

Name: Sandra StraubeAufwand 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 `private` 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 Integer (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.

Inhaltsverzeichnis

Beispiel 1 Operatoren überladen:.....	1
Lösungsidee.....	1
Testfälle.....	4
Fall 1.....	4
Fall 2.....	4
Fall 3-4.....	4
Fall 5-9.....	4
Fall 10-13.....	5
Fall 14.....	5
Fall 15.....	5
Fall 16-19.....	5
Fall 20-26.....	6
Fall 27-40.....	7
Fall 41-47.....	8
Fall 48-54.....	8
Fall 55-58.....	9
Fall 59-62.....	9
Fall 63-66.....	9
Fall 66-71.....	10
Fall 72-73.....	10
Fall 74-75.....	10
Fall 76-77.....	11
Fall 78-80.....	11
Fall 81-83.....	11
Fall 84-86.....	11
Fall 87-89.....	12

Fall 90-92.....	12
Fall 93-95.....	12
Fall 96-98.....	12
Fall 99.....	13
Fall 100.....	13
Fall 101.....	13

Beispiel 1 Operatoren überladen:

Lösungsidee

Es soll eine Klasse erstellt werden zur Berechnung von Brüchen. Diese Brüche sollen hier nur aus Integern bestehen, also ganzen Zahlen.

Zur Erstellung dieser sollen verschiedene Konstruktoren erstellt werden, einen Copy Constructor, einen Default Constructor und einen, der einen oder zwei Integer übernimmt. Falls keine Zahlen übergeben werden, sollen standardmäßig 0 als Zähler und 1 als Nenner festgelegt werden. Zudem soll das Vorzeichen extra angelegt werden und Nenner und Zähler an sich positiv bleiben. Um dies zu erreichen, soll unter anderem die Methode `normalize()` verwendet werden. Hierzu ist zu beachten, dass $(-1)/3$ und $1/(-3)$ beide negativ sein sollen, $(-1)/(-3)$ ist jedoch positiv. Zudem soll `normalize()` hauptsächlich die kanonische Form des übergebenen Bruchs herbeiführen, indem der größte gemeinsame Teiler ermittelt wird.

Ebenfalls ist an diversen Stellen zu überprüfen, ob der Bruch an sich valide ist. Der Nenner darf niemals 0 sein, sollte dieser Fall auftreten ist eine Exception mittels Erweiterung der Exception-Klasse auszugeben.

Daneben sollen noch mehrere weitere Methoden zur Überprüfung des Status des aktuellen Bruchs implementiert werden: Ob dieser 0 ist, also der Zähler gleich 0 und der Nenner gleich 1 ist, ob er positiv oder ob er negativ ist. Auch Getter-Methoden für den Zähler und den Nenner sollen vorhanden sein. Diese Methoden sollen entsprechend bereits überall dort angewendet werden, wo es sinnvoll ist.

Die Eingabe soll neben der einfachen Übergabe auch per `istream` möglich sein innerhalb von `scan()`. Hierbei soll auf eine korrekte Form der Eingabe geachtet werden, ansonsten soll der User die Eingabe erneuern. Ganze Zahlen sollen an dieser Stelle nicht behandelt werden, stattdessen ist hier $z/1$ anzugeben. Ebenso sollen Kommazahlen und Text entsprechend behandelt werden. Negative Brüche sollen ebenfalls möglich sein anzugeben. Nach dem Einlesen soll die Normalisierung stattfinden.

Mittels `print()` soll der aktuell übergebene Bruch in korrekter Form ausgegeben werden. Negative Brüche sind zu berücksichtigen, ebenso ganze „Brüche“ wie $5/1$, welche als ganze Zahl ausgegeben werden sollen. Zuhilfe genommen werden soll

hier die Methode `as_string()`, welche den Bruch vorher in einen String umwandeln soll.

Die Übergabe und Ausgabe der Streams soll ebenfalls mittels der Operatoren `<<` und `>>` möglich sein, weshalb diese explizit überladen werden sollen. Die entsprechende Logik soll per Delegation zu `print()` und `scan()` eingefügt werden.

