Ü B U N G E N zu Programmiersprache 1 (Python, C++) WS 2021/2022

Übungsblatt Nr. 10: in der Übungsstunde

1) Nehmen Sie das Beispiel mit der umfassendsten Implementierung von `Bruch` (HÜ 9, Aufgabe 2). Extrahieren Sie daraus eine Headerdatei `Bruch.hpp` und verwenden Sie diese.

2) a) Berechnen Sie `a + Bruch{1,1}`. Das wird gehen. Berechnen Sie ebenso `a + 1` und `a + 1`. Funktioniert das Programm immer noch? Welche Fehlermeldungen bekommen Sie jetzt?
   b) Schreiben Sie einen Konstruktor `Bruch(int, int)`. Geht es jetzt?
   c) Schreiben Sie einen Konstruktor `Bruch(int)`. Geht es jetzt?
   d) Schreiben Sie auch einen Konstruktor `Bruch()` (Default Konstruktor).

3) a) Starten Sie mit v2e) Kopieren Sie aus dem Vorlesungstext `Mathematik mit C++` Abschnitt `Folgen, Reihen` den Code der `ggt()`-Funktion in die Headerdatei. Verbessern Sie den Konstruktor, dass er den Nenner positiv macht, den GGT von Zähler und Nenner berechnet und den Bruch kürzt. Ist der Nenner 0, soll das Programm mit einer Fehlermeldung abgebrochen werden.

4) a) Machen Sie aus zwei Konstruktoren einen mit Defaultargument und verwenden Sie eine Initialisierungsliste statt Zuweisungen.
   b) Schreiben Sie einen Streaminput für `Bruch`. Geben Sie 0 beim Nenner ein. Kommt die Fehlermeldung? Beheben Sie diesen Bug.

Übungsblatt Nr. 10 Hausübung: Die folgenden Aufgaben sind Pflicht und zählen 1 Punkt!

1) a) Lesen Sie in einer Endlosschleife von der Tastatur Brüche ein und speichern Sie diese in einem `vector<Bruch> v`. Die Schleife soll enden, wenn nichts mehr eingelesen werden kann. Geben Sie anschließend alle eingelesenen Brüche mit einem Lambda wieder aus.

   b) Lesen Sie die Brüche aus der Datei `zahlenpaare.txt`. Sie müssen dazu nur ihr Programm mit `./a <zahlenpaare.txt` starten.

   c) Sortieren Sie den Vektor absteigend nach den Zählern, aufsteigend nach den Nennern und machen jedes Mal einen Kontrollausdruck.

   d) Sortieren dieses mal mit `std::ranges:sort()`: Sie inkludieren Sie `functional` und `ranges` und verwenden zum Sortieren:

```
// absteigend nach z
std::ranges::sort(v, std::greater<>{}, &Bruch::z);
std::ranges::sort(v, {}, &Bruch::n);    // aufsteigend nach n
// statt std::greater<>{} geht auch std::ranges::greater{}
```

   Fleißaufgabe: Können Sie die Brüche auch aufsteigend nach Wert sortieren?

   e) Berechnen Sie auch die Summe der Brüche.

2) Wiederholen Sie die vorige Aufgabe mit `Vec2`.

   a) Definieren Sie die Streameingabe für `Vec2` und lesen damit mit der Endlosschleife in einen `vector<Vec2> v;` ein.

   b) Sortieren Sie analog `v` absteigend nach den x- und aufsteigend nach den y-Koordinaten.

   c) Sortieren Sie die Vektoren absteigend nach ihrer Länge.

   d) Addieren Sie alle Vektoren.

3) a) Probieren Sie alle Programme aus Vorlesung 10, eigene Datentypen, SimpleStat aus. Sie können den Datenfile `input.txt` verwenden mit
`./a.exe <input.txt`

b) Programmieren sie die `operator+=()` Methode und die Streamausgabe im Programm `SimpleStat3.cpp` so, dass genau dieselbe Auswertung wie bei Version 2 gedruckt wird (d.h kopieren Sie die entsprechenden Programmzeilen an die richtige Stelle und passen Sie sie an).

c) Erweitern Sie die Statistik um die Varianz und die Standardabweichung:

$$\text{Varianz} = (\sum_{i=1}^{n} x_i^2 - n \cdot \text{Mittelwert}^2)/(n-1),$$

Standardabweichung = $\sqrt{\text{Varianz}}$. Sie brauchen dazu noch eine weitere Summenvariable (z.B. `sum2`), in der Sie die Stichprobenquadrate aufsummieren.

d) Sie sollten die Zeile mit dem Mittelwert nur ausgeben, wenn $n > 0$ ist. Bauen Sie eine entsprechende `if`-Konstruktion ein. Sie sollten die Varianz und die Standardabweichung nur ausgeben, wenn $n > 1$ ist. Programmieren Sie auch diese Bedingung.

