SEK_UE03

1.Beispiel

Ansatz

Um dieses Problem zu lösen, muss eine Klasse *rational_t* implementiert werden. Es geht dabei darum, mit rationalen Zahlen möglichst aufwandslos in C++ arbeiten zu können. Um dies zu ermöglichen, muss die Klasse in der Lage sein, sowohl den Zähler als auch den Nenner von Brüchen zu speichern. Um Instanzen dieser Klasse besser verwenden zu können werden mithilfe des Operator-Overloading Konzeptes die Operatoren +, -, /, *, ==, !=, <=, >=, < und > überladen. Dadurch wird es möglich, einzelne Instanzen dieser Klasse mit den genannten Operatoren zu vergleichen oder zu verknüpfen. Zusätzlich wird für die Grundrechnungsarten eine Überladung hinzugefügt, welche es ermöglicht, Integer mit *rational_t* Objekten zu verknüpfen. Wichtig ist auch, dass beim Anlegen von rationalen Zahlen darauf geachtet wird, dass der Nenner nicht Null ist.

Umsetzung

Zur Umsetzung ist in der Solution “UE03_Fallmann” im Projekt “T01” zu finden..

Testcases

complex

- Test invalid rational:

```
Test01:
Testing creation of invalid rational
Divide by 0 Error!
success
```

- Test division by zero:

```
Test02:
Testing division by zero
Divide by 0 Error!
success
```

- Test normalization and addition of zero:

```
Test03:
Testing normalisation of neg. numerator and denominator and addition of zero
success
```

- Test normalization and multiplication with zero:

```
Test04:
Testing normalisation of normalized rational and multiplication of zero
success
```

- Test normalization and subtraction of zero:

```
Test05:
Testing normalisation of non normalized rational nr and subtraction of zero
success
```

- Test print whole nr:

```
Test06:
Testing print of whole number
Test print
< 5 >
```

- Test print rational nr:

```
Test07:
Testing print of rational number
Test print
< 5/6 >
```

- Test scan whole nr:

```
Test08:
Testing scan of whole number
success
```

- Test scan rational nr:

```
Test09:
Testing scan of rational number
success
```

- Test scan invalid file:

```
Test10:
Testing scan of invalid file
success
```

- Test constructors:

```
Test11:
Testing default constructor
< 1 >
success

Test12:
Testing int constructor
< 2 >
success

Test13:
Testing constructor init with two numbers
< 2/3 >
success

Test14:
Testing copy constructor
success
```

- Test comparison operators:
  negative – der Vergleichsoperator soll false zurückliefern
  positive – der Vergleichsoperator soll true zurückliefern

```
Test15:
Testing >= positive
success

Test16:
Testing >= negative
success

Test17:
Testing != positive
success
```

```
Test18:
Testing != negative
success

Test19:
Testing == positive
success

Test20:
Testing == negative
success

Test21:
Testing < pos
success

Test22:
Testing < negative
success

Test23:
Testing > positive
success

Test24:
Testing > negative
success

Test25:
Testing <= positive
success

Test26:
Testing <= negative
success
```

- Test compound assign operators:

```
Test27:
Testing compound assignment operator +=
success

Test28:
Testing compound assignment operator -=
success

Test29:
Testing compound assignment operator *=
success

Test30:
Testing compound assignment operator /=
success
```

- Test basic operations:
  Für diese Tests wurde jeweils ein Integer auf der linken Seite des Operators verwendet.

```
Test31:
Testing integer-operation add
success

Test32:
Testing integer-operation subtract
success

Test33:
Testing integer-operation multiply
success

Test34:
Testing integer-operation divide
success
```