Richard Kaiser

www.rkaiser.de

Dezimale Gleitkommabibliothek

(C++ Quelltext Bibliothek)

für

Microsoft Visual Studio 2010/2012/2013/2015

Version 1.0

Version 1.0- 07/15

Inhalt

1 Dezimale Gleitkommazahlen – Grundlagen

1.1 Binäre Gleitkommaformate – in C/C++ *float, double, long double*

Gleitkommawerte werden in einem sogenannten **Gleitkommaformat** dargestellt. Ein solches Format verwendet eine vorgegebene **Basis b**, für die in der Praxis nur die Werte 2 (**binäres Gleitkommaformat**) und 10 (**dezimales Gleitkommaformat**) üblich sind. Die **Gleitkommadarstellung** einer (im mathematischen Sinn) reellen Zahl r besteht dann aus 3 ganzzahligen Werten s (für das Vorzeichen), m (für die sogenannte Mantisse, die auch als **Signifikand** bezeichnet wird) und e (dem Exponenten), so dass

$r = s*m*b^e$

gilt oder möglichst gut angenähert wird. Um eine eindeutige Darstellung zu erreichen, wird der Exponent in der Regel so gewählt, dass entweder $0{,}1 \le m < 1$ oder $1 \le m < b$ gilt. Die C/C++-Datentypen *float*, *double* und *long double* verwenden binäre Gleitkommaformate, die in IEEE 754 standardisiert sind.

Beispiel: Für die Zahl 3,14 im Dezimalsystem (b=10) erhält man die Darstellung

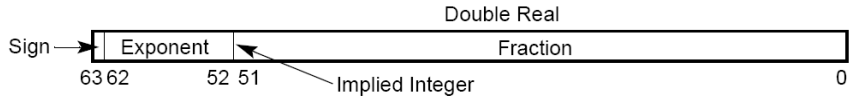
$3{,}14 = (+1) * 0{,}314 * 10^1$, also s=1, m=0,314 und e=1

Die Zahl 314 hat dieselbe Mantisse und dasselbe Vorzeichen, aber den Exponenten 3:

$314 = (+1) * 0{,}314 * 10^3$, also s=1, m=0,314 und e=3

Da mit diesem Datenformat die Zahl Null nicht dargestellt werden kann, wird dafür meist ein spezielles Bitmuster von s, m und e verwendet.

Visual C++ verwendet die Gleitkommaformate der Floating Point Unit (FPU) der Intel Prozessoren. Beim Datentyp ***double*** *sind die 8 Bytes folgendermaßen auf Vorzeichen, Exponent und Mantisse (Fraction) verteilt:*



Der Wert v der dargestellten Zahl ergibt sich folgendermaßen aus dem Vorzeichen s, der Fraction m und dem Exponenten e:

if $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.m)$.
if $e = 0$ and $m \neq 0$, then $v = (-1)^s * 2^{(-1022)} * (0.m)$.
if $e = 0$ and $m = 0$, then $v = (-1)^s * 0$.
if $e = 2047$ and $m = 0$, then $v = (-1)^s *$ Inf. // +8 und –8
if $e = 2047$ and $m \neq 0$, then v is a NaN. // **n**ot **a** **N**umber – keine Zahl

Bei der Bestimmung des binären Gleitkommadarstellung einer reellen Zahl r geht man ähnlich wie bei Ganzzahlen vor und sucht Koeffizienten $...b_1b_0b_{-1}b_{-2}...$, so dass

$r = ... b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} ...$

gilt. Von diesen Koeffizienten nimmt man ab dem ersten, von Null verschiedenen, so viele für die Mantisse m, wie diese Bits hat. Die Position der ersten Stelle wird dann durch eine Multiplikation mit 2^e berücksichtigt.

Beispiele:
$5_{10} = 1*2^2 + 0*2^1 + 1*2^0 = 1{,}01_2*2^2$ (m = 1,01, e = 2)
$0{,}5_{10} = 1*2^{-1}$ (m = 1,0, e = –1)
$0{,}1875_{10} = 1*2^{-3} + 1*2^{-4} = 1{,}1_2*2^{-3}$ (m = 1,1, e = –3)

Ein Algorithmus zur Bestimmung der Koeffizienten b_1, b_0, b_{-1}, b_{-2}.usw. soll am Beispiel der Zahl $0{,}1_{10}$ gezeigt werden:

Beispiel: $0{,}1_{10} = 0*2^{-1}$ Rest $0{,}1_{10}$
$0{,}1_{10} = 0*2^{-2}$ Rest $0{,}1_{10}$
$0{,}1_{10} = 0*2^{-3}$ Rest $0{,}1_{10}$
$0{,}1_{10} = 1*2^{-4}$ Rest // 1/10 – 1/16 = 8/80 – 5/80 = 3/80
$3/80_{10} = 1*2^{-5}$ Rest // 3/80 – 1/32 = 1/160 = (1/16)*(1/10)

Offensichtlich wiederholen sich die Ziffern anschließend, d.h. die Ziffernfolge wird ein nichtabbrechender, periodischer Dezimalbruch:

0.1_{10} $= 0{,}0(0011)_2$ // Periode in Klammern
$= 1{,}(1001)2^{-4}$ // normiert, so dass die Mantisse mit 1 beginnt

Dieses Beispiel zeigt, dass eine Zahl, die in einem bestimmten Zahlensystem eine abbrechende Dezimalbruchentwicklung hat, in einem anderen Zahlensystem ein nichtabbrechender periodischer Dezimalbruch sein kann. Weitere Beispiele aus anderen Zahlensystemen:

$1/3_{10} = 0{,}1_3$ Der im Dezimalsystem periodische Dezimalbruch 1/3 ist im System zur Basis 3 abbrechend.
$1/7_{10} = 0{,}1_7$ Der im Dezimalsystem periodische Dezimalbruch 1/7 ist im System zur Basis 7 abbrechend.

Generell gilt: Ein Bruch z/n lässt sich genau dann als abbrechender Dezimalbruch in einem Zahlensystem zur Basis B darstellen, wenn alle Primfaktoren des Nenners Teiler von B sind.

Deshalb können in einem binären Gleitkommaformat alle die reellen Zahlen r exakt als Gleitkommazahlen dargestellt werden, für die

$r = \ldots b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} \ldots$

gilt und bei denen die Mantisse für die Anzahl der Koeffizienten ausreicht. Alle anderen reellen Zahlen werden entweder nur durch Näherungswerte dargestellt (falls die Mantisse nicht breit genug ist) oder können nicht dargestellt werden, weil der Exponent e zu klein (Unterlauf, underflow) oder zu groß (Überlauf, overflow) wird.

Bei einem Gleitkommaformat werden also alle reellen Zahlen, die sich erst ab der letzten Stelle der Mantisse unterscheiden, durch dasselbe Bitmuster dargestellt. Deshalb ist die Darstellung einer reellen Zahl im Gleitkommaformat **nur relativ genau, aber nicht immer exakt**. In diesem Punkt unterscheiden sich Gleitkommadatentypen grundlegend von den Ganzzahldatentypen: Bei einem Ganzzahldatentyp entspricht jedem Bitmuster genau eine Zahl im Wertebereich, und diese Darstellung ist immer exakt.

Allerdings ist eine Gleitkommadarstellung auch nicht besonders ungenau. In vielen Anwendungen wirkt sich diese Ungenauigkeit beim Rechnen mit Gleitkommazahlen überhaupt nicht aus.

Bei der Subtraktion von fast gleichgroßen Gleitkommazahlen hat das Ergebnis jedoch oft wesentlich weniger richtige Stellen als die Ausgangszahlen:

Beispiel: f1 und f2 unterscheiden sich in der 7. Stelle:

```
float f1=1.0000010; // float hat 6 signifikante
float f2=1.0000001;                  // Stellen
float f=f1-f2; // f=8.34465026855469E-7
```

Beim Ergebnis f ist bereits die erste Stelle falsch. Ersetzt man hier *float* durch *double*, erhält man bessere Ergebnisse. Aber auch mit *double* kann der Fehler in der Größenordnung von 10% liegen, wenn sich die beiden Operanden nur auf den letzten signifikanten Stellen unterscheiden.

