

# Übungen zu Computergrundlagen WS 2019/2020

## Übungsblatt 12: C++

24. Januar 2019

### Allgemeine Hinweise

- Abgabetermin für die Lösungen ist **Freitag, 31.01.2020, 11:00 Uhr**
- Schickt die Lösungen bitte per Email an Euren Tutor:
  - Montag 14:00–15:30: Moritz Schumacher (mschumacher@icp.uni-stuttgart.de)
  - Dienstag 9:45–11:15: Samuel Tovey (stovey@icp.uni-stuttgart.de)
  - Dienstag 15:45–17:15: Philipp Stärk (pstaerk@icp.uni-stuttgart.de)
  - Mittwoch 15:45–17:15: Marco Brückner (mbrueckner@icp.uni-stuttgart.de)
  - Donnerstag 9:45–11:15: Ingo Tischler (itischler@icp.uni-stuttgart.de)
- Die Übungen sollen von Gruppen von jeweils *zwei* (nur in Ausnahmefällen drei) Leuten bearbeitet werden. Bitte gebt *nur eine Lösung pro Gruppe* ab und nennt in eurer Abgabe alle Mitglieder eurer Gruppe!
- Als Lösung der Aufgabe soll ein einziges Python-Skript erstellt werden, welche ihr dann per E-Mail an euren Tutor schickt.

### Hinweise:

- Wenn der Compiler lange Fehlermeldungen ausgibt, ist es meist hilfreich, die Zeile zu suchen, in der `Error:` steht. Wenn dort auf ein Header-File (z.B. `vector`) verwiesen wird, kann man so lange nach unten suchen, bis man die Stelle findet, die in der eigenen `.cpp`-Datei die entsprechende Funktion aus dem Header aufruft.
- Verwendet einen Texteditor, der Syntaxhervorhebung für C++ bietet. Beispielsweise: `nano`, `vim`, `gedit`, `emacs`, ... unter Linux; `TextMate`, `BBEdit`, ... unter macOS; `Notepad++`, `Visual Studio Code`, ... unter Windows.
- Kompiliert euren Code mit `g++ -std=c++17 -Wall -o programm programm.cpp` und führt in aus mit `./programm`.
- Ein laufendes Programm kann jederzeit mit der Tastenkombination `Strg-C` abgebrochen werden. Dies ist nützlich, wenn man beim Programmieren z.B. versehentlich eine Schleife geschrieben hat, die nie abbricht.

### Aufgabe 12.1: Datentypen (1 Punkt)

Sagt vorher, welches Ergebnis (Wert und Datentyp, oder einen Compilerfehler) folgende Ausdrücke produzieren. Gebt jeweils den Grund dafür an. Nehmt an, dass alle nötigen Header-Files (z.B. `<string>`, `<cmath>`) eingebunden wurden. (1 Punkt)

- `3 + 5`
- `3 + 5.0`

- "3" + "5"
- std::string("3") + "5"
- 3 / 2
- 3.0 / 2
- int(2.71828)
- std::round(2.71828)

### Aufgabe 12.2: std::vector (2 Punkte)

Schreibt eine Funktion, die alle geraden Zahlen unterhalb einer vorgegebenen Grenze als std::vector zurückgibt. Verwendet dafür das Codegerüst unten und testet sie damit. (2 Punkte)

```
#include <cassert>
#include <vector>

... gerade_zahlen(...) {
    std::vector<unsigned int> v ...
    for (...) {
        ...
    }
    return v;
}

void main() {
    assert(gerade_zahlen(5) == std::vector<unsigned int>({2,4}));
    assert(gerade_zahlen(6) == std::vector<unsigned int>({2,4,6}));
}
```

### Aufgabe 12.3: Templates (1 Punkt)

Schreibt eine Funktion quadriere(x), die Zahlen eines beliebigen Datentyps quadriert und als eine Zahl desselben Datentyps zurückgibt. Verwendet dafür das Codegerüst unten. (1 Punkt)

```
#include <cassert>

template <...>
... quadriere(... x) {
    ...
}

void main() {
    auto a = quadriere(2);
    assert(a == 4);
    assert(typeid(int) == typeid(a));

    auto b = quadriere(2.0);
    assert(b == 4);
    assert(typeid(double) == typeid(b));
}
```

```

    auto c = quadriere(2.0f);
    assert(c == 4);
    assert(typeid(float) == typeid(c));
}