Zur Berechnung der Brüche sind nun sowohl Methoden zu implementieren, als auch die entsprechenden Operatoren mittels Delegation dieser Methoden zu überladen:

- Der Operator `+` soll zu der entsprechenden Methode delegieren, um zwei Brüche miteinander zu addieren. Dies könnte mithilfe des größten gemeinsamen Vielfachen implementiert werden oder indem die Nenner beider Brüche auf einen gemeinsamen Nenner erhoben werden, zusammen mit den jeweiligen Zählern und diese dann addiert werden. Hier ist darauf zu achten, dass Brüche auch negativ werden können, indem mit einem größeren negativen Bruch addiert wird.
- Der `-` Operator mit der entsprechenden Methode soll auf die gleiche Art funktionieren, jedoch zuletzt die Zähler subtrahieren. Zusätzlich ist hier drauf zu achten, dass negative Brüche positiv werden, wenn sie subtrahiert werden.
- Die Multiplikation ist ebenfalls auf die entsprechende Methode zu delegieren, dazu sollen Zähler und Zähler, sowie Nenner und Nenner miteinander multipliziert werden. Auf das Vorzeichen ist ebenfalls zu achten, da zwei negative Brüche positiv werden.
- Zuletzt der Operator `/`, welcher im Grunde gleich wie die Multiplikation funktioniert, jedoch hier Zähler * Nenner gerechnet werden sollen. Wichtig hierbei ist, die Division durch 0 zu vermeiden und eine entsprechende Fehlermeldung auszuwerfen.

Diese Methoden/Operatoren sollen ebenfalls Berechnungen mit Integer an der linken und rechten Seite des Operanden zulassen können. Dazu sollen entsprechende Operatoren außerhalb der eigentlichen Klasse erstellt und an die korrekten Methoden delegiert werden.

Neben der eigentlichen Berechnung sollen diese Operatoren noch mit dem Zuweisungsoperator verknüpft werden. Dieser soll die Werte aus der rechten Seite

dessen an den linken Bruch übergeben, außer es ist derselbe Bruch, und anschließend normalisieren. Der Zuweisungsoperator soll mit den Berechnungen zu Compound Assignment Operatoren erweitert werden.

Zuletzt sollen nun noch gängige Vergleichsoperatoren implementiert werden. Da diese oftmals nur den gegenteiligen Wert eines anderen Vergleichsoperators ausgeben, Beispiel == und !=, sind die Funktionen entsprechend zu delegieren. Hierbei können ebenfalls Integer auf der linken Seite sein, da dies jedoch nur explizit bei den Berechnungs-Operatoren erwünscht ist, soll dieser Schritt hier übersprungen werden. In der Theorie sind diese ansonsten auf die gleiche Art außerhalb der Klasse anzulegen.

Es ist darauf zu achten, dass die Brüche korrekt übergeben werden, dass invalide Brüche zeitig berücksichtigt werden, sowie dass Methoden, die ihr Objekt nicht verändern, als Konstant gekennzeichnet werden. Teilungen durch 0 sollen vermieden werden. Vorzeichen müssen korrekt berücksichtigt werden.

Testfälle

Fall 1: Test erfolgreich

Test `as_string()` auf korrekte Ausgabe. Setzt korrekte Funktion von `<<` voraus.

Input: `RationalType r(-1, 2); cout << r;`



```
Microsoft Visual Studio-Debugging-Konsole
--- test_as_string:
- Output (Expect -1/2): <-1/2>
```

Fall 2: Test erfolgreich

Test `print()` auf korrekte Ausgabe.

Input: `RationalType r(-1, 2); r.print();`

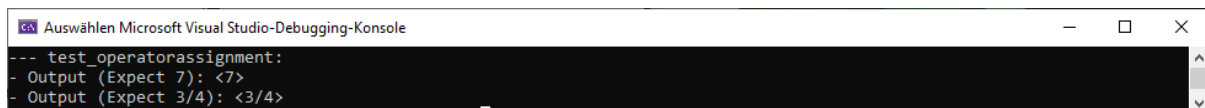


```
Microsoft Visual Studio-Debugging-Konsole
--- test_print:
- Output (Expect -1/2):
<-1/2>
```

Fall 3-4: Tests erfolgreich

Test overload operator= auf

- assign: `RationalType r(-1, 2); int ti{ 7 }; r = ti;`
- copy: `RationalType trt(3, 4); r = trt;.`