Solche Rundungsfehler können sich im Lauf einer Folge von Rechnungen so weit aufschaukeln, dass das berechnete Ergebnis deutlich vom tatsächlichen abweicht. Es empfiehlt sich deshalb, die Ergebnisse von Rechnungen mit Gleitkommazahlen immer nachzuprüfen (z.B. eine Probe ins Programm aufzunehmen).

Die bisherigen Ausführungen zeigen insbesondere, dass bei der Addition einer kleinen Zahl zu einer großen Zahl die Summe gleich der großen Zahl sein kann, wenn sich die kleine Zahl erst nach der letzten Stelle der Mantisse auf das Ergebnis auswirkt. Damit kann im Unterschied zu den reellen Zahlen der Mathematik

$a + x = a$

sein, ohne dass dabei x = 0 ist.

Außerdem kann das Ergebnis des Ausdrucks a + b + c davon abhängen, in welcher Reihenfolge der Compiler die Zwischensummen berechnet. Wie das Beispiel (mit B = 10 und einer Mantisse mit 3 Stellen)

a = –123,0 b = 123,0 c = 0,456

zeigt, gilt (a + b) + c = 0,456 ≠ 0 = a + (b + c), d.h. das **Assoziativgesetz muss bei der Addition von Gleitkommazahlen nicht gelten**.

Sobald ein zum Vergleich herangezogener Wert das **Ergebnis von Rechenoperationen** ist, muss das Ergebnis dieses Vergleichs nicht mehr dem erwarteten Ergebnis entsprechen, da es durch Rundungsfehler verfälscht sein kann:

Beispiel: Diese Anweisungen geben „ungleich“ aus:

```
float f=0;
for (int i=1;i<=2;i++) f=f+0.1;
if (f==0.2f) // muss nicht gelten!
  cout<<"gleich"<<endl;
else cout<<"ungleich"<<endl;
```

Ersetzt man hier *float* durch *double* , erhält man dagegen „gleich“. Bei einer 10fachen Summation der Werte 0.1 und einem Vergleich mit 1 erhält man für alle Gleitkommatypen das Ergebnis „ungleich“.

Wenn man aber schon nicht feststellen kann, ob zwei Gleitkommawerte gleich sind, die bei einer exakten Darstellung und Rechnung gleich sein müssten, kann man auch bei einem Vergleich von zwei annähernd gleich großen Werten mit >, >=, < und <= nie sicher sein, ob das Ergebnis in die richtige Richtung ausschlägt.

Beispiel: Diese Anweisungen geben die Meldung „größer“ aus.

```
float f=0.1;
if (f > 0.1) textBox1->Text = "größer";
else textBox1->Text = "nicht größer";
```

Ersetzt man hier *float* durch *double* , erhält man jedoch „nicht größer“.

Deshalb sollte man annähernd gleich große **Gleitkommawerte weder mit dem Operator„==“ noch mit einem anderen Vergleichsoperator vergleichen.** Das gilt für jede Programmiersprache und nicht nur für einen speziellen Compiler oder eine spezielle Programmiersprache wie z.B. C++.

1.2 Gleitkommaformate für exakte Arithmetik: Festpunktformate, BCD

Ein **Festkommadatentyp mit n Nachkommastellen** stellt eine Zahl mit Nachkommastellen intern durch eine Ganzzahl dar. Von dieser Zahl werden dann die letzten n Stellen als Nachkommastellen interpretiert.

Beispiel: Die Zahl 3.14 wird bei einem Festkommadatentyp mit

- 2 Nachkommastellen intern durch 314 dargestellt, und mit
- 4 Nachkommastellen durch 31400.

Festkommadatentypen sind für Geldbeträge oft ausreichend. Bei Multiplikationen mit Nachkommastellen gehen aber Nachkommastellen verloren (z.B. 1.01*1.01 = 1.0201). Will man Einheiten wie Gramm in Tonnen konvertieren, sind mehr Stellen notwendig.

1.3 Dezimale Gleitkommaformate

Da Rechnungen mit den binären Gleitkommadatentypen *float* und *double* nicht exakt sind, sind sie für **kaufmännische Rechnungen** oft nicht geeignet.

Beispiel: Addiert man zum Betrag €0,70 eine Steuer von 5%, ergibt das bei einer exakten Rechnung €0,735. Dieser Wert wird auf €0,74 aufgerundet. Bei einer Rechnung mit *double*

```
double tax=0.05;      double amount=0.70;
double result=(1+tax)*amount;
```

erhält man

```
result=0.7349999....;
```

Dieser Wert wird auf €0.73 abgerundet und unterscheidet sich um €0.01 vom richtigen Ergebnis.

Weitere Beispiele dieser Art findet man bei Mike Cowlishaw (z.B. unter http://speleotrove.com/decimal/). Zur Vermeidung solcher Probleme müssen andere Datentypen verwendet werden:

Ein **dezimaler Gleitkommadatentyp** stellt die Werte intern in einem Gleitkommaformat $s*m*10^e$ mit einer ganzzahligen Mantisse m zur Basis 10 dar. Da keine Rundungsfehler durch eine Konversion zur Basis 2 entstehen, können alle Werte aus dem Dezimalsystem exakt dargestellt werden, für die die Mantisse breit genug ist.

Beispiel: Einige Dezimalzahlen und ihre Darstellung in einem dezimalen Gleitkommaformat (nicht normalisiert):

25: $25*10^0=11001_2*10^0$, also m=11001 und e=0
2.5: $25*10^{-1}=11001_2*10^{-1}$, also m=11001 und e=-1
0.1: $1*10^{-1}=1_2*10^{-1}$, also m=1 und e=-1

In Abschnitt 1.1 wurde gezeigt, dass die Dezimalzahl 0.1 in einem binären Gleitkommaformat nicht exakt dargestellt werden kann.

Da die Rechnungen mit einem dezimalen Gleitkommatyp exakt sind, können sie auch für kaufmännische Rechnungen verwendet werden. Da dieses Format aber derzeit von den meisten Prozessoren noch nicht unterstützt wird, sind die Operationen mit diesem Format langsamer als mit einem binären Gleitkommaformat.

1.4 Dezimale Gleitkommaformate in C/C++

Im Gegensatz zu C#, Java, Visual Basic und anderen Sprachen enthalten der C- und der C++-Standard keine dezimalen Gleitkommadatentypen.

Es gibt allerdings für C die IEEE754 konforme declib-Bibliotheken von Mike Cowlishaw

http://speleotrove.com/decimal/

und einen Technical Report (ISO/IEC TR 24733) von 2011 auf der Basis von C++03:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2849.pdf
(D:\cpp.boo\decNumber_2015\Lit\DTR-24733-decimal-Cpp-n2849.pdf)

Dieser TR 24733 ist in gcc implementiert, aber nicht in Visual C++. Mir ist keine Aussage von Microsoft bekannt, dass dieser TR in Visual C++ aufgenommen wird, ohne dass es dazu einen offiziellen Standard gibt.

Von Dietmar Kühl (Mitglied des C++ Standard Komitees) gibt es einen Vorschlag von 2012, den TR 24733 an C++11 und C++14 anzupassen:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3407.html
(D:\cpp.boo\decNumber_2015\Lit\Kuehl-Proposal to Add Decimal Floating Point Support to C-N3407.pdf)

Ich kenne allerdings keine Reaktionen des C++ Standard Komitees auf diesen Vorschlag.

Die hier vorliegende Declib-Bibliothek implementiert eine Teilmenge der Spezifikation von TR 24733. Der gesamte Umfang kann mit den Bibliotheken von Mike Cowlishaw nicht implementiert werden.

