```
/**
 * @file        MyNumber.cpp
 * @synopsis    MyNumber-Klasse Definition
 * @author      Jan Tammen (FH Konstanz), <jan.tammen@fh-konstanz.de>
 * @date        2005-03-30
 */

#include "MyNumber.h"

// {{{ Konstruktoren, Destruktor
// ------------------------------------------------------------------- //

// {{{ Default-Konstruktor: "0" erzeugen
MyNumber::MyNumber(void)
{
    this->mNumber.reserve(mMaxDecimalPlaces);
    this->mSign    = 1;
    this->mIsNull = true;
    this->mComma  = 0;
    this->mFractionLength = 0;
}
// }}}

// {{{ Copy-Konstruktor
MyNumber::MyNumber (const MyNumber& x)
{
    this->mSign             = x.mSign;
    this->mComma            = x.mComma;
    this->mIsNull           = x.mIsNull;
    this->mNumber           = x.mNumber;
    this->mFractionLength   = x.mFractionLength;
    this->mIsNull           = x.mIsNull;
}
// }}}

// {{{ Konstruktor: Langzahl aus String erzeugen
MyNumber::MyNumber (string completeNumber)
{
    this->mSign    = 1;
    this->mIsNull = true;
    this->mFractionLength = 0;
    stringstream ssCompleteNumber;
    ssCompleteNumber << completeNumber;

    string sTmp, sFraction, sNumber;
    long unsigned int iIntLength = 0;
    long unsigned int iFractionLength = 0;

    /// String am Dezimalpunkt 'splitten'
    while (getline (ssCompleteNumber, sTmp, '.')) {
        /// Negatives Vorzeichen erkennen
        if (sTmp.at(0) == '-')
        {
            this->setSign(-1);
            sTmp = sTmp.substr(1);
        }

        iIntLength += sTmp.length();
        stringstream ssNumber(sTmp);

        /// Komma an richtiger Stelle setzen
        this->setComma(iIntLength);
        sNumber = sTmp;

        /// Naechsten Teil holen: Nachkommastellen
        getline (ssCompleteNumber, sTmp, '.');
        iFractionLength += sTmp.length();
        sFraction = sTmp;
    }
```



```
    /// Max. erlaubte Anzahl Nachkommastellen ueberschritten
    if (iFractionLength > mMaxDecimalPlaces)
    {
        throw InputException(string("Anzahl der Nachkommastellen zu hoch, breche ab!
"));
    }

    /// Komplette Zahl in Array einlesen. Dazu zunaechst passenden 'vector' anlegen:
    this->mNumber.reserve(iIntLength+mMaxDecimalPlaces);
    string sCompleteNumber = sNumber+sFraction;

    /// String-Iterator holen
    string::const_iterator siCompleteNumber;
    for (siCompleteNumber  = sCompleteNumber.begin();
         siCompleteNumber != sCompleteNumber.end();
         ++siCompleteNumber)
    {
        /// Stelle ans Ende des Feldes einfuegen
        this->mNumber.push_back((*siCompleteNumber)-'0');
    }

    /// Feld mit Nullen auffuellen
    this->mNumber.resize(mMaxDecimalPlaces+iIntLength);
    this->mFractionLength = iFractionLength;
    this->mIsNull = false;
}
// }}}

// {{{ Konstruktor: Langzahl aus Integer erzeugen
MyNumber::MyNumber (long int number)
{
    if (number == 0) { this->mIsNull = true; return; }

    this->mSign   = (number >= 0) ? 1 : -1;
    this->mIsNull = false;
    this->mFractionLength = 0;

    stringstream ssNumber;
    ssNumber << number;
    string sNumber = ssNumber.str();

    /// String-Iterator holen
    string::const_iterator siNumber;
    for (siNumber  = sNumber.begin();
         siNumber != sNumber.end();
         ++siNumber)
    {
        //cout << "[DEBUG] Fuege Stelle ein: " << *siNumber << endl;
        /// Stelle ans Ende des Feldes einfuegen
        this->mNumber.push_back((*siNumber)-'0');
    }

    /// Komma setzen
    this->setComma(sNumber.length());

    /// Feld mit Nullen auffuellen
    this->mNumber.resize(mMaxDecimalPlaces + sNumber.length());
}
// }}}

// {{{ Destruktor
MyNumber::~MyNumber (void)
{
    /// Speicher freigeben
    std::vector<int>().swap(mNumber);
}
// }}}
```



```
// ------------------------------------------------------------------- //
// }}}

// {{{ Operatoren
// ------------------------------------------------------------------- //

// {{{ Operatoren: Zuweisung
MyNumber& MyNumber::operator= (const MyNumber& z)
{
    if (&z != this)
    {
        this->mSign             = z.mSign;
        this->mComma            = z.mComma;
        this->mIsNull           = z.mIsNull;
        this->mNumber           = z.mNumber;
        this->mFractionLength   = z.mFractionLength;
        this->mIsNull           = z.mIsNull;
    }

    return *this;
}
// }}}

// {{{ Operatoren: Addition
MyNumber MyNumber::operator+ (const MyNumber& z) const
{
    // {{{ Haben die Faktoren eine unterschiedliche Anzahl Vorkommastellen?