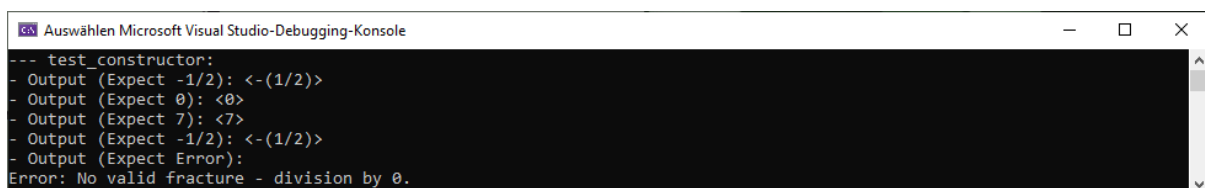


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_operatorassignment:
- Output (Expect 7): <7>
- Output (Expect 3/4): <3/4>
```

Fall 5-9: Tests erfolgreich

Test korrekte Funktion der Konstruktoren auf

- negative Brüche: `RationalType r(-1, 2);`
- leer: `RationalType r2;`
- einzelner Integer: `RationalType r3(7);`
- Copy constructor: `RationalType r4(r);`
- Eingabe invalider Bruch /0: `RationalType r5(1, 0);.`

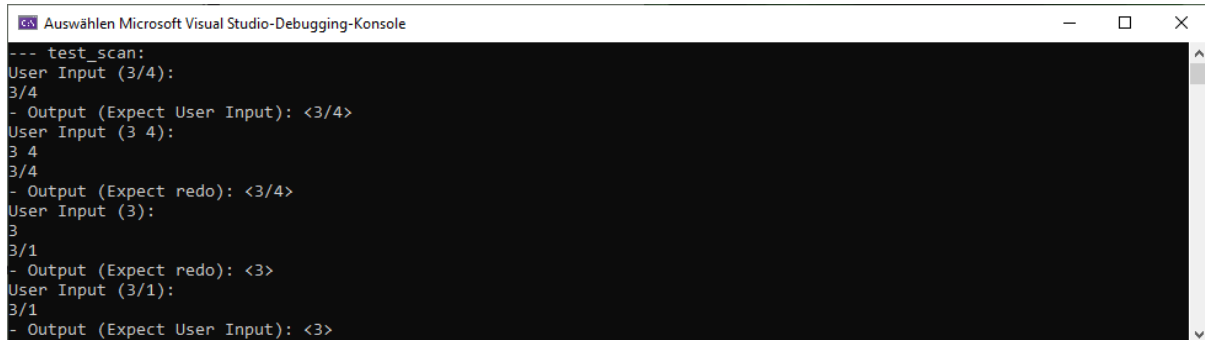


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_constructor:
- Output (Expect -1/2): <-1/2>
- Output (Expect 0): <0>
- Output (Expect 7): <7>
- Output (Expect -1/2): <-1/2>
- Output (Expect Error):
Error: No valid fracture - division by 0.
```


Fall 10-13: Tests erfolgreich

Test scan auf

- Bruch (Erfolg auch bei negativen Brüchen)
- invalide Inputs (selbes Ergebnis bei „“, String)
- ganze Zahl (3/1, keine 3).



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_scan:
User Input (3/4):
3/4
- Output (Expect User Input): <3/4>
User Input (3 4):
3 4
3/4
- Output (Expect redo): <3/4>
User Input (3):
3
3/1
- Output (Expect redo): <3>
User Input (3/1):
3/1
- Output (Expect User Input): <3>
```

Fall 14: Test erfolgreich

Test get_numerator.

Input: `RationalType r(11, 15);`

```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_get_numerator:
- Output (Expect 11): 11
```

Fall 15: Test erfolgreich

Test get_denominator.

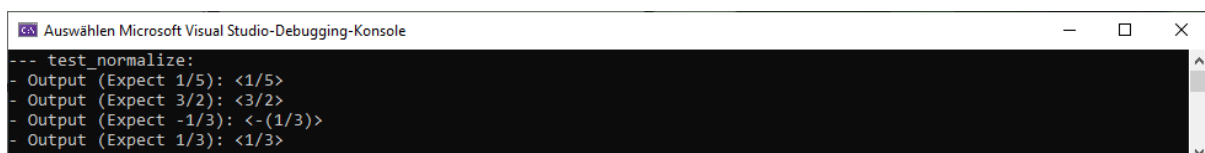
Input: `RationalType r(11, 15);`

```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_get_denominator:
- Output (Expect 15): 15
```

Fall 16-19: Tests erfolgreich

Test normalize auf

- n>d: `RationalType r(3, 15);`
- n<d: `r = { 42, 28 };`
- negative denominator: `r = { 1, -3 };`
- negative numerator and negative denominator: `r = { -1, -3 };`

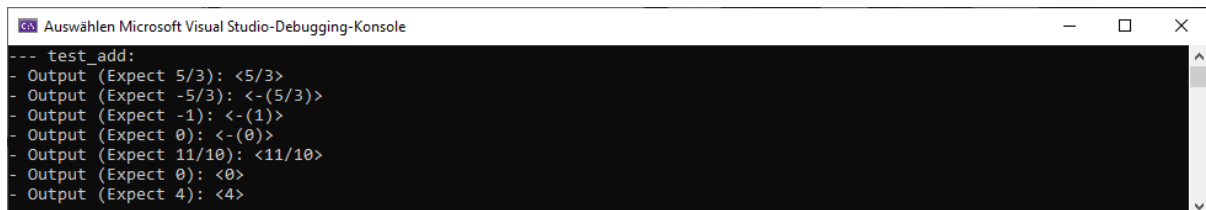


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_normalize:
- Output (Expect 1/5): <1/5>
- Output (Expect 3/2): <3/2>
- Output (Expect -1/3): <-1/3>
- Output (Expect 1/3): <1/3>
```