Wertebereiche in Visual Studio:

Datentyp	Wertebereich (pos./negativ) in Visual Studio	signifikante Stellen (Genauigkeit)	Größe in Bytes
float	$1{,}18 \times 10^{-38}$ .. $3{,}40 \times 10^{38}$	6	4
double	$2{,}23 \times 10^{-308}$ .. $1{,}79 \times 10^{308}$	15	8
long double	wie *double*	wie *double*	wie *double*

Declib Wertebereiche (http://speleotrove.com/decimal/dbover.html):

Datentyp	Wertebereich (pos./negativ) in Visual Studio	signifikante Stellen (Genauigkeit)	Größe in Bytes
decimal128	Exponenten von -6143 bis +6144	34	128 Bit (16 Byte)

2 Installation

Alle Definitionen sind in der Header-Datei *decimal_TR24732.h*.

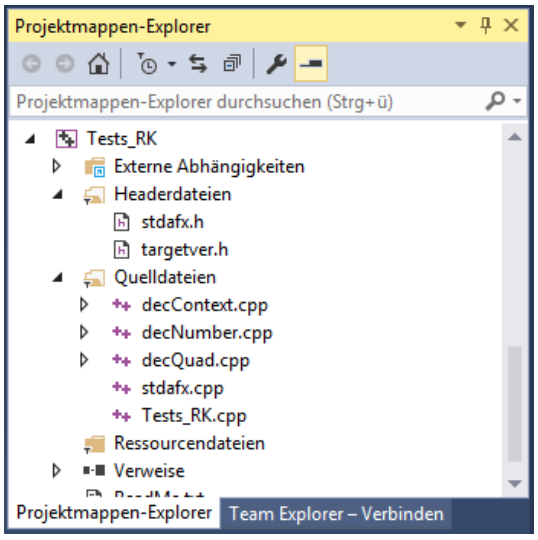
1. Die zip-Datei in ein Verzeichnis entpacken. Für die folgenden Beispiele wurde das Verzeichnis

   C:\DecLib_Include\

   angenommen.

2. Ein C++-Projekt mit Visual Studio (Win32 Konsolen-Anwendung) anlegen. Für die folgenden Beispiele wurde das Projekt *DecLib_Testprojekt_2* angelegt.

3. Diesem Projekt werden die Quelldateien *decContext.cpp* und *decQuad.cpp* aus dem Verzeichnis unter 1. hinzugefügt:



4. In die Quelltextdatei des Projekts wird die #include-Anweisung

   ```
   #include "C:\DecLib_Include\decimal_TR24732.h"
   ```

   aufgenommen. Das Projekt müsste dann ohne Fehlermeldungen des Compilers übersetzt werden.

5. Der dezimale Gleitkommadatentyp *decimal128* ist vorläufig im Namensbereich *rk1::decimal* enthalten (gemäß dem TR24733 soll er im Namensbereich *std::decimal* enthalten sein).

??? In welchen Namensbereich soll decimal128 endgültig aufgenommen werden? „std“ nur ungern?

3 Die Verwendung der decimal_TR24732.h

Der dezimale Gleitkommadatentyp *rk1::decimal::decimal128* kann im Wesentlichen wie einer der vordefinierten Gleitkommadatentypen *float* oder *double* verwendet werden:

```
rk1::decimal::decimal128  d;
```

Nach

```
using rk1::decimal::decimal128;
// bzw.
using namespace rk1::decimal;
```

können die Namensbereiche auch weggelassen werden:

```
decimal128 d;
```

Alle Bibliotheken und Beispiele wurden mit Visual Studio 2010, 2012, 2013 und 2015 kompiliert und getestet.

Alle Deklarationen und Definitionen der ursprünglichen DecLib-Bibliotheken in C von Mike Cowlishaw wurden in den Namensbereich *N_MikeCowlishaw_Sources* gepackt. In diesem Namensbereich steht ebenfalls ein Datentyp *decimal128* zur Verfügung. Dieser hat eine völlig andere Bedeutung. Deswegen sollte **nie** ein

```
using namespace N_MikeCowlishaw_Sources;
```

verwendet werden, da das Fehlermeldungen zur Folge haben. Es dürfte auch nie eine Notwendigkeit dafür bestehen. Falls doch einmal ein Element x aus diesem Namensbereich benötigt wird, kann man es mit der expliziten Angabe eines Namensbereichs verwenden:

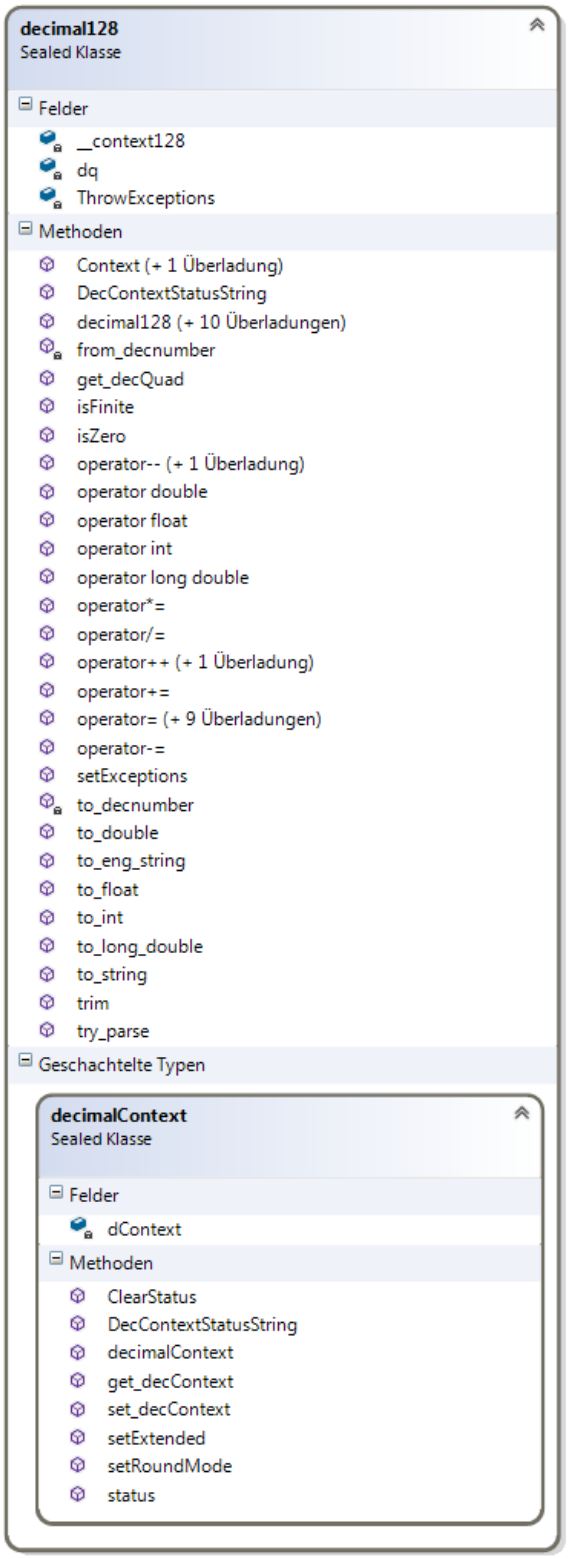
```
N_MikeCowlishaw_Sources::x
```

Die Beispiele in diesem Dokument sind alle aus „SimpleTests_1.h“.

