

Objektorientierte Programmierung, Übungsaufgabe 1

Schnittstelle für Langzahl-Arithmetik

Jan Tammen <foobar@fh-konstanz.de>

6. April 2005

Inhaltsverzeichnis

1	Einleitung	3
2	Schnittstelle der Langzahl-Klasse	3
3	Anwendung: Reihenentwicklung	5
4	Klasse mit Datentyp double	5
4.1	Anwendung im Testprogramm	6
5	Klasse mit Langform	7
6	Arithmetik mit Langzahlen	7
6.1	Addition	7
6.1.1	Vorzeichenbehaftete Faktoren	8
6.1.2	Beispiel zur Addition	8
6.1.3	Pseudocode	9
6.2	Subtraktion	9
6.2.1	Beispiel zur Subtraktion	10
6.2.2	Pseudocode	10
6.3	Multiplikation	11
6.3.1	Vorzeichenbehaftete Faktoren	11
6.3.2	Beispiel zur Multiplikation	11
6.3.3	Pseudocode	12
6.4	Kehrwert einer ganzen Zahl	13
6.4.1	Pseudocode	13

1 Einleitung

Um die begrenzte Genauigkeit des Datentyps `double` zu erweitern, muss eine andere Speicherungsform für gebrochene Zahlen gewählt werden. Mögliche Optionen sind die Speicherung als String bzw. das Ablegen jeder einzelnen Ziffer in einem `integer`-Feld.

Solch eine „Langzahl“ soll in einer Klasse gekapselt werden, welche die üblichen arithmetischen Operationen zur Verfügung stellt. Um Implementierungsdetails vor dem Anwender der Klasse zu verbergen, wird ihm ein einfaches Interface, eine Schnittstelle, zur Verfügung gestellt. In der folgenden Dokumentation sollen Ansatz und Entwurf dieser Schnittstelle dargestellt werden.

2 Schnittstelle der Langzahl-Klasse

```
1  /**
2   * @file    Langzahl.h
3   * @synopsis Langzahl Interface
4   * @author  Jan Tammen (FH Konstanz), <jan.tammen@fh-konstanz.de>
5   * @date    2005-03-21
6   */
7
8  #ifndef LANGZAHL_H
9  #define LANGZAHL_H
10
11 class Langzahl
12 {
13     public:
14         /// virtueller Destruktor
15         virtual ~Langzahl() {};
16
17         /// Ueberladene Operatoren
18         Langzahl& operator= (const Langzahl& z);
19
20         Langzahl& operator+ (const Langzahl& z);
21         Langzahl& operator- (const Langzahl& z);
22         Langzahl& operator* (const Langzahl& z);
23         Langzahl& operator/ (const Langzahl& z);
24
25         Langzahl& operator+= (const Langzahl& z);
26         Langzahl& operator-= (const Langzahl& z);
27         Langzahl& operator*= (const int x);
28         Langzahl& operator/= (const Langzahl& z);
29
```

```

30     bool operator> (const Langzahl& z);
31     bool operator>= (const Langzahl& z);
32     bool operator< (const Langzahl& z);
33     bool operator<= (const Langzahl& z);
34
35     bool  isNull ()    const { return mIsNull; }
36     void  setIsNull (bool x) { mIsNull = x; }
37     long int getNumber(void) const;
38
39     protected:
40         /// bool'scher Wert: ist Zahl = 0?
41         bool      mIsNull;
42
43         /// Vorzeichen: -1: negativ, 1: positiv
44         short int mSign;
45 };
46
47 #endif

```

Das Konzept beruht auf der abstrakten Basis-Klasse `Langzahl`, von welcher die späteren konkreten Implementierungen abgeleitet werden. Im Einzelnen müssen die Kindklassen mindestens folgende Methoden implementieren:

operator= Zuweisungsoperator. Kopiert die Datenkomponenten des Parameter-Objektes in das Zielobjekt. Gibt Selbst-Referenz zurück.

operator+ Additionsoperator. Addiert die gekapselte Zahl des Parameter-Objektes zur Zahl des Zielobjektes. Gibt Selbst-Referenz zurück.

operator- Subtraktionsoperator. Subtrahiert die gekapselte Zahl des Parameter-Objektes von der Zahl des Zielobjektes. Gibt Selbst-Referenz zurück.

operator* Multiplikationsoperator. Multipliziert die gekapselte Zahl des Parameter-Objektes mit der Zahl des Zielobjektes. Gibt Selbst-Referenz zurück.

operator/ Divisionsoperator. Dividiert die Zahl des Zielobjektes durch die gekapselte Zahl des Parameter-Objektes. Gibt Selbst-Referenz zurück.

operator+= Zusammengesetzter Zuweisungsoperator: Addition. Gibt Selbst-Referenz zurück.

operator-= Zusammengesetzter Zuweisungsoperator: Subtraktion. Gibt Selbst-Referenz zurück.

operator*= Zusammengesetzter Zuweisungsoperator: Multiplikation. Gibt Selbst-Referenz zurück.

operator/= Zusammengesetzter Zuweisungsoperator: Division. Gibt Selbst-Referenz zurück.

getNumber Gibt den **ganzzahligen** Anteil der Zahl zurück.

isNull Abfrage der Datenkomponente `isNull`. Gibt bool'schen Wert zurück: `true`, falls Zahl = 0, `false`, falls Zahl $\neq 0$.

