Bonusaufgaben zum C/C++-Praktikum

Grundlagen

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 1

Die Aufgaben für das C/C++-Praktikum sind thematisch sortiert. Zu Beginn jedes Themengebiets der Vortragsfolien ist vermerkt, welche Aufgaben zu diesem Themengebiet gehören (`[G]`: Grundlagen; `[S]`: Speicherverwaltung; `[O]`: Objektorientierung; `[F]`: Fortgeschrittene Themen; `[C]`: (Embedded) C).

Hinweise zum Bonus:

- Für jedes der ersten 4 Teilgebiete (`[G]`, `[S]`, `[O]`, `[F]`) wird es eine Bonusaufgabe geben. Das letzte Teilgebiet `[C]` wird weiter unterteilt in C und Embedded-C und stellt damit zwei Aufgaben. Insgesamt wird es also 6 Bonusaufgaben geben.
- Für die Bearbeitung jeder Aufgabe ist ab Veröffentlichung 2 Wochen Zeit. Danach wird die Abgabe auf Moodle geschlossen.
- Eine Bonusaufgabe gilt dann als bestanden wenn mindestens 75% der auf dem Zettel angegebenen Punkte erreicht werden. Sie gilt entweder bestanden oder nicht bestanden.
- Die Punkte einer Aufgabe haben keine Aussage über deren Anteil am Bonus. Jede der 6 Aufgaben wird für die Berechnung gleich gewichtet. Daraus ergibt sich die folgende Formel:
  Gesamtbonus = 1,0 * (Anzahl bestandener Bonusaufgaben / 6)

Hinweise zur Abgabe:

- Für alle Bonuszettel findet ihr Vorlagen im Moodle, die ihr als Ausgangspunkt für eure Lösung verwenden müsst. Diese enthalten:
  - Für alle außer dieser ersten Übung alle abzugebenden Dateien. In dieser Übung sind die erforderlichen Dateien noch selbst zu erstellen.
  - Für jede abzugebende `.cpp`-Datei auch eine `.test.cpp`-Datei. Diese Dateien enthalten Tests die ihr nutzen solltet um eure Implementationen zu testen. Das erfolgreiche Durchlaufen der Tests ist keine Garantie für volle Punktzahl im Testat, wenn die Tests jedoch nicht durchlaufen werden die Aufgaben mit 0 Punkten bewertet.
  - Jeweils eine `CMakeLists.txt`-Datei. Diese Datei kann mit dem Build-Tool `CMake` verwenden werden um den Code zu kompilieren und die Tests auszuführen. Für eine Bewertung der Abgabe darf die `CMakeLists.txt` **nicht** verändert werden. Eine Erklärung wie `CMake` verwendet wird findet ihr am Ende dieses Übungsblatts.
  - **Wichtig:** Kenntnisse über CMake werden für die Klausur **nicht** vorausgesetzt.
- Für diese Übung sind folgende Dateien im Ordner `src` zu erstellen:
  `rational.hpp, rational.cpp, main.cpp`

Aufgabe 1.1: [G] Rationale Zahlen (4 Punkte)

1.1a) Klasse Rational (1 Punkt)

Deklariere eine Klasse `Rational`, die rationale Zahlen darstellt:

$$\frac{Z}{N} \tag{1}$$

Dabei ist Z $\in \mathbb{Z}$ der Zähler und N $\in \mathbb{Z}\backslash\{0\}$ der Nenner. Die Klasse enthält folgende Elemente:

- Zwei public Attribute: `int, int`
- Konstruktor ohne Parameter
- Konstruktor mit Parametern: `Rational (int counter, int denominator)`
- Copy-Konstruktor
- Destruktor

1.1b) Implementation (1 Punkt)

Implementiere die Elemente der Klasse in rational.cpp. Die Rationale, die ohne Parameter instanziert werden, bekommen eine 1 für den Zähler und eine 2 für den Nenner. Alle Konstruktoren/Destruktoren müssen eine Meldung auf der Konsole ausgeben, wenn sie aufgerufen werden, zum Beispiel „Rational z/n created “

Binde den Header rational ein und implementiere die Funktion main in der Datei main.cpp. Erstelle darin Rationalen a=$\frac{1}{4}$ und b=$\frac{1}{2}$.

1.1c) Operatorenüberladung (2 Punkte)

Überlade nun den Operator <<, um die Ausgabe eines Rationals auf der Konsole zu ermöglichen. Für ein Rational mit dem Zählerwert 2 und Nennerwert 3 soll beispielsweise `std::string("2/3")` von der überladenen Methode ausgegeben werden.