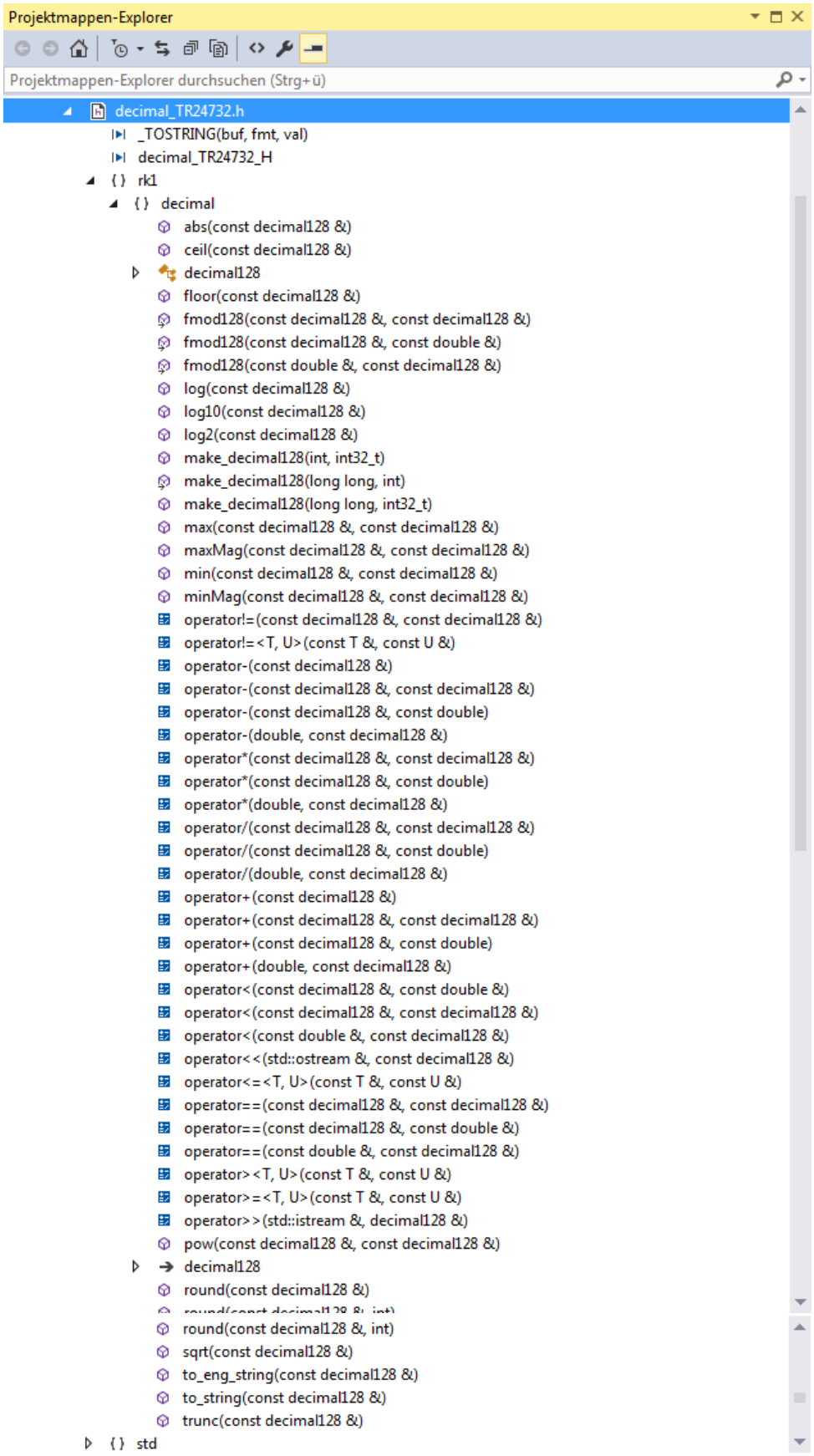
Die in den Beispielen gelegentlich verwendeten Anweisungen *rk1::Assert::AreEqual(T expected, T actual)* prüfen, ob die Argumente für *actual* und *expected* gleich sind (einfache Unittests).

3.1 Übersicht

Elemente der Klasse *decimal123* (Klassenansicht von Visual Studio 2015):



Elemente im Namensbereich *rk1::decimal* (im Projektmappen-Explorer):


Projektmappen-Explorer
Projektmappen-Explorer durchsuchen (Strg+ü)
decimal_TR24732.h
_TOSTRING(buf, fmt, val)
decimal_TR24732_H
rk1
decimal
abs(const decimal128 &)
ceil(const decimal128 &)
decimal128
floor(const decimal128 &)
fmod128(const decimal128 &, const decimal128 &)
fmod128(const decimal128 &, const double &)
fmod128(const double &, const decimal128 &)
log(const decimal128 &)
log10(const decimal128 &)
log2(const decimal128 &)
make_decimal128(int, int32_t)
make_decimal128(long long, int)
make_decimal128(long long, int32_t)
max(const decimal128 &, const decimal128 &)
maxMag(const decimal128 &, const decimal128 &)
min(const decimal128 &, const decimal128 &)
minMag(const decimal128 &, const decimal128 &)
operator!=(const decimal128 &, const decimal128 &)
operator!=<T, U>(const T &, const U &)
operator-(const decimal128 &)
operator-(const decimal128 &, const decimal128 &)
operator-(const decimal128 &, const double)
operator-(double, const decimal128 &)
operator*(const decimal128 &, const decimal128 &)
operator*(const decimal128 &, const double)
operator*(double, const decimal128 &)
operator/(const decimal128 &, const decimal128 &)
operator/(const decimal128 &, const double)
operator/(double, const decimal128 &)
operator+(const decimal128 &)
operator+(const decimal128 &, const decimal128 &)
operator+(const decimal128 &, const double)
operator+(double, const decimal128 &)
operator<(const decimal128 &, const double &)
operator<(const decimal128 &, const decimal128 &)
operator<(const double &, const decimal128 &)
operator<<(std::ostream &, const decimal128 &)
operator<=<T, U>(const T &, const U &)
operator==(const decimal128 &, const decimal128 &)
operator==(const decimal128 &, const double &)
operator==(const double &, const decimal128 &)
operator><T, U>(const T &, const U &)
operator>=<T, U>(const T &, const U &)
operator>>(std::istream &, decimal128 &)
pow(const decimal128 &, const decimal128 &)
decimal128
round(const decimal128 &)
round(const decimal128 &, int)
sqrt(const decimal128 &)
to_eng_string(const decimal128 &)
to_string(const decimal128 &)
trunc(const decimal128 &)
std

3.2 Konstruktoren

Der **Standardkonstruktor** erzeugt eine Variable mit dem Wert 0:

```
decimal128 d;
string s = d.to_string(); // “0
#ifdef SIMPLEUNITTESTS_H__
        rk1::Assert::AreEqual("0", s, "Tests_Konstruktoren D d");
#endif
```

Die Konstruktoren

```
decimal128(int z)
decimal128(unsigned int z)
decimal128(long z)
decimal128(unsigned long z)
decimal128(long long z)
decimal128(unsigned long long z)

explicit decimal128(double r)
explicit decimal128(long double r)
explicit decimal128(float r)
```

erzeugen eine *decimal128*-Variable mit dem als Argument angegebenen Wert. Da die Gleitkommakonstruktoren *explicit* sind, ist eine Initialisierung mit = nicht möglich. Dieses unterschiedliche Verhalten wurde nur deswegen so implementiert, weil das in ISO/IEC DTR 24733 WG21 N2849 so gefordert wird. Falls diese Asymmetrie als störend empfunden wird, kann man das *explicit* im Konstruktor entfernen. Alle Tests sind damit genau gelaufen.

