	**Wertebereich**	**FreePascal**	**C/C++** `stdint.h`	**Java**
8	0 ... 255	Byte	uint8_t	–
16	0 ... 65535	Word	uint16_t	–
32	0 ... 4294967295	Longword	uint32_t	–
64	0 ... 18446744073709551615	QWord	uint64_t	–

2.3 Ganze negative Zahlen

Zur Darstellung ganzer negativer Zahlen kommt bei vielen CPUs – und damit auch bei vielen Programmiersprachen – das 2er-Komplement zum Einsatz. Dabei hat das höchstwertigste Bit einer n-Bit-Zahl die Wertigkeit -2^{n-1}. Bei vier-Bit-Zahlen also -8 und bei acht-Bit-Zahlen also -128. In Abbildung 1 ist dieser Sachverhalt den positiven Bitmustern gegenübergestellt.
Um das beispielsweise die 8-bittige 2er-Komplement-Darstellung der Zahl -53 zu ermitteln, wird die Gleichung $-128 + r = -53$ nach n umgestellt und mit $r = 128 - 53 = 75$ gelöst ($75 =$ `0x4b` $=$ `0b1001011`). Der Zahl r werden nun die nötigen Einsen vorangestellt, so dass die 8-bittige 2er-Komplement-Darstellung der Zahl -53 die binärzahl `11001011` ist.

Abbildung 1: Positive ganze 4-Bit Zahlen sowie negative ganze Zahlen im 2er-Komplement.



Die Dezimalzahl `-5` sieht im 2er-Komplement in verschiedenen Breiten so aus:

- 8 Bit: `0xFB`
- 16 Bit: `0xFFFB`
- 32 Bit: `0xFFFFFFFB`
- 64 Bit: `0xFFFFFFFFFFFFFFFB`

Die höherwertigen Stellen werden bei Bedarf mit Einsen gefüllt.
Datentypen und Wertebereiche von vorzeichenbehafteten Ganzzahltypen in verschiedenen Programmiersprachen [3]:

Anzahl Bits	**Wertebereich**	**FreePascal**	**C/C++** `stdint.h`	**Java**
8	−128 … 127	ShortInt	int8_t	byte
16	−32768 … −32767	SmallInt	int16_t	short
32	−2147483648 … −2147483647	LongInt	int32_t	int
64	−9223372036854775808 … −9223372036854775807	Int64	int64_t	long

2.4 Ganze Zahlen im Speicher

Das Programm `a.out_typedMemory` (gebaut mit `./build.sh` aus `typedMemory.hpp`, `typedMemory.cpp` und `main.cpp`) zeigt, wie ganze Zahlen im Speicher abgelegt werden:

a.out_typedMemory

```
bash$ ./a.out_typedMemory
0000:  41 61 30 39 68 65 6c 6c 6f 00 05 68 65 6c 6c 6f Aa09hello..hello
0010:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0020:  41 31 21 11 cc cc cc cc cc cc cc cc cc cc cc cc A1!.............
0030:  81 71 61 51 41 31 21 11 cc cc cc cc cc cc cc cc .qaQA1!.........
0040:  ff ff ff fe cc cc cc cc cc cc cc cc cc cc cc cc ................
0050:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0060:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0070:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
```

Oben ist an der Adresse `0x0020` die Zahl positive 32-Bit Zahl `0x41312111` abgelegt; an Adresse `0x0030` die Zahl positive 64-Bit Zahl `0x8171615141312111`. Ab Adresse `0x0040` findet sich die negative 32-Bit Zahl `-0x00000002` im 2er-Komplement.

2.5 Stellenwertsysteme

Zahlen lassen sich in verschiedenen Stellenwertsysteme darstellen. Diese Darstellungen lassen sich ineinander umwandeln.

2.6 Stellenwertsysteme: Ziffernanzahl

Mit dieser Formel lässt sich berechnen, wie viele Ziffern n zur Darstellung einer Zahl z in einem bestimmten Stellenwertsystem zur Basis B nötig sind:
$n = \frac{ln(z)}{ln(B)} + 1$
Hierbei ist ein Ergebnis der Art 3.9957 als „weniger als vier, also drei“ zu interpretieren.