4) a) Anstelle Zahlen aus einer Datei einzulesen, können Sie solche Daten auch selbst generieren, z.B. mit einem Zufallszahlengenerator. Die Beschreibung dazu finden Sie im Vorlesungsskript. Um Zufallszahlen zu erzeugen, müssen Sie `random` inkludieren und einen Zufallszahlengenerator erzeugen. Ich empfehle die Mersenne-Twisterengine `mt19937_64` (die auch Python verwendet!), am besten als globales Objekt vor dem eigentlichen Programm:
`std::mt19937_64 g;`
Dann brauchen Sie noch eine Verteilungsfunktion, die die Zahlen des Generators geeignet transformiert. Wir wollen hier eine Normalverteilung mit Mittelwert 5. und Standarabweichung 3. erzeugen:
`std::normal_distribution<double> nv{5., 3.};`
Jetzt liefert der Code `nv(g)` eine solche Zufallszahl. Erzeugen Sie eine Million Zufallszahlen und werten Sie diese statistisch aus!

b) Jetzt wenden wir eine bekannte Taktik der (getürkten) Meinungsforschung an. Wir nehmen nur jene Stichproben, die uns gefallen. Definieren Sie dazu eine weiteres SimpleStat-Objekt mit Titel `getuerkte_NV`. In dieser speichern Sie die Stichproben nur, wenn sie > 1 sind. Geben Sie beide Statistiken aus!

Challenge-Aufgabe(n): Diese Aufgaben sind freiwillig, zählen 3-8 Punkte und sind bis zum angegeben Termin abzugeben!

5) Mein unglaubliches Bus- bzw. Tram-Beispiel:
   a) Verifizieren Sie analog zum letzten Beispiel, dass die Verteilung
   `std::exponential_distribution wartezeit{0.1};`
   eine Zufallszahl mit Erwartungswert 10 liefert. Die Exponentialverteilung wird gerne zur Simulation von Wartezeiten eingesetzt.
   b) Verwenden Sie nun `w` dazu, um einen zufälligen Fahrplan einer fiktiven Tramlinie 2 zu generieren, wobei alle Zeiten in Minuten und alle Zeitpunkte in Minuten ab Mitternacht angegeben werden. In der Funktion `ein_tag()` generieren Sie den Fahrbetrieb eines Tages: Der Betrieb beginnt um Punkt 6 Uhr am Morgen (= 6*60), wann die erste Bim losfährt. Nach einer zufälligen Wartezeit `w` startet die nächste, nach einer weiteren Wartezeit `w` die nächste usw., bis der Betrieb um 23 Uhr (= 23*60) endet. Keine Bahn fährt nach diesem Zeitpunkt ab. Geben Sie die Abfahrtszeiten aller Tagesfahrten aus:

      ```
      Tram 1: 360
      Tram 2: ...
      ```

   c) Wenn ihr Fahrplan funktioniert, entfernen Sie die Ausgabe des Fahrplans und wiederholen den Tag 10000 mal. Erheben Sie nun folgende Statistiken (legen Sie diese Objekte am Programmanfang an):

      ```
      SimpleStat alle_trams{"Alle Abfahrtsintervalle w"};
      SimpleStat trams_per\_tag{"Nummer der letzten Tram des Tages"};
      ```

      Kontrollieren Sie, ob wirklich im Schnitt alle 10 Minuten eine Abfahrt stattfindet.
   d) Sie fahren jetzt jeden Tag um genau 13 Uhr 13 nach der Programmiersprache-Vorlesung mit der Bim. Sie nehmen genau jene Bahn, die als erstes nach diesem Zeitpunkt ankommt. Machen Sie eine Statistik

      ```
      SimpleStat meine_bim{"Abfahrtsintervall meiner Bim"};
      ```

      Können Sie erklären, warum diese Tram ein durchschnittliches Abfahrtsintervall von 20 Minuten hat?
      (3 Punkte, Abgabe bis Sonntag, 16.01.2022)

6) a) Speichern Sie `double` Inputdaten (z.B. der Datei `zahlenpaare.dat` in einem Vector `std::vector<double> v` und geben Sie ihn wieder aus.
   b) Kopieren Sie `v`, sortieren Sie die Kopie aufsteigend mit `std::sort()` und geben Sie sie aus.
   c) Kopieren Sie `v`, sortieren Sie die Kopie aufsteigend mit `std::ranges::sort()` und geben Sie sie aus.
   d) Geben Sie in b) und c) aus, wie viele Vergleichsoperationen beim Sortieren durch den Algorithmus gemacht wurden. Benutzen Sie dazu eine selbstgeschriebene Vergleichsoperation, die zählt, wie oft sie aufgerufen wurde.
   (3 Punkte, Abgabe bis Sonntag, 16.01.2022)

7) Lösen Sie das vorige Beispiel mit dem Objekttyp `struct count_comparisons`, der den Vergleich mit `operator()` erledigt und die Aufrufanzahl in einer internen Variable `mutable int calls` zählt. Der Destruktor soll diese Zahl ausgeben. Das `mutable` bedeutet, dass eine Änderung dieses Attributs auch in einem `const` Objekt erlaubt ist. Dann könnten Sie das Sortieren und die Ausgabe der Vergleichsoperationen so erledigen:
   `std::sort(v, count_comparisons{});`
   Sie werden allerdings feststellen, dass `std::sort` und `std::ranges::sort()` völlig unterschiedliche Resultate liefern, weil offenbar die Vergleichsfunktion unterschiedlich oft kopiert wird.
   (3 Punkte, Abgabe bis Sonntag, 16.01.2022)

8) Modifizieren Sie ihr `struct count_comparisons`, sodass obige Programmzeile richtig funktioniert. Jede Instanz der Struktur soll ihre Vergleichsoperationen ausgeben und in einer Zusammenfassung soll die gesamte Anzahl von Vergleichen sowie die Anzahl der Kopien des Vergleichsobjektes gedruckt werden.
   (5 Punkte, Abgabe bis Sonntag, 23.01.2022)