Überlade die Operatoren +, −, * und / für die Berechnung der entsprechenden Operation mit Rationalen. Berechne dann in `main` die folgenden Ausdrücke:

- c = $\frac{1}{2} + \frac{1}{4}$
- d = $\frac{1}{2} - \frac{1}{4}$
- e = $\frac{1}{2} * \frac{1}{4}$
- f = $\frac{1}{2} / \frac{1}{4}$

und drucke sie anschließend mit std::cout und dem Operator << auf der Konsole in der oben angegebenen Reihenfolge aus. Nach jeden Ergebnis soll ein Zeilenumbruch folgen.

Jetzt muss der Operator < überladen werden. Der Operator gibt zurück, ob das Rational links vom Operator kleiner ist als das andere Rational. Teste den Operator in main, indem du den kleinsten Ausdruck (c, d, e oder f) findest und auf der Konsole ausgibst.

Hinweise

- Die Ergebnisse müssen nicht gekürzt werden.

1.1d) Rationale Zahlen vereinfachen (1 Punkt)

Überlege dir eine Funktion `Rational simplify(Rational rhs)`, die den vereinfachten Term einer rationalen Zahl zurückgibt. Teste die Funktion mit c, d, e und f aus der vorherigen Aufgabe, indem du jeweils simplify auf das Rational anwendest. Gebe das Ergebnis anschließend wie in 1.1c auf der Konsole aus.

Mit CMake arbeiten

Wichtig: Kenntnisse über CMake werden für die Klausur **nicht** vorausgesetzt.

`CMake` ist ein Buildtool, das alle möglichen Aufgaben wie das Kompilieren, Testen, Packen und Installieren von Software automatisieren kann. Um dies zu erreichen definiert `CMake` eine eigene Beschreibungssprache die unabhängig von sowohl dem verwendeten Compiler als auch dem verwendeten Buildsystem ist. So kann ein `CMake`-Projekt unter Windows mit `Visual Studio`, aber auch unter Linux mit `make` und `gcc` oder `clang` gebaut werden. Der Inhalt der `CMakeLists.txt` muss für die Bearbeitung der Bonusaufgaben und die Prüfung nicht näher verstanden werden. Für die weitere Erklärung wird davon ausgegangen, dass ihr entweder die im Moodle bereitgestellte Virtuelle Maschine verwendet oder eine eigene Linux Installation nutzt, auf der `CMake` und ein C++-Compiler installiert sind.

Bevor die Dateien in einem `CMake`-Projekt kompiliert werden können, müssen zuerst aus der `CMakeLists.txt` Builddateien generiert werden. Dafür muss folgenden Befehl in demselben Ordner, in dem sich die `CMakeLists.txt` befindet ausgeführt werden:

```
cmake -B build -DCMAKE_BUILD_TYPE=Debug
```

`build` bedeutet, dass alle benötingten Daten im Unterordner `build` erzeugt werden. `CMAKE_BUILD_TYPE` kann entweder auf `Debug` oder `Release` gesetzt werden. `Debug` bedeutet, dass unter Anderem vom Compiler keine Optimierungen des Codes vorgenommen werden und beispielsweise Klassen der C++-Standardbibliothek Fehlerüberprüfungen durchführen, die unter `Release` für eine bessere Performance deaktiviert werden. `Debug` empfiehlt sich somit für das Bearbeiten der Aufgaben und das Suchen von Fehlern.

Das eigentliche Bauen der Dateien wird mit dem folgenden Befehl angestoßen:

```
cmake --build build --parallel
```

Du wirst feststellen, dass hier viel mehr passiert als für so wenige zu kompilierende Dateien zu erwarten wäre. Das liegt daran, dass automatisiert auch ein Testframework, das für das Kompilieren und Ausführen der Tests benötigt wird, mitkompiliert wird. Läuft der Befehl erfolgreich durch, solltest du in dem Ordner `build` zwei auszuführende Dateien `tasks` und `tests` finden.

Möchtest du deine eignene `main`-Funktion ausführen kannst du dies wie folgt tun:

```
./build/tasks
```

Um die Tests auszuführen kannst den folgenden Befehl verwenden:

```
cmake --build build --parallel --target test -- ARGS=--output-on-failure
```

Hierbei musst du nachdem du deinen Code bearbeitet hast daran denken, zunächst den Code noch einmal zu bauen bevor du die Tests erneut ausführst. Um beides in einem Schritt zu machen, kannst du den Befehl zum Testen wie folgt ergänzen:

```
cmake --build build --parallel --target all test -- ARGS=--output-on-failure
```