[3] Bei C/C++ werden im Quellcode meist die Typen `signed char`, `signed int`, `signed long`, und `signed long long` verwendet, die vom Compiler auf einen der unten genannten Typen abgebildet werden

bc -l

```
bash$ bc -l
>>> l(1000)/l(10)+1
4.0000000000000000000
l(999)/l(10)+1
3.9995654882259823087
>>> l(65535)/l(2)+1
16.999977986052736044 65
```

Oben ist zu sehen, dass die Zahl `1000` vier Dezimalstellen benötigt, die Zahl `999` drei Dezimalstellen. Die Zahl `65535` benötigt im Binärsystem 16 Stellen (Bit).

2.7 Stellenwertsysteme: Ziffernfolge in Wert umrechnen

Ermittlung des Wertes einer Ziffernfolge in Groovy

```
// file src_etc/07_Numbers/conversions0.groovy
String s ="48879";
chars    = s.toCharArray();
int result = 0;
int base = 10;
for (int i = 0; i < chars.length; i++){
  result *= base;
  char c = chars[i];
  int v  = Character.getNumericValue(c);
  result += v;
  println("i=" + i + ":   char c=" + c + "   int v=" + v + "    " + "result=" + result);
}
println(" converted s=>" + s + "< to >" + result + "<");
```

Im Rumpf der `for`-Schleife finden sich zwei wichtige Anweisungen. Die Anweisung `value *= Base;` schiebt das Ergebnis um eine Stelle nach links. Die Anweisung `value += v;` hängt die zuvor ermittelte nächste Ziffer an.
Ausgabe dazu:

groovy conversions.groovy

```
i=0:   char c=4   int v=4    result=4
i=1:   char c=8   int v=8    result=48
i=2:   char c=8   int v=8    result=488
i=3:   char c=7   int v=7    result=4887
i=4:   char c=9   int v=9    result=48879
 converted s=>48879< to >48879<
```

2.8 Stellenwertsysteme: Wert in Ziffernfolge umrechnen

Das Horner-Schema findet seine Verwendung vor allem beim Berechnen von Polynomen, wodurch Potenzen durch Multiplikation und Addition ersetzt werden. (siehe Arndt Brünner – Umrechnung von Zahlensystemen[4]).
Im Folgenden wird eine Variante des Horner-Schemas gezeigt, mit der sich die Ziffernfolge eines Wertes in einem beliebigen Stellenwertsystem berechnen lässt. Um die Ziffern eines Wertes zu ermitteln, sind diese beiden Eigenschaften essentiell:

[4] `http://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm`

1. Der Divisionsrest einer ganzzahligen Division liefert die letzte Ziffer.
2. Eine ganzzahlige Division durch die Basis schiebt die Zahl nach rechts.

Die wiederholte Anwendung von Ganzzahldivision und Divisionsrest sieht wie folgt aus:

Ermittlung der Ziffern nach Horner-Schema

```
48879 = 4*10.000 + 8*1000 + 8*100  + 7*10   + 9
      = 4*10^4   + 8*10^3 + 8*10^2 + 7*10^1 + 9*10^0

48879 / 10 ergibt  4887  Rest 9
4887  / 10 ergibt   488  Rest 7
488   / 10 ergibt    48  Rest 8
48    / 10 ergibt     4  Rest 8
4     / 10 ergibt     0  Rest 4
0 => Ende
```

Die Ziffernfolge „48879“ ergibt sich oben durch das Lesen der Divisionsreste von unten nach oben.

Ermittlung der Ziffern nach Horner-Schema in Groovy

```
// file src_etc/07_Numbers/conversions2.groovy

// convert int to corresponding ASCII char
// (hence "i2c"; i to c; int to char)
// e.g. 0 to '0'
def i2c (int digit_value){
  return Character.forDigit(digit_value, 10)
}

int n = 48879;
int k = n;
int Base = 10;
String result = "";
while (k>0) {
  int digit = k % Base;
  println(k + " " + digit);
  result = "" + i2c(digit) + result;
  k /= Base;
}
println("converted n= >" + n + "<   to   >" + result + "<");
```

Im Rumpf der `while`-Schleife finden sich zwei wichtige Anweisungen. Die Anweisung `digit = k % Base;` ermittelt die letzte Ziffer. Die Anweisung `k /= Base;` schiebt die bearbeitete Zahl nach rechts und löscht dadurch die letzte Ziffer. Ausgabe dazu:

groovy conversions.groovy

```
48879 9
4887 7
488 8
48 8
4 4
converted n= >48879<   to   >48879<
```

