SWE3 - Übung 3

WS 2025/26

Aufwand in h: 4

Tim Peko

1. Aufgabe: Rationale Zahl als Datentyp

1.1. Lösungsidee

Die Klasse `rational_t` repräsentiert rationale Zahlen als gekürzte Brüche `Zähler/Nenner` mit `int` als `value_type`. Invarianten: Nenner ist nie `0`, ist immer positiv; `0` wird als `0/1` repräsentiert. Konstruktion und alle Operationen rufen `normalize()` auf, welche mittels euklidischem Algorithmus kürzt und das Vorzeichen in den Zähler schiebt. Ungültige Eingaben (Nenner `0`) lösen `invalid_rational_error` aus; Division durch `0` in $\models$ löst `division_by_zero_error` aus. Vergleiche nutzen Kreuzmultiplikation, um Rundungsfehler zu vermeiden. Streams werden als `"<n/d>"` ausgegeben und `n` oder `n/d` eingelesen.

1.1.1. Designentscheidungen

- Datentyp: `int` als `value_type` gemäß Aufgabenstellung; API könnte später auf `long long`/`std::int64_t` erweitert werden. Um Überläufe zu vermeiden, werden Zwischenrechnungen in `long long` durchgeführt und anschließend normalisiert/gekürzt.
- Invariante Darstellung: Der Nenner ist stets positiv; das Vorzeichen liegt ausschließlich im Zähler. Die Null wird kanonisch als `0/1` gespeichert. Die Methode `normalize()` erzwingt diese Regeln und kürzt mit Euklidischem Algorithmus (`gcd`).
- Ausnahmen: Konstruktion mit Nenner `0` ist ein Programmfehler in der Nutzung der API und wird als `invalid_rational_error` (Unterklasse von `std::invalid_argument` → logic_error-Kategorie) gewertet. Division durch eine rationale Null in $\models$ ist ein Laufzeitproblem im konkreten Rechenschritt und wird als `division_by_zero_error` (Unterklasse von `std::domain_error` → runtime_error-Kategorie) gewertet. Diese Trennung erleichtert die Fehleranalyse.
- Operatoren: Die zusammengesetzten Operatoren (`+=`, `-=`, `*=`, $\models$) bilden die zentrale Implementierung; die binären Operatoren (`+`, `-`, `*`, `/`) delegieren darauf, um Code-Duplikation zu vermeiden. Vergleichsoperatoren nutzen Kreuzmultiplikation (`a/b < c/d` $\iff$ `ad < cb`) in `long long`, wodurch Rundung vermieden wird.
- Interoperabilität mit `int`: Für Ausdrücke mit linkem `int`-Operand (z. B. `3 + rational_t(2,3)`) sind freie Operatoren definiert, um symmetrisches Verhalten zu gewährleisten.
- Streams: `operator<<` gibt in der geforderten Form `"<n/d>"` bzw. `"<n>"` für ganze Zahlen aus. `operator>>` akzeptiert `n` oder `n/d` und setzt `failbit` bei ungültigem Format, wirft aber auch eine Ausnahme bei `n/0`.

1.1.2. Komplexität und Korrektheit

- `normalize()`: Der Euklidische Algorithmus terminiert in `O(log(min(|Z|,|N|)))`. Durch Normalisierung nach jeder arithmetischen Operation bleibt die Darstellung kanonisch; Gleichheit ist folglich strukturelle Gleichheit (`Z1==Z2 && N1==N2`).
- Vergleich: Kreuzmultiplikation ist korrekt, solange das Produkt im `long long`-Bereich bleibt. Die anschließende Normalisierung begrenzt die Größe; realistisch innerhalb typischer Übungsdaten unkritisch.
- Robustheit: Jede Methode, die potenziell die Invariante verletzen kann, ruft `normalize()` oder prüft mit `is_consistent()`.

1.1.3. Teststrategie

- Konstruktorfälle: Default, aus `int`, aus `(Z,N)` inkl. Kürzung, Vorzeichenführung, `0`-Repräsentation.
- Prädikate: `is_negative`, `is_positive`, `is_zero` für repräsentative Randfälle.
- Arithmetik: `+=, -=, *=, /=`; daraus abgeleitete `+, -, *, /`; Mischtypen (`int` links/rechts). Vergleich mit erwarteten kanonischen Strings (`as_string`).
- Fehlerfälle: Konstruktion mit Nenner `0` (logic error), Division durch `0` (runtime error) – jeweils per Exception geprüft.
- I/O: Ausgabeformat `"<...>"`, Basiseinlesen `n`/`n/d` und Fehlerbehandlung.

1.2. Ergebnisse

1.2.1. Testfälle

Getestet werden:
- Konstruktion: Default, aus `int`, aus `(Z, N)`, Normalisierung und Vorzeichenlage
- Prädikate: `is_negative`, `is_positive`, `is_zero`
- Arithmetik: `+=, -=, *=, /=` und abgeleitete `+, -, *, /`, inkl. Mischung mit `int`
- Ausnahmen: Konstruktion mit Nenner 0, Division durch 0
- Vergleich: `==, !=, <, <=, >, >=`
- Streams: `operator<<` (Format) und Grundlagen für `operator>>`

Siehe Tests in `01_Beispiel_Test/test.cpp`.

1.2.2. Implementierungsdetails

- gcd (Euklidischer Algorithmus): Iterative Variante auf `int` mit Absolutwerten; Terminiert in `O(log min(|a|,|b|))`. Im Grenzfall `a=0,b=0` wird `1` zurückgegeben, damit `normalize()` definierte Ergebnisse liefert (Zero-Fall wird davor bereits zu `0/1` kanonisiert).
- normalize():
  - Erzwingt positiven Nenner (`-` auf Zähler/ Nenner flippen bei Bedarf)
  - Kanonisiert `0` zu `0/1`
  - Kürzt durch `gcd`
- Vergleiche: Kreuzmultiplikation (`a.n * b.d` und `b.n * a.d`) in `long long`, um Überlauf-Spielräume zu erhöhen; vermeidet Fließkommafehler.
- Streams: `<<` gibt `"<n/d>"` bzw. `"<n>"` aus. `>>` akzeptiert `n` oder `n/d`, setzt `failbit` bei Leseformatfehlern und wirft `invalid_rational_error` bei `n/0`.