Die weiteren Implementierungsdetails, wie z.B. die interne Speicherung der Zahl, müssen jeweils in den abgeleiteten Klassen festgelegt werden.

3 Anwendung: Reihenentwicklung

In einem Anwendungsprogramm soll die Verwendung der Schnittstelle demonstriert werden. Dabei werden durch das Programm zwei einfache unendliche Reihenentwicklungen simuliert. Folgende Reihen werden dazu hier betrachtet:

$$\sum_{k=1}^{\infty} \frac{1}{k(k+1)} = \frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \dots = 1 \quad (1)$$

$$\sum_{k=1}^{\infty} (-1)^{k+1} \frac{1}{2k-1} = \pm \dots = \frac{\pi}{4} \quad (2)$$

4 Klasse mit Datentyp double

Eine konkrete Klasse `MyDouble` welche von `Langzahl` abgeleitet wird, benutzt zur Speicherung der Zahl den C++-Standard-Datentyp `double`. Dazu der passenden Ausschnitt aus der Klassen-Deklaration:

```
1 #include "Langzahl.h"
2 class MyDouble : public virtual Langzahl
3 {
4     [...]
5     private:
6         long double mNumber;
7 };
```

Listing 1: Ausschnitt `MyDouble.h`

Die o.g. Methoden/Operatoren können nun ohne weitere Umwege direkt implementiert werden, z.B. für den Additionsoperator:

```

1 MyDouble& MyDouble::operator+ (const MyDouble& z)
2 {
3     this->mNumber = (this->mNumber+z.mNumber);
4     return (*this);
5 }

```

Listing 2: Additionsoperator MyDouble

4.1 Anwendung im Testprogramm

Im Testprogramm sieht die Verwendung der MyNumber-Klasse für die in (1) beschriebene Reihenentwicklung folgendermaßen aus:

```

1 #include "MyDouble.h"
2 int main ()
3 {
4     /// [...] Abfrage Anzahl Iterationen
5
6     MyDouble sum;
7     MyDouble* potenz = new MyDouble(1.0);
8     for (int i = 0; i < numIterations; ++i)
9     {
10         sum += potenz->getKehrwert();
11         potenz->setNumber(potenz->getNumber()*2.0);
12     }
13
14     std::cout << "Summe: " << sum << std::endl;
15 }

```

Listing 3: Verwendung MyDouble im Testprogramm

Die dafür benutzte Hilfsmethode getKehrwert ist wie folgt definiert:

```

1 MyDouble& MyDouble::getKehrwert(void)
2 {
3     MyDouble* tmp = new MyDouble((1.0/this->mNumber));
4     return *tmp;
5 }

```

Listing 4: Deklaration MyDobule::getKehrwert

5 Klasse mit Langform

Für die Klassenversion, welche die Zahl intern in einer Langform speichert, werden folgende Erweiterungen des Datenteils in der Deklaration vorgenommen:

```
1 #include "MyNumber.h"
2 class MyNumber : public virtual Langzahl
3 {
4     [...]
5     private:
6         static const unsigned int mMaxDecimalPlaces = 100;
7         long mNumber;
8         vector<int> mDecimalPlaces;
9 };
```

Listing 5: Ausschnitt MyNumber.h

Beschreibung der Datenkomponenten:

mMaxDecimalPlaces Anzahl der maximal speicherbaren Nachkommastellen.

mNumber Vorkommateil.

mDecimalPlaces Integer-Feld mit den Nachkommastellen.

6 Arithmetik mit Langzahlen

In den folgenden Abschnitten werden die Algorithmen für das Rechnen (Addition, Subtraktion, Multiplikation) mit den durch die zu implementierende `Langzahl`-Klasse repräsentierten Langzahlen (in Pseudocode) vorgestellt.

6.1 Addition

Bei der **Addition** von Langzahlen werden jeweils die Vorkommateile der Faktoren getrennt summiert. Dies ist durch den bei diesem Design gewählten Datentyp (`long int`) des Vorkommateils ohne weitere Hilfsmittel durchführbar. Hierbei besteht natürlich die Gefahr eines Überlaufes, auf den der Anwender entsprechend hingewiesen werden muss.