2.9 Reelle Zahlen: Notationen

Viele Reelle Zahlen lassen sich gar nicht als Dezimalbrüche darstellen, da sie sehr viele oder gar unendlich viele Vor- oder Nachkommastellen haben (z.B. $\frac{1}{7} = 0{,}\overline{142857}$).
In den Wissenschaften hat sich bei Berechnungen mit sehr großen und sehr kleinen Zahlen die wissenschaftliche Notation durchgesetzt. Dabei wird eine Zahl Z aufgeteilt in die Mantisse M und den Zehnerpotenzfaktor: K ($Z = M \cdot 10^K$).
So lässt sich beispielsweise die Astronomische Einheit (mittlerer Abstand Sonne zur Erde; ca. 150 Millionen km) lesbarer darstellen:

- exakt festgelegter Wert: $149597870700m = 1{,}495978707 \cdot 10^{11}$ (Internationale Astronomische Union 2009, $\pm 3m$)
- Wissenschaftliche Notation (gerundet): 1,496e11 m.
- Technische Notation (gerundet): 149,6e9 m.

Bei der zuletzt gezeigten technischen Notation werden die Zehnerpotenzfaktoren so gewählt, dass sie ein vielfaches von drei sind. So haben sie eine Entsprechung zu den bereits bekannten Präfixen (kilo, mega, milli, nano, etc.). Oben wurden die Werte $1{,}496 \cdot 10^{11}$ und 149,6e9 auf vier signifikante Stellen gerundet.
Beispielberechnung: Mit der Lichtgeschwindigkeit von $299792458\frac{m}{s}$ lässt sich ausrechnen, wie alt das Licht ist, das von der Sonne auf die Erde trifft (d.h. wie lange es benötigt):

$$t = \frac{1{,}496\text{e}11m}{2{,}9998\text{e}8\frac{m}{s}}$$

$$= \frac{1{,}496}{2{,}9998} \cdot \frac{m}{\frac{m}{s}} \cdot \frac{1\text{e}11}{1\text{e}8}$$

$$= \frac{1{,}496}{2{,}9998} \cdot \frac{ms}{m} \cdot 1\text{e}3$$

$$= 0{,}499\text{e}3s = 499s$$

Datentypen[5] in verschiedenen Programmiersprachen, die Dezimalzahlen unterstützen:

Sprache, Typ	**signifikante Dezimalstellen**	**Wertebereich**	**Größe in Byte**
Java, `java.math.BigDecimal`	einstellbar	$unscaledvalue \cdot 10^{-scale}$	dynamisch
C#, `System.Decimal`	28	$1 \cdot 10^{-28}$ ... $7,9228 \cdot 10^{28}$	16
FreePascal, `Currency`	19	-922337203685477, 5808 ... 922337203685477, 5807	8

[5]Dies Typen sind typischerweise nicht Teil der Sprache sondern Teil dazu mitgelieferten Standardbibliothek. Das ist darin begründet, dass gängige CPUs diese Datentypen nicht unterstützen.

2.10 Reelle Zahlen: binäre Darstellung

Die wissenschaftliche Schreibweise lässt sich auch auf binäre Zahlen anwenden:

$$5{,}5625 = 0\text{b}0101{,}1001 = 0\text{b}0101{,}1001 \cdot 10^{0} = 0\text{b}1{,}011001 \cdot 10^{10}$$

Dabei ist zu beachten, dass die Stellenwertigkeiten hinter dem Komma negative Exponenten von 2 sind:

$$0b10^{-1} + 0b10^{-100} = 2^{-1} + 2^{-4} = 0{,}5 + 0{,}0625 = 0{,}5625$$

Gängige CPUs und Programmiersprachen binäre Gleitkommatypen aus dem Standard IEEE 754 mit 32 oder 64 Bit. Ein solcher 32-Bit single precision `float`-Typ verfügt über diese Eigenschaften:

- Vorzeichen: 1Bit
- Mantisse: 24 Bit
- Exponent: 8 Bit
- Spezielle Bitmuster für Not a Number (NaN, für nicht mathematisch definierte Ergebnisse) und ±infinity (für Überläufe)

