Draft der C++-Vorlesung vom 13.März 2026

Nachtrag 6. März:
Unter Linux ist `long int` 8 Byte lang, siehe demo_sizeof[1].

1. Strukturiertes Programmieren und Struktogramme.
   Code: v_2a[2] mit docu[3].

   Script: §4.1-4.7

2. Zyklen: In Beispiel Loops[4] sind die drei Zyklenarten Zählzyklus (`for`), abweisender Zyklus (`while`) und nichtabweisender Zyklus (`do{}while;`) aufgeführt.
   - `for`- und `while`-Zyklus sind im angeführten Beispiel identisch und für $n < 0$ wird der Schleifenkörper `{...}`. Dagegen wird im `do-while`-Zyklus zuerst der Schleifenkörper `{...}` bevor auf weiter Ausführung getestet wird, für $n < 0$ ist dieser Zyklus also nicht identisch zu den beiden anderen.
   - Für die Variable n sind in die folgenden, hier äquivalenten Definitionen möglich:
     - `int n=5;` // klassische Zuweisung
     - `int n(5);` // Parameterkonstruktor der Klasse `int`
     - `int n{5};` // via initializer list der Klasse, C++11

     In obigem Fall ist die initializer list exakt dasselbe wie der Parameterkonstruktor. Das gilt **nicht generell**, siehe `vector`!

3. Programmierstil: Die Beispiele Loops_BadStyle1.cpp[5] (keine Kommentare) und Loops_BadStyle2.cpp[6] (zusätzlich keine Einrückungen) zeigen auf wie sie nicht programmiere sollen.
   C++ erlaubt auch, wie in Loops_BadStyle3.cpp[7], daß der gesamte Code in einer Zeile geschrieben werden kann. Dies kann automatisch durch Formatierungstool korrigiert werden, siehe *Plugins* $\longrightarrow$ *Source code formatter* in der IDE CodeBlocks.

4. Demonstration zur Genauigkeit von Gleikommaoperationen.
   Warum ist im Code ab Zeile 41 die wiederholte Addition mit 1.0/1024 genauer als mit 1.0/10 (Binärdarstellung der Zahlen)?
   Code: v_2d[8]

5. Beispiel Geheimzahl (Errate Zahl im Intervall), Struktogramm[9] an Tafel.
   Code: v_2b[10] mit docu[11].
   - Funktionen `eingabe` und `geheimZahl`.
   - Umsetzung des Struktogramms in `geheimZahl` unter Verwendung der Funktion `eingabe`.
   - Unterscheide *Deklaration* (Prototype [Ankündigung]) von *Definition* (Implementierung) einer Funktion.
   - Zufallszahlengenerierung (`main.cpp:76`) im Intervall $[anf, ende]$ noch mit C-Zufallszahlen (`#include <cstdlib>`).

6. Vektoren: Nachzulesen in [Haase, §5.1] und Beispielen v_3b[12] und v_3c[13].

   Script: §5.1

[1] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/demo_sizeof.zip
[2] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2a.zip
[3] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2a/html
[4] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/Loops.cpp
[5] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/Loops_BadStyle1.cpp
[6] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/Loops_BadStyle2.cpp
[7] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/Loops_BadStyle3.cpp
[8] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2d.zip
[9] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/5_PDFsam_vorlesung_1_2.pdf
[10] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2b.zip
[11] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2b/html
[12] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3b.zip
[13] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3c.zip

- Deklaration, benötigt vorher `#include <vector>` und `using namesapce std` :
  `vector<double> v;` // Vektor der Länge 0.
  wobei der Template parameter `double` (fast) jeder andere Datentyp sein kann.
- Definition gleich in Deklaration, empfehlenswert falls möglich. Die hier benutzte Elementanzahl 5 kann auch eine, zur Compilezeit unbekannte, Variable sein.
  - via Parameterkonstruktor:
    `vector<double> v(5);` // **nicht initialisierter** Vektor der Länge 5.
  - via Parameterkonstruktor:
    `vector<double> v(5,-1.0);` // mit -1.0 initialisierter Vektor der Länge 5.
  - via Initializer List geschweifte Klammern:
    `vector<double> v{-1.0,2.2,1,2,3};` // Vektor der Länge 5 $[-1.0, 2.2, 1, 2, 3]$.
  - Vorsicht! Unterscheide zwischen
    `vector<int> v(5);` // **nicht initialisierter** int-Vektor der Länge 5.
    und
    `vector<int> v{5};` // int-Vektor der Länge 1 mit einzigem Element 5.
- Lese-/Schreibzugriff auf Vektorelement v_2 (Indizes starten mit 0).
  - `v[2]` // ohne Indexüberprüfung
  - `v.at(2)` // mit Indexüberprüfung (langsamer)
- Dynamisches Wachsen des Vektors $v = [-1.0, 2.2, 1, 2, 3]$, durch Anhängen eines Elementes:
  `v.push_back(-1.234);` // Vektor der Länge 6 $[-1.0, 2.2, 1, 2, 3, -1.234]$
  `cout << v.size();` // Gibt 6 als Anzahl der Elemente aus.
- Vektor als Funktionsparameter in Bsp. v_3c[14] (html[15]) und .
  Script: §7.4
  - als Input-Parameter, siehe `main.cpp:59`:
    `void print_vek(vector<float> const & v)`
  - als Output (InOut)-Parameter, siehe `main.cpp:32`:
    `void vec_init_2(const int n, (vector<float> & v)`
  - als Return-Parameter, siehe `main.cpp:17`:
    `vector<float> vec_init_1(const int n)`
- Bei **statisch**en Vektoren `array<double, 5> sv;` muß die Elementanzahl zur **Compilezeit** feststehen.
  Script: §5.1.2

7. Demonstration der Nutzung von `string`, `vector`.
   Code: demoString[16], demoVector[17].
   Script: §2.3.1-2.3.3
8. Dynamischer Vektor und dessen Nutzung als Input- oder Outputparameter in Funktionen.
   Code: v_2c[18] mit docu[19].
   Script: §5.1.1

   Aufg 4.: Kleine Funktion für elementweise Eingabe eines dynamischen Vektors v, solange $v_i > 0$.
9. Beispiel Geheimzahl mit Speicherung aller Versuche (dynamischer Vektor) und einem Vektor von Strings zur flexiblen Ausgabe.
   Code: v_3a[20] mit docu[21].

[14] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3c.zip
[15] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3c/html
[16] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/demoString.cpp
[17] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/demoVector.cpp
[18] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2c.zip
[19] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_2c/html
[20] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3a.zip
[21] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_3a/html

- Auch mit `array` [Haase, §5.1.2] von Strings möglich (`main.cpp:51`).
- Zufallszahlengenerierung (`main.cpp:108-112`) im Intervall $[anf, ende]$ mit C++