SWE3 HÜ 3: Rechnen mit rationalen Zahlen

Erik Pitzer <erik.pitzer@fh-hagenberg.at>

Medizin- und Bio-Informatik – WS 2025/26

Name

Aufwand (in h)

Punkte

Aufgabe	Lösungsidee	Implementierung	Tests	Gesamtpunkte
1	20% / 20P	50% / 50P	30% / 30P	100

Aufgabe 1: Klasse Rational

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

- Beispiel

```
int main() {
  rational_t r{1, 2};
  std::cout << r * -10 << '\n'
            << r * rational_t(20, 2) << '\n';
  r = 7;
  std::cout << r + rational_t(2, 3) << '\n'
            << 10 / r / 2 + rational_t(6, 5) << '\n';
}
```

- ergibt

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

Beachten Sie diese Vorgaben und Implementierungshinweise:

1. Die Datenkomponenten für Zähler und Nenner sind vom Datentyp int. Am besten Sie verwenden wieder einen re-export für `value_type` ()
2. Bei einer Division durch Null wird eine Ausnahme erzeugt. Erstellen Sie hierfür eine eigene Exception-Klasse. Argumentieren Sie ob es sich dabei um einen `logic_error` oder einen `runtime_error` handelt.
3. Werden vom Anwender der Klasse `rational_t` ungültige Zahlen übergeben, so soll ebenfalls eine passende Ausnahme erzeugt werden.
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 immer wieder zu verifizieren.
5. Schreiben Sie eine private Methode `normalize`, mit deren Hilfe eine rationale Zahl in ihren kanonischen Repräsentanten konvertiert werden kann, wo also wenn möglich gekürzt wird. Verwenden Sie diese Methode in Ihren anderen Methoden.
6. Schreiben Sie eine Methode `as_string`, die eine rationale Zahl als Zeichenkette vom Typ `std::string` liefert. (Verwenden Sie dazu einen `std::ostringstream` wie in der Übung.)
7. Verwenden Sie Referenzen, `const` und `noexcept` so oft wie möglich und sinnvoll. Vergessen Sie nicht auf konstante Methoden.

8. Schreiben Sie die Methoden `get_numerator` und `get_denominator` mit der entsprechenden Semantik.

9. Schreiben Sie die Methoden `is_negative`, `is_positive` und `is_zero` mit der entsprechenden Semantik.

10. 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). Argumentieren Sie ob hier die Verwendung eines `explicit` Konstruktors sinnvoll ist oder nicht.

11. Überladen Sie den Zuweisungsoperator (assignment operator, copy assignment), der bei Selbstzuweisung entsprechend reagiert.

12. Überladen Sie die Vergleichsoperatoren (wählen Sie eine Variante aus):
    - entweder `==`, `!=`, `<`, `<=`, `>` und `>`. Implementieren Sie diese, indem Sie auch Delegation verwenden.
    - oder `==` und `<=>` (C++ 20 wird benötigt)

13. Überladen Sie die Operatoren `+=`, `-=`, `*=` und `/=` (compound assignment operators).

14. Überladen Sie die Operatoren `+`, `-`, `*` und `/`. Implementieren Sie diese, indem Sie Delegation verwenden. Denken Sie daran, dass der linke Operand auch vom Datentyp `int` sein kann. Es soll also möglich sein ganze Zahlen (`int`) mit einem `rational_t` zu verknüpfen. Beispiel: $3 + rational\_t(2, 3)$

15. Überladen Sie die Operatoren `<<` und `>>`, um rationale Zahlen "ganz normal" auf Streams schreiben und von Streams einlesen zu können.

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. (Optional) Bei guter Verwendung des GoogleTest Frameworks erhalten Sie 10 Bonuspunkte.