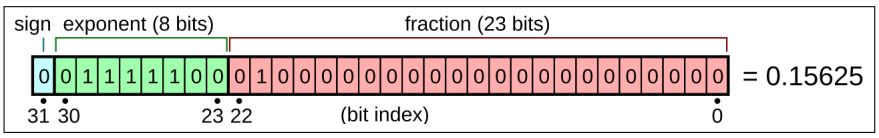


Abbildung 2: Aufbau von 32-Bit Gleitkommazahlen nach IEEE 754. Fraction bezeichnet die Mantisse. Quelle: `https://en.wikipedia.org/wiki/Single-precision_floating-point_format`

Die folgende Tabelle gibt einen Überblick über wichtige binäre Gleitkommatypen in verschiedenen Programmiersprachen:

Sprache, Typ	**signifikante Dezimalstellen**	**Wertebereich (positiv)**	**Größe in Byte**
FreePascal, `single`; Java, `float`	6	1,5e-45 …3,4e38	4
FreePascal, `double`; Java, `double`	15	5,0e-324 … 1,7e308	8
FreePascal, `extended`	19	1,9e-4932 …1,1e4932	10

2.11 Probleme bei der Programmierung: Genauigkeit

In der folgenden Tabelle ist zu sehen, dass Gleitkommazahlen bei kleinen Werten viele Nachkommastellen darstellen können. Bei größeren Zahlen, sinkt die Anzahl der Nachkommastellen und bei sehr großen Zahlen, lassen sich sogar ganze Zahlen nicht mehr genau darstellen (der Abstand zwischen zwei Zahlen ist dann größer als 1).

darzustellende Zahl	**dargestellte Zahl**	**nächstgrößere Zahl**	**Abstand**
1	1	1,0000001	0,0000001
10	10	10,000001	0,000001
100	100	100,00001	0,00001
1000 = 1e3	1000	1000,00006	0,00006
10000 = 1e4	10000	10000,001	0,001
100000 = 1e5	100000	100000,01	0,01
1000000 = 1e6	1000000	1000000,06	0,06
1000000 = 1e7	10000000	10000001	1
100000000 = 1e8	100000000	100000008	8
1000000000 = 1e9	1000000000	1000000064	64
10000000000 = 1e10	10000000000	10000001024	1024
100000000000 = 1e11	99999997952	100000006144	8192
1000000000000 = 1e12	999999995904	1000000061440	65536

In der Tabelle ist auch zu sehen, dass das Komma nach rechts und links gleitet. Daher der Name Gleitkommazahl.
Dass die Anzahl der Nachkommastellen von der absoluten Größe abhängt, sorgt bei Berechnungen mit Gleitkommazahlen für einige Probleme (bzw. die signifikanten Stellen weit vom Komma entfernt sein können). Hier ein Beispiel mit einer Gleitkommazahlen mit 6 signifikanten Stellen:

$$
\begin{aligned}
a &= 123456{,}0 \\
b &= 123456000{,}0 \\
c &= 123456000000{,}0 \\
d &= 0{,}123456 \\
e &= 0,000123456 \\
f &= 0,000000123456
\end{aligned}
$$

Durch die Begrenzung der Anzahl der signifikanten Stellen ergibt sich folgendes Fehlverhalten: mathematisch korrekt ist $b + d = 123456000{,}123456$; das auf sechs signifikante Stellen begrenzte Ergebnis ist jedoch wieder der Wert von b: 123456000,0. Ebenso ergibt $a + d - a$ den Wert 0,0 und eben nicht d.

2.12 Probleme bei der Programmierung: Vergleiche

Ein Problem bei der Programmierung mit Gleitkommazahlen stellen Vergleiche dar. Dies wird im Folgenden an einem Beispiel gezeigt. Die Funktion `comp()` gibt Werte der Funktion `polynom1()` in einem Intervall mit einer bestimmten Schrittweite aus.

float_int_pitfalls.groovy

```
// file float_int_pitfalls.groovy

def polynom1(x) {
  return  ( x - 1.0 ) * ( x - 3.0 ) * ( x - 5.0 ) // returns 0.0 on x = 1.0, 3.0, 5.0
}

def comp(float x_start, float x_end, float x_delta){
    x = x_start
```

```
    while ( x != x_end ) {
      println("x=" + x + "  y=" +  polynom1(x));
      x += x_delta
    }
}

comp(0.0, 10.0, 1.0)
comp(0.0, 10.0, 0.1)
```