Beispiel: Die Ganzzahl-Konstruktoren können sowohl zu einer Initialisierung mit = als auch mit () verwendet werden:

```
int i1=1; unsigned int i2=1;
long i3=1; unsigned long i4=1;
long long i5=1; unsigned long long i6=1;
using rk1::decimal::decimal128;
decimal128 d1sa = i1; decimal128 d1sb(i1);
decimal128 d1ua = i2; decimal128 d2ub(i2);
decimal128 d2sa = i3; decimal128 d2sb(i3);
decimal128 d2ua = i4; decimal128 d2ub2(i4);
decimal128 d3sa = i5; decimal128 d3sb(i5);
decimal128 d3ua = i6; decimal128 d3ub(i6);
```

Die expliziten Gleitkomma-Konstruktoren können nicht mit =, sondern nur mit () verwendet werden:

```
float f=1; double d=1; long double ld=1;
decimal128 df2(f);   // decimal128 df1 = f;   // geht nicht
decimal128 dd2(d);   // decimal128 dd1 = d;   // geht nicht
decimal128 dld2(ld); // decimal128 dld1 = ld; // geht nicht
```

Einige Beispiel für die erzeugten Werte:

```
decimal128 d1(3.14f);
string s1 = d1.to_string();
decimal128 d2(0.1f);
string s2 = d2.to_string();
decimal128 d3(0.5f);
string s3 = d3.to_string();

#ifdef SIMPLEUNITTESTS_H__
 rk1::Assert::AreEqual("3.140000", s1, "Tests_Konstruktoren D d1(3.14f)");
 rk1::Assert::AreEqual("0.100000", s2, "Tests_Konstruktoren D d2(0.1f)");
 rk1::Assert::AreEqual("0.500000", s3, "Tests_Konstruktoren D d3(0.5f)");
 const int max = 10;
 for (int i = -max; i < max; i++)
 {
   float f = 0 + 1.0f*i / max;
```

```
    rk1::decimal::decimal128 f1(f);
    rk1::Assert::AreEqual(to_string(f), f1.to_string(), "Tests_Konstruktoren float
D d1(i)");
 }
```

Entsprechende Ergebnisse erhält man auch mit ***double***- und ***long double*** Parametern.

Dem *string*-Konstruktor

```
// Dieser string-Konstruktor ist im TR24733 nicht enthalten
explicit decimal128(std::string s)
```

kann man längere Mantissen als bei *double*-Werten übergeben. Falls das Argument keinen zulässigen *decimal128*-Wert darstellt, ist das Ergebnis NaN.

Beispiel:
```
decimal128 e("2.71828182845904523536028747135 27");
string se = e.to_string(); // se = "2.71828182845904523536028747135 27"
decimal128 d1("3.14");
string s1 = d1.to_string(); // s1 = "3.14"
decimal128 d2("1.2.");
string s2 = d2.to_string(); // s2 = "NaN"
decimal128 d3("0x");
string s3 = d3.to_string(); // s3 = "NaN"
decimal128 d4("x");
string s4 = d4.to_string(); // s4 = "NaN"
```

3.3 make_decimal128

Die globalen Funktionen

decimal128 make_decimal128(long long coeff, int32_t exponent)
decimal128 make_decimal128(unsigned long long coeff, int32_t exponent)

erzeugen ein *decimal128* aus einem Koeffizienten und einem Exponenten, analog zu TR24733 (3.2.5 Initialization from coefficient and exponent):

decimal128 make_decimal128(long long coeff, int exponent);
decimal128 make_decimal128(unsigned long long coeff, int exponent);

Beispiel: Die Anweisungen

```
const int Max = 10;
for (int i = -Max; i < Max; i++)
  for (int j = -Max; j < Max; j++)
  {
  long long coeff = i;
  int exponent = j;
  rk1::decimal::decimal128 d = rk1::decimal::make_decimal128(coeff, exponent);
  std::cout << d.to_string() << std::endl;
  }
```

erzeugen die Ausgabe

```
-1.0E-9
-1.0E-8
-1.0E-7
-0.0000010
-0.000010
-0.00010
-0.0010
-0.010
-0.10
-1.0
```

-10
-1.0E+2
-1.0E+3
-1.0E+4
-1.0E+5
-1.0E+6
usw.

3.4 Arithmetische Operatoren +, -, * und /

Für *rk1::decimal::decimal128* stehen die üblichen arithmetischen Operatoren zur Verfügung:

Beispiel: Die nächsten beiden Funktionen enthalten alle Operatoren +, -, * und /. Da beide dasselbe Ergebnis erzielen, sind sie gut zum Testen geeignet:

```
static decimal128 geom_Reihe_Addition(decimal128 a, decimal128 q, int n)
{
  decimal128 s = a; // decimal128 s(a); geht auch
  for (int i = 0; i < n; i++)
  {
    a = a*q;
    s = s + a;
  }
  return s;
}

static decimal128 geom_Reihe_Summenformel(decimal128 a, decimal128 q, int n)
{
  decimal128 p = 1; // decimal128 p(1); geht auch
  for (int i = 0; i < n + 1; i++)
    p = p*q;
  decimal128 r = a*(p - decimal128(1)) / (q - decimal128(1));
  return r;
}
```

Diese Operatoren können auch *rk1::decimal::decimal128*-Ausdrücke mit elementaren Datentypen kombinieren.

3.5 Inkrement und Dekrement-Operatoren ++, --

Die Präfix- und Postfix-Operatoren ++ und – haben für *rk1::decimal::decimal128* dieselbe Bedeutung wie für die elementaren Datentypen.

Beispiel: Hier steht D für den Datentyp *rk1::decimal::decimal128*.

```
{ // Postfix
  D i = 0, j = 0, k1, k2;
  k1 = i++; // k=0, i=1
  k2 = j--; // k=0, j=-1

#ifdef SIMPLEUNITTESTS_H__
  rk1::Assert::AreEqual(D(0), k1, "N_SimpleTests_1::Increment_Decrement_1: Postfix
k1");
  rk1::Assert::AreEqual(D(1), i, "N_SimpleTests_1::Increment_Decrement_1: Postfix i");
  rk1::Assert::AreEqual(D(0), k2, "N_SimpleTests_1::Increment_Decrement_1: Postfix
k2");
  rk1::Assert::AreEqual(D(-1), j, "N_SimpleTests_1::Increment_Decrement_1: Postfix j");
#endif
}
{ // Präfix
  D i = 0, j = 0, k1, k2;
```

```
  k1 = ++i; // k=1, i=1
  k2 = --j; // k=-1, j=-1
#ifdef SIMPLEUNITTESTS_H__
  rk1::Assert::AreEqual(D(1), k1, "N_SimpleTests_1::Increment_Decrement_1: Präfix k1");
  rk1::Assert::AreEqual(D(1), i, "N_SimpleTests_1::Increment_Decrement_1: Präfix i");
  rk1::Assert::AreEqual(D(-1), k2, "N_SimpleTests_1::Increment_Decrement_1: Präfix
k2");
  rk1::Assert::AreEqual(D(-1), j, "N_SimpleTests_1::Increment_Decrement_1: Präfix j");
#endif
}
```

3.6 Zuweisungsoperatoren mit elementaren Datentypen, Strings und *decimal128*

Der Zuweisungsoperator akzeptiert als Operanden auf der rechten Seite elementare Datentypen und Strings. Falls der String auf der rechten Seite kein zulässiger *decimal128*-Wert ist, ist das Ergebnis NaN.

Beispiel:
```
decimal128 e;
e="2.718281828459045235602874713527";
string se = e.to_string(); // se = "2.718281828459045235602874713527"
decimal128 d1;
d1="3.14";
string s1 = d1.to_string(); // s1 = "3.14"
decimal128 d2;
d2="1.2.";
string s2 = d2.to_string(); // s2 = "NaN"
decimal128 d3;
d3="0x";
string s3 = d3.to_string(); // s3 = "NaN"
decimal128 d4;
d4="x";
string s4 = d4.to_string(); // s4 = "NaN"
```

Die kombinierten Zuweisungsoperatoren +=, -= usw. haben dieselbe Bedeutung wie bei den elementaren Datentypen. Der rechte Operand kann sowohl den Datentyp *rk1::decimal::decimal128* als auch einen elementaren Datentyp haben.

Beispiel:
```
rk1::decimal::decimal128 d1 = 1, d2 = 2;
d1 += d2;
string s1 = d1.to_string();
d1 += 1; //
string s2 = d1.to_string();
d1 += 1U;
string s3 = d1.to_string();
d1 += 1L;
string s4 = d1.to_string();
d1 += 1UL;
string s5 = d1.to_string();
d1 += 1LL;
string s6 = d1.to_string();
d1 += 1ULL;
string s7 = d1.to_string();
```

3.7 Vergleichsoperatoren

Die Vergleichsoperatoren <, == usw. haben dieselbe Bedeutung wie bei den elementaren Datentypen. Der rechte Operand kann sowohl den Datentyp *rk1::decimal::decimal128* als auch einen elementaren Datentyp haben.