Fall 20-26: Tests erfolgreich

Test add() auf

- positive Brüche: `RationalType r(4, 3); RationalType r2(1, 3);`
- negative Brüche: `r = { -4, 3 }; r2 = { -1, 3 };`
- negativer Bruch rhs: `r = { 1, 3 }; r2 = { -4, 3 };`
- negativer Bruch lhs: `r = { -1, 3 }; r2 = { 1, 3 };`
- unterschiedliche denominator: `r = { 3, 10 }; r2 = { 4, 5 };`
- beide Brüche 0: `r = {}; r2 = {};`
- ganzzahlige „Brüche“: `r = { 7 }; r2 = { -3 };`

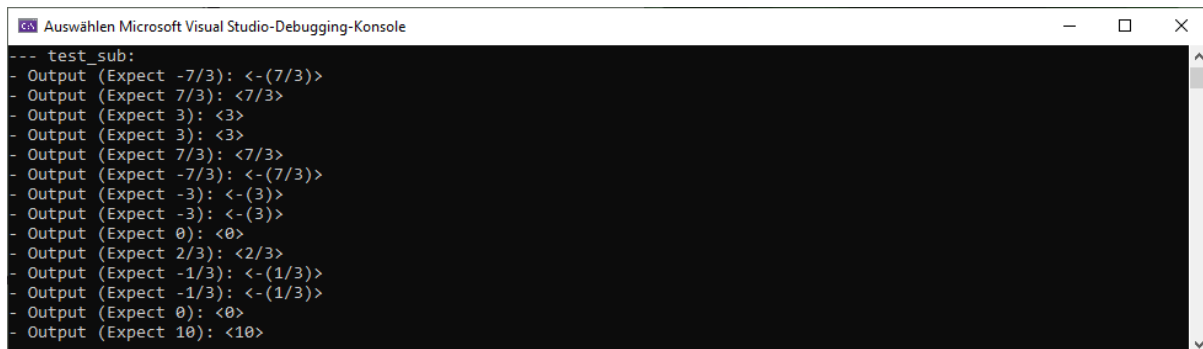


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_add:
- Output (Expect 5/3): <5/3>
- Output (Expect -5/3): <-(5/3)>
- Output (Expect -1): <-(1)>
- Output (Expect 0): <-(0)>
- Output (Expect 11/10): <11/10>
- Output (Expect 0): <0>
- Output (Expect 4): <4>
```

Fall 27-40: Tests erfolgreich

Test sub() auf

```
- +r < +r2:      RationalType r(1, 3); RationalType r2(8, 3);
- +r > +r2:      r = { 8, 3 }; r2 = { 1, 3 };
- +r < -r2:      r = { 1, 3 }; r2 = { -8, 3 };
- +r > -r2:      r = { 8, 3 }; r2 = { -1, 3 };
- -r < -r2:      r = { -1, 3 }; r2 = { -8, 3 };
- -r > -r2:      r = { -8, 3 }; r2 = { -1, 3 };
- -r < +r2:      r = { -1, 3 }; r2 = { 8, 3 };
- -r > +r2:      r = { -8, 3 }; r2 = { 1, 3 };
- +r == +r2:     r = { 1, 3 }; r2 = { 1, 3 };
- +r == -r2:     r = { 1, 3 }; r2 = { -1, 3 };
- lhs = 0:       r = { 0, 3 }; r2 = { 1, 3 };
- rhs = 0:       r = { -1, 3 }; r2 = { 0, 3 };
- beide Brüche 0: r = {}; r2 = {};
- ganzzahlige "Brüche": r = { 7 }; r2 = { -3 };
```

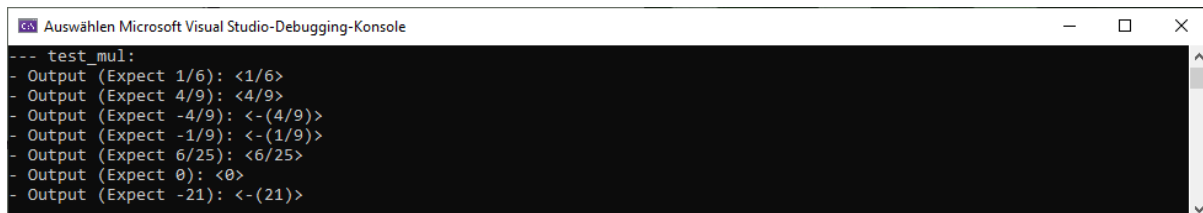


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_sub:
- Output (Expect -7/3): <-(7/3)>
- Output (Expect 7/3): <7/3>
- Output (Expect 3): <3>
- Output (Expect 3): <3>
- Output (Expect 7/3): <7/3>
- Output (Expect -7/3): <-(7/3)>
- Output (Expect -3): <-(3)>
- Output (Expect -3): <-(3)>
- Output (Expect 0): <0>
- Output (Expect 2/3): <2/3>
- Output (Expect -1/3): <-(1/3)>
- Output (Expect -1/3): <-(1/3)>
- Output (Expect 0): <0>
- Output (Expect 10): <10>
```