Bei der Ausführung mit `groovy float_int_pitfalls.groovy` gibt das Programm zunächst korrekt die Polynomwerte im Bereich `0.0` bis `19.0` in `1.0`-er Schritten aus. Der Aufruf `comp(0.0, 10.0, 0.1)` für `0.1`-er Schritte führt jedoch dazu, dass dass Programm kein Ende findet und abgebrochen werden muss:

Ausgabe von `groovy float_int_pitfalls.groovy`

```
bash$ groovy float_int_pitfalls.groovy
x=0.0  y=-15.0
x=1.0  y=0.0
x=2.0  y=3.0
x=3.0  y=-0.0
x=4.0  y=-3.0
x=5.0  y=0.0
x=6.0  y=15.0
x=7.0  y=48.0
x=8.0  y=105.0
x=9.0  y=192.0
x=0.0  y=-15.0
x=0.10000000149011612  y=-12.788999968364834
x=0.20000000298023224  y=-10.751999941825865
x=0.30000000447034836  y=-8.882999920114875
x=0.4000000059604645  y=-7.175999902963639
x=0.5000000074505806  y=-5.624999890103936
...
x=9.700000144541264  y=273.96301888720734
x=9.80000014603138  y=287.2320196733479
x=9.900000147521496  y=300.9090204804097
x=10.000000149011612  y=315.00002130866096
x=10.100000150501728  y=329.51102215836994
x=10.200000151991844  y=344.4480230298047
...
x=227.7000033929944  y=1.1344224936924214E7
x=227.80000339448452  y=1.1359378306607664E7
x=227.90000339597464  y=1.1374545164291717E7
x=228.00000339746475  y=1.1389725515976379E7
x=228.10000339895487  y=1.1404919367661644E7
x=228.20000340044498  y=1.1420126725347517E7
...
```

Das Programm beendet sich nicht, da der Verglich in der `while`-Bedingung niemals `false` wird; der Wert `10.0` wird nicht getroffen (nur `9.900000147521496` und `10.000000149011612`). Eine Lösung für dieses Problem ist es, den Vergleich `x != x_end` durch `x <= x_end` zu ersetzen, so dass größere `x`-Werte `false` liefern.

2.13 Probleme bei der Programmierung: Wertebereiche

float_int_pitfalls.groovy

```
// file float_int_pitfalls.groovy

def int tax1(int gross){
  return gross * 19 / 100
}

def int tax2(int gross){
  // return gross / 100 * 19
  int deci = gross / 100
  return  deci * 19
}

println(tax1(10) + " " + tax2(10))
println(tax1(100) + " " + tax2(100))
println(tax1(1000000000) + " " + tax2(1000000000))
```

Das Programm soll den Wert einer 19%-igen Steuer berechnen.

Ausgabe von groovy float_int_pitfalls.groovy

```
1 0
19 19
18201308 190000000
```

Die Berechnung der Steuer funktioniert nur für den Wert `100`. Beim Wert `1` wird in `tax2()` der Quotient `gross / 100` zu `0`. Beim Wert `1000000000` wird in `tax1()` das Produkt `gross * 19` zu groß für den Wertebereich von `int`. Eine Lösung für dieses Problem ist es, die Werte vor der Berechnung in einen größeren Datentypen umzuwandeln (hier: `long`).

3 Material zum aktiven Lernen

3.1 Vertiefungsaufgaben zu 'Ganze Zahlen'

a) Wandeln Sie die 4-Bittige 2er-Komplementzahl `0b0101` in eine Dezimalzahl um.

b) Wandeln Sie die 4-Bittige 2er-Komplementzahl `0b1101` in eine Dezimalzahl um.

c) Ermitteln Sie den Wert des 32-bittigen `unsigned int` ab Adresse `0x34`:

a.out_typedMemory

```
bash$ ./a.out_typedMemory
0000:  41 61 30 39 68 65 6c 6c 6f 00 05 68 65 6c 6c 6f Aa09hello..hello
0010:  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
0020:  41 31 21 11 cc cc cc cc cc cc cc cc cc cc cc cc A1!.............
0030:  81 71 61 51 41 31 21 11 cc cc cc cc cc cc cc cc .qaQA1!.........
```

