Name: Nikolic Maja **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.

Inhaltsverzeichnis

Beispiel 1 Klasse rational_t erweitern

Lösungsidee

In dieser Übung wird die Klasse rational_t in einen generischen Datentyp umgewandelt, sodass sie mit verschiedenen Arten von numerischen Datentypen arbeiten kann.

Damit der rational_t richtig funktionieren kann werden folgende Operationen für den generischen Typ vorausgesetzt:

- Addition
- Subtraktion
- Mulitplikation
- Division
- Schreiben auf Stream
- Lesen von Stream
- Vergleich auf Gleicheit
- Vergleich auf größer
- Vergleich auf kleiner
- Modulo
- Neutrale Elemente besitzen

Um den Besitz von neutralen Elementen zu gewährleisten werden die generischen Methoden zero() und one() erstellt. Zero() gibt dabei das neutrale Element entsprechend der Addtion zurück und one() das neutrale Element entsprechend der Multiplikation.

Im Namespace ops sind folgende Methoden enthalten:

- Abs() gibt den Absolutwert des übergebenen Wertes zurück
- Divides() prüft ob der erste Wert durch den zweiten Wert ohne Rest dividierbar ist.
- Equals() prüft ob zwei Werte gleich hoch sind
- Gcd() gibt den größten gemeinsamen Teiler zweier Werte zurück
- Is_negative() prüft ob der übergebene Wert kleiner als 0 bzw. das neutrale Element der Addition ist.
- Is_zero() prüft ob der übergebene Wert gleich 0 bzw. dem neutralen Element der Addition ist.
- Negate() gibt den negierten Wert zurück
- Remainder() gibt den Rest zweier Werte nach der Division aus (Modulo)

Der Operator = kann nicht als friend-Funktion implementiert werden, da diese Funktion nicht statisch sein darf.

