

1. Operatoren überladen

1.1 Lösungsidee

Ähnlich wie in der Übung soll eine Klasse erstellt werden, damit mit rationalen Zahlen gerechnet werden kann. Da mit Klassenobjekten jedoch per Default keine Rechenoperationen durchgeführt werden können, müssen die entsprechenden Operatoren überladen, also für die jeweilige Klasse neu definiert werden. Auch die Shorthand-Operanden (abgekürzte Schreibweise mit beispielsweise +=) sowie Zuweisungsoperatoren müssen überladen werden. Eine Besonderheit entsteht, wenn zu einem Klassenobjekt eine ganze Zahl addiert werden soll; hier sind zwei Parameter notwendig, die außerhalb der Klasse stehen und somit als *non-member-methods* bezeichnet werden. Bei binären Operationen werden die Parameter als „*left hand side*“ und „*right hand side*“ bezeichnet und beschreiben damit ihre Position im Verhältnis zum Operanden.

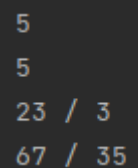
Achtung, unterscheide zwischen überladenen Methoden mit gleichnamigen Methoden, aber abweichenden Parameterlisten, sowie überladenen Operatoren mit abweichenden Definitionen für bestimmte Operationen mit Klassenobjekten.

1.2 Quelltext

Die Rechenoperationen bilden die allgemeinen Grundrechnungsarten mit rationalen Zahlen ab und werden von den jeweiligen Methoden mit den zugehörigen überladenen Operatoren aufgerufen. Um einen Bruch zu kürzen, wird vor der Ausgabe die Methode *normalize()* aufgerufen, die mit dem größten gemeinsamen Vielfachen den Bruch soweit wie möglich rekursiv vereinfacht. Damit nur gültige Brüche gespeichert werden, prüft die Methode *isConsistent()* ab, ob im Nenner keine Null steht. Sollte dies zutreffen, wird eine entsprechende Exception geworfen, also eine individuelle Fehlermeldung, die das Programm vor ungültigen Fehlern und dem Absturz schützt. Abgefangen wird die Exception im *try-catch()*-Block, wobei hier mehrere *catch()*-Blöcke definiert werden können und somit eine beliebig feingranulare Fehlerbehandlung möglich wird. Die Überprüfung auf eine ungültige Bruchzahl kann somit auch sehr einfach mit den Settern realisiert werden.

1.3 Testfälle

Für jeden Operator steht eine entsprechende Testmethode zur Verfügung. Die berechneten Ergebnisse sind korrekt und werden hier nicht weiter behandelt. Ergebnisse der in der Angabe exemplarischen Anwendung:



```
5
5
23 / 3
67 / 35
```

Test auf positiv, negativ oder Null:

```
rationalType a(2,-6);  
rationalType a(2,6);  
rationalType a(0,4);
```

```
Testing negative, positive, zero and normalize:  
- (1 / 3)  
Rational number is negative.  
1 / 3  
Rational number is positive.  
0  
Rational number is zero.
```

Anm.: Die Brüche werden gekürzt dargestellt.

Einfache mathematische Operationen mit Methodenaufruf:

```
rationalType a(4,12);  
rationalType b(6,12);  
a.add(b);  
rationalType a(4,9);  
rationalType b(3,6);  
a.sub(b);
```

```
Testing regular mathematical operations:  
5 / 6  
- (1 / 18)  
1 / 4  
1
```

[gekürzte Methodenaufrufe]

Überladene mathematische Operationen:

```
rationalType a(4,12);  
rationalType b(6,12);  
a = a + b;  
rationalType a(4,9);  
rationalType b(3,6);  
a = a - b;
```

```
Testing overloaded mathematical operations:  
5 / 6  
- (1 / 18)  
1 / 4  
1
```

[gekürzte Methodenaufrufe]

Überladene mathematische Operationen mit gemischten Operanden und Zuweisung:

```
rationalType a(-1,2);  
rationalType b = a * -10; [Ergebnis der Multiplikation: 5]  
  
rationalType a(1,3);  
rationalType b = 1 / a; [Ergebnis der Division: 3]
```

```
Testing mixed calculations (integers with rationals)  
4 / 3  
2 / 3  
5  
3
```

Beachte: unterschiedliche Positionen der Integer-Werte

Vergleichsoperatoren:

Da die Methoden nur einen boolschen Wert zurückgeben, hier die Methodenaufrufe mit den jeweiligen Ergebnissen:

```
testRationalType::testEqual();  
testRationalType::testNotEqual();  
testRationalType::testBiggerEqual();  
testRationalType::testSmallerEqual();  
testRationalType::testSmaller();  
testRationalType::testBigger();
```

```
Testing comparisons:  
0  
1  
1  
0  
0  
1
```

In jeder Methode werden die folgenden beiden Brüche verglichen:

```
rationalType a(3,4);  
rationalType b(1,2);
```

Zuweisung einer rationalen Zahl und eines Integers:

Anmerkung: die erste Ausgabe zeigt den Originalzustand, die zweite den überschriebenen Wert.

```
rationalType a(4,5);  
rationalType b(1,3);  
a = b;  
  
rationalType a(4,5);  
rationalType::printResult(a);  
a = 45;
```

Testing assignments:
4 / 5
1 / 3
4 / 5
45

Einlesen von Datei und von der Konsole:

```
std::fstream file("inputFile.txt");  
a.scan(file);  
std::cout << "Please enter two [...]: ";  
b.scan();
```

Testing scan:
9 / 4
Please enter two seperated digits as rational number: 4 4
3 / 2

Anm.: Auch hier wird der fertig gekürzte Bruch ausgegeben. Inhalt der Datei:

19 . 4
2

Ungültige rationale Zahl und Division durch Null: Exceptions

```
try {  
    rationalType a(2,0);  
} catch (InvalidRationalException &e) {  
    std::cout << "Exception caught: rational number is invalid." << std::endl;  
}  
  
try {  
    rationalType a(1,4);  
    rationalType b(0,4);  
    a.div(b);  
} catch (DivByZeroException &e) {  
    std::cout << "Exception caught: division by zero is  
invalid." << std::endl;  
}
```

Testing exceptions:
Exception caught: rational number is invalid.
Exception caught: division by zero is invalid.

Process finished with exit code 0

Anm.: „Finished with exit code 0“ deutet auf ordnungsgemäßes Beenden des Programms hin.

Aufwand in Stunden: 10