Name: Robin Berger **Aufwand in h:** 10

Punkte: **Kurzzeichen Tutor/in:**

Beispiel 1 (100 Punkte): Operatoren überladen

Schreiben Sie eine Klasse **`rational_t`**, die es ermöglicht, mit Bruchzahlen (über den ganzen Zahlen) zu rechnen.

Ein Beispiel: Das Programmfragment

```
1 rational_t r (-1, 2);
2
3 std::cout << r * -10                  << '\n'
4           << r * rational_t (20, -2) << '\n';
5
6 r = 7;
7
8 std::cout << r + rational_t (2, 3)          << '\n'
9           << 10 / r / 2 + rational_t (6, 5) << '\n';
```

führt zu dieser Ausgabe:

```
<5>
<5>
<23/3>
<67/35>
```

Beachten Sie diese Vorgaben und Implementierungshinweise:

1. Die Datenkomponenten für Zähler und Nenner sind vom Datentyp **`int`**.
2. Bei einer Division durch Null wird ein Fehler ausgegeben bzw. geworfen. Erstellen Sie hierfür eine eigene Exception-Klasse.
3. Werden vom Anwender der Klasse `rational_t` ungültige Zahlen übergeben, so wird ein Fehler ausgegeben bzw. geworfen.
4. Schreiben Sie ein privates Prädikat `is_consistent`, mit dessen Hilfe geprüft werden kann, ob `*this` konsistent (gültig, valide) ist. Verwenden Sie diese Methode an sinnvollen Stellen Ihrer Implementierung, um die Gültigkeit an verschiedenen Stellen im Code zu gewährleisten.
5. Schreiben Sie eine private Methode `normalize`, mit deren Hilfe eine rationale Zahl in ihren kanonischen Repräsentanten konvertiert werden kann. Verwenden Sie diese Methode in Ihren anderen Methoden.
6. Schreiben Sie eine Methode `print`, die eine rationale Zahl auf einem `std::ostream` ausgibt.
7. Schreiben Sie eine Methode `scan`, die eine rationale Zahl von einem `std::istream` einliest.

8. Schreiben Sie eine Methode `as_string`, die eine rationale Zahl als Zeichenkette vom Typ `std::string` liefert. (Verwenden Sie dazu die Funktion `std::to_string`.)
9. Verwenden Sie Referenzen und `const` so oft wie möglich und sinnvoll. Vergessen Sie nicht auf konstante Methoden.
10. Schreiben Sie die Methoden `get_numerator` und `get_denominator` mit der entsprechenden Semantik.
11. Schreiben Sie die Methoden `is_negative`, `is_positive` und `is_zero` mit der entsprechenden Semantik.
12. Schreiben Sie Konstruktoren ohne Argument (default constructor), mit einem Integer (Zähler) sowie mit zwei Integern (Zähler und Nenner) als Argument. Schreiben Sie auch einen Kopierkonstruktor (copy constructor, copy initialization).
13. Überladen Sie den Zuweisungsoperator (assignment operator, copy assignment), der bei Selbstzuweisung entsprechend reagiert.
14. Überladen Sie die Vergleichsoperatoren ==, !=, <, <=, > und >=. Implementieren Sie diese, indem Sie auch Delegation verwenden.
15. Überladen Sie die Operatoren +=, -=, *= und /= (compound assignment operators).
16. Überladen Sie die Operatoren +, -, * und /. Implementieren Sie diese, indem Sie Delegation verwenden. Denken Sie daran, dass der linke Operand auch vom Datentyp `int` sein können muss.
17. Überladen Sie die Operatoren << und >>, um rationale Zahlen „ganz normal“ auf Streams schreiben und von Streams einlesen zu können. Implementieren Sie diese, indem Sie Delegation verwenden.

Anmerkungen: (1) Geben Sie für Ihre Problemlösungen auch Lösungsideen an. (2) Kommentieren Sie Ihre Algorithmen ausführlich. (3) Strukturieren Sie Ihre Programme sauber. (4) Geben Sie ausreichend Testfälle ab und prüfen Sie alle Eingabedaten auf ihre Gültigkeit.

Lösungsidee

Rational_t

Die Klasse rational_t ermöglicht das Rechnen mit Objekten, die Bruchzahlen repräsentieren, das Vereinfachen (Kürzen) dieser, sowie sämtliche Operationen (Grundrechnungsarten, assignment, logische Operatoren, etc…)

Konstruktoren, default Werte 0/1

In der Klasse rational_t stehen mehrere Konstruktoren zu Verfügung. Dabei wird, sofern die zu initialisierenden Werte nicht übergeben wurden, für den Zähler 0 und für den Nenner 1 als Standardwert übernommen. Wird nur eine ZFahl übergeben, dann wird diese zum Zähler – der Nenner wird zu eins. Es wurde also bezüglich des Wertes eine Ganze Zahl übergeben.

