Schleifensteuerung

- Eine Schleife läuft, solange die Fortsetzungsbedingung (z.B. `i<=100`) erfüllt ist
- Dieses Verhalten aus logischen oder Performance-Gründen nicht immer erwünscht
  - Die Schleife hat ihr Ziel bereits erfüllt, weitere Iterationen sind überflüssig
  - Nach wenigen Iterationen wird ersichtlich, dass die Aufgabe nicht lösbar ist
- Abhilfe mit `break` → `mod2finder.cpp`

- Der Code im Inneren einer Schleife kann i.A. sehr kompliziert sein
- Vielleicht möchte man alle Iterationen starten aber nicht jede Iteration vollständig ausführen
- `break` würde hier die Schleife vollständig beenden, also nicht hilfreich
- Gewünschtes Ergebnis mit `continue` → `sqrtcond.cpp`
- Durch geschicktes Kombinieren von `break` und `continue` lassen sich mit wenig Code nichttriviale Sachen programmieren

Auswahl beschränken mit switch

- Wenn eine Anweisung nur bestimmte Werte (z.B. 1, 2 oder 3) ausgeben darf, kann man die entsprechende Programmlogik mit `if - else if - else` implementieren.
- Es gibt aber eine viele bequemere Konstruktion, die sich genau für solche Fälle eignet: `switch` $\rightarrow$ `verkaufsautomat.cpp`

Datentypen

- Wertebereiche der integrierten Datentypen sind enorm wichtig
- Typische Fehler in der C++ Numerik durch lässigen Umgang mit Datentypen: Genauigkeitsverlust (precision loss), Unterlauf (underflow), Überlauf (overflow)$\rightarrow$ `overflow.cpp`
- Häufiger Denkfehler: Die Addition ist doch kommutativ, d.h. $a + b = b + a$? $\rightarrow$ `add.cpp`

Nachbesprechung Übungsaufgaben

- Aufgabe 4: Verallgemeinerung der Fakultät, die Werte wachsen sehr schnell und es kommt zu einem Überlauf → `a4a-produkt.cc`
- Mit zwei Schleifen gibt es $N(N+1)/2$ Iterationen, die Laufzeit des Algorithmus beträgt $\sim \mathcal{O}(N^2)$ → `a4b-summe.cc`
- Betrachte die Berechnung der Summanden

$$\text{Schritt 1:} \quad \frac{1}{1}\left(\frac{1}{1}\right)$$

$$\text{Schritt 2:} \quad \frac{1}{2}\left(\frac{1}{1}+\frac{1}{2}\right)$$

$$\text{Schritt 3:} \quad \frac{1}{3}\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}\right)$$

$$\ldots$$

- Die Innere Summe ändert sich jeweils um $1/i$, muss also nicht jedes mal neu berechnet werden!
- Mit einer einzigen Schleife verbessert sich die Laufzeit auf $\sim \mathcal{O}(N)$ → `a4b-summe-v2.cc`

- Aufgabe 5: Wiederholung 3. Vorlesung: Schritt ist die kleinste Zahl, so dass gilt

$$1.0 + \text{Schritt} \neq 1.0$$

- Genau diesen Schritt wollen wir hier experimentell bestimmen → `a5-genauigk.cc`
- Aufgabe 6: Darstellung einer Zahl im Dreiersystem

$$z = c_2 \cdot 3^2 + c_1 \cdot 3^1 + c_0 \cdot 3^0, \quad 0 \leq c_i \leq 2$$

$$16 = 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \Rightarrow 16 = 121_{\text{Basis } 3}$$

- Wir iterieren über die Stellen einer Zahl zu Basis 3 mit einer 3-fachen Schleife → `a6-zaehl3.cc`
- Die Laufzeit von 3 verschachtelten Schleifen ist sehr ungünstig...

- Andere Möglichkeit:
  - Ganzzahlige Division: $14 \text{ div } 6 = 2$, in C++ `14 / 6`
  - Modulo: $14 \mod 6 = 2$, in C++ `14 % 6`
- Beispiel Dezimalsystem: $697 = 6 * 100 + 9 * 10 + 7$, wie extrahiert man die Koeffizienten 6, 9 und 7?

$$(697 \text{ div } 100) \,\%\, 10 = 6 \,\%\, 10 = 6$$
$$(697 \text{ div } 10) \,\%\, 10 = 69 \,\%\, 10 = 9$$
$$(697 \text{ div } 1) \,\%\, 10 = 697 \,\%\, 10 = 7$$

- Genauso lässt sich der Algorithmus auch im Dreiersystem verwenden $\rightarrow$ `a6-zaehl3-v2.cc`

Vorbesprechung Übungsaufgaben

- Aufgabe 7: `if` und else if benutzen
- Aufgabe 8: Formatierungshilfen $\rightarrow$ `format.cpp`
- Aufgabe 9: Nützlich zum Testen: Unix-Befehl `date`, z.B.

```
date -d "2019-10-02" +%A
```