Beispiel:
```
rk1::decimal::decimal128 d1 = 1, d2 = 1;
```

```
using std::cout;
using std::endl;
// comparison operators
if (d1 < d2)  cout << d1 << " less " << d2 << endl;
if (d1 <= d2) cout << d1 << " less equal" << d2 << endl;
if (d1 > d2)  cout << d1 << "greater" << d2 << endl;
if (d1 >= d2) cout << d1 << "greater equal" << d2 << endl;
if (d1 == d2) cout << d1 << "equal" << d2 << endl;
if (d1 != d2) cout << d1 << "not equal" << d2 << endl;

#if _MSC_FULL_VER > 16004000 // das geht ab VS 2010 SP 3
bool b0a = d1 < 0;
bool b0b = d1 == 0;
bool b0c = d1 > 0;

bool b1a = d1 < 1;
bool b1b = d1 == 1;
bool b1c = d1 > 1;

bool b2a = d1 < 2;
bool b2b = d1 == 2;
bool b2c = d1 > 2;
```

3.8 IO mit den Operatoren << und >>

Mit den Ein- und Ausgabeoperatoren << und >> können Werte des Datentyps *rk1::decimal::decimal128* wie elementare Datentypen in einen stream geschrieben werden.

Beispiel:

```
int i1 = 1, i2 = 2, i3 = 3;
rk1::decimal::decimal128 d1 = i1, d2 = i2, d3 = i3;
// Teste cout:
cout << "d1=" << d1 << endl;
cout << "d2=" << d2 << endl;
cout << "d1*d2=" << d3 << endl;

std::ostringstream sout;
sout << d1;
string s_out = sout.str();
 SIMPLEUNITTESTS_H__
rk1::Assert::AreEqual(s_out, std::to_string(i1), "N_SimpleTest



string fstream_filename = "fstream.txt";
std::ofstream fout(fstream_filename);
fout << d1 << endl;
fout.close();

std::ifstream fin(fstream_filename);
fin >> d2;
 SIMPLEUNITTESTS_H__
rk1::Assert::AreEqual(d1, d2, "N_SimpleTests_1::IO-Tests:");
```

3.9 Fehler und die Werte NaN und Infinity

Fehler wie eine Division durch 0 werden durch NAN- oder Infinity-Werte dargestellt. Mit der Funktion

```
bool isFinite() const
```

kann man prüfen, ob ein solcher Wert aufgetreten ist. Sie gibt *true* zurück, wenn der Wert weder Infinity noch NaN ist, und sonst *false*.

Beispiel:
```
decimal128 d0 = 0;
decimal128 d1 = 1;
decimal128 r1 = d0 / d1; // 0
decimal128 r2 = d1 / d0;
decimal128 r3 = r2 * r1;
decimal128 r4 = r2 / r2;
string s1 = r1.to_string(); // "0"
string s2 = r2.to_string(); // "Infinity"
string s3 = r3.to_string(); // "NaN"
string s4 = r4.to_string(); // "NaN"

bool b1 = r1.isFinite(); // true
bool b2 = r2.isFinite(); // false
bool b3 = r3.isFinite(); // false
bool b4 = r4.isFinite(); // false
```

Fehler können über

```
rk1::decimal::decimal128::DecContextStatusString();
```

abgefragt werden. Dabei wird einer der Strings

```
#define DEC_Condition_CS "Conversion syntax"
#define DEC_Condition_DZ "Division by zero"
#define DEC_Condition_DI "Division impossible"
#define DEC_Condition_DU "Division undefined"
#define DEC_Condition_IE "Inexact"
#define DEC_Condition_IS "Insufficient storage"
#define DEC_Condition_IC "Invalid context"
#define DEC_Condition_IO "Invalid operation"
#if DECSUBSET
#define DEC_Condition_LD "Lost digits"
#endif
#define DEC_Condition_OV "Overflow"
#define DEC_Condition_PA "Clamped"
#define DEC_Condition_RO "Rounded"
#define DEC_Condition_SU "Subnormal"
#define DEC_Condition_UN "Underflow"
#define DEC_Condition_ZE "No status"
#define DEC_Condition_MU "Multiple status"
```

zurückgegeben. Ein solcher Fehlerstatus bleibt stehen, bis er mit

```
rk1::decimal::decimal128::Context().ClearStatus();
```

manuell wieder gelöscht wird.

Beispiel: Mit den Werten von oben:

```
string s8 = rk1::decimal::decimal128::DecContextStatusString();
     // "Conversion syntax; Division by zero; Inexact; Invalid operation; "

rk1::decimal::decimal128::Context().ClearStatus();
string s9 = rk1::decimal::decimal128::DecContextStatusString();  // ""


decimal128 d10("100000000000"); // too big for int
int i1 = d10.to_int();
string s10 = decimal128::DecContextStatusString();  // ""
```

3.10 Mathematische Funktionen

3.10.1 *sqrt, pow*

Die Funktionen

```
decimal128 sqrt(const decimal128& x);
decimal128 pow(const decimal128& x1, const decimal128& y1);
```

berechnen wie die entsprechenden C-Funktionen die Quadratwurzel und die Potenz und können wie folgendermaßen verwendet werden:

```
      rk1::decimal::decimal128 d = 4;
      string engs = d.to_eng_string();
      d = sqrt(d);
      string sd = d.to_string();
      string engs1 = d.to_eng_string();
      const int Max_i = 100;
      const double Max_j = 10;
      for (int i = 0; i < Max_i; i++)
        for (int j = 0; j < Max_j; j++)
        {
        D x(i + j / Max_j);
        D xx = x*x;
        D r = sqrt(xx);
#ifdef SIMPLEUNITTESTS_H__
        rk1::Assert::AreEqual(x, r, "N_SimpleTests_1::Math_1: sqrt");
#endif
        }
```

3.10.2 log, log10, log2

Die Funktionen

```
decimal128 log(const decimal128& x1);
decimal128 log10(const decimal128& x1);
decimal128 log2(const decimal128& x1);
```

berechnen wie die entsprechenden C-Funktionen die Logarithmen zur Basis e, 2 und 10 und können wie folgendermaßen verwendet werden:

log-Beispiele:

```
decimal128 e("2.71828182845904523536028747135 27");
const int Max = 100;
for (int i = 1; i < Max; i++)
{  // prüfe, ob exp(log(x)) = x gilt
  decimal128 x = i;
  decimal128 log_x = log(x);
  decimal128 res = pow(e, log_x);
  decimal128 rnd = round(res,10);
#ifdef SIMPLEUNITTESTS_H__
  rk1::Assert::AreEqual(x, rnd, "N_SimpleTests_1::Math_1: log");
#endif
}
```

log2-Beispiele:

```
const int Max = 100;
for (int i = 1; i < Max; i++)
{ // prüfe, ob 2^(log2(x)) = x gilt
  decimal128 x = i;
  decimal128 log2_x = log2(x);
  decimal128 res = pow(2, log2_x);
```

```
  decimal128 rnd = round(res,10);
#ifdef SIMPLEUNITTESTS_H__
          rk1::Assert::AreEqual(x, rnd, "N_SimpleTests_1::Math_1: log2");
#endif
}
```

log10-Beispiele:

```
const int Max = 100;
for (int i = 1; i < Max; i++)
{ // prüfe, ob 10^(log(x)) = x gilt
  decimal128 x = i;
  decimal128 log10_x = log10(x);
  decimal128 res = pow(10, log10_x);
  decimal128 rnd = round(res,10);
#ifdef SIMPLEUNITTESTS_H__
          rk1::Assert::AreEqual(x, rnd, "N_SimpleTests_1::Math_1: log10");
#endif
```

3.10.3 trunc, round, ceil, floor, fmod

Die Funktionen

```
decimal128 trunc(const decimal128& x1);
decimal128 round(const decimal128& x1);
decimal128 ceil(const decimal128& x1);
decimal128 floor(const decimal128& x1);
```

haben die Ergebnisse der entsprechenden C-Funktionen:

```
value   round   floor   ceil    trunc
-----   -----   -----   ----    -----
2.3     2.0     2.0     3.0     2.0
3.8     4.0     3.0     4.0     3.0
5.5     6.0     5.0     6.0     5.0
-2.3    -2.0    -3.0    -2.0    -2.0
-3.8    -4.0    -4.0    -3.0    -3.0
-5.5    -6.0    -6.0    -5.0    -5.0
```

Die Funktion

```
decimal128 round(const decimal128& x1, int n)
```

rundet auf n Nachkommastellen.