Grundrechnungsarten

In diesen Methoden sind die grundlegenen Regeln des Bruchrechnens abgebildet. So müssen beispielsweise bei einer Addition oder Substraktion beide Objekte zuerst auf den gleichen Nenner gebracht und daher auch die Zähler entsprechend multipliziert werden. Dafür dienen mitunter die Hilfsfunktionen gcd und lcm (kgV ergibt sich aus (Nenner1 * Nenner2)/ggT -> Ergebnis wird neuer gemeinsamer Nenner). Da der Nenner entsprechend erweitert werden muss, ist das übergebene Objekt bei der Addition und Subtraktion nicht const. Bei der Multiplikation und Division verhält es sich leichter, da hier nur Zähler und Nenner multipliziert werden müssen (bzw. Kehrwert). Nach einem Rechenschritt werden die Ergebnisse durch die Methode normalize() normalisiert.

Prädikat is_consistent

Grundsätzlich sind alle Zahlen erlaubt, die durch einen int repräsentiert werden können – bis auf die 0 im Nenner. Ist der Nenner eines Bruchs also 0, gibt is_consistent false zurück.

Methode normalize

Wie bereits erwähnt, werden die Brüche in dieser Methode mithilfe des ggT gekürzt. Weiters wird auch überprüft, ob sowohl Zähler als auch Nenner ein negatives Vorzeichen haben – in diesem Fall werden beide mit -1 multipliziert.

Print Funktion

Diese Methode bedient sich der Methode as_string() und gibt daher einen string auf der Konsole aus. In der Methode as_string wird unterschieden, in welchen Zustand ein Bruch annehmen kann. So wird beispielsweise überprüft, ob die Division von Zähler und Nenner ohne Rest möglich ist. In diesem Fall wird das Ergebnis der Division (als Ganze Zahl) in den string gespeichert (das gilt auch wenn 0 im Zähler steht). Wird eine Zahl mit Nenner gleich 0 übergeben wird ein leerer string zurückgegeben und ein error geworfen (und gefangen).

Scan Funktion

Diese Methode liest eine einzelne Zahl aus einem File ein, und speichert die Werte in ein rational_t Objekt. Ist die Eingabe ungültig, wird das Element mit den Standard Werten 0/1 befüllt.

Force error functions

Diese Methoden (eine statisch, eine nicht statisch) werden in den catch_blocks aufgerufen und provizieren eine Division durch 0 und wirft gegebenenfalls einen divide_by_zero_error(), welcher in einer eigenen Klasse programmiert ist (aus Übung übernommen).

Testfälle

`test_all_calculations();`

```
Testing all operation methods:

Addition:
<11/4>
<29/5>
<11/12>

Subtraction:
<1/-4>
<11/5>
<-7/4>

Multiplication:
<15/8>
<36/5>
<5/-9>

Division:
<5/6>
<20/9>
<-5/16>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berger_U
 (Prozess "13308") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

`test_all_calculations_op()`

```
Testing all operations with overloaded operator:

Addition:
<11/4>
<29/5>
<11/12>

Subtraction:
<1/-4>
<11/5>
<-7/4>

Multiplication:
<15/8>
<36/5>
<5/-9>

Division:
<5/6>
<20/9>
<-5/16>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berger_Ue03\x64\
 (Prozess "11148") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

```
test_all_calculations_op_assign();
```

```
Testing all operations with overloaded assign and operator:

Addition:
<11/4>
<29/5>
<11/12>

Subtraction:
<1/-4>
<11/5>
<-7/4>

Multiplication:
<15/8>
<36/5>
<5/-9>

Division:
<5/6>
<20/9>
<-5/16>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berger_Ue03\x64\D
 (Prozess "6728") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

```
test_all_calculations_op_int();
```

```
Testing the calculations of an rational object with an int in both directions:
Addition:
<55/12>
<55/12>

Subtraction:
<65/-12>
<65/12>

Multiplication:
<25/-12>
<25/-12>

Division:
<1/-12>
<-12>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berg
 (Prozess "22240") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

(Addition und Multiplikation sind Kommutativ (Ergebnisse beider Rechnungen müssen gleich sein), Division und Subtraktion wurden in diesem Beispiel ebenfalls richtig gerechnet)

```
test_divide_by_zero();
```

```
Divide By Zero Error
Processing with execution ...


C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung0
 (Prozess "11884") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

```
test_scan_print();
```

```
Testing the scan and print function:
<10/3>
<4>
<14/5>
Wrong input. Standard values are <0/1>
<0>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berger_
 (Prozess "18300") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

```
test_all_logical_operators();
```

```
Testing all logical operators:

Testing equal (==):
a<5/4> is not the same as b<3/2>
a<5/4> is the same as c<5/4>

Testing unequal (!=):
a<5/4> is not the same as b<3/2>
a<5/4> is the same as c<5/4>

Testing smaller (<):
a<2/3> is smaller than b<5/4>
a<2/3> is not smaller than c<1/8>

Testing smaller or equal (<=):
a<2/3> is smaller or equal b<2/3>
a<2/3> is smaller or equal c<7/8>

Testing bigger (>):
a<2/3> is not bigger than b<5/4>
a<2/3> is bigger than c<1/8>

Testing bigger or equal (>=):
a<2/3> is bigger or equal b<2/3>
a<2/3> is bigger or equal c<1/8>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berge
 (Prozess "28564") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

```
test_example_from_instruction();
```

```
Testing the calculations from the instruction:

<5>
<5>
<23/3>
<67/35>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung03\SWE_Berger_Ue
 (Prozess "28808") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```