Fall 41-47: Tests erfolgreich

Test mul() auf

- positive Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- negative Brüche: `r = { -4, 3 }; r2 = { -1, 3 };`
- negativer Bruch rhs: `r = { 1, 3 }; r2 = { -4, 3 };`
- negativer Bruch lhs: `r = { -1, 3 }; r2 = { 1, 3 };`
- unterschiedliche denominator: `r = { 3, 10 }; r2 = { 4, 5 };`
- beide Brüche 0: `r = {}; r2 = {};`
- ganzzahlige „Brüche“: `r = { 7 }; r2 = { -3 };`

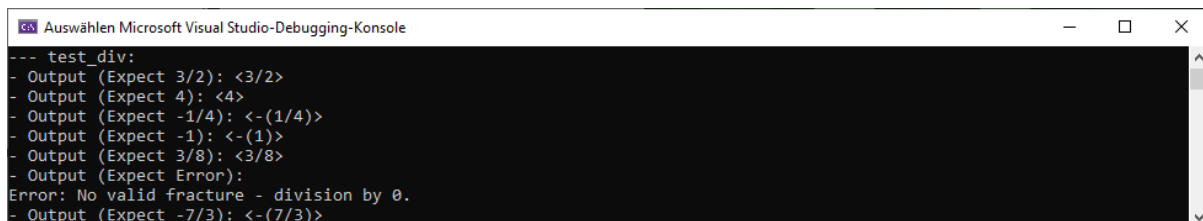


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_mul:
- Output (Expect 1/6): <1/6>
- Output (Expect 4/9): <4/9>
- Output (Expect -4/9): <-(4/9)>
- Output (Expect -1/9): <-(1/9)>
- Output (Expect 6/25): <6/25>
- Output (Expect 0): <0>
- Output (Expect -21): <-(21)>
```

Fall 48-54: Tests erfolgreich

Test div() auf

- positive Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- negative Brüche: `r = { -4, 3 }; r2 = { -1, 3 };`
- negativer Bruch rhs: `r = { 1, 3 }; r2 = { -4, 3 };`
- negativer Bruch lhs: `r = { -1, 3 }; r2 = { 1, 3 };`
- unterschiedliche denominator: `r = { 3, 10 }; r2 = { 4, 5 };`
- beide Brüche 0 / Teilung durch 0: `r = {}; r2 = {};`
- ganzzahlige „Brüche“: `r = { 7 }; r2 = { -3 };`

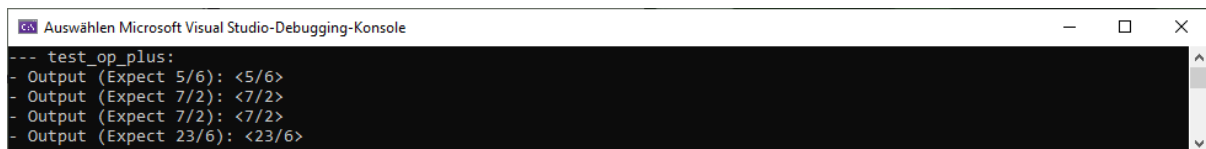


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_div:
- Output (Expect 3/2): <3/2>
- Output (Expect 4): <4>
- Output (Expect -1/4): <-(1/4)>
- Output (Expect -1): <-(1)>
- Output (Expect 3/8): <3/8>
- Output (Expect Error):
Error: No valid fracture - division by 0.
- Output (Expect -7/3): <-(7/3)>
```

