Name: Theresia Schwaiger **Aufwand in h:** 12

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.

Übung 03

Lösungsidee:

Grundlagen

Operatoren Überladen

Im Allgemeinen erleichtert, das Operatoren überladen die Lesbarkeit von Methoden.
Hier bei wird in unserem Fall zuerst eine Grundfunktion erstellt und diese wird dann in der Operatoren-Methode aufgerufen (Delegation).

Rationale Zahlen:

Das sind alle Zahlen, die Man als Bruchdarstellen kann.

Exception:

Wenn man nun zum Beispiel eine Division durch 0 durchführen möchte, wirft man eine Exception. Das bedeutet, dass das Programm nicht sofort abstürzt, sondern man bekommt eine Error-Warnung, die man zuvor beispielsweise in einer Exception-Klasse erstellt hat und danach läuft das Programm weiter.

Programm

Grundrechnungsarten:

Hier wird einfach jeweils der Zähler und Nenner – addiert, subtrahiert, dividiert, multipliziert – diese werden dann auf die jeweiligen Operatoren-Funktionen erweitert.

- Bei den Funktionen ohne Assignment wird eine Hilfsvariable dazugenommen und diese wird dann auch wieder zurückgegeben
- Bei den Funktionen mit Assignment wir die Funktion mit dem Parameter aufgerufen und dann *this zurückgeben.

Normalisieren:

Diese Funktion ist für das Kürzen der Ergebnisse da. Das heißt es nimmt, den absoluten Wert (abs in C++) des Zählers und Nenners und vergleicht diese, ob man durch sie dividieren kann und das ohne Rest. Wenn ja wird hier auf den Kleinsten Wert dividiert.
Abs – gibt den absoulten wert zurück

Vergleichsoperatoren:

Auch hier werden zuerst die Grundfunktionen erstellt, damit man diese auch in den Operatoren Methoden aufgerufen werden können (Delegation).

Ist Gleich oder ist ungleich:

Es werden Werte verglichen und man gibt true zurück, wenn dieses, je nach Funktionen stimmen oder nicht.

Ist kleiner / kleiner gleich:

Es werden wieder Zahlen verglichen, ob diese kleiner bzw. kleiner gleich ist und es wird true zurückgegeben, wenn dies zutrifft.

Ist größer/größer gleich

Es werden wieder Zahlen verglichen, ob diese größer bzw. größer gleich ist und es wird true zurückgegeben, wenn dies zutrifft.

Getter:
Gibt die Werte des Zählers oder Nenners zurück

Is-Consistent:
Kontrolliert, ob die Zahl auch valide ist, das heißt dass, keine Rationale Zahl mit (5/0) geschrieben wird.

Positiv/Negativ/Null:
Gibt zurück, ob die Zahl positiv, negativ oder null ist.

Print/Scan/as-string:
As_string schreibt die Rationale Zahl als string, damit man das Ergebnis leichter ausgeben kann-
Print gibt eine Rationale Zahl aus, (auch mit Hilfe von as-string).
Scan kontrolliert, ob die eingegeben Zahl valide ist.

Implementierung:

Siehe Visual Studio:

- Rational_t.h
- Rational_t.cpp
- Rational_test.h
- Rational_test.cpp
- Excpetions.h
- Main.cpp

Testfälle

Allgemeine Anmerkung:
Die Werte können aus den Testfunktionen entnommen werden!

Operationen ohne Normalisieren:

Hier wurde noramlize() in den Methoden auskommentiert

```
Testing normal operations without normalization:
6 / 10
2 / 3
20 / 6
2 / 8
```

Operationen mit Normalisieren

```
Testing normal operations:
3 / 5
2 / 3
10 / 3
1 / 4
```

Überladene Operatoren mit und ohne Assignment

```
Testing overloading operations:
3 / 5
2 / 3
10 / 3
1 / 4


Testing overloading operations with assignment:
3 / 5
2 / 3
10 / 3
1 / 4
```

Operatoren mit Input

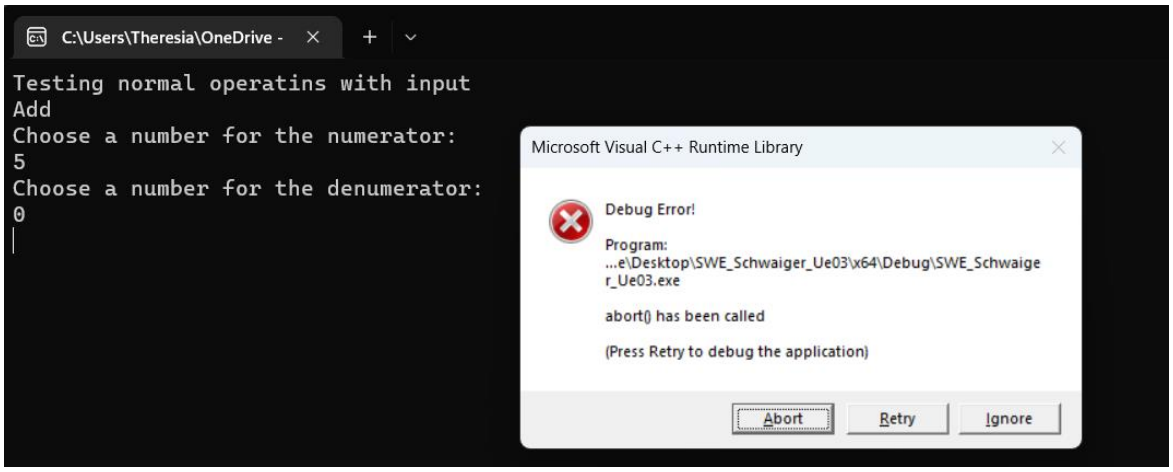
```
Testing normal operatins with input
Add
Choose a number for the numerator:
5
Choose a number for the denumerator:
3
3 / 5
Proceeding with execution...

Sub
Choose a number for the numerator:
4
Choose a number for the denumerator:
2
-1 / 5
Proceeding with execution...
```

```
Mul
Choose a number for the numerator:
6
Choose a number for the denumerator:
2
4 / 1
Proceeding with execution...

Div
Choose a number for the numerator:
4
Choose a number for the denumerator:
2
1 / 1
Proceeding with execution...
```

Zweiter Durchgang:



Vergleichs Operatoren

```
Testing compare operators:

Are the same!
Are the same!
A is smaler than b
A is bigger than b
A is bigger than b
A is smaler than b



 Second round:
Are the same!
Are not the same !
A is smaler than b
A is bigger than b
A is bigger than b
A is bigger than b
```

Vergleichsoperatoren Überladen

Getter

```
Testing getter:
Numerator: 5
Denominator: 6
```

Positiv, Null, Negativ

```
Testing zero, positive, negative:
 This number is negative
This number is positive
 THis number is zero
```

Division by zero