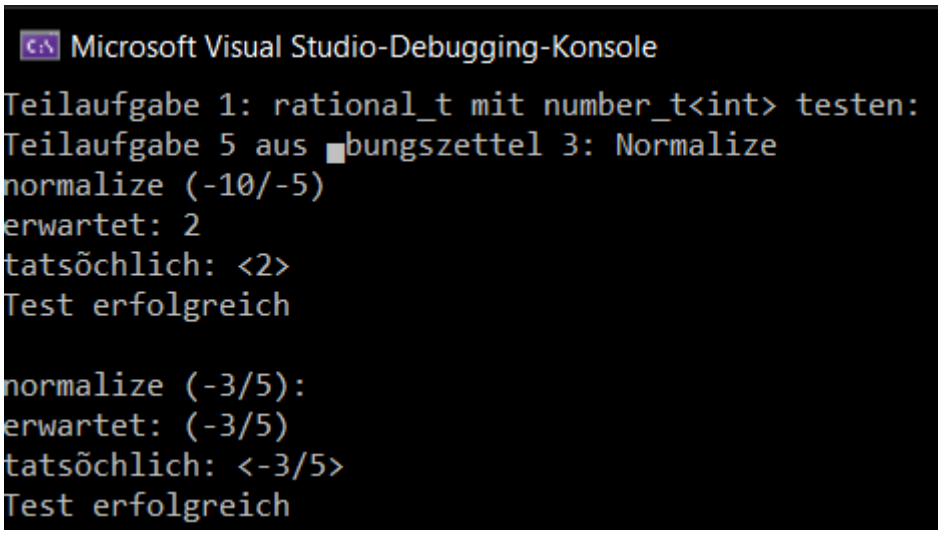
Quelltext: Rational

Tests

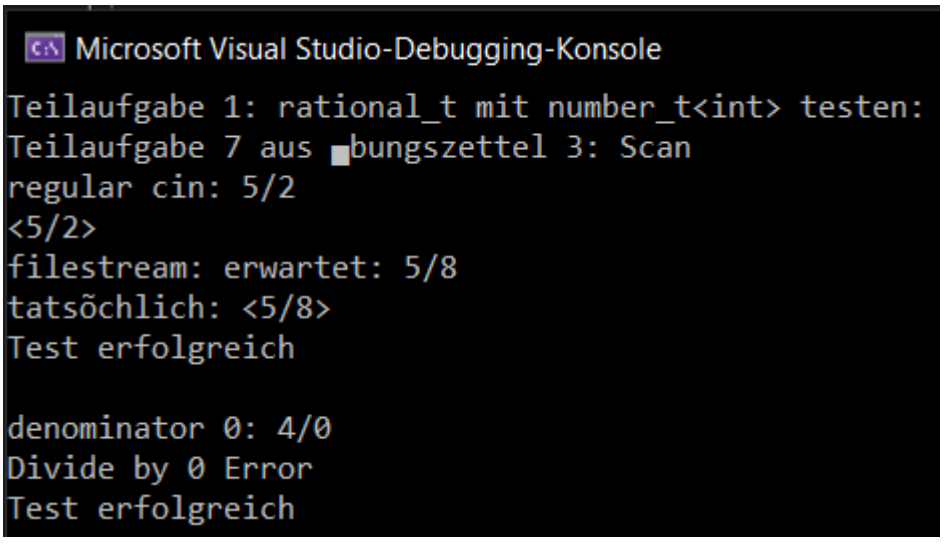
Da in jedem Test print bzw. as_string ausgeführt wird, wird dies nicht explizit getestet.

Rational_t mit number_t<int> testen:

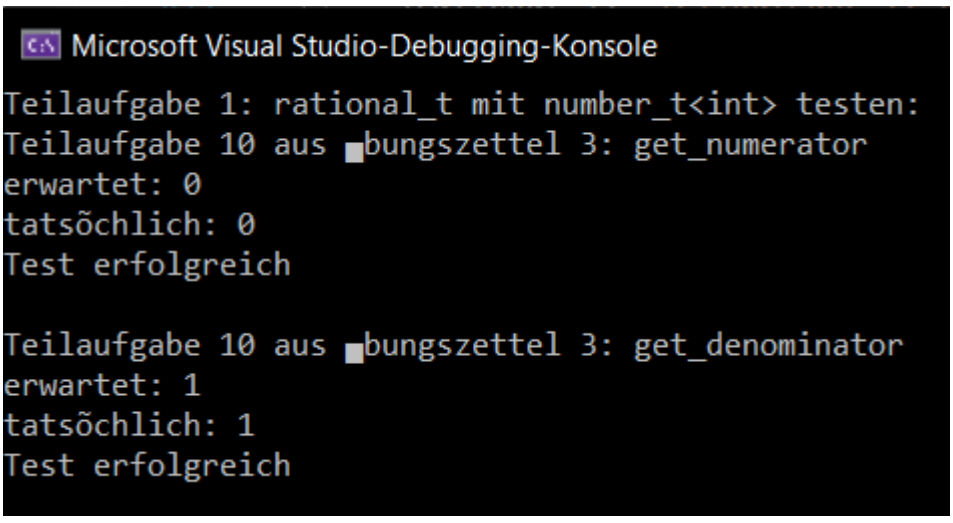
Normalize:



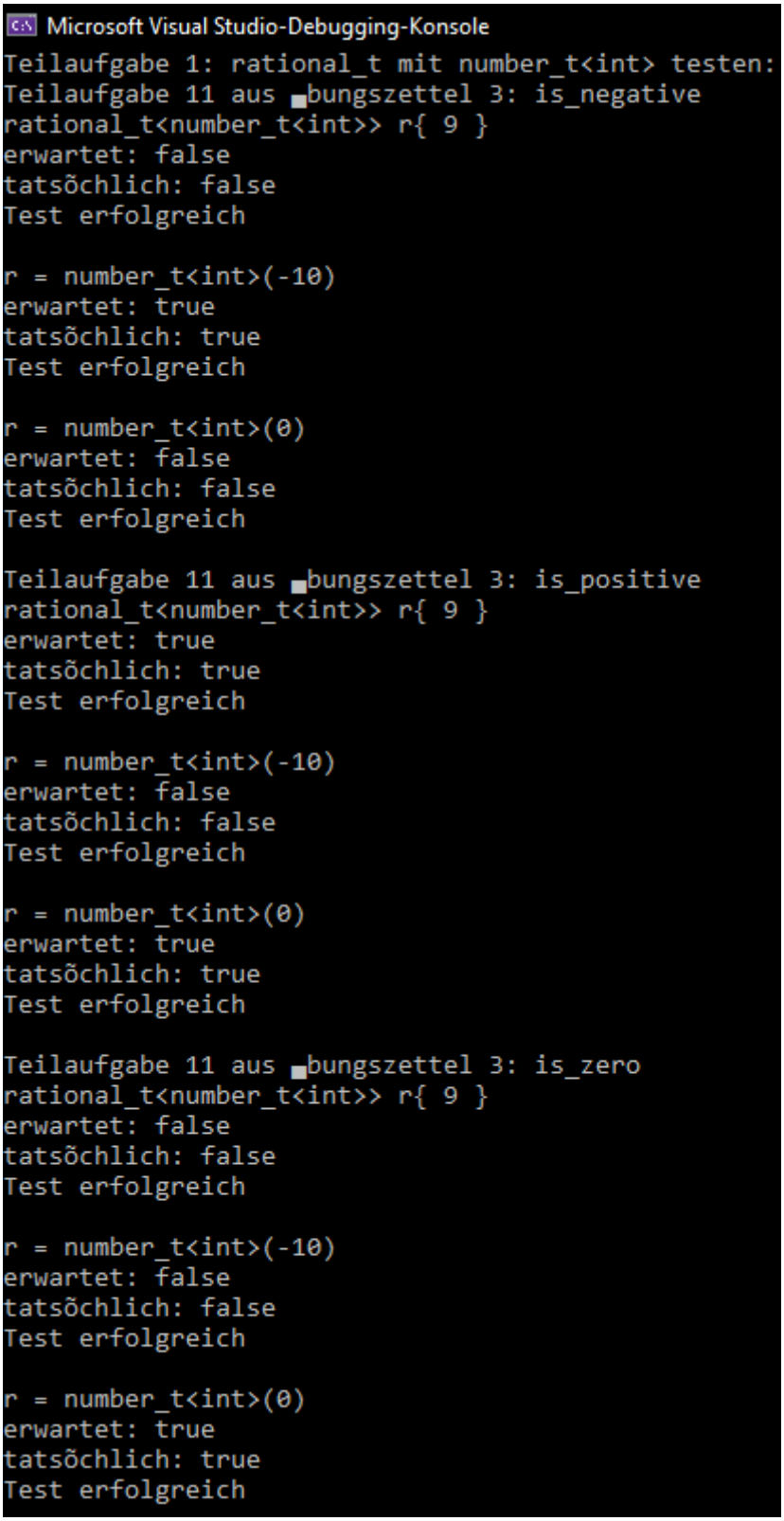
Scan:



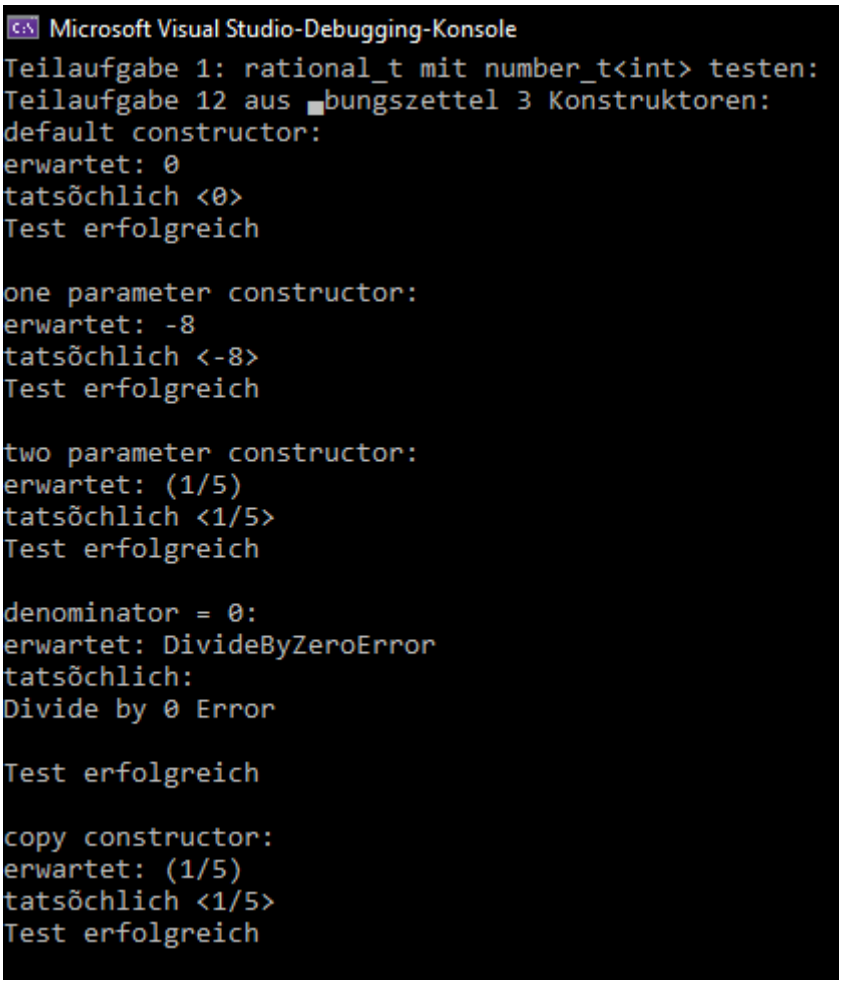
Get_nominator und get_denominator:



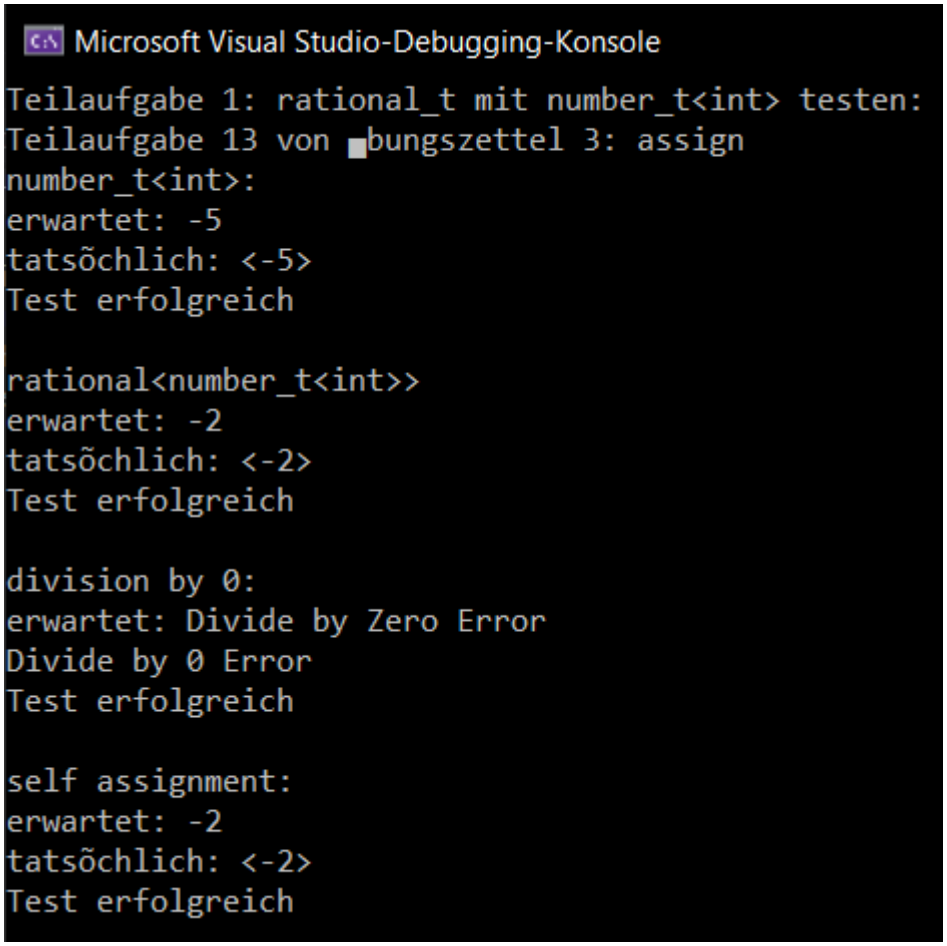
Is_negative, is_positive, is_zero:



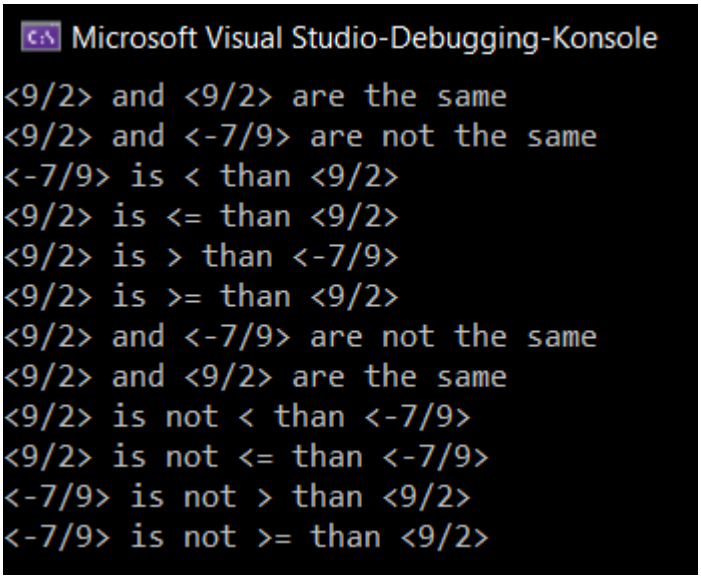
Konstruktoren:



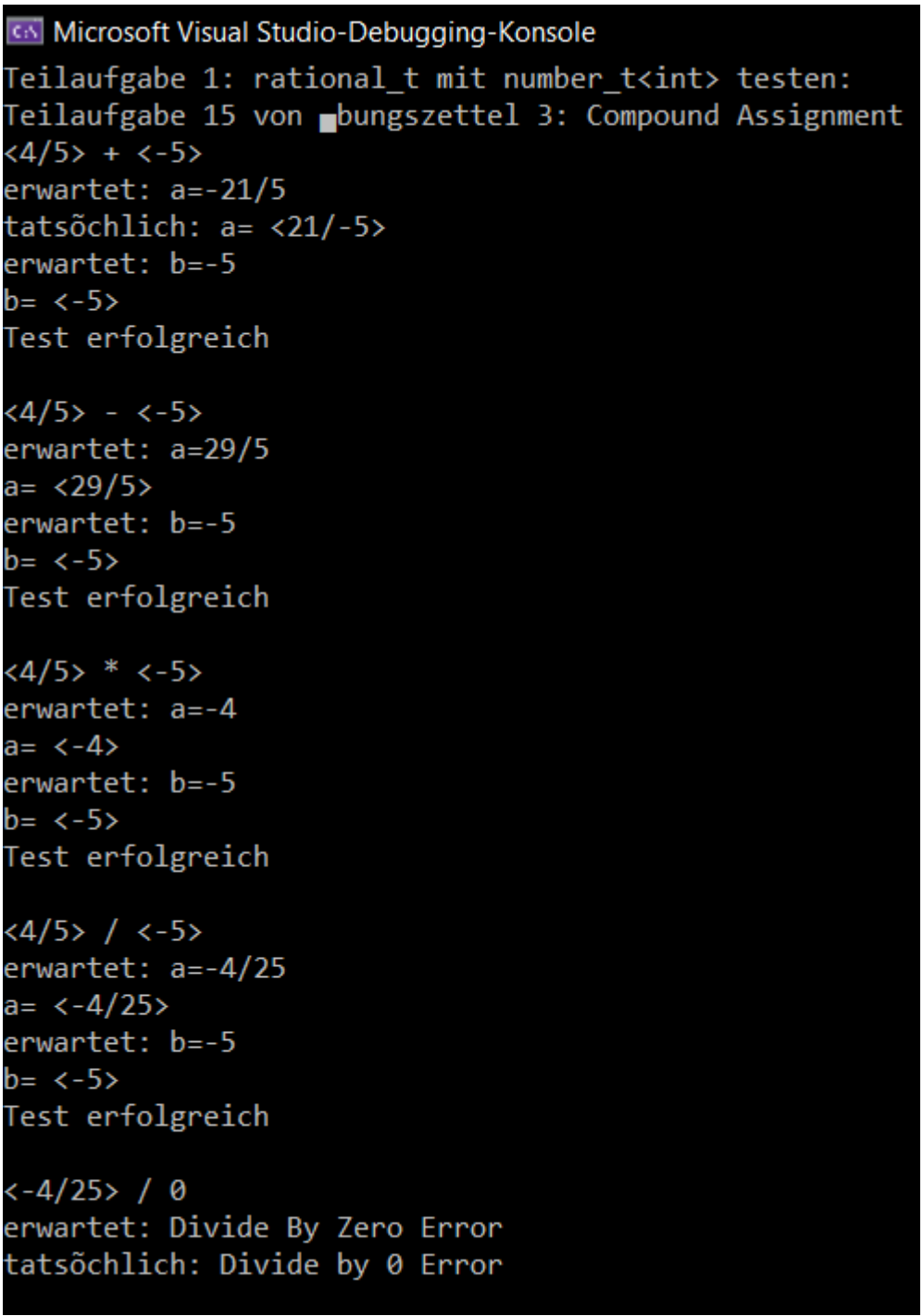
assign:


Microsoft Visual Studio-Debugging-Konsole
Teilaufgabe 1: rational_t mit number_t<int> testen:
Teilaufgabe 13 von bungszettel 3: assign
number_t<int>:
erwartet: -5
tatsõchlich: <-5>
Test erfolgreich
rational<number_t<int>>
erwartet: -2
tatsõchlich: <-2>
Test erfolgreich
division by 0:
erwartet: Divide by Zero Error
Divide by 0 Error
Test erfolgreich
self assignment:
erwartet: -2
tatsõchlich: <-2>
Test erfolgreich

Compare:



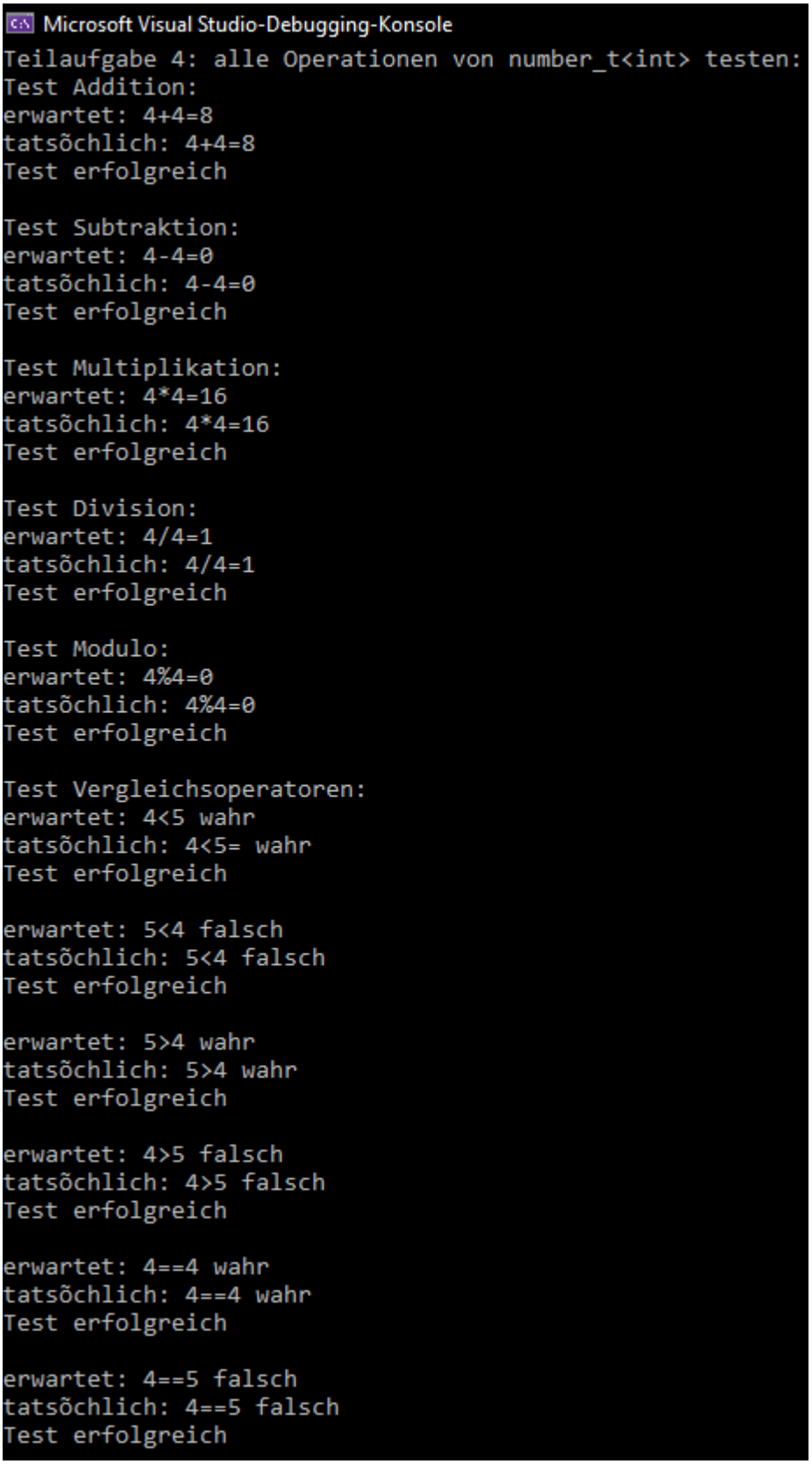
Compound_assign:



calculations:

```
Microsoft Visual Studio-Debugging-Konsole
Teilaufgabe 1: rational_t mit number_t<int> testen:
Teilaufgabe 16 von ▄bungszettel 3: Rechenoperationen
erwartet: <4/5> + <-5> = -21/5
tatsõchlich: <4/5> + <-5> = <21/-5>
Test erfolgreich

erwartet: <4/5> - <-5> = 29/5
tatsõchlich: <4/5> - <-5> = <29/5>
Test erfolgreich

erwartet: <4/5> * <-5> = -4
tatsõchlich: <4/5> * <-5> = <-4>
Test erfolgreich