3.2 Vertiefungsaufgaben zu 'Stellenwertsysteme'

a) Setzen sie eine alternative Implementierung zu „Ermittlung des Wertes einer Ziffernfolge in Groovy“ in Kapitel 2.7 um. Diese summiert Zwischenwerte `v`

`* faktor` auf, wobei Faktor mit `1` initialisiert wurde und in jedem Durchlauf mit der Basis multipliziert wird. Verwenden Sie eine beliebige Programmiersprache (Groovy, Java oder Python).

b) Wandeln Sie schriftlich die dezimale Zahl 164 in die binäre Darstellung um (Horner-Schema).

3.3 Vertiefungsaufgaben zu 'Reelle Zahlen'

a) Wandeln Sie 3,4e2 in die normale Schreibweise um.

b) Wandeln Sie 59824 in die wissenschaftliche Schreibweise mit drei signifikanten Stellen um.

c) Wandeln Sie 5,6e4 in die technische Schreibweise um.

3.4 Vertiefungsaufgaben zu 'Probleme bei der Programmierung'

a) Erstellen Sie in `float_int_pitfalls.groovy` (oder einer Java-Variante davon) die Funktionen `comp_fixed()`, die korrekt arbeitet.

3.5 Verständnisfragen zu 'Reelle Zahlen'

a) Lässt sich jede reelle Zahl in einem Computer darstellen?

b) Welche Vorteile hat die wissenschaftliche Notation?

c) Welche weiteren Vorteile hat die technische Notation?

3.6 Verständnisfragen zu 'Probleme bei der Programmierung'

a) Welche Probleme treten bei der Programmierung aufgrund des Wertebereichs von `int` auf?

b) Welche Probleme treten bei der Programmierung aufgrund der Genauigkeit von `float` und `double` auf?

3.7 Testate zu 'Stellenwertsysteme'

a) Bauen Sie die Datei `src_etc/07_Numbers/conversions4.groovy` so um, dass Hexadezimalzahlen umgewandelt werden.

3.8 Testate zu 'Reelle Zahlen'

a) Wandeln Sie 2,93e3 in die normale Schreibweise um.

b) Wandeln Sie 82434 in die wissenschaftliche Schreibweise mit drei signifikanten Stellen um.

3.9 Testate zu 'Probleme bei der Programmierung'

a) Erstellen Sie in `float_int_pitfalls.groovy` (oder einer Java-Variante davon) die Funktionen `tax1_fixed()` und `tax2_fixed()`, die korrekt rechnen.

4 Anhang: Literatur und weiterführendes Material

Bücher und Papers:

- Donald Knuth: The Art of Computer Programming. 3. Auflage. Band 2. Addison-Wesley, Boston 1998, ISBN 0-201-89684-2, Positional Number Systems, S. 194–213 (englisch).
- David Goldberg, March, 1991 issue of Computing Surveys: What Every Computer Scientist Should Know About Floating-Point Arithmetic[6]

nützliche URLs:

- Mathematik-Seiten von Arndt Brünner – Umrechnung von Zahlensystemen[7]
- wikipedia – Vorsätze für Maßeinheiten[8]
- wikipedia – Messergebnisse Astronomische Einheit[9]
- The Bare Minimum about Floating-Point[10]
- Josh Haberman – Floating Point Demystified, Part 1[11]
- Josh Haberman – Floating Point Demystified, Part 2[12]

Fundgrube:

- Float Toy – build intuition for the IEEE floating-point format.[13]

[6] https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
[7] http://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm
[8] https://de.wikipedia.org/wiki/Vorsätze_für_Maßeinheiten
[9] https://de.wikipedia.org/wiki/Astronomische_Einheit#Messergebnisse
[10] https://computational-discovery-on-jupyter.github.io/Computational-Discovery-on-Jupyter/Appendix/floating-point.html
[11] https://blog.reverberate.org/2014/09/what-every-computer-programmer-should.html
[12] https://blog.reverberate.org/2016/02/06/floating-point-demystified-part2.html
[13] https://evanw.github.io/float-toy/