Diese Funktionen können wie folgendermaßen verwendet werden:

```
int Fall = 0;
const int Max = 100;
for (int i = -Max; i < Max; i++)
  for (int j = -Max; j < Max; j++)
    if (j != 0)
    {
      decimal128 x = i;
      decimal128 y = j;
      decimal128 arg = x + y / Max;

      decimal128 act_trunc = trunc(arg);
      decimal128 act_ceil = ceil(arg);
      decimal128 act_floor = floor(arg);
      decimal128 act_round = round(arg);

      double d_arg = i + (double)j / Max;
```

```
        double d_trunc = std::trunc(d_arg);
        double d_ceil = std::ceil(d_arg);
        double d_floor = std::floor(d_arg);
        double d_round = std::round(d_arg);

        decimal128 exp_trunc(d_trunc);
        decimal128 exp_ceil(d_ceil);
        decimal128 exp_floor(d_floor);
        decimal128 exp_round(d_round);

        string s_arg = arg.to_string();
        string s_act_trunc = act_trunc.to_string();
        string s_act_ceil = act_ceil.to_string();
        string s_act_floor = act_floor.to_string();
        string s_act_round = act_round.to_string();

#ifdef SIMPLEUNITTESTS_H__
      rk1::Assert::AreEqual(act_trunc, exp_trunc, "N_SimpleTests_1::Math_1: trunc");
      rk1::Assert::AreEqual(act_ceil, exp_ceil, "N_SimpleTests_1::Math_1: ceil");
      rk1::Assert::AreEqual(act_floor, exp_floor, "N_SimpleTests_1::Math_1: floor");
      rk1::Assert::AreEqual(act_round, exp_round, "N_SimpleTests_1::Math_1: round");
#endif
    }
```

Die Funktionen

```
inline decimal128 fmod128(const decimal128& x, const decimal128& y)
inline decimal128 fmod128(const double& x, const decimal128& y)
inline decimal128 fmod128(const decimal128& x, const double& y)
```

haben meist dasselbe Ergebnis wie die *fmod*-Funktionen aus der Standardbibliothek. Rundungsfehler können aber zu Abweichungen führen.

3.11 Konversionen

Falls ein String einen zulässigen *decimal128*-Wert darstellt, gibt

```
static inline bool try_parse(const std::string& s, decimal128& result)
```

true zurück, und schreibt den konvertierten Wert in das Argument für *result*. Falls das Argument für s keinen zulässigen Wert darstellt, gibt diese Funktion *false* zurück und verändert das Argument für *result* nicht.

Beispiel: Die Strings in *src1* erzeugen mit *try_parse* die Werte in *exp*. Die Rückgabewerte sind die Werte in *succ*:

```
        const int max = 8;
        std::string src1[max] = { "123", "0", "", "a", "1.23",
          "1a23", "1.23.45", "1.23e4" };
        decimal128 exp[max] =   { 123,    0,  0,  0,   decimal128(1.23),
          0, decimal128(0), decimal128(1.23e4) };
        bool succ[max] = { true, true, false, false, true,
          false, false, true };
        for (int i = 0; i < max; i++)
        {
          decimal128 act;
          bool success = decimal128::try_parse(src1[i], act);
          if (success)
          {
          }
          std::string s_act = act.to_string();
        #ifdef SIMPLEUNITTESTS_H__
          rk1::Assert::AreEqual(exp[i], act, "N_SimpleTests_1::Conversion_Tests::try_parse
        value "+std::to_string(i));
```

```
  rk1::Assert::AreEqual(succ[i], success,
"N_SimpleTests_1::Conversion_Tests::try_parse success" + std::to_string(i));
#endif
}
```

Ein *decimal128*-Wert kann mit den Elementfunktionen

```
std::string to_string() const
std::string to_eng_string() const
```

sowie den globalen Funktionen

```
std::string to_string(const decimal128& d)
std::string to_eng_string(const decimal128& d)
```

in einen String konvertiert werden. Die Funktion mit „eng“ im Namen erzeugen identische Strings, nur dass bei einer Darstellung mit Exponenten die Exponenten ein Vielfaches von 3 sind.

Beispiele: Die Anweisungen

```
const int max = 50;
decimal128 factor("1.234567890123456789");
for (int i = 0; i < max; i=i+2)
{
  long long coeff = i;
  factor = factor * 100;
  decimal128 act= factor+rk1::decimal::make_decimal128(coeff,-i);
  cout << act.to_string() << " - " << act.to_eng_string() << endl;
}
```

erzeugen die Ausgabe:

```
123.456789012345678900000000 - 123.456789012345678900000000
12345.698901234567890000000000000 - 12345.698901234567890000000000000
1234567.890523456789000000000000000 - 1234567.890523456789000000000000000
123456789.012351678900000000000000 - 123456789.012351678900000000000000
12345678901.234567970000000000000000 - 12345678901.234567970000000000000000
1234567890123.456789001000000000000 - 1234567890123.456789001000000000000
123456789012345.678900000120000000 - 123456789012345.678900000120000000
12345678901234567.89000000000014000 - 12345678901234567.89000000000014000
123456789123456789.000000000000002 - 123456789123456789.000000000000002
12345678901234567890.0000000000001 - 12345678901234567890.0000000000001
1234567890123456789000.00000000001 - 1234567890123456789000.00000000001
123456789012345678900000.000000001 - 123456789012345678900000.000000001
12345678901234567890000000.0000001 - 12345678901234567890000000.0000001
1234567890123456789000000000.00001 - 1234567890123456789000000000.00001
123456789012345678900000000000.001 - 123456789012345678900000000000.001
12345678901234567890000000000000.1 - 12345678901234567890000000000000.1
1.2345678901234567890000000000000001E+34 - 12.345678901234567890000000000000001E+33
1.2345678901234567890000000000000001E+36 - 1.2345678901234567890000000000000001E+36
1.2345678901234567890000000000000001E+38 - 123.45678901234567890000000000000001E+36
1.2345678901234567890000000000000001E+40 - 12.345678901234567890000000000000001E+39
1.2345678901234567890000000000000001E+42 - 1.2345678901234567890000000000000001E+42
1.2345678901234567890000000000000001E+44 - 123.45678901234567890000000000000001E+42
1.2345678901234567890000000000000001E+46 - 12.345678901234567890000000000000001E+45
1.2345678901234567890000000000000001E+48 - 1.2345678901234567890000000000000001E+48
1.2345678901234567890000000000000001E+50 - 123.45678901234567890000000000000001E+48
```

Die Elementfunktion

```
decimal128 trim()
```

gibt einen *decimal128*-Wert zurück, bei dem die Nullen am Ende einer Zahl entfernt sind.