```

### Aufgabe 12.4: Iteratoren (3 Punkte)

Schreibt eine Funktion, die einen beliebigen Container (z.B. `std::list`, `std::vector`) mit Zahlen bekommt und die Standardabweichung von diesen berechnet. Verwendet dafür ein Template und testet sie mit Hilfe des Codegerüsts unten. (3 Punkte)

```

#include <list>
#include <vector>

template <typename T>
typename T::value_type standardabweichung(const T & daten) {
    typename T::value_type mittelwert = 0, varianz = 0;

    for (...) {
        ...
    }
    ...
    .
    .
    ...

    ...

    return std::sqrt(varianz);
}

void main() {
    std::vector<double> daten1 = { 525.8, 605.7, 843.3, 1195.5,
                                1945.6, 2135.6, 2308.7, 2950.0};
    std::list<float> daten2 = { 727.7, 1086.5, 1091.0, 1361.3,
                              1490.6, 1956.1};

    auto s1 = standardabweichung(daten1);
    assert(std::abs(s1 - 894.373) < 1e-3);
    auto s2 = standardabweichung(daten2);
    assert(std::abs(s2 - 420.972) < 1e-3);
}

```

### Aufgabe 12.5: <iostream> (1 Punkt)

Der Code unten, der die Funktion von Aufgabe 12.4 benutzt, gibt immer mehrere Dezimalstellen auf dem Bildschirm aus. Wir wollen aber nur eine Dezimalstellen sehen. Schlagt in der [Dokumentation](#) nach, was ihr hinzufügen musst, um die Genauigkeit für die Ausgabe in einen `iostream` einzustellen. (1 Punkt)

```

#include <iostream>

void main() {
    std::list<float> daten2 = { 727.7, 1086.5, 1091.0, 1361.3,
                               1490.6, 1956.1};
    auto s2 = standardabweichung(daten2);

    std::cout << s2 << std::endl;
}

```

## Aufgabe 12.6: Ausführungsgeschwindigkeit (2 Punkte)

In der Vorlesung habt ihr gelernt, dass C++ viel schneller sei als Python. Diese Behauptung gilt es, zu überprüfen.

**12.6.1** Generiert mit `numpy.random.random` ein Array mit Zufallszahlen. Benutzt anschließend das Python-Modul `timeit`, um zu testen, wie lange die folgenden beiden Python-Funktionen pro Datenpunkt brauchen. (1 Punkt)

```

import numpy
import timeit

N = 10000000

def standardabweichung_numpy(daten):
    return numpy.std(daten, ddof=1)

def standardabweichung(daten):
    mittelwert = sum(daten)/len(daten)
    varianz = sum([(x - mittelwert) ** 2 for x in daten]) / (len(daten) - 1)
    return varianz**0.5;

daten = numpy.random.random(...)

t1 = timeit.timeit(... standardabweichung(daten) ...)
print(...)

t2 = timeit.timeit(... standardabweichung_numpy(daten) ...)
print(...)

```

**12.6.2** Verwendet die C++-Bibliothek `std::chrono`, um mit folgendem Code zu messen, wie lange eure Funktion von Aufgabe 12.4 braucht. Fügt anschließend `-O3` zu eurem Compiler-Aufruf hinzu und messt nochmal.

```

#include <chrono>
#include <algorithm>
#include <vector>

void main() {
    auto start = std::chrono::high_resolution_clock::now();

    unsigned int N = 10000000;

    std::vector<double> daten(N);

```

```

std::generate(daten.begin(), daten.end(), []() {
    return std::rand();
});

auto s = standardabweichung(daten);

auto end = std::chrono::high_resolution_clock::now();
auto sekunden = std::chrono::duration_cast<
    std::chrono::nanoseconds>(end - start).count();

std::cout << "Berechnung von " << s << " dauerte " << (sekunden/N)
    << " Nanosekunden pro Datenpunkt" << std::endl;
}

```

Warum ist C++ mit -O3 schneller als ohne, warum ist Python langsamer als C++ und Python mit Numpy schneller als ohne? (1 Punkt)

**Hinweis:** Erhöht die Anzahl der Datenpunkte so lange, bis die Zeit pro Datenpunkt konstant wird.