Fall 55-58: Tests erfolgreich

Test overload operator+() auf

- Addition zweier Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- Addition Bruch mit Integer: `int r3 = 3; r + r3`
- Addition Integer mit Bruch: `r3 + r`
- Mehrere Additionen aneinandergereiht: `r3 + r + r2.`

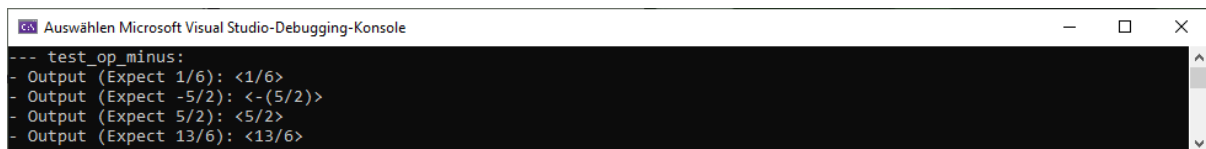


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus:
- Output (Expect 5/6): <5/6>
- Output (Expect 7/2): <7/2>
- Output (Expect 7/2): <7/2>
- Output (Expect 23/6): <23/6>
```

Fall 59-62: Tests erfolgreich

Test overload operator-() auf

- Subtraktion zweier Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- Subtraktion Bruch mit Integer: `int r3 = 3; r - r3`
- Subtraktion Integer mit Bruch: `r3 - r`
- Mehrere Subtraktionen aneinandergereiht: `r3 - r - r2.`

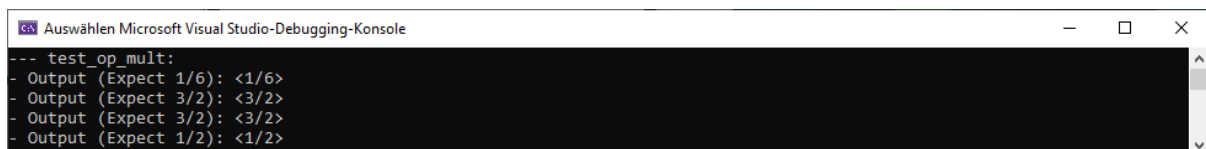


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus:
- Output (Expect 1/6): <1/6>
- Output (Expect -5/2): <-(5/2)>
- Output (Expect 5/2): <5/2>
- Output (Expect 13/6): <13/6>
```

Fall 63-66: Tests erfolgreich

Test overload operator*() auf

- Multiplikation zweier Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- Multiplikation Bruch mit Integer: `int r3 = 3; r * r3`
- Multiplikation Integer mit Bruch: `r3 * r`
- Mehrere Multiplikationen aneinandergereiht: `r3 * r * r2.`

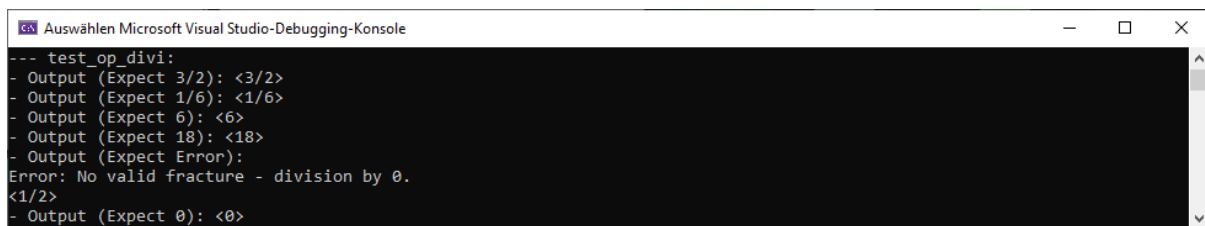


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_mult:
- Output (Expect 1/6): <1/6>
- Output (Expect 3/2): <3/2>
- Output (Expect 3/2): <3/2>
- Output (Expect 1/2): <1/2>
```

Fall 66-71: Tests erfolgreich

Test overload operator/() auf

- Division zweier Brüche: `RationalType r(1, 2); RationalType r2(1, 3);`
- Division Bruch mit Integer: `int r3 = 3; r / r3`
- Division Integer mit Bruch: `r3 / r`
- Mehrere Divisionen aneinandergereiht: `r3 / r / r2`
- Teilung durch 0: `r / 0`
- Teilung 0 durch Bruch: `0 / r.`

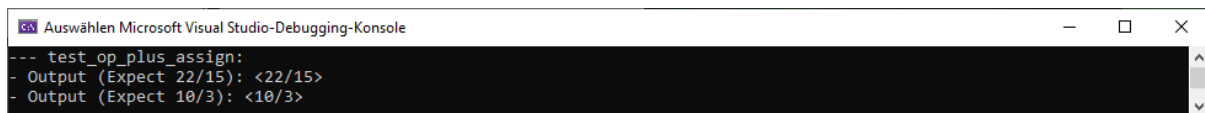


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_divi:
- Output (Expect 3/2): <3/2>
- Output (Expect 1/6): <1/6>
- Output (Expect 6): <6>
- Output (Expect 18): <18>
- Output (Expect Error):
Error: No valid fracture - division by 0.
<1/2>
- Output (Expect 0): <0>
```