erwartet: <4/5> * <-5> = -4/25
tatsõchlich: <4/5> / <-5> = <-4/25>
Test erfolgreich


test with number_t<int>:
erwartet: <4/5> + 10 = 54/5
tatsõchlich: <4/5> + 10 = <54/5>
Test erfolgreich

erwartet: <4/5> - 10 = -46/5
tatsõchlich: <4/5> - 10 = <-46/5>
Test erfolgreich

erwartet: <4/5> * 10 = 8
tatsõchlich: <4/5> * 10 = <8>
Test erfolgreich

erwartet: <4/5> / 10 = 2/25
tatsõchlich: <4/5> / 10 = <2/25>
Test erfolgreich

erwartet: <4/5> / 10 = Divide by Zero Error
tatsõchlich: Divide by 0 Error
Test erfolgreich


test with number_t<int> first:
erwartet: 10 + <4/5> = 54/5
tatsõchlich: 10 + <4/5> = <54/5>
Test erfolgreich

erwartet: 10 - <4/5> = 46/5
tatsõchlich: 10 - <4/5> = <46/5>
Test erfolgreich

erwartet: 10 * <4/5> = 8
tatsõchlich: 10 * <4/5> = <8>
Test erfolgreich

erwartet: 10 / <4/5> = 25/2
tatsõchlich: 10 / <4/5> = <25/2>
Test erfolgreich
```

Operatoren von number_t<int>:



Operatoren von number_t<double>:

```
Teilaufgabe 4: alle Operationen von number_t<double> testen:
Test Addition:
erwartet: 4.7+4.7=9.4
tatsöchlich: 4.7+4.7=9.4
Test erfolgreich

Test Subtraktion:
erwartet: 4.7-4.7=0
tatsöchlich: 4.7-4.7=0
Test erfolgreich

Test Multiplikation:
erwartet: 4.7*4.7=22.09
tatsöchlich: 4.7*4.7=22.09
Test erfolgreich

Test Division:
erwartet: 4.7/4.7=1
tatsöchlich: 4.7/4.7=1
Test erfolgreich

Test Modulo:
erwartet: 4.7%4.7=0
tatsöchlich: kompiliert nicht, weil Modulo nicht sinnvoll auf Double angewendet werden kannTest nicht erfolgreich

Test Vergleichsoperatoren:
erwartet: 4.7<5.2 wahr
tatsöchlich: 4.7<5.2= wahr
Test erfolgreich

erwartet: 5.2<4.7 falsch
tatsöchlich: 5.2<4.7 falsch
Test erfolgreich

erwartet: 5.2>4.7 wahr
tatsöchlich: 5.2>4.7 wahr
Test erfolgreich

erwartet: 4.7>5.2 falsch
tatsöchlich: 4.7>5.2 falsch
Test erfolgreich

erwartet: 4.7==4.7 wahr
tatsöchlich: 4.7==4.7 wahr
Test erfolgreich

erwartet: 4.7==5.2 falsch
tatsöchlich: 4.7==5.2 falsch
Test erfolgreich
```

Methoden in operations.h mit number_t<int>

```
Teilaufgabe 5: alle Methoden in operations.h mit number_t<int> testen:
Test abs() mit positivem Wert:
erwartet: abs(5)=5
tatsõchlich: abs(5)=5
Test erfolgreich

Test abs() mit negativem Wert:
erwartet: abs(-8)=8
tatsõchlich: abs(-8=8
Test nicht erfolgreich

Test divides():
erwartet: divides(5,5) = wahr
tatsõchlich: divides(5,5) =  wahr
Test erfolgreich

Test divides():
erwartet: divides(5,-8) = falsch
tatsõchlich: divides(5,-8) =  falsch
Test erfolgreich

Test equals():
erwartet: equals(5,5) = wahr
tatsõchlich: equals(5,5) =  wahr
Test erfolgreich

Test equals():
erwartet: equals(5,-8) = falsch
tatsõchlich: equals(5,-8) =  falsch
Test erfolgreich

Test gcd():
erwartet: gcd(5,5) =5
tatsõchlich: gcd(5,5) = 5
Test erfolgreich

Test gcd():
erwartet: gcd(5,-8) =1
tatsõchlich: gcd(5,-8) = 1
Test erfolgreich

Test is_negative() mit positivem Wert:
erwartet: is_negative(5) falsch
tatsõchlich: is_negative(5)= falsch
Test erfolgreich

Test equals(): mit negativem Wert
erwartet: is_negative(-8) wahr
tatsõchlich: is_negative(-8)= wahr
Test erfolgreich

Test is_zero() mit Nicht 0:
erwartet: is_zero(5) falsch
tatsõchlich: is_zero(5)= falsch
Test erfolgreich

Test is_zero() mit 0:
erwartet: is_zero(0) wahr
tatsõchlich: is_zero(0)= wahr
Test erfolgreich
```

```
Test negate() mit positivem Wert:
erwartet: negate(5) =-5
tatsõchlich: negate(5)=-5
Test erfolgreich

Test negatve() mit negativem Wert:
erwartet: negate(-8)=-8
tatsõchlich: negate(-8)=-5
Test erfolgreich

Teilaufgabe 6: alle Methoden im namespace nelms mit number_t<int> testen:
Test zero<number_t<int>>():
erwartet: zero<number<int>>()=0
tatsõchlich: zero<number_t<int>>()=0
Test erfolgreich

Test one<number_t<int>>():
erwartet: one<number<int>>()=1
tatsõchlich: one<number_t<int>>()=1
Test erfolgreich
```