Beispiel: Fügt man im letzten Beispiel nach

```
decimal128 act= factor+rk1::decimal::make_decimal128(coeff,-i);
```

noch die Anweisung

```
act=act.trim();
```

ein, erhält man für die ersten 10 Zeilen die Ausgabe

```
123.4567890123456789 - 123.4567890123456789
12345.69890123456789 - 12345.69890123456789
1234567.890523456789 - 1234567.890523456789
123456789.0123516789 - 123456789.0123516789
12345678901.23456797 - 12345678901.23456797
1234567890123.456789001 - 1234567890123.456789001
123456789012345.678900000012 - 123456789012345.678900000012
12345678901234567.89000000000014 - 12345678901234567.89000000000014
1234567890123456789.000000000000002 - 1234567890123456789.000000000000002
12345678901234567890.0000000000001 - 12345678901234567890.0000000000001
```

Werte der elementaren Datentypen können einem decimal128 zugewiesen werden:

```
decimal128& operator=(int rhs)
decimal128& operator=(unsigned int rhs)
decimal128& operator=(long rhs)
decimal128& operator=(unsigned long rhs)
decimal128& operator=(long long rhs)
decimal128& operator=(unsigned long long rhs)
decimal128& operator=(float rhs)
decimal128& operator=(double rhs)
decimal128& operator=(long double rhs)
```

Die Elementfunktionen von *decimal128*

```
int to_int() const
float to_float() const
double to_double() const
long double to_long_double() const
```

können folgendermaßen verwendet werden:

Ab Visual Studio 2013 stehen diese Konversionen auch noch als explizite Konversionsoperatoren zur Verfügung:

```
explicit operator int() const
explicit operator float() const
explicit operator long double() const
```

4 Die Tests von Mike Cowlishaw

Für die Tests von Mike Cowlishaw wurde ein Interpreter geschrieben, der diese Tests mit der Klasse *decimal128* ausführt. Alle wesentlichen Tests wurden erfolgreich absolviert.

4.1 abs

abs.decTest_msc.log: ###number of tests passed=64 failed=0

4.2 add

add.decTest_msc.log: ###number of tests passed=1082 failed=0

4.3 compare, comparesig

compare.decTest_msc.log: ###number of tests passed=605 failed=34

Die nicht erfolgreich absolvierten Tests ergaben sich lediglich daraus, dass der Interpreter die NANs noch nicht versteht:

```
-- PROPAGATING NANS
COMX860 COMPARE  NAN9 -INF   ->  NAN9
  ### failed:  :: NaN9 ### failed: ??? could not understand result=NaN9 -INF  NaN9 valid
ops:  <=  >=  ==  Invalid operation;
COMX861 COMPARE  NAN8  999   ->  NAN8
  ### failed:  :: NaN8 ### failed: ??? could not understand result=NaN8 999  NaN8 valid
ops:  <=  >=  ==  Invalid operation;
COMX862 COMPARE  NAN77 INF   ->  NAN77
  ### failed:  :: NaN77 ### failed: ??? could not understand result=NaN77 INF  NaN77
valid ops:  <=  >=  ==  Invalid operation;
COMX863 COMPARE -NAN67 NAN5  -> -NAN67
  ### failed:  :: -NAN67 ### failed: ??? could not understand result=-NAN67 NaN5  -NAN67
valid ops:  <=  >=  ==  Invalid operation;
COMX864 COMPARE -INF  -NAN4  -> -NAN4
  ### failed:  :: -INF ### failed: ??? could not understand result=-NAN4 -NAN4  -NAN4
valid ops:  <=  >=  ==  Invalid operation;
COMX865 COMPARE -999  -NAN33 -> -NAN33
  ### failed:  :: -999 ### failed: ??? could not understand result=-NAN33 -NAN33  -NAN33
valid ops:  <=  >=  ==  Invalid operation;
COMX866 COMPARE  INF   NAN2  ->  NAN2
  ### failed:  :: INF ### failed: ??? could not understand result=NaN2 NaN2  NaN2 valid
ops:  <=  >=  ==  Invalid operation;
COMX867 COMPARE -NAN41 -NAN42 -> -NAN41
  ### failed:  :: -NAN41 ### failed: ??? could not understand result=-NAN41 -NAN42  -
NAN41 valid ops:  <=  >=  ==  Invalid operation;
COMX868 COMPARE +NAN41 -NAN42 ->  NAN41
  ### failed:  :: +NAN41 ### failed: ??? could not understand result=NaN41 -NAN42  NaN41
valid ops:  <=  >=  ==  Invalid operation;
```

```
COMX869 COMPARE -NAN41 +NAN42 -> -NAN41
  ### failed:  :: -NAN41 ### failed: ??? could not understand result=-NAN41 +NAN42  -
NAN41 valid ops:  <=  >=  ==  Invalid operation;
COMX870 COMPARE +NAN41 +NAN42 ->  NAN41
  ### failed:  :: +NAN41 ### failed: ??? could not understand result=NaN41 +NAN42  NaN41
valid ops:  <=  >=  ==  Invalid operation;

COMX871 COMPARE -SNAN99 -INF    -> -NAN99 INVALID_OPERATION
  ### failed:  :: -SNAN99 ### failed: ??? could not understand result=-NAN99 -INF  -NAN99
valid ops:  <=  >=  ==  Invalid operation;
COMX872 COMPARE  SNAN98 -11     ->  NAN98 INVALID_OPERATION
  ### failed:  :: sNaN98 ### failed: ??? could not understand result=NaN98 -11  NaN98
valid ops:  <=  >=  ==  Invalid operation;
COMX873 COMPARE  SNAN97  NAN    ->  NAN97 INVALID_OPERATION
  ### failed:  :: sNaN97 ### failed: ??? could not understand result=NaN97 NaN  NaN97
valid ops:  <=  >=  ==  Invalid operation;
COMX874 COMPARE  SNAN16 SNAN94  ->  NAN16 INVALID_OPERATION
  ### failed:  :: sNaN16 ### failed: ??? could not understand result=NaN16 sNaN94  NaN16
valid ops:  <=  >=  ==  Invalid operation;
COMX875 COMPARE  NAN85  SNAN83  ->  NAN83 INVALID_OPERATION
  ### failed:  :: NaN85 ### failed: ??? could not understand result=NaN83 sNaN83  NaN83
valid ops:  <=  >=  ==  Invalid operation;
COMX876 COMPARE -INF    SNAN92  ->  NAN92 INVALID_OPERATION
  ### failed:  :: -INF ### failed: ??? could not understand result=NaN92 sNaN92  NaN92
valid ops:  <=  >=  ==  Invalid operation;
COMX877 COMPARE  088    SNAN81  ->  NAN81 INVALID_OPERATION
  ### failed:  :: 088 ### failed: ??? could not understand result=NaN81 sNaN81  NaN81
valid ops:  <=  >=  ==  Invalid operation;
COMX878 COMPARE  INF    SNAN90  ->  NAN90 INVALID_OPERATION
  ### failed:  :: INF ### failed: ??? could not understand result=NaN90 sNaN90  NaN90
valid ops:  <=  >=  ==  Invalid operation;
COMX879 COMPARE  NAN   -SNAN89  -> -NAN89 INVALID_OPERATION
  ### failed:  :: NaN ### failed: ??? could not understand result=-NAN89 -SNAN89  -NAN89
valid ops:  <=  >=  ==  Invalid operation;
```

comparesig.decTest_msc.log: ###number of tests passed=591 failed=34

4.4 divide, divideint

divide.decTest_msc.log: ###number of tests passed=488 failed=0

divideint.decTest_msc.log: ###number of tests passed=389 failed=0

4.5 max, maxmag

max.decTest_msc.log: ###number of tests passed=298 failed=0

maxmag.decTest_msc.log: ###number of tests passed=268 failed=1

```
MXGX353 MAXMAG -1E+777777777  1E+411111111 -> -1E+777777777
  ### failed: -1E+777777777 maxMag 1E+411111111=Infinity
```

??? Fehler beim Parsen von NANs

4.6 min, minmag

min.decTest_msc.log: ###number of tests passed=287 failed=0

minmag.decTest_msc.log: ###number of tests passed=292 failed=1

```
   MNGX352 MINMAG -1E+777777777 +1E+411111111 ->  1E+411111111
     ### failed: -1E+777777777 minMag +1E+411111111=-Infinity
```

??? Fehler beim Parsen von NANs

4.7 Unäres +, -

minus.decTest_msc.log: ###number of tests passed=87 failed=0

plus.decTest_msc.log: ###number of tests passed=92 failed=0

4.8 multiply

###number of tests passed=356 failed=0

4.9 subtract

###number of tests passed=427 failed=0

Index