Fall 72-73: Tests erfolgreich

Test overload operator+=() auf

- zwei Brüche: `RationalType r(4, 5); RationalType r2(2, 3); r += r2;`
- Bruch auf Integer: `r = { 1, 3 }; int r3 = 3; r += r3;.`



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus_assign:
- Output (Expect 22/15): <22/15>
- Output (Expect 10/3): <10/3>
```

Fall 74-75: Tests erfolgreich

Test overload operator-=() auf

- zwei Brüche: `RationalType r(4, 5); RationalType r2(2, 3); r -= r2;`
- Bruch auf Integer: `r = { 1, 3 }; int r3 = 3; r -= r3;.`



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus_assign:
- Output (Expect 2/15): <2/15>
- Output (Expect -8/3): <-(8/3)>
```

Fall 76-77: Tests erfolgreich

Test overload operator*=() auf

- zwei Brüche: `RationalType r(4, 5); RationalType r2(2, 3); r *= r2;`
- Bruch auf Integer: `r = { 1, 3 }; int r3 = 3; r *= r3;.`

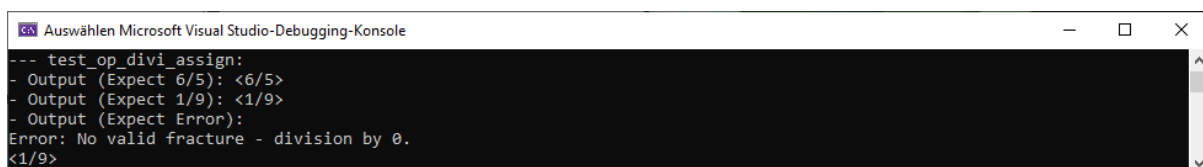


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_mult_assign:
- Output (Expect 8/15): <8/15>
- Output (Expect 1): <1>
```

Fall 78-80: Tests erfolgreich

Test overload operator/=(()) auf

- zwei Brüche: `RationalType r(4, 5); RationalType r2(2, 3); r /= r2;`
- Bruch auf Integer: `r = { 1, 3 }; int r3 = 3; r /= r3;`
- Teilung durch 0: `r /= 0;.`

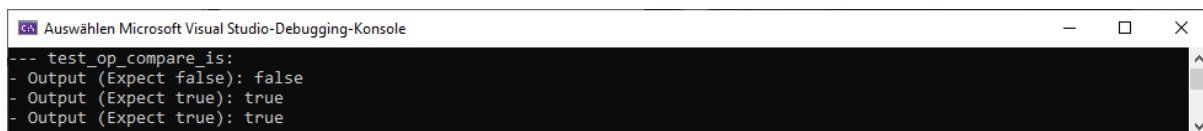


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_divi_assign:
- Output (Expect 6/5): <6/5>
- Output (Expect 1/9): <1/9>
- Output (Expect Error):
Error: No valid fracture - division by 0.
<1/9>
```

Fall 81-83: Tests erfolgreich

Test overload operator==(()) auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3;.`



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_is:
- Output (Expect false): false
- Output (Expect true): true
- Output (Expect true): true
```

Fall 84-86: Tests erfolgreich

Test overload operator!=(()) auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3;.`

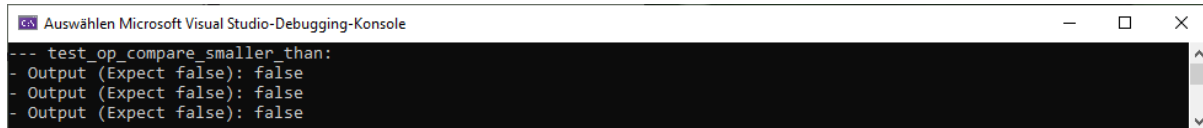


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_is_not:
- Output (Expect true): true
- Output (Expect false): false
- Output (Expect false): false
```

Fall 87-89: Tests erfolgreich

Test overload operator<() auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3; .`