    MyNumber x, y;
    x = *this;
    y = z;

    int delta_vk = int(x.getComma() - y.getComma());

    /// x bzw. y hat weniger Vorkommastellen!
    if (delta_vk < 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            x.mNumber.insert(x.mNumber.begin(), 0);

        x.setComma(x.getComma()+abs(delta_vk));
    }
    else if (delta_vk > 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            y.mNumber.insert(y.mNumber.begin(), 0);

        y.setComma(y.getComma()+abs(delta_vk));
    }
    // }}}

    // {{{ Sonderfaelle behandeln
    /// 0 + y = y
    //if (x.isNull()) return y;

    /// x + 0 = x
    //if (y.isNull()) return x;

    /// Haben wir eigentlich eine Subtraktion? -> verschiedene Vorzeichen
    if (x.getSign() != y.getSign()) return x - (-y);
    // }}}

    /// Leere Summe = s erstellen, Vorzeichen setzen
    MyNumber* s = new MyNumber();
    s->setSign(x.getSign());

    /// Zahlen stellenweise addieren
    unsigned short int carry = 0, extension = 0;
    for (int i = y.mNumber.size()-1; i >= 0; i--)
    {
```



```
        /// Ziffern addieren
        extension = y.mNumber[i] + x.mNumber[i];
        extension += carry;
        carry = 0;

        /// Wenn Zw.ergebnis >= Basis (hier: 10), ist ein Uebertrag entstanden
        if (extension >= 10)
        {
            extension -= 10;
            carry = 1;
        }

        if (extension != 0 && s->isNull()) s->mIsNull = false;

        /// Errechnete Dezimalstelle am Anfang des Feldes einfuegen
        s->mNumber.insert(s->mNumber.begin(), extension);
    }

    /// Komma an richtiger Stelle setzen
    s->setComma(x.getComma());

    /// Der Uebertrag muss noch eingefuegt sowie das Komma um eine
    /// Stelle nach rechts verschoben werden
    if (carry > 0)
    {
        s->mNumber.insert(s->mNumber.begin(), carry);
        s->setComma(s->getComma()+1);
    }

    /// Abschliessend das Ergebnis noch mit Nullen fuellen
    s->resize(s->getComma()+s->mMaxDecimalPlaces);
    return *s;
}

MyNumber& MyNumber::operator+= (const MyNumber& z)
{
    return (*this) = (*this) + z;
}
// }}}

// {{{ Operatoren: Subtraktion
MyNumber MyNumber::operator- (const MyNumber& z) const
{
    // {{{ Haben die Faktoren eine unterschiedliche Anzahl Vorkommastellen?
    MyNumber x, y;
    x = *this;
    y = z;

    int delta_vk = int(x.getComma() - y.getComma());

    /// x bzw. y hat weniger Vorkommastellen!
    if (delta_vk < 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            x.mNumber.insert(x.mNumber.begin(), 0);

        x.setComma(x.getComma()+abs(delta_vk));
    }
    else if (delta_vk > 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            y.mNumber.insert(y.mNumber.begin(), 0);

        y.setComma(y.getComma()+abs(delta_vk));
    }
    // }}}

    // {{{ Sonderfaelle behandeln
    /// 0 - y = -y
```

```
    if (x.isNull()) return -y;

    /// x - 0 = x
    if (y.isNull()) return x;

    /// Haben wir eigentlich eine Addition? -> verschiedene Vorzeichen
    if (x.getSign() != y.getSign()) return x + (-y);
    // }}}

    /// Leere Differenz = d erstellen
    MyNumber* d = new MyNumber();

    bool bSwapped = false;
    /// Betraglich groessere Zahl finden
    if (::abs(x) < ::abs(y))
    {
        MyNumber tmp = x;
        x = y;
        y = tmp;
        bSwapped = true;
    }

    d->setSign(x.getSign());

    /// Zahlen stellenweise subtrahieren
    unsigned short int carry = 0, extension = 0;
    for (int i = y.mNumber.size()-1; i >= 0; i--)
    {
        /// Stelle des Minuenden ist *kleiner* als die des Subtrahenden -> Uebertrag
        if (x.mNumber[i] < (y.mNumber[i] + carry))
        {
            extension = (x.mNumber[i] + 10) - y.mNumber[i] - carry;
            carry = 1;
        }
        else
        {
            extension = x.mNumber[i] - y.mNumber[i] - carry;
            carry = 0;
        }

        if (extension != 0 && d->isNull()) d->mIsNull = false;

        /// Errechnete Dezimalstelle am Anfang des Feldes einfuegen
        d->mNumber.insert(d->mNumber.begin(), extension);
    }

    /// Komma an richtiger Stelle setzen
    d->setComma(d->mNumber.size() - this->mMaxDecimalPlaces);

    /// Falls die Faktoren vertauscht wurden, muss das
    /// Vorzeichen invertiert werden.