Es folgt die Summation der Nachkommateile der Faktoren. Dabei wird nach der Schulmethode vorgegangen – die einzelnen Ziffern der Faktoren werden von rechts beginnend addiert, ein eventuell auftretender Übertrag wird in die nächsthöhere Stelle übernommen. Sollte am „Ende“

der Zahl ebenfalls ein Übertrag aufgetreten sein, so muss dieser zum Vorkommateil-Summanden addiert werden.

6.1.1 Vorzeichenbehaftete Faktoren

Neben dem Standardfall $s = x + y$ müssen bei den Additionsoperatoren folgende Spezialfälle behandelt werden:

- $(-x) + y \equiv y - x$
- $x + (-y) \equiv -(y - x)$
- $(-x) + (-y) \equiv -(x + y)$

Die Vorgehensweise lässt sich dabei folgendermaßen zusammenfassen: Sind die Vorzeichen der Faktoren identisch, so ist der Betrag der Summe die Summe der Faktoren-Beträge und das Vorzeichen das gemeinsame Vorzeichen der Faktoren.

Wenn die Faktoren unterschiedliche Vorzeichen haben, muss zunächst der größere von beiden Faktoren-Beträgen ermittelt werden. Der Betrag der Summe ergibt sich nun aus der Differenz des größeren und des kleineren Faktor-Betrages. Das Vorzeichen des größeren Faktor-Betrages ist gleichzeitig auch das Vorzeichen der Summe.

6.1.2 Beispiel zur Addition

```
1 gegeben:
2 x = 12,854571, x_v = 12, x_n = 0,854571
3 y = 98,957235, y_v = 98, y_n = 0,957235
4
5 gesucht:
6 s = (x_v+x_n) + (y_v+y_n)
7
8 -----
9 Vorkommateil:
10 s_v = x_v + y_v = 12 + 98 = 110
11
12 Nachkommateil:
13 s_n = x_n + y_n:
14
15     0,854571
16 +   0,957235
17 -----
18     0,701706
19 +   1,11010 Übertrag
```

```

20 -----
21 = 1,811806
22
23 Addieren des Übertrags zu s_v: s_v + 1 = 111
24
25 Gesamtergebnis: s = 111,811806

```

Listing 6: Beispiel Addition Schulmethode

6.1.3 Pseudocode

Hinweis: Hier wird lediglich der „Normalfall“, also die Addition zweier positiven Zahlen betrachtet. Die anderen Fälle lassen sich auf diesen Fall bzw. die Subtraktion zurückführen.

```

1  Langzahl x, y, s
2
3  // extension: Zwischenergebnis, carry: Übertrag
4  carry, extension in  $\mathbb{N}$ 
5  carry, extension  $\leftarrow$  0
6
7  s.setVorkommateil(x.vorkommaTeil + y.vorkommaTeil)
8
9  for <i> from 0 to <Länge y> do
10     extension  $\leftarrow$  y[i] + x[i]
11     extension  $\leftarrow$  (extension + carry)
12     carry  $\leftarrow$  0
13
14     if <extension  $\geq$  10> then
15         extension  $\leftarrow$  (extension - 10)
16         carry  $\leftarrow$  1
17     end if
18
19     s.addNachkommastelle(extension)
20 end for
21
22 if <carry > 0> then
23     s.vorkommateil  $\leftarrow$  (s.vorkommateil + 1)
24 end if

```

Listing 7: Pseudocode Addition

6.2 Subtraktion

Da bei diesem Entwurf die Speicherung der Zahl intern im Dezimalsystem erfolgt, wird für die **Subtraktion** ebenfalls die bekannte Schulmethode angewandt. Zunächst sollten ebenfalls wieder die Vorzeichen der Faktoren betrachtet werden, um Spezialfälle zu behandeln: $x - (-y) \equiv x + y$, $(-x) - y \equiv -(x + y)$, $(-x) - (-y) \equiv -(y - x)$. Das Vorgehen zur Aufteilung der Rechnung in Vor- und Nachkommateil ist dabei analog zur Additions-Methode.