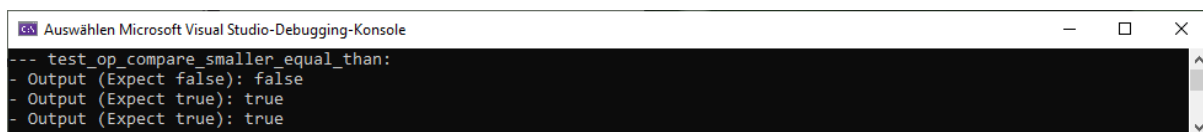


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_smaller_than:
- Output (Expect false): false
- Output (Expect false): false
- Output (Expect false): false
```

Fall 90-92: Tests erfolgreich

Test overload operator<=() auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3; .`

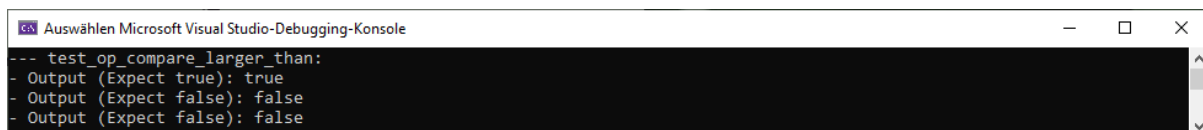


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_smaller_equal_than:
- Output (Expect false): false
- Output (Expect true): true
- Output (Expect true): true
```

Fall 93-95: Tests erfolgreich

Test overload operator>() auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3; .`

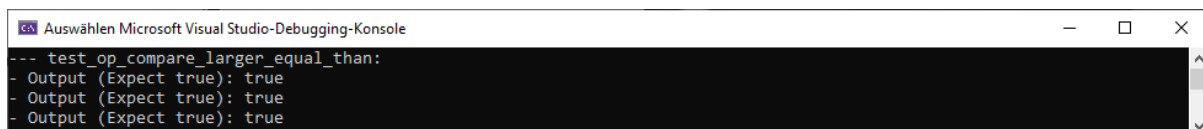


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_than:
- Output (Expect true): true
- Output (Expect false): false
- Output (Expect false): false
```

Fall 96-98: Tests erfolgreich

Test overload operator>=() auf

- zwei ungleiche Brüche: `RationalType r(4, 5); RationalType r2(2, 3);`
- zwei gleiche Brüche: `r2 = r;`
- ein ganzzahliger „Bruch“ auf einen Integer: `r = { 3, 1 }; int r3 = 3; .`

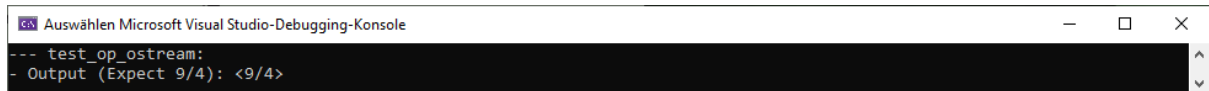


```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_equal_than:
- Output (Expect true): true
- Output (Expect true): true
- Output (Expect true): true
```


Fall 99: Test erfolgreich

Test overload operator<<() auf funktionelle Ausgabe. Da es auf print() basiert, sind weitere Tests hier als gegeben gesehen.

Input: `cout << RationalType(9, 4);`



```
Auswählen Microsoft Visual Studio-Debugging-Konsole
--- test_op_ostream:
- Output (Expect 9/4): <9/4>
```

Fall 100: Test erfolgreich

Test overload operator>>() auf funktionelle Ausgabe. Da es auf scan() basiert, sind weitere Tests hier als gegeben gesehen.

Input: `cin >> r;`

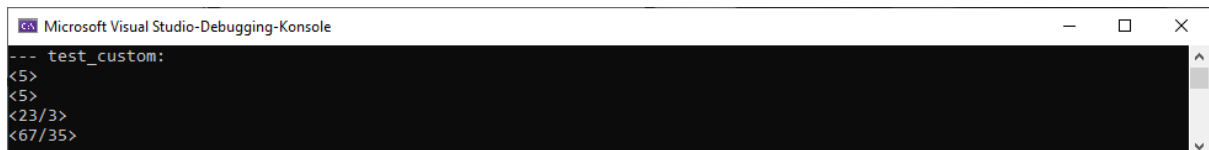


```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_istream (enter a fracture format n/d):
100/30
- Output (Expect User Input): <10/3>
```

Fall 101: Test erfolgreich

Test Programmfragment aus dem Übungsbeispiel 3.

Input: `RationalType r(-1, 2); cout << r * -10; cout << r * RationalType(20, -2);
r = 7; cout << r + RationalType(2, 3); cout << 10 / r / 2 + RationalType(6, 5);`



```
Microsoft Visual Studio-Debugging-Konsole
--- test_custom:
<5>
<5>
<23/3>
<67/35>
```