    if (bSwapped) d->setSign(d->getSign()*-1);

    /// Abschliessend das Ergebnis noch mit Nullen fuellen
    d->resize(d->getComma()+d->mMaxDecimalPlaces);
    return *d;
}

MyNumber& MyNumber::operator-= (const MyNumber& z)
{
    return (*this) = (*this) - z;
}

/// Vorzeichen 'invertieren'
MyNumber MyNumber::operator- (void) const
{
    MyNumber result = *this;
    result.setSign(result.getSign()*(-1));
    return result;
```



```
}
// }}}

// {{{ Operatoren: Multiplikation
MyNumber MyNumber::operator* (const MyNumber& z) const
{
    // {{{ Sonderfaelle behandeln
    /// 0 * z = z * 0 = 0
    if (this->isNull() || z.isNull()) return MyNumber();
    // }}}

    MyNumber x, y;
    x = *this;
    y = z;

    /// Leeres Produkt = p erstellen, Vorzeichen setzen
    MyNumber* p = new MyNumber();
    p->mIsNull = false;

    /// Laenge des Nachkommateils von p = (Laenge Nachkommateil this) + (Laenge Nach
kommateil z)
    p->mNumber.resize(x.mNumber.size()+y.mNumber.size());

    /// Zahlen stellenweise multiplizieren
    unsigned short int carry, extension = 0;
    for (unsigned int i = 0; i < y.mNumber.size(); ++i)
    {
        carry = 0;
        MyNumber* tmp = new MyNumber();
        tmp->mIsNull = false;

        /// Jede Ziffer des 1. Faktors mit akt. Ziffer des 2. Faktors multiplizieren
        for (int j = x.mNumber.size()-1; j >= 0; --j)
        {
            /// Die 0 brauchen wir nicht zu addieren
            if (y.mNumber[i] == 0) { tmp->mIsNull = true; continue; }

            /// Ziffern multiplizieren
            extension = y.mNumber[i]*x.mNumber[j];
            extension += carry;
            carry = 0;

            /// Wenn Zw.ergebnis >= Basis (hier: 10), ist ein Uebertrag entstanden
            if (extension >= 10)
            {
                carry = extension/10;
                extension %= 10;
            }

            /// Errechnete Dezimalstelle am Anfang des Feldes einfuegen
            tmp->mNumber.insert(tmp->mNumber.begin(), extension);
            tmp->mFractionLength++;
        }

        if (tmp->isNull()) continue;

        /// Der Uebertrag muss am Anfang der Zahl eingefuegt werden,
        /// ausserdem muss das Komma um eine Position verschoben werden.
        int correctCarry = 0;
        if (carry > 0) { tmp->mNumber.insert(tmp->mNumber.begin(), carry); correctCa
rry = 1;};

        /// Zahl auf neue Laenge bringen
        int newTmpSize = x.mNumber.size()+y.mNumber.size()+correctCarry-1-i;
        tmp->resize(newTmpSize);

        /// Komma an richtiger Stelle positionieren
        int newCommaPos = tmp->mNumber.size()-(2*mMaxDecimalPlaces);
        tmp->setComma(newCommaPos);
```



```
        /// Schliesslich das Zw.ergebnis zum Endergebnis addieren
        *p += *tmp;
        delete tmp;
    }

    /// TESTING: Resizen auf Max. Groesse
    p->resize(p->getComma()+p->mMaxDecimalPlaces);

    /// Vorzeichen: s. Dokumentation
    p->setSign(this->getSign()*z.getSign());
    return *p;
}

MyNumber& MyNumber::operator*= (const MyNumber& z)
{
    return (*this) = (*this) * z;
}
// }}}

// {{{ Operatoren: Ausgabe
std::ostream& operator<< (std::ostream& s, const MyNumber& z)
{
    /// Minuszeichen bei neg. Zahl ausgeben
    if (z.isNegative()) s << '-';

    /// Iterator fuer Zahlen-Array erstellen
    std::vector<int>::const_iterator itNumber;

    /// Nachkomma-Teil ausgeben
    unsigned int i = 0;
    for (itNumber  = z.mNumber.begin();
         itNumber != z.mNumber.end();
         ++itNumber, ++i)
    {
        if (z.getComma() == i) cout << ".";
        s << (*itNumber);
    }

    return s;
}
// }}}

// {{{ Operatoren: Vergleich
bool MyNumber::operator> (const MyNumber& z) const
{
    return !(*this <= z);
}

bool MyNumber::operator>= (const MyNumber& z) const
{
    return !(*this < z);
}

bool MyNumber::operator< (const MyNumber& z) const
{
    /// Vorzeichen vergleichen
    if (this->getSign() != z.getSign())
        return (this->getSign() > z.getSign());

    for (unsigned int i = 0; i < this->mNumber.size(); i++)
    {
        if (z.mNumber[i] > this->mNumber[i])
            return true;
        else if (z.mNumber[i] < this->mNumber[i])
            return false;
    }

    return false;
```



```
}

bool MyNumber::operator<= (const MyNumber& z) const
{
    return (*this < z || *this == z);
}

bool MyNumber::operator== (const MyNumber& z) const
{
    if (this->mSign != z.getSign()) return false;

    for (unsigned int i = 0; i < this->mNumber.size(); i++)
    {
        if (z.mNumber[i] != this->mNumber[i]) return false;
    }

    return true;
}

bool MyNumber::leq (const MyNumber& z) const
{
    MyNumber x, y;
    x = *this;
    y = z;

    int delta_vk = int(x.getComma() - y.getComma());

    /// x bzw. y hat weniger Vorkommastellen!
    if (delta_vk < 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            x.mNumber.insert(x.mNumber.begin(), 0);

        x.setComma(x.getComma()+abs(delta_vk));
    }
    else if (delta_vk > 0)
    {
        for (unsigned int i = abs(delta_vk); i > 0; --i)
            y.mNumber.insert(y.mNumber.begin(), 0);

        y.setComma(y.getComma()+abs(delta_vk));
    }

    return (x <= y);
}
// }}}

// ------------------------------------------------------------------ //
// }}}

// {{{ Hilfsfunktionen
// ------------------------------------------------------------------ //

/// Komma an Stelle x setzen
void MyNumber::setComma (int x)
{
    if (x > 0)
    {
        this->mComma = x;
        return;
    }
    else if (x < 0)
    {
        /// Zunaechst muessen wir Nullen einfuegen
        for (int i = abs(x)+1; i > 0; --i)
            this->mNumber.insert(this->mNumber.begin(), 0);

        this->setComma(1);
    }
```

```
    return;
}

/// Kehrwert der Langzahl berechnen
///
/// Aufgrund der Tatsache, das fuer die interen Berechnungen
/// bei diesem Algorithmus wiederum Langzahl-Objekt verwendet werden,
/// ist diese Methode bei 'vielen' Nachkommastellen recht traege.
MyNumber MyNumber::getKehrwert (void)
{
    unsigned int stellen = mMaxDecimalPlaces;
    MyNumber kehrwert;
    MyNumber nenner = *this;

    if (nenner.isNull() || stellen == 0) return kehrwert;

    MyNumber zaehler = MyNumber(1);  /// Zaehler
    int x;                           /// Stelle, die eingefuegt wird

    for (x = 0; ((nenner * MyNumber(x+1)).leq(zaehler)); x++);      /// <größte ganz
e Zahl mit n * x <= z>
    kehrwert.mNumber.push_back(x);                                  /// Vorkommateil
 einfuegen
    kehrwert.setComma(1);

    /// Nackommastellen berechnen
    while (stellen != 0)
    {
        zaehler = MyNumber(10)*(zaehler - (nenner*MyNumber(x)));
        if (zaehler.isNull()) { kehrwert.resize(stellen); break; }

        for (x = 0; ((nenner * MyNumber(x+1)).leq(zaehler)); x++);  /// <größte ganz
e Zahl mit n * x <= z>

        /// Stelle einfuegen
        kehrwert.mNumber.push_back(x);
        stellen--;

        if (kehrwert.isNull() && x != 0) kehrwert.setIsNull(false);
    }

    return kehrwert;
}

// ---------------------------------------------------------------------- //
// }}}

/* vim: set expandtab tabstop=4 shiftwidth=4 softtabstop=4 foldmethod=marker: */
```