6.2.1 Beispiel zur Subtraktion

```
1 gegeben:
2 x = 12,854571, x_v = 12, x_n = 0,854571
3 y = 98,957235, y_v = 98, y_n = 0,957235
4
5 gesucht:
6 d = (x_v+x_n) - (y_v+y_n)
7
8 // Bestimmung des größeren Betrages -> "Tausch" der Variablen!
9
10 -----
11 Vorkommateil:
12 s_v = x_v - y_v = 98 - 12 = 86
13
14 Nachkommateil:
15 s_n = x_n - y_n:
16
17     0,957235
18 -   0,854571
19 -----
20     0,103764
21 -   0,00110 Übertrag
22 -----
23 =    0,102664
24
25 Gesamtergebnis: da getauscht wurde: d = -d = -86,102664
```

Listing 8: Beispiel Subtraktion Schulmethode

6.2.2 Pseudocode

Hinweis: Hier wird lediglich der „Normalfall“, also die Subtraktion zweier positiven Zahlen betrachtet. Die anderen Fälle lassen sich auf diesen Fall bzw. die Addition zurückführen. Es wird weiterhin

angenommen, dass in x bereits die betraglich größere Zahl vorliegt.

```
1 Langzahl x, y, d
2
3 // extension: Zwischenergebnis, carry: Übertrag
4 carry, extension in  $\mathbb{N}$ 
5 carry, extension  $\leftarrow 0$ 
6
7 d.setVorkommateil(x.vorkommateil - y.vorkommateil)
8
9 for <i> from 0 to <Länge y> do
10     if <y[i]> > (x[i] + carry) then
11         extension  $\leftarrow$  (x[i] + 10) - y[i] - carry
12         carry  $\leftarrow$  1
13     else
14         extension  $\leftarrow$  x[i] - y[i] - carry
15         carry  $\leftarrow$  0
16     end if
17
18     d.addNachkommastelle(extension)
19 end for
```

Listing 9: Pseudocode Subtraktion

6.3 Multiplikation

Bei der Schulmethode für die **Multiplikation** muss zunächst der Betrag der zu multiplizierenden Faktoren gebildet werden. Anschließend wird der Multiplikand der Reihe nach von rechts nach links mit den einzelnen Ziffern des Multiplikators multipliziert. Dabei muss der Wertigkeit der Ziffern durch ein Herausrücken nach links Rechnung getragen werden. Nun werden die Teilsummen zum Gesamtergebnis aufsummiert.

Schließlich muss noch das Komma an der korrekten Stelle positioniert werden – die Anzahl der Nachkommastellen im Endergebnis ergibt sich dabei aus der Summe der Nachkommastellen der Multiplikanden.

6.3.1 Vorzeichenbehaftete Faktoren

Das Vorzeichen des Endergebnisses ist **positiv**, wenn kein oder beide Faktoren negativ sind und **negativ**, wenn einer der Faktoren negativ ist.

6.3.2 Beispiel zur Multiplikation

```
1 gegeben:
2 x = 12,345678, y = 87,654321
3
4 gesucht:
5 p = x * y
6
7 12345678 * 87654321
8 -----
9      98765424
10 +    86419746
11 +    74074068
12 +    61728390
13 +    49382712
14 +    37037034
15 +    24691356
16 +    12345678
17 -----
18 = 1082152022374638
19
20 Setzen des Kommas nach 6+6=12 Stellen (von rechts):
21
22 p = 1082,152022374638
```

Listing 10: Beispiel Multiplikation Schulmethode

6.3.3 Pseudocode

```
1 Langzahl x, y, p
2 x ← abs(x)
3 y ← abs(y)
4
5 for <i> from 0 to <Länge y> do
6   p +← <multipliziere x mit i-ter Ziffer von y>
7   <y eine Stelle nach rechts verschieben>
8   <x eine Stelle nach links verschieben>
9 end for
```

Listing 11: Pseudocode Multiplikation Schulmethode

Anmerkung: Um o.g. Algorithmus zu verwenden, müsste eine entsprechende Methode implementiert werden, welche eine komplette Langzahl mit einem integer multipliziert und dabei die

„Wertigkeit“ der Stelle berücksichtigt. Alternativ könnte man dieses Vorgehen auch getrennt für Vor- und Nachkommateil wählen.

6.4 Kehrwert einer ganzen Zahl

Der Kehrwert einer ganzen Zahl x ergibt sich allgemein durch die Bildung von $\frac{1}{x}$. Der folgende Algorithmus erreicht entweder sein Ende, falls das Ergebnis „gerade“, also endlich ist oder falls die gewünschte Anzahl an Nachkommastellen (a) erreicht ist.

6.4.1 Pseudocode

```
1  z ← 1                // Zähler
2  n ← x                // Nenner
3  a in N
4  Langzahl ergebnis
5
6  while <true> do
7      x ← <größte ganze Zahl mit n * x ≤ z>
8      ergebnis.setVorkommateil(x)
9
10     if <a = 0> then
11         exit
12     else
13         z ← (10*(z-n * x))
14     end if
15
16     if <z = 0> then
17         exit
18     end if
19
20     x ← <größte ganze Zahl mit n * x ≤ z>
21     ergebnis.addNachkommastelle(x)
22
23     a ← a-1
24
25 end while
```

Listing 12: Pseudocode Kehrwertbildung