Name: Robin Berger **Aufwand in h:** 15

Punkte: **Kurzzeichen Tutor/in:**

Beispiel 1 (100 Punkte): Klasse `rational_t` erweitern

Erweitern Sie Ihre Klasse `rational_t` vom vorhergehenden Übungszettel um die folgenden Funktionalitäten:

1. Parametrieren Sie Ihre Klasse und wandeln Sie sie zu einem generischen Datentyp `rational_t<T>` um. `T` ist dabei jener Datentyp, von dem Zähler und Nenner sind. Der Defaultwert von `T` ist der Datentyp `int`.
2. Implementieren Sie Ihre Klasse `rational_t<T>` so, dass man damit nicht nur rationale Zahlen über $\mathbb{Z}$ sondern über beliebige Bereiche bilden kann. Implementieren Sie zu Testzwecken einen Datentyp `number_t<T>` und bilden Sie damit rationale Zahlen vom Typ `rational_t<number_t<T>>`.
3. Überlegen Sie, welche Operationen der Datentyp `T` unterstützen muss, damit dieser von Ihrer Klasse `rational_t<T>` verwendet werden kann. Listen Sie diese Anforderungen explizit in der Dokumentation auf. Erstellen Sie in Folge auf Grundlage dieser Anforderungen ein C++ *Concept* `numeric` und passen Sie die Klasse `rational_t<T>` an, sodass dieses *Concept* die Anforderungen an den Typparameter `T` festlegt.
4. Erstellen Sie nun die Klasse number_t<T> und tragen Sie Sorge dafür, dass sämtliche Anforderungen des *Concepts* `numeric` unterstützt werden. Beschränken Sie sich bei den Tests der Klasse `number_t<T>` auf `rational_t<number_t<int>>`
5. Schreiben Sie die Klasse `rational_t<T>` so, dass sie möglichst wenig Vorgaben an den Datentyp `T` stellt. Erstellen Sie eine Datei `operations.h`, die im Namensraum `ops` die folgenden Funktionen implementiert:

```
T abs (T const & a);

bool divides (T const & a, T const & b);

bool equals (T const & a, T const & b);

T gcd (T a, T b);

bool is_negative (T const & a);

bool is_zero (T const & a);

T negate (T const & a);

T remainder (T const & a, T const & b);
```

6. Definieren Sie überdies in der Datei `operations.h` im Namensraum `nelms` die benötigten Funktionen für die Bildung *neutraler Elemente* und verwenden Sie diese an entsprechenden Stellen in Ihrer Lösung.
7. Obige Funktionen sind generisch (Typvariable `T`) zu implementieren sowie inline auszuführen. Spezialisieren Sie außerdem alle Funktionen für den Datentyp int. Ihre Klasse `rational_t<T>` verwendet natürlich alle angegebenen Funktionen.
8. Implementieren Sie eine Methode `inverse`, die eine rationale Zahl durch ihren Kehrwert ersetzt.

9. Die Klassen `rational_t<T>` und `number_t<T>` implementieren ihre Operatoren inline als zweistellige friend-Funktionen („Barton-Nackman Trick“).

Bitte beachten Sie: Testfälle sind ein Musskriterium bei der Punktevergabe. Enthält eine Ausarbeitung keine Testfälle, so werden dafür auch keine Punkte vergeben. Testfälle sind auf die entsprechenden Teilaufgaben zu beziehen. Es muss aus der Testfallausgabe klar ersichtlich sein, auf welche Teilaufgabe sich ein Testfall bezieht.

Die Testfälle sind entsprechend den Teilaufgaben laut Angabe zu reihen. Ein Testfall schreibt die folgenden Informationen aus: Testfallname, was wird getestet (Bezug zur Angabe), erwarteter Output, tatsächlicher Output, Test erfolgreich/nicht erfolgreich. Ist ein Test nicht erfolgreich, so kann eine Beschreibung der vermuteten Fehlerursache bzw. der durchgeführten Fehlersuche doch noch Punkte bringen.

50% der Punkte für dieses Beispiel entfallen auf die Testfälle!

Hinweis: Achten Sie darauf, dass für die Unterstützung von *Concepts*, der Sprachlevel Ihrer Entwicklungsumgebung gegebenenfalls auf **C++20** angepasst werden muss.

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:

Allgemein rational_type

Die template Klasse rational_t ermöglich das Rechnen mit Objekten, die Bruchzahlen repräsentieren, das Vereinfachen (Kürzen) dieser, sowie sämtliche Operationen wie die 4 Grundrechnungsarten, Zuweisungen und logische Operationen. Zähler und Nenner können von unterschiedlichen Datentypen sein – Vorraussetzung ist das, dass der Datentyp den Kriterien des Concepts NumericType entspricht.

Concept NumericType & Number.h

Das Concept NumericType definiert, welche Operatoren für den übergebenen Datentyp überladen sein müssen. Enthalten müssen in diesem Fall sein:

- Operatoren der Grundrechnungsarten (+, -, *, /)
- Operatoren Assign/Grundrechnungsarten (+=, -=, *=, /=)
- Alle logischen Operatoren (==, !=, <, >, <=, >=)
- Stream Operator (<<) -> Ausgaben von Objekten
- Modulo Operator (%)
    - Dieser wird in der ops:gcd methode benötigt, daher ist er auch als Voraussetzung enthalten. Dies hat allerdings zur Folge, dass rational_t<double> beispielsweise nicht erlaubt wird, da für einen double grundsätzlich die Modulo Operation nicht definiert ist.

All diese Operationen sind in Form von friend inline Funktionen im File Number.h definiert.

Friend Deklarationen

Die Operatoren für rational_t Objekte werden inline als zweistellige friend-Funktionen implementiert. Durch das Schlüsselwort inline wird die Funktion direkt innerhalb der Defintion ausgeführt und overhead für den gewöhnlichen Funktionsaufruf kann verhindert werden.

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.

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_

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 function

Diese Methode verwendet in diesem Programm nicht mehr die Funktion as_string, da für die Methode as_string std::to_string benötigt werden würde, diese Standardfunktion für eigene Datentypen (Number<T>) nicht überladen ist. Es werden Zähler und Nenner als Zahlen direkt auf den Stream geschrieben.

Operations & nelms

Im Namespace nelms werden die Null- und Einserelemente der verschiedenen Datentypen definiert. Im Fall von einen int sind das 0 und 1, im Falle eines doubles sind das 0.0 bzw. 1.0. Der Namespace nelms wird zudem in Number.h um die Null- und Einserelemente der Datentypen Number<int> erweitert.

Im Namespace ops sind viele Operationen definiert, die an verschiedenen Stellen im Code Anwendung finden. Beispiele dafür sind etwa gcd (größter gemeinsamer Teiler) oder die Prädikate is_zero oder is_negative. Diese verwenden als Referenz für 0 das jeweilige 0er Element des Datentyps aus nelms.

Testfälle:

1.) Standardvalue int

```
template<NumericType T = int>
class complex {
private:
```

2.) Klasse Number<T> void print_numbers()

```
Printing numbers ...
42
21.42
72
```

3.) Siehe Lösungsidee

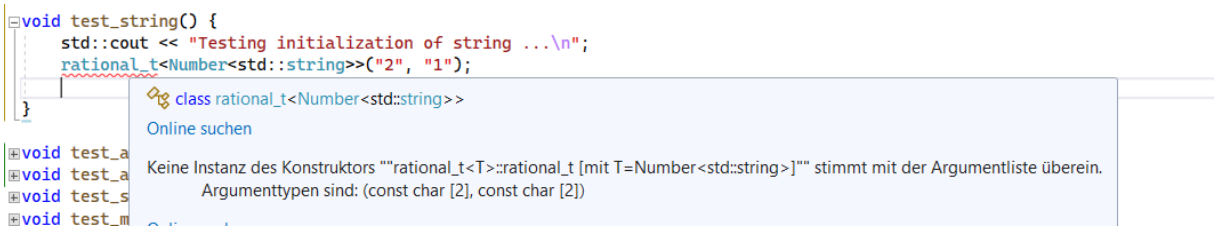
4, 5, 6, 7 implizit in anderen Testfällen

8.) Funktion inverse

```
Testing method invers ...
<3/4>
<4/3>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung04\SWE_Berge
 (Prozess "1116") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Test_string()



10) Testfälle Operatoren/Grundrechnungsarten gesamt

Test_all_calculations():

```
Testing all operation methods:

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

Subtraction:
<1/2>
<-5/1>
<1/-12>

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\Übung04\SWE_Berger_Ue04\x64\Debug\
 (Prozess "13120") 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:
<1/2>
<-5/1>
<1/-12>

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\Übung04\SWE_Berger_Ue04\x64\
 (Prozess "19164") 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:
<1/2>
<-5/1>
<1/-12>

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\Übung04\SWE_Berger_Ue04\x64\Debug
 (Prozess "32908") 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<10/8>

Testing unequal (!=):
a<5/4> is not the same as b<3/2>
a<5/4> is the same as c<10/8>

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<4/6>
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<4/6>
a<2/3> is bigger or equal c<1/8>

C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung04\SWE_Berger_Ue04\x64\Debug\S
 (Prozess "9580") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

test_divide_by_zero()

```
Processing with execution ...


C:\Users\robin\OneDrive - FH OOe\Dokumente\3. Semester\SWE\Übungen\Übung04\SWE_Berger_Ue04\x64\D
 (Prozess "29984") wurde mit Code "0" beendet.
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```