

Name: Sandra StraubeAufwand in h: 10h

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.

### Inhaltsverzeichnis

Beispiel 1 Klasse rational_t erweitern: .....	1
Lösungsidee.....	1
Testfälle .....	5
Teil 1 - .....	5
Fall 1.....	5
Fall 2.....	5
Fall 3.....	6
Fall 4.....	6
Fall 5.....	6
Fall 6.....	6
Fall 7.....	6
Fall 8.....	7
Fall 9.....	7
Fall 10.....	7
Fall 11.....	8
Fall 12.....	9
Fall 13.....	9
Fall 14.....	10
Teil 2 - .....	11
Fall 1.....	11
Teil 3 - .....	12
Fall 1.....	12
Teil 4 - .....	13
Fall 1.....	13
Fall 2.....	13

Fall 3.....	13
Fall 4.....	13
Fall 5.....	14
Fall 6.....	14
Fall 7.....	14
Fall 8.....	14
Fall 9.....	15
Fall 10.....	16
Fall 11.....	17
Fall 12.....	17
Fall 13.....	18
Teil 5 - .....	19
Fall 1.....	19
Fall 2.....	19
Fall 3.....	19
Fall 4.....	20
Fall 5.....	20
Fall 6.....	20
Fall 7.....	20
Fall 8.....	21
Teil 6 - .....	22
Fall 1.....	22
Fall 2.....	22
Teil 7 - .....	23
Fall 1.....	23
Fall 2.....	23

Fall 3.....	23
Fall 4.....	23
Fall 5.....	23
Fall 6.....	24
Fall 7.....	24
Fall 8.....	24
Fall 9.....	24
Fall 10.....	25
Teil 8 - .....	26
Fall 1.....	26
Fall 2.....	26
Teil 9 - .....	27
Fall 1.....	27
Fall 2.....	27
Fall 3.....	28
Fall 4.....	28
Fall 5.....	29
Fall 6.....	29
Fall 7.....	29
Fall 8.....	30
Fall 9.....	30
Fall 10.....	30
Fall 11.....	31
Fall 12.....	31
Fall 13.....	31
Fall 14.....	32

Fall 15.....	32
Fall 16.....	32
Fall 17.....	32
Fall 18.....	33
Fall 19.....	33
Fall 20.....	33
Fall 21.....	34
Fall 22.....	34
Fall 23.....	34
Fall 24.....	35
Fall 25.....	35
Fall 26.....	35
Fall 27.....	36
Fall 28.....	36
Fall 29.....	36
Fall 30.....	37
Fall 31.....	37
Fall 32.....	37
Fall 33.....	38
Fall 34.....	38

## Beispiel 1 Klasse `rational_t` erweitern:

### Lösungsidee

Die Klasse `rational_t` aus der vorherigen Aufgabe ist als generisches Template umzuarbeiten, wodurch zwar als Default der Datentyp `int`, aber auch andere Datentypen theoretisch unterstützt werden können. Um dieses Verhalten zu testen, soll zusätzlich die Klasse `number_t` implementiert und als Template-Datentyp/Template-Klasse verwendet werden. Voraussetzung hier ist u.a. ein Defaultkonstruktor und ein Kopierkonstruktor.

Bei der Umsetzung von `rational_t` sollen entsprechend alle Methoden und Eigenschaften angepasst werden für den generischen Datentyp `T`. Da hier z.B. `scan()` aufgrund der Verwendung von `to_string()` nichtmehr funktioniert, ist die Möglichkeit entweder das Überladen der Klasse oder eine alternative Funktion, die allgemeiner angewandt werden kann. Hinweis: Bei der Ausarbeitung habe ich mich für letzteres entschieden, um die Klasse allgemeiner zu halten. Da `scan()` mit `to_string()` jedoch Bestandteil der letzten Aufgabe war, ist diese Methode dennoch vorhanden, für `print()` jedoch wurde sie mit `as_ostream()` ersetzt.

Da ein generisches Template an dieser Stelle nun mehrere Datentypen zulässt, kann dies an bestimmten Stellen im Code Fehler auswerfen. Dies kann nun durch Spezialisierung/Überladen der Klasse behoben werden oder indem man Vorlagen an diese Datentypen übergibt. Letzteres lässt sich durch Concepts realisieren. Für die Klasse `rational_t` sollen die folgenden Anforderungen an den Datentyp `T` gestellt werden:

```
concept NumericType = requires(S t) {  
    t + t;  
    t - t;  
    t * t;  
    t / t;  
    t += t;  
    t -= t;  
    t *= t;  
    t /= t;  
    t == t;  
    t != t;  
    t < t;  
    t <= t;  
    t > t;  
    t >= t;  
    cout << t;  
    cin >> t;  
};
```

```
t % t; // No support for double here.  
t = t;  
};
```

Die Anforderungen sollen sämtliche verwendeten und unterstützten Operatoren umfassen, sowie ein ostream und istream mittels cout und cin möglich sein. Zudem soll es möglich sein für die Normalisierung und die Berechnung des größten gemeinsamen Teilers das Modulo für T zu berechnen. Falls ein Datentyp die Vorlagen nicht erfüllt, wird ein Compilerfehler ausgegeben, den man nicht mittels Try & Catch-Block abfangen kann. Hinweis: Dies ist der Grund, weswegen dieser Test für Teilaufgabe 3 im Code auskommentiert wurde.

Die Klasse `number_t<T>` soll nun sämtliche Concepts von `rational_t` erfüllen. Dazu sollen sämtliche erforderlichen Operatoren für `number_t` in der Klasse überladen werden. Diese Überladungen sollen relativ simpel sein und auf den Datentypen T angewandt werden, welcher in diesem Fall für sämtliche Tests ein Integer ist.

Natürlich soll T dennoch als genereller Datentyp behandelt werden.

Als nächstes sollen diverse Operationen angelegt werden in einem separaten Namensraum, auf das u.a. mit `rational_t` und `number_t` zugegriffen werden kann.

Diese Operationen sollen umfassen:

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

Dies soll den absoluten Betrag von T ermitteln und zurückgeben. Der absolute Betrag ist immer der positive Betrag. Hinweis: Diese Methode ist nicht in `rational_t` verwendet worden. Der Grund ist, dass dafür die gesamte Methode `normalize()` umgeschrieben werden müsste und aufgebläht, nur um diese Methode aufzurufen. Die Möglichkeit zur Anwendung wurde jedoch im Code markiert.

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

Es soll ermittelt werden, ob Ta sich durch T b teilen lässt und ein entsprechender Wahrheitswert zurückgegeben werden. Diese Methode soll remainder verwenden. Hinweis: `divides()` wird indirekt bereits bei der Methode `ops::gcd()` angewendet, jedoch hätte diese Methode stark umgeschrieben und aufgebläht werden müssen, nur um diese eine Methode mit der gleichen Funktionalität aufzurufen.



- `bool equals (T const & a, T const & b);`  
Hier soll ermittelt werden, ob T a und T b gleich sind und ein entsprechender Wahrheitswert ausgegeben werden.
- `T gcd (T a, T b);`  
Zur Ermittlung des größten gemeinsamen Teilers für die Funktion `normalize(). S. divides()`.
- `bool is_negative (T const & a);`  
Diese Methode soll true zurückgeben, wenn der übergebene Wert vom Datentyp her negativ ist.
- `bool is_zero (T const & a);`  
Hier soll true wiedergegeben werden, wenn der übergebene Wert vom Datentyp her neutral Null ist
- `T negate (T const & a);`  
Mit dieser Funktion soll der Wert a negiert werden: Positiv zu negativ und negativ zu positiv. Neutral 0 ist hier extra zu beachten.
- `T remainder (T const & a, T const & b);`  
Diese Methode soll von zwei Werten den Rest wiedergeben. Diese Operation soll in `divides()` angewendet werden.

Die angegebenen Operationen aus dem Namensraum `ops::` sollen zudem für Integer spezialisiert werden. Es soll nicht extra für `number_t` oder `rational_t` spezialisiert werden, da hier von Default ausgegangen wird und durch die Verarbeitung von `rational_t` der eigentliche Datentyp T an `ops::` weitergegeben wird. Hinweis: Default funktioniert in meinem Fall auch für Integer, weshalb im Grunde die Spezialisierung für Integer nicht notwendig ist, jedoch ist dies im Gegensatz zur Spezialisierung für `number_t` explizit gewünscht.

Neben dem Namensraum `ops::` soll zudem ein Namensraum `nelms::` erstellt werden und Methoden beinhalten, welche die neutralen Null- und Eins-Elemente des jeweiligen Datentyps ermitteln. Diese sollen ebenfalls für den Datentyp Integer spezialisiert werden. Hier soll ebenfalls für den Datentyp `number_t<T>` spezialisiert werden, da sich hier der Wert von Default unterscheiden soll. Für `rational_t<T>` ist keine Spezialisierung für die Funktion der Template-Klasse notwendig.

Für die Klasse `rational_t<T>` soll zudem noch eine neue Methode `inverse()` erstellt werden, welche den Kehrwert der rationalen Zahl ermittelt und für das Objekt ersetzt. Hierbei ist darauf zu achten, dass kein Bruch /zero möglich ist, in solch einem Fall soll eine Exception ausgegeben werden.

Für die Operatoren in den Klassen `rational_t` und `number_t` sollen die Operatoren inline mithilfe des Barton-Nackman Tricks implementiert werden. Operatoren wurden in der vorherigen Aufgabe als Member- und Non-Member implementiert und überladen, um links- und rechtsseitige Operationen zu ermöglichen. Die Non-Member wurden dazu zusätzlich als friend eingefügt, damit diese Zugriff auf Eigenschaften und Methoden der Klasse haben. Da die Klasse nun jedoch als Template umgebaut wurde, ist dies nichtmehr möglich, denn der Non-Member Operator kann nicht als Template implementiert werden. Der Barton-Nachman Trick hilft hier aus, indem die Member zu inline friends umgebaut werden. Dies lässt nun links- und rechtsseitige Operationen zu, sowie Zugriff auf die Klasse mit Methoden und die Operatoren werden generalisiert für die Klasse `rational_t<T>` und `number_t<T>`. Inline weist den Compiler an, den eigentlichen Code innerhalb der Funktion für jeden Funktionsaufruf zu setzen. Dies kann die Laufzeit des Programms verkürzen, da so weiterer Aufwand für den eigentlichen Funktionsaufruf vermieden wird. Dieser Trick lässt sich allerdings auf bestimmte Operatoren, wie den Assignment Operator, nicht anwenden, da diese echte Member der Klasse sein müssen.

Es ist generell darauf zu achten, dass möglichst generell geschrieben wird, wobei die Funktionalitäten von `rational_t` nicht zerstört werden darf. Es sollen theoretisch andere Datentypen möglich sein bzw. die Möglichkeit gegeben werden die entsprechenden Klassen und Methoden für den Datentypen zu überladen. Es ist wichtig, dass rationale Zahlen weiterhin nicht durch 0 dividiert werden können, durch die neue Implementierung sollen Vorkommen von Integer 0 und 1 mit den relativen neutralen Null- und Eins-Werten aus `nelms::` ersetzt werden. Ebenso sind die Operationen aus `ops::` möglichst großflächig zu verwenden und Methoden, die spezielle Anforderungen an den Datentypen T haben zu vermeiden oder zu spezialisieren oder mittels der Concepts vorauszusetzen.

## Testfälle

Folgend gibt es nicht für jeden Testfall eine eigene Erklärung, da davon ausgegangen wird, dass sich die Bedeutung des Tests in diesem Falle aus dem Methodennamen ergibt. Hinweis: `_nt` im Methodennamen deutet auf einen Test mit `Number_t<int>` hin.

### Teil 1 - test\_nr\_one

Folgende Tests beziehen sich auf Teilaufgabe 1 und testen die Funktionalität nach der erfolgreichen Konvertierung von `rational_t` auf einen generischen Datentypen `T`.

#### Fall 1: test\_constructor

Test erfolgreich

Hierbei werden die Konstruktoren mit und ohne übergebenen Datentyp getestet.

Test auf Konstruktion auf:

- Default Fallback
- negative Brüche
- leer
- einzelner Integer
- Copy constructor:
- Eingabe invalider Bruch `/0`.

```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_one:
----- 1) Test generic datatype rational_t<T> mit T = int.
--- test_constructor:
- Input: -1/2 without datatype int; Output (Expect -1/2): <-(1/2)>
- Input: -1/2; Output (Expect -1/2): <-(1/2)>
- Input: ''; Output (Expect 0): <0>
- Input: 7; Output (Expect 7): <7>
- Input: r(-1/2); Output (Expect -1/2): <-(1/2)>
- Input: 1/0; Output (Expect Error):
Error: No valid fracture - division by 0.
```

#### Fall 2: test\_as\_string

Test erfolgreich

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

## Fall 3: test\_print

Test erfolgreich

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

## Fall 4: test\_scan

Test erfolgreich

Test scan auf

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

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

## Fall 5: test\_get\_numerator

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_get_numerator:
- Input: 11/15; Output (Expect 11): 11
```

## Fall 6: test\_get\_denominator

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_get_denominator:
- Input: 11/15; Output (Expect 15): 15
```

## Fall 7: test\_is\_negative

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_is_negative:
- Input: 11/15; Output (Expect false): false
- Input: -11/15; Output (Expect true): true
```

## Fall 8: test\_is\_positive

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_is_positive:
- Input: 11/15; Output (Expect true): true
- Input: -11/15; Output (Expect false): false
```

## Fall 9: test\_normalize

Test erfolgreich

Test normalize auf

- $n > d$
- $n < d$
- negative denominator
- negative numerator and negative denominator.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_normalize:
- Input: 3/15; Output (Expect 1/5): <1/5>
- Input: 42/28; Output (Expect 3/2): <3/2>
- Input: 1/-3; Output (Expect -1/3): <-(1/3)>
- Input: -1/-3; Output (Expect 1/3): <1/3>
```

## Fall 10: test\_add

Test erfolgreich

Test add() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0
- ganzzahlige „Brüche“.

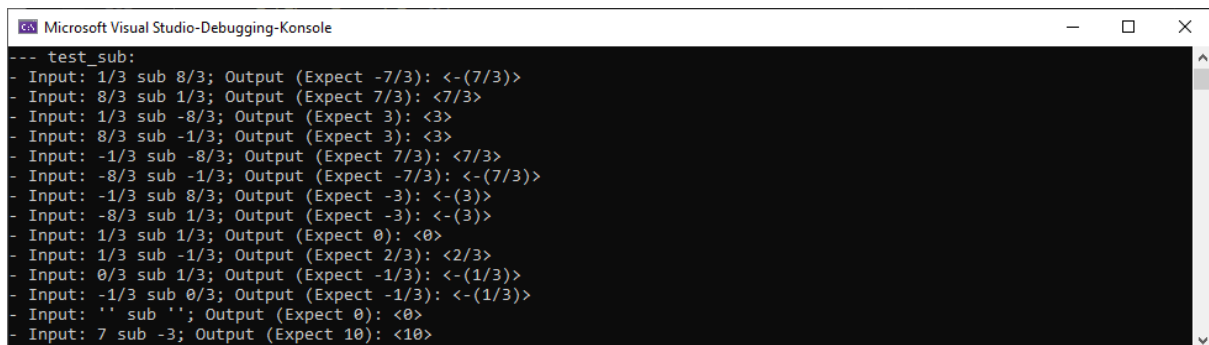
```
Microsoft Visual Studio-Debugging-Konsole
--- test_add:
- Input: 4/3 add 1/3; Output (Expect 5/3): <5/3>
- Input: -4/3 add -1/3; Output (Expect -5/3): <-(5/3)>
- Input: 1/3 add -4/3; Output (Expect -1): <-(1)>
- Input: -1/3 add 1/3; Output (Expect 0): <-(0)>
- Input: 3/10 add 4/5; Output (Expect 11/10): <11/10>
- Input: '' add ''; Output (Expect 0): <0>
- Input: 7 add -3; Output (Expect 4): <4>
```

Fall 11: test\_sub

Test erfolgreich

Test sub() auf

- $+r < +r2$
- $+r > +r2$
- $+r < -r2$
- $+r > -r2$
- $-r < -r2$
- $-r > -r2$
- $-r < +r2$
- $-r > +r2$
- $+r == +r2$
- $+r == -r2$
- lhs = 0
- rhs = 0
- beide Brüche 0
- ganzzahlige "Brüche".



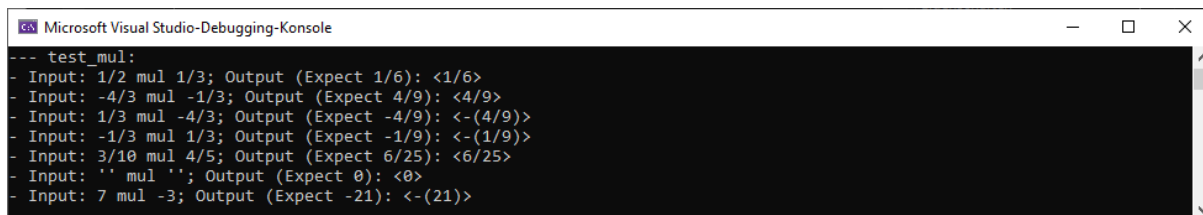
```
Microsoft Visual Studio-Debugging-Konsole
--- test_sub:
- Input: 1/3 sub 8/3; Output (Expect -7/3): <-(7/3)>
- Input: 8/3 sub 1/3; Output (Expect 7/3): <7/3>
- Input: 1/3 sub -8/3; Output (Expect 3): <3>
- Input: 8/3 sub -1/3; Output (Expect 3): <3>
- Input: -1/3 sub -8/3; Output (Expect 7/3): <7/3>
- Input: -8/3 sub -1/3; Output (Expect -7/3): <-(7/3)>
- Input: -1/3 sub 8/3; Output (Expect -3): <-(3)>
- Input: -8/3 sub 1/3; Output (Expect -3): <-(3)>
- Input: 1/3 sub 1/3; Output (Expect 0): <0>
- Input: 1/3 sub -1/3; Output (Expect 2/3): <2/3>
- Input: 0/3 sub 1/3; Output (Expect -1/3): <-(1/3)>
- Input: -1/3 sub 0/3; Output (Expect -1/3): <-(1/3)>
- Input: '' sub ''; Output (Expect 0): <0>
- Input: 7 sub -3; Output (Expect 10): <10>
```

## Fall 12: test\_mul

Test erfolgreich

Test mul() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0
- ganzzahlige „Brüche“.



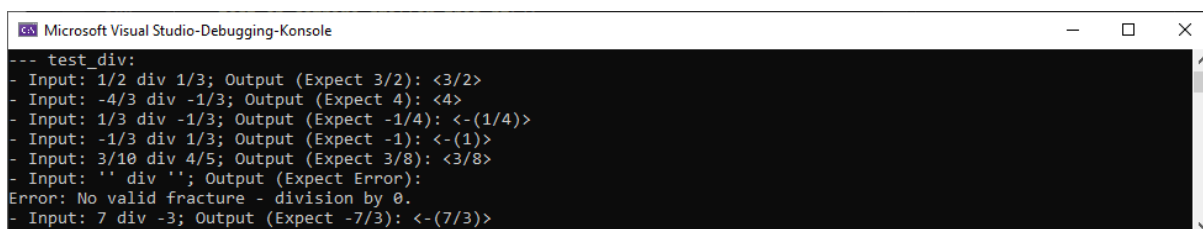
```
Microsoft Visual Studio-Debugging-Konsole
--- test_mul:
- Input: 1/2 mul 1/3; Output (Expect 1/6): <1/6>
- Input: -4/3 mul -1/3; Output (Expect 4/9): <4/9>
- Input: 1/3 mul -4/3; Output (Expect -4/9): <-(4/9)>
- Input: -1/3 mul 1/3; Output (Expect -1/9): <-(1/9)>
- Input: 3/10 mul 4/5; Output (Expect 6/25): <6/25>
- Input: '' mul ''; Output (Expect 0): <0>
- Input: 7 mul -3; Output (Expect -21): <-(21)>
```

## Fall 13: test\_div

Test erfolgreich

Test div() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0 / Teilung durch 0
- ganzzahlige „Brüche“.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_div:
- Input: 1/2 div 1/3; Output (Expect 3/2): <3/2>
- Input: -4/3 div -1/3; Output (Expect 4): <4>
- Input: 1/3 div -1/3; Output (Expect -1/4): <-(1/4)>
- Input: -1/3 div 1/3; Output (Expect -1): <-(1)>
- Input: 3/10 div 4/5; Output (Expect 3/8): <3/8>
- Input: '' div ''; Output (Expect Error):
Error: No valid fracture - division by 0.
- Input: 7 div -3; Output (Expect -7/3): <-(7/3)>
```

Fall 14: test\_custom

Test erfolgreich

Test auf gemischte Operatoren.

Input:

```
Rational_t<int> r(-1, 2) * 10
```

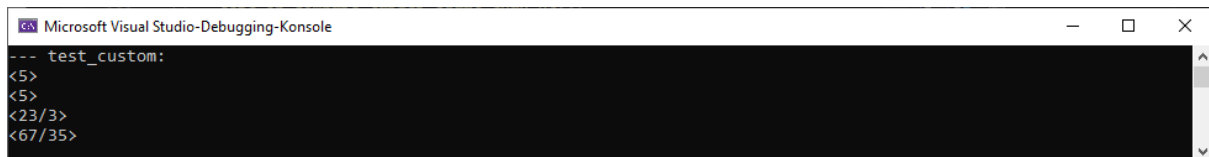
```
Rational_t<int> r(-1, 2) * Rational_t(20, -2)
```

```
r = 7
```

```
r + Rational_t(2, 3)
```

```
10 / r / 2 + Rational_t(6, 5)
```

Output: (Erwartet: 5, 5, 23/3, 67/35)



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



## Teil 2 - test\_nr\_two

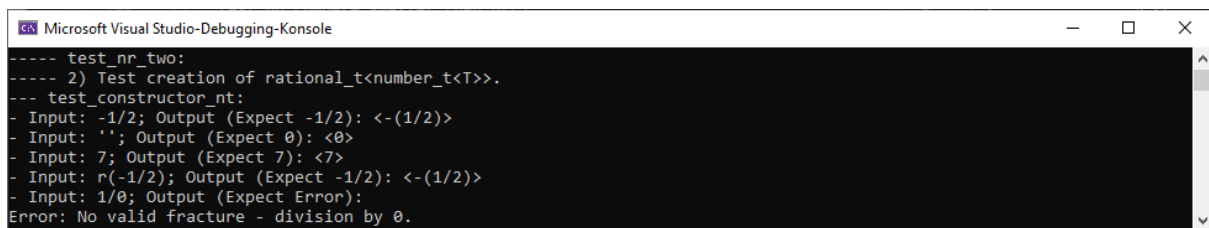
Test auf Konstruktion `rational_t<number_t<T>>`. `Number_t` wurde hier als Klasse als Datentyp implementiert. Die Aufgabenstellung lässt Überschneidungen mit Aufgabe 4 zu, weitere Tests sind dort angegeben.

## Fall 1: test\_constructor\_nt

Test erfolgreich

Test auf Konstruktion von

- negative Brüchen
- leer
- einzelner Integer von `Number_t`
- Copy constructor:
- Eingabe invalider Bruch `/0`.

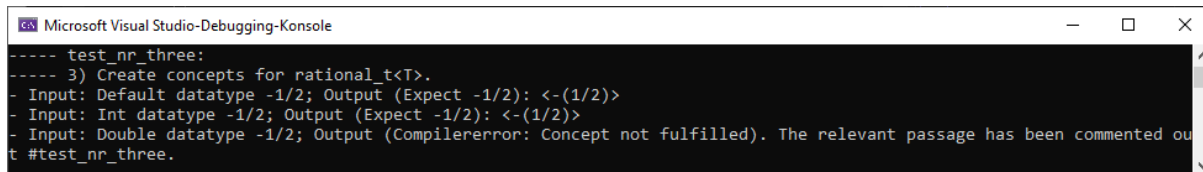


```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_two:
----- 2) Test creation of rational_t<number_t<T>>.
--- test_constructor_nt:
- Input: -1/2; Output (Expect -1/2): <-1/2>
- Input: ''; Output (Expect 0): <0>
- Input: 7; Output (Expect 7): <7>
- Input: r(-1/2); Output (Expect -1/2): <-1/2>
- Input: 1/0; Output (Expect Error):
Error: No valid fracture - division by 0.
```

## Teil 3 - test\_nr\_three

Erstellung von Concept von rational\_t und Test auf default(int) und int. Ebenfalls Test auf double, da dies jedoch ein Compilerfehler auswirft, ist diese Stelle auskommentiert zur Dokumentation im Code. Double wirft einen Fehler, da Modulo nicht unterstützt wird.

## Fall 1: Test erfolgreich



```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_three:
----- 3) Create concepts for rational_t<T>.
- Input: Default datatype -1/2; Output (Expect -1/2): <-(1/2)>
- Input: Int datatype -1/2; Output (Expect -1/2): <-(1/2)>
- Input: Double datatype -1/2; Output (Compilererror: Concept not fulfilled). The relevant passage has been commented out
t #test_nr_three.
```

## Teil 4 - test\_nr\_four

Test auf Funktionalität Methoden von `rational_t` mit `number_t<int>` unter Berücksichtigung der Concepts. Durch Überschneidungen mit Aufgabe 2 ist hier der Konstruktor nicht nochmal extra getestet bzw. nur indirekt.

## Fall 1: test\_as\_string\_nt

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_four:
----- 4) Test concepts and methods for rational_t<T> with number_t<T>. Please also see test_nr_two for constructors and
test_nr_nine for operators.
--- test_as_string_nt:
- Input: -1/2; Output (Expect -1/2): <-(1/2)>
```

## Fall 2: test\_print\_nt

Test erfolgreich

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

## Fall 3: test\_scan\_nt

Test erfolgreich

Test scan auf

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

```
Microsoft Visual Studio-Debugging-Konsole
--- test_scan_nt:
User Input (3/4):
3/4
- Output (Expect user input): <3/4>
User Input (3 4 then 3/4):
3 4
3/4
- Output (Expect redo): <3/4>
User Input (3 then 3/1):
3
3/1
- Output (Expect redo): <3>
User Input (3/1):
3/1
- Output (Expect user input): <3>
```

## Fall 4: test\_get\_numerator\_nt

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_get_numerator_nt:
- Input: 11/15; Output (Expect 11): 11
```

## Fall 5: test\_get\_denominator\_nt

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_get_denominator_nt:
- Input: 11/15; Output (Expect 15): 15
```

## Fall 6: test\_is\_negative\_nt

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_is_negative_nt:
- Input: 11/15; Output (Expect false): false
- Input: -11/15; Output (Expect true): true
```

## Fall 7: test\_is\_positive\_nt

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_is_positive_nt:
- Input: 11/15; Output (Expect true): true
- Input: -11/15; Output (Expect false): false
```

## Fall 8: test\_normalize\_nt

Test erfolgreich

Test normalize auf

- $n > d$
- $n < d$
- negative denominator
- negative numerator and negative denominator.

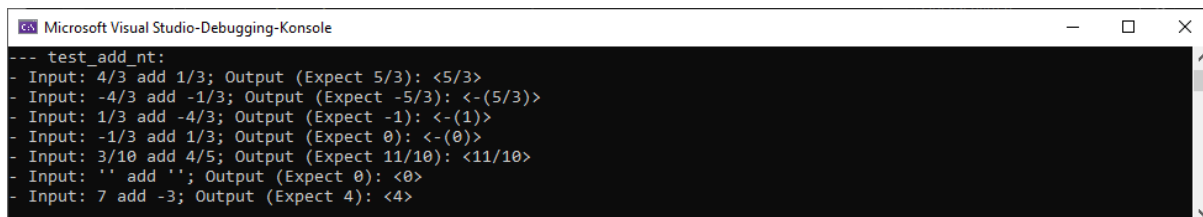
```
Microsoft Visual Studio-Debugging-Konsole
--- test_normalize_nt:
- Input: 3/15; Output (Expect 1/5): <1/5>
- Input: 42/28; Output (Expect 3/2): <3/2>
- Input: 1/-3; Output (Expect -1/3): <-(1/3)>
- Input: -1/-3; Output (Expect 1/3): <1/3>
```

## Fall 9: test\_add\_nt

Test erfolgreich

Test add() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0
- ganzzahlige „Brüche“.



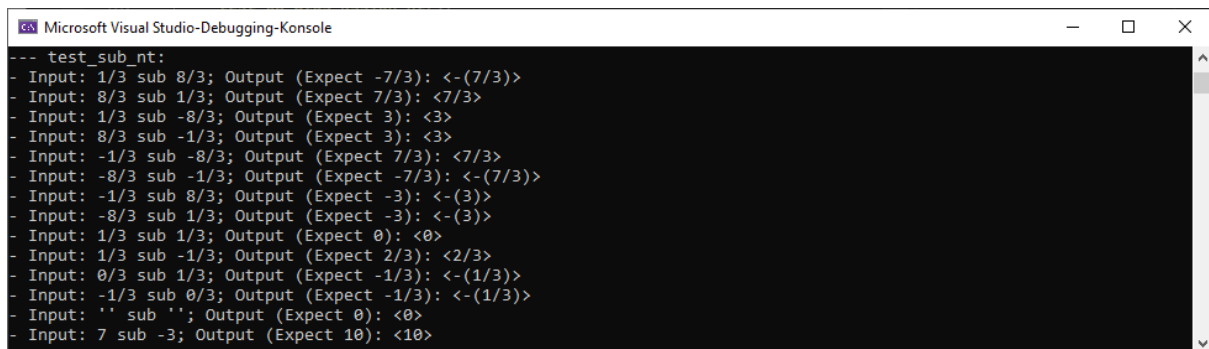
```
Microsoft Visual Studio-Debugging-Konsole
--- test_add_nt:
- Input: 4/3 add 1/3; Output (Expect 5/3): <5/3>
- Input: -4/3 add -1/3; Output (Expect -5/3): <-(5/3)>
- Input: 1/3 add -4/3; Output (Expect -1): <-(1)>
- Input: -1/3 add 1/3; Output (Expect 0): <-(0)>
- Input: 3/10 add 4/5; Output (Expect 11/10): <11/10>
- Input: '' add ''; Output (Expect 0): <0>
- Input: 7 add -3; Output (Expect 4): <4>
```

## Fall 10: test\_sub\_nt

Test erfolgreich

Test sub() auf

- $+r < +r2$
- $+r > +r2$
- $+r < -r2$
- $+r > -r2$
- $-r < -r2$
- $-r > -r2$
- $-r < +r2$
- $-r > +r2$
- $+r == +r2$
- $+r == -r2$
- lhs = 0
- rhs = 0
- beide Brüche 0
- ganzzahlige "Brüche".



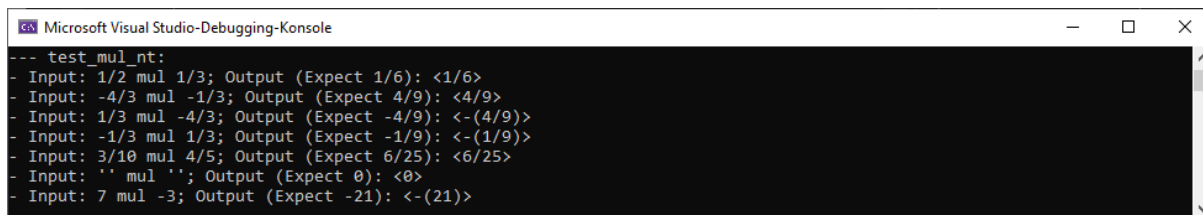
```
Microsoft Visual Studio-Debugging-Konsole
--- test_sub_nt:
- Input: 1/3 sub 8/3; Output (Expect -7/3): <-(7/3)>
- Input: 8/3 sub 1/3; Output (Expect 7/3): <7/3>
- Input: 1/3 sub -8/3; Output (Expect 3): <3>
- Input: 8/3 sub -1/3; Output (Expect 3): <3>
- Input: -1/3 sub -8/3; Output (Expect 7/3): <7/3>
- Input: -8/3 sub -1/3; Output (Expect -7/3): <-(7/3)>
- Input: -1/3 sub 8/3; Output (Expect -3): <-(3)>
- Input: -8/3 sub 1/3; Output (Expect -3): <-(3)>
- Input: 1/3 sub 1/3; Output (Expect 0): <0>
- Input: 1/3 sub -1/3; Output (Expect 2/3): <2/3>
- Input: 0/3 sub 1/3; Output (Expect -1/3): <-(1/3)>
- Input: -1/3 sub 0/3; Output (Expect -1/3): <-(1/3)>
- Input: '' sub ''; Output (Expect 0): <0>
- Input: 7 sub -3; Output (Expect 10): <10>
```

## Fall 11: test\_mul\_nt

Test erfolgreich

Test mul() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0
- ganzzahlige „Brüche“.



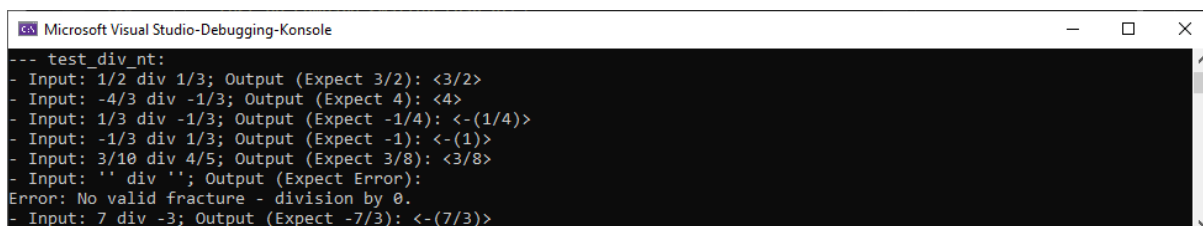
```
Microsoft Visual Studio-Debugging-Konsole
--- test_mul_nt:
- Input: 1/2 mul 1/3; Output (Expect 1/6): <1/6>
- Input: -4/3 mul -1/3; Output (Expect 4/9): <4/9>
- Input: 1/3 mul -4/3; Output (Expect -4/9): <-(4/9)>
- Input: -1/3 mul 1/3; Output (Expect -1/9): <-(1/9)>
- Input: 3/10 mul 4/5; Output (Expect 6/25): <6/25>
- Input: '' mul ''; Output (Expect 0): <0>
- Input: 7 mul -3; Output (Expect -21): <-(21)>
```

## Fall 12: test\_div\_nt

Test erfolgreich

Test div() auf

- positive Brüche
- negative Brüche
- negativer Bruch rhs
- negativer Bruch lhs
- unterschiedliche denominator
- beide Brüche 0 / Teilung durch 0
- ganzzahlige „Brüche“.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_div_nt:
- Input: 1/2 div 1/3; Output (Expect 3/2): <3/2>
- Input: -4/3 div -1/3; Output (Expect 4): <4>
- Input: 1/3 div -1/3; Output (Expect -1/4): <-(1/4)>
- Input: -1/3 div 1/3; Output (Expect -1): <-(1)>
- Input: 3/10 div 4/5; Output (Expect 3/8): <3/8>
- Input: '' div ''; Output (Expect Error):
Error: No valid fracture - division by 0.
- Input: 7 div -3; Output (Expect -7/3): <-(7/3)>
```

Fall 13: test\_custom\_nt

Test erfolgreich

Test auf gemischte Operatoren.

Input:

```
Rational_t<int> r(-1, 2) * 10
```

```
Rational_t<int> r(-1, 2) * Rational_t(20, -2)
```

```
r = 7
```

```
r + Rational_t(2, 3)
```

```
10 / r / 2 + Rational_t(6, 5)
```

Output: (Erwartet: 5, 5, 23/3, 67/35)



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



## Teil 5 - test\_nr\_five

Tests der Operationen im Namensraum ops::. Getestet wird mit number\_t<int>, nicht mit rational\_t<number\_t<int>>, da in rational\_t intern durch die Verarbeitung der Methoden der Datentyp T weitergegeben wird und dies somit auf dasselbe Ergebnis hinausläuft.

## Fall 1: test\_ops\_abs

Test erfolgreich

Test auf

- negative
- 0.

```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_five:
----- 5) Test ops default.
--- test_ops_abs:
- Input: Number_t<int> -4; Output (Expect 4): 4
- Input: Number_t<int> 0; Output (Expect 0): 0
```

## Fall 2: test\_ops\_divides

Test erfolgreich

Test auf

- rhs teilt lhs
- rhs teils lhs nicht
- Teilung 0.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_divides:
- Input: Number_t<int> 10, 2; Output (Expect true): true
- Input: Number_t<int> 2, 10; Output (Expect false): false
- Input: Number_t<int> 10, 0; Output (Expect false): false
```

## Fall 3: test\_ops\_equals

Test erfolgreich

Test auf

- Ungleichheit
- Gleichheit.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_equals:
- Input: Number_t<int> 10, 2; Output (Expect false): false
- Input: Number_t<int> 2, 2; Output (Expect true): true
```

## Fall 4: test\_ops\_gcd

Test erfolgreich

Test auf

- lhs > rhs
- rhs > lhs.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_gcd:
- Input: Number_t<int> 25, 10; Output (Expect 5): 5
- Input: Number_t<int> 10, 25; Output (Expect 5): 5
```

## Fall 5: test\_ops\_negative

Test erfolgreich

Test auf

- nicht negativ
- negative
- 0.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_negative:
- Input: Number_t<int> 10; Output (Expect false): false
- Input: Number_t<int> -2; Output (Expect true): true
- Input: Number_t<int> 0; Output (Expect false): false
```

## Fall 6: test\_ops\_zero

Test erfolgreich

Test auf 0 oder nicht.

```
Microsoft Visual Studio-Debugging-Konsole
---- test_ops_zero:
--- test_is_zero_nt:
- Input: 0/15; Output (Expect true): true
- Input: -11/15; Output (Expect false): false
```

## Fall 7: test\_ops\_negate

Test erfolgreich

Test auf

- positive -> negative
- negative -> positive
- 0.

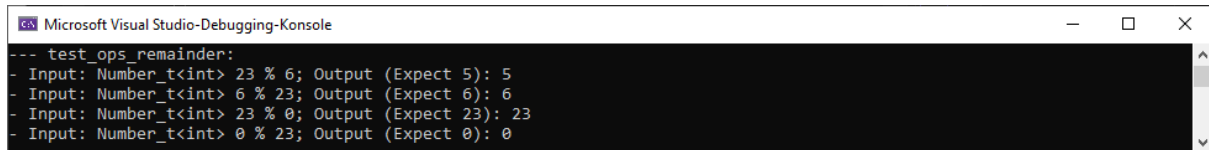
```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_negate:
- Input: Number_t<int> 25; Output (Expect -25): -25
- Input: Number_t<int> -5; Output (Expect 5): 5
- Input: Number_t<int> 0; Output (Expect 0): 0
```

Fall 8: test\_ops\_remainder

Test erfolgreich

Test auf

- lhs > rhs
- lhs < rhs
- Modulo 0
- 0 Modulo.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_remainder:
- Input: Number_t<int> 23 % 6; Output (Expect 5): 5
- Input: Number_t<int> 6 % 23; Output (Expect 6): 6
- Input: Number_t<int> 23 % 0; Output (Expect 23): 23
- Input: Number_t<int> 0 % 23; Output (Expect 0): 0
```

## Teil 6 - test\_nr\_six

Test auf die Bildung neutraler Elemente für `number_t<int>`. Da durch die Verarbeitung der Methoden von `rational_t<number_t<int>>` der Datentyp gleichwertig weitergegeben wird, wird dies nicht extra getestet.

## Fall 1: test\_nelems\_zero

Test erfolgreich



```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_six:
----- 6) Test nelems default.
--- test_nelems_zero:
- Input: Number_t<int>; Output (Expect 0): 0
```

## Fall 2: test\_nelems\_one

Test erfolgreich



```
Microsoft Visual Studio-Debugging-Konsole
--- test_nelems_one:
- Input: Number_t<int>; Output (Expect 1): 1
```

## Teil 7 - test\_nr\_seven

Test auf nelms:: und ops:: auf Int / Spezialisierung Templates für Int.

## Fall 1: test\_nelms\_zero\_int

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_seven:
----- 7) Test ops and nelms for int.
--- test_nelms_zero_int:
- Input: int; Output (Expect 0): 0
```

## Fall 2: test\_nelms\_one\_int

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_nelms_one_int:
- Input: Number_t<int>; Output (Expect 1): 1
```

## Fall 3: test\_ops\_abs\_int

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_abs_int:
- Input: int -4; Output (Expect 4): 4
- Input: int 0; Output (Expect 0): 0
```

## Fall 4: test\_ops\_divides\_int

Test erfolgreich

Test auf

- lhs > rhs && teilbar
- lhs < rhs && nicht teilbar
- Teilung 0.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_divides_int:
- Input: int 10, 2; Output (Expect true): true
- Input: int 2, 10; Output (Expect false): false
- Input: int 10, 0; Output (Expect false): false
```

## Fall 5: test\_ops\_equals\_int

Test erfolgreich

```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_equals_int:
- Input: Number_t<int> 10, 2; Output (Expect false): false
- Input: Number_t<int> 2, 2; Output (Expect true): true
```

## Fall 6: test\_ops\_gcd\_int

Test erfolgreich

Test auf

- lhs &gt; rhs

- lhs &lt; rhs.

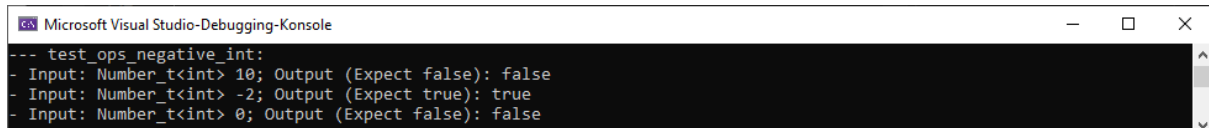


```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_gcd_int:
- Input: int 25, 10; Output (Expect 5): 5
- Input: int 10, 25; Output (Expect 5): 5
```

## Fall 7: test\_ops\_negative\_int

Test erfolgreich

Test ob int negativ.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_negative_int:
- Input: Number_t<int> 10; Output (Expect false): false
- Input: Number_t<int> -2; Output (Expect true): true
- Input: Number_t<int> 0; Output (Expect false): false
```

## Fall 8: test\_ops\_zero\_int

Test erfolgreich

Test ob int 0.



```
Microsoft Visual Studio-Debugging-Konsole
---- test_ops_zero_int:
--- test_is_zero:
- Input: 0/15; Output (Expect true): true
- Input: -11/15; Output (Expect false): false
```

## Fall 9: test\_ops\_negate\_int

Test erfolgreich

Test auf

- positiv -&gt; negativ

- negativ -&gt; positiv

- 0.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_negate_int:
- Input: int 25; Output (Expect -25): -25
- Input: int -5; Output (Expect 5): 5
- Input: int 0; Output (Expect 0): 0
```

Fall 10: test\_ops\_remainder\_int

Test erfolgreich

Test auf

- lhs > rhs && teilbar
- lhs < rhs && nicht teilbar
- Modulo 0
- 0 Modulo.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_ops_remainder_int:
- Input: int 23 % 6; Output (Expect 5): 5
- Input: int 6 % 23; Output (Expect 6): 6
- Input: int 23 % 0; Output (Expect 23): 23
- Input: int 0 % 23; Output (Expect 0): 0
```

## Teil 8 - test\_nr\_eight

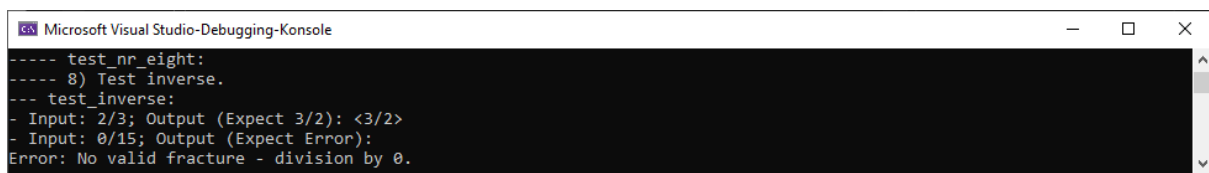
Test auf die Methode inverse.

## Fall 1: test\_inverse

Test erfolgreich

Test auf

- Invertierung
- Invertierung mit /0 als Ergebnis.



```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_eight:
----- 8) Test inverse.
--- test_inverse:
- Input: 2/3; Output (Expect 3/2): <3/2>
- Input: 0/15; Output (Expect Error):
Error: No valid fracture - division by 0.
```

## Fall 2: test\_inverse\_nt

Test erfolgreich

Test mit number\_t auf

- Invertierung
- Invertierung mit /0 als Ergebnis.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_inverse_nt:
- Input: 2/3; Output (Expect 3/2): <3/2>
- Input: 0/15; Output (Expect Error):
Error: No valid fracture - division by 0.
```



## Teil 9 - test\_nr\_nine

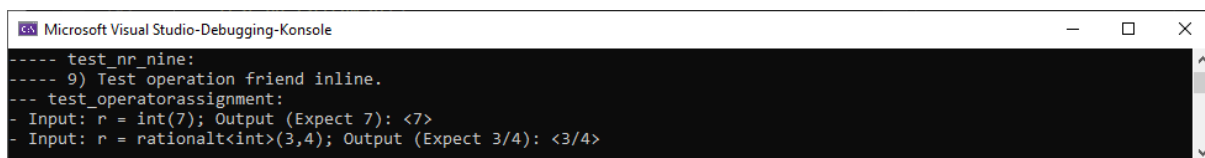
Tests auf sämtliche Operatoren für `rational_t<int>` und `rational_t<number_t<int>>`.  
Operatoren hier sind friend und inline.

## Fall 1: test\_operatorassignment

Test erfolgreich

Test overload operator= auf

- assign
- copy.



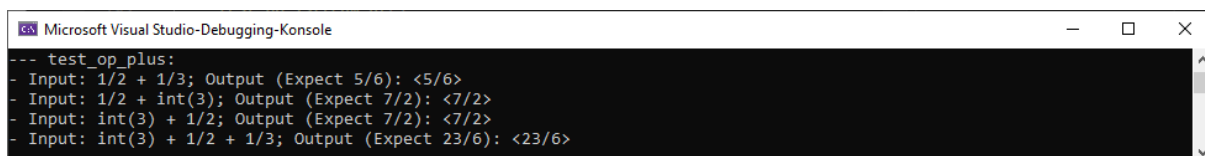
```
Microsoft Visual Studio-Debugging-Konsole
----- test_nr_nine:
----- 9) Test operation friend inline.
--- test_operatorassignment:
- Input: r = int(7); Output (Expect 7): <7>
- Input: r = rational_t<int>(3,4); Output (Expect 3/4): <3/4>
```

## Fall 2: test\_op\_plus

Test erfolgreich

Test overload operator+() auf

- Addition zweier Brüche
- Addition Bruch mit Integer
- Addition Integer mit Bruch
- Mehrere Additionen aneinandergereiht.



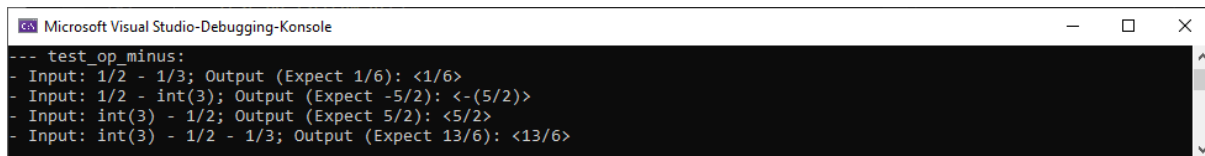
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus:
- Input: 1/2 + 1/3; Output (Expect 5/6): <5/6>
- Input: 1/2 + int(3); Output (Expect 7/2): <7/2>
- Input: int(3) + 1/2; Output (Expect 7/2): <7/2>
- Input: int(3) + 1/2 + 1/3; Output (Expect 23/6): <23/6>
```

## Fall 3: test\_op\_minus

Test erfolgreich

Test overload operator-() auf

- Subtraktion zweier Brüche
- Subtraktion Bruch mit Integer
- Subtraktion Integer mit Bruch
- Mehrere Subtraktionen aneinandergereiht.



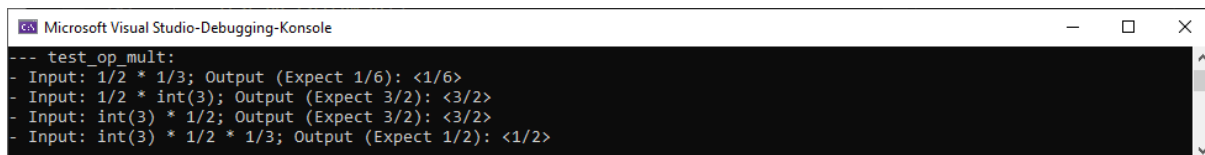
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus:
- Input: 1/2 - 1/3; Output (Expect 1/6): <1/6>
- Input: 1/2 - int(3); Output (Expect -5/2): <-(5/2)>
- Input: int(3) - 1/2; Output (Expect 5/2): <5/2>
- Input: int(3) - 1/2 - 1/3; Output (Expect 13/6): <13/6>
```

## Fall 4: test\_op\_mult

Test erfolgreich

Test overload operator\*() auf

- Multiplikation zweier Brüche
- Multiplikation Bruch mit Integer
- Multiplikation Integer mit Bruch
- Mehrere Multiplikationen aneinandergereiht.



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

## Fall 5: test\_op\_divi

Test erfolgreich

Test overload operator/(()) auf

- Division zweier Brüche
- Division Bruch mit Integer
- Division Integer mit Bruch
- Mehrere Divisionen aneinandergereiht
- Teilung durch 0
- Teilung 0 durch Bruch.

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

## Fall 6: test\_op\_plus\_assign

Test erfolgreich

Test overload operator+=(()) auf

- zwei Brüche
- Bruch auf Integer.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus_assign:
- Input: 4/5 += 2/3; Output (Expect 22/15): <22/15>
- Input: 1/3 += int(3); Output (Expect 10/3): <10/3>
```

## Fall 7: test\_op\_minus\_assign

Test erfolgreich

Test overload operator-=(()) auf

- zwei Brüche
- Bruch auf Integer.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus_assign:
- Input: 4/5 -= 2/3; Output (Expect 2/15): <2/15>
- Input: 1/3 -= int(3); Output (Expect -8/3): <-8/3>
```

## Fall 8: test\_op\_mult\_assign

Test erfolgreich

Test overload operator\*=(()) auf

- zwei Brüche
- Bruch auf Integer.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_mult_assign:
- Input: 4/5 *= 2/3; Output (Expect 8/15): <8/15>
- Input: 1/3 *= int(3); Output (Expect 1): <1>
```

## Fall 9: test\_op\_divi\_assign

Test erfolgreich

Test overload operator/=(()) auf

- zwei Brüche
- Bruch auf Integer
- Teilung durch 0.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_divi_assign:
- Input: 4/5 /= 2/3; Output (Expect 6/5): <6/5>
- Input: 1/3 /= int(3); Output (Expect 1/9): <1/9>
- Input: 1/3 /= int(0); Output (Expect Error):
Error: No valid fracture - division by 0.
<1/9>
```

## Fall 10: test\_op\_compare\_is

Test erfolgreich

Test overload operator==(()) auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_is:
- Input: 4/5 == 2/3; Output (Expect false): false
- Input: 2/3 == 4/5; Output (Expect true): true
- Input: 3/1 == int(3); Output (Expect true): true
```

## Fall 11: test\_op\_compare\_is\_not

Test erfolgreich

Test overload operator!=() auf

- zwei ungleiche Brüche
- zwei gleiche Brüche:
- ein ganzzahliger „Bruch“ auf einen Integer.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_is_not:
- Input: 4/5 != 2/3; Output (Expect true): true
- Input: 2/3 != 4/5; Output (Expect false): false
- Input: 3/1 != int(3); Output (Expect false): false
```

## Fall 12: test\_op\_compare\_smaller\_than

Test erfolgreich

Test overload operator&lt;() auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_smaller_than:
- Input: 4/5 < 2/3; Output (Expect false): false
- Input: 2/3 < 4/5; Output (Expect false): false
- Input: 3/1 < int(3); Output (Expect false): false
```

## Fall 13: test\_op\_compare\_smaller\_equal\_than

Test erfolgreich

Test overload operator&lt;=() auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_smaller_equal_than:
- Input: 4/5 <= 2/3; Output (Expect false): false
- Input: 2/3 <= 4/5; Output (Expect true): true
- Input: 3/1 <= int(3); Output (Expect true): true
```

## Fall 14: test\_op\_compare\_larger\_than

Test erfolgreich

Test overload operator&gt;() auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_than:
- Input: 4/5 > 2/3; Output (Expect true): true
- Input: 2/3 > 4/5; Output (Expect false): false
- Input: 3/1 > int(3); Output (Expect false): false
```

## Fall 15: test\_op\_compare\_larger\_equal\_than

Test erfolgreich

Test overload operator&gt;=() auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_equal_than:
- Input: 4/5 >= 2/3; Output (Expect true): true
- Input: 2/3 >= 4/5; Output (Expect true): true
- Input: 3/1 >= int(3); Output (Expect true): true
```

## Fall 16: test\_op\_ostream

Test erfolgreich

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

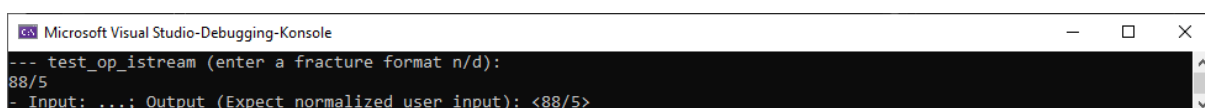


```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_ostream:
- Input: 9/4; Output (Expect 9/4): <9/4>
```

## Fall 17: test\_op\_istream

Test erfolgreich

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



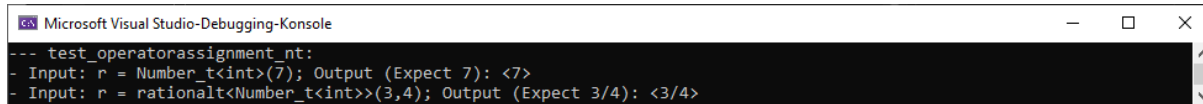
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_istream (enter a fracture format n/d):
88/5
- Input: ...; Output (Expect normalized user input): <88/5>
```

## Fall 18: test\_operatorassignment\_nt

Test erfolgreich

Test overload operator= mit Number\_t&lt;int&gt; auf

- assign
- copy.



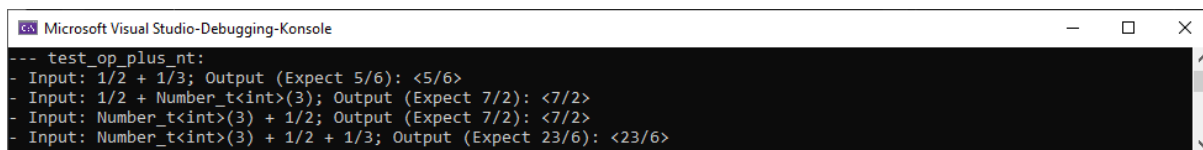
```
Microsoft Visual Studio-Debugging-Konsole
--- test_operatorassignment_nt:
- Input: r = Number_t<int>(7); Output (Expect 7): <7>
- Input: r = rational<Number_t<int>>(3,4); Output (Expect 3/4): <3/4>
```

## Fall 19: test\_op\_plus\_nt

Test erfolgreich

Test overload operator+() mit Number\_t&lt;int&gt; auf

- Addition zweier Brüche
- Addition Bruch mit Integer von Number\_t
- Addition Integer von Number\_t mit Bruch
- Mehrere Additionen aneinandergereiht.



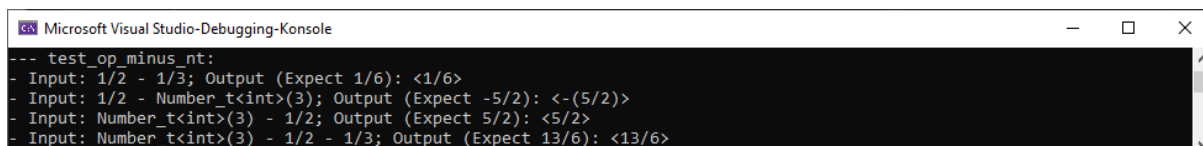
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus_nt:
- Input: 1/2 + 1/3; Output (Expect 5/6): <5/6>
- Input: 1/2 + Number_t<int>(3); Output (Expect 7/2): <7/2>
- Input: Number_t<int>(3) + 1/2; Output (Expect 7/2): <7/2>
- Input: Number_t<int>(3) + 1/2 + 1/3; Output (Expect 23/6): <23/6>
```

## Fall 20: test\_op\_minus\_nt

Test erfolgreich

Test overload operator-() mit Number\_t&lt;int&gt; auf

- Subtraktion zweier Brüche
- Subtraktion Bruch mit Integer von Number\_t
- Subtraktion Integer von Number\_t mit Bruch
- Mehrere Subtraktionen aneinandergereiht.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus_nt:
- Input: 1/2 - 1/3; Output (Expect 1/6): <1/6>
- Input: 1/2 - Number_t<int>(3); Output (Expect -5/2): <-(5/2)>
- Input: Number_t<int>(3) - 1/2; Output (Expect 5/2): <5/2>
- Input: Number_t<int>(3) - 1/2 - 1/3; Output (Expect 13/6): <13/6>
```

## Fall 21: test\_op\_mult\_nt

Test erfolgreich

Test overload operator\*() mit Number\_t&lt;int&gt; auf

- Multiplikation zweier Brüche
- Multiplikation Bruch mit Integer von Number\_t
- Multiplikation Integer von Number\_t mit Bruch
- Mehrere Multiplikationen aneinandergereiht.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_mult_nt:
- Input: 1/2 * 1/3; Output (Expect 1/6): <1/6>
- Input: 1/2 * Number_t<int>(3); Output (Expect 3/2): <3/2>
- Input: Number_t<int>(3) * 1/2; Output (Expect 3/2): <3/2>
- Input: Number_t<int>(3) * 1/2 * 1/3; Output (Expect 1/2): <1/2>
```

## Fall 22: test\_op\_divi\_nt

Test erfolgreich

Test overload operator/() mit Number\_t&lt;int&gt; auf

- Division zweier Brüche
- Division Bruch mit Integer von Number\_t
- Division Integer von Number\_t mit Bruch
- Mehrere Divisionen aneinandergereiht
- Teilung durch 0
- Teilung 0 durch Bruch.

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

## Fall 23: test\_op\_plus\_assign\_nt

Test erfolgreich

Test overload operator+=() mit Number\_t&lt;int&gt; auf

- zwei Brüche
- Bruch auf Integer von Number\_t.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_plus_assign_nt:
- Input: 4/5 += 2/3; Output (Expect 22/15): <22/15>
- Input: 1/3 += Number_t<int>(3); Output (Expect 10/3): <10/3>
```



## Fall 24: test\_op\_minus\_assign\_nt

Test erfolgreich

Test overload operator-=() mit Number\_t&lt;int&gt; auf

- zwei Brüche
- Bruch auf Integer von Number\_t.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_minus_assign_nt:
- Input: 4/5 -= 2/3; Output (Expect 2/15): <2/15>
- Input: 1/3 -= Number_t<int>(3); Output (Expect -8/3): <-(8/3)>
```

## Fall 25: test\_op\_mult\_assign\_nt

Test erfolgreich

Test overload operator\*=() mit Number\_t&lt;int&gt; auf

- zwei Brüche
- Bruch auf Integer von Number\_t.

```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_mult_assign_nt:
- Input: 4/5 *= 2/3; Output (Expect 8/15): <8/15>
- Input: 1/3 *= Number_t<int>(3); Output (Expect 1): <1>
```

## Fall 26: test\_op\_divi\_assign\_nt

Test erfolgreich

Test overload operator/=() mit Number\_t&lt;int&gt; auf

- zwei Brüche
- Bruch auf Integer von Number\_t
- Teilung durch 0.

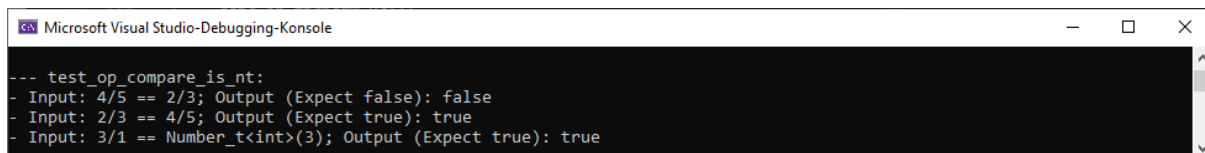
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_divi_assign_nt:
- Input: 4/5 /= 2/3; Output (Expect 6/5): <6/5>
- Input: 1/3 /= Number_t<int>(3); Output (Expect 1/9): <1/9>
- Input: 1/3 /= Number_t<int>(0); Output (Expect Error):
Error: No valid fracture - division by 0.
```

## Fall 27: test\_op\_compare\_is\_nt

Test erfolgreich

Test overload operator==(()) mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



```
Microsoft Visual Studio-Debugging-Konsole

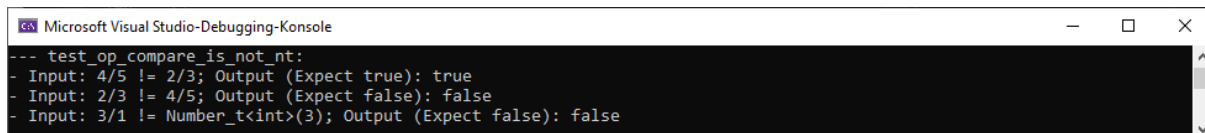
--- test_op_compare_is_nt:
- Input: 4/5 == 2/3; Output (Expect false): false
- Input: 2/3 == 4/5; Output (Expect true): true
- Input: 3/1 == Number_t<int>(3); Output (Expect true): true
```

## Fall 28: test\_op\_compare\_is\_not\_nt

Test erfolgreich

Test overload operator!=(()) mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche:
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



```
Microsoft Visual Studio-Debugging-Konsole

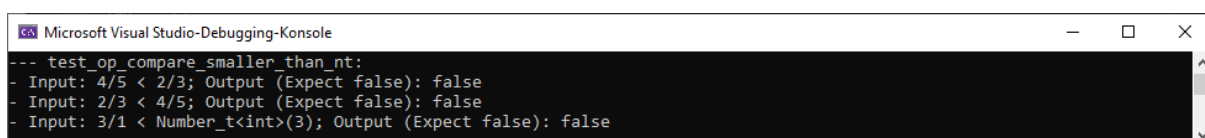
--- test_op_compare_is_not_nt:
- Input: 4/5 != 2/3; Output (Expect true): true
- Input: 2/3 != 4/5; Output (Expect false): false
- Input: 3/1 != Number_t<int>(3); Output (Expect false): false
```

## Fall 29: test\_op\_compare\_smaller\_than\_nt

Test erfolgreich

Test overload operator&lt;() mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



```
Microsoft Visual Studio-Debugging-Konsole

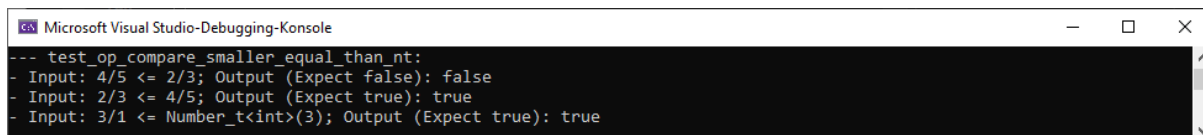
--- test_op_compare_smaller_than_nt:
- Input: 4/5 < 2/3; Output (Expect false): false
- Input: 2/3 < 4/5; Output (Expect false): false
- Input: 3/1 < Number_t<int>(3); Output (Expect false): false
```

## Fall 30: test\_op\_compare\_smaller\_equal\_than\_nt

Test erfolgreich

Test overload operator&lt;=() mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



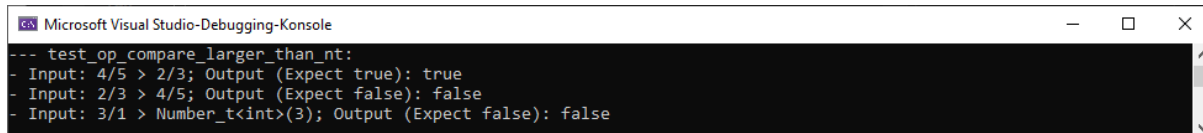
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_smaller_equal_than_nt:
- Input: 4/5 <= 2/3; Output (Expect false): false
- Input: 2/3 <= 4/5; Output (Expect true): true
- Input: 3/1 <= Number_t<int>(3); Output (Expect true): true
```

## Fall 31: test\_op\_compare\_larger\_than\_nt

Test erfolgreich

Test overload operator&gt;() mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



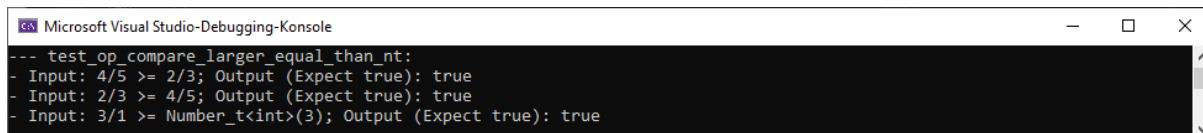
```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_than_nt:
- Input: 4/5 > 2/3; Output (Expect true): true
- Input: 2/3 > 4/5; Output (Expect false): false
- Input: 3/1 > Number_t<int>(3); Output (Expect false): false
```

## Fall 32: test\_op\_compare\_larger\_equal\_than\_nt

Test erfolgreich

Test overload operator&gt;=() mit Number\_t&lt;int&gt; auf

- zwei ungleiche Brüche
- zwei gleiche Brüche
- ein ganzzahliger „Bruch“ auf einen Integer von Number\_t.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_compare_larger_equal_than_nt:
- Input: 4/5 >= 2/3; Output (Expect true): true
- Input: 2/3 >= 4/5; Output (Expect true): true
- Input: 3/1 >= Number_t<int>(3); Output (Expect true): true
```

## Fall 33: test\_op\_ostream\_nt

Test erfolgreich

Test overload operator<<() mit Number\_t<int> auf funktionelle Ausgabe. Da es auf print() basiert, sind weitere Tests hier als gegeben gesehen.

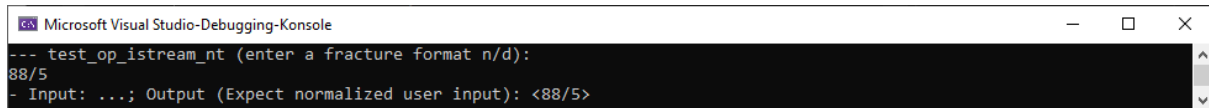


```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_ostream_nt:
- Input: 9/4; Output (Expect 9/4): <9/4>
```

## Fall 34: test\_op\_istream\_nt

Test erfolgreich

Test overload operator>>() mit Number\_t<int> auf funktionelle Ausgabe. Da es auf scan() basiert, sind weitere Tests hier als gegeben gesehen.



```
Microsoft Visual Studio-Debugging-Konsole
--- test_op_istream_nt (enter a fracture format n/d):
88/5
- Input: ...; Output (Expect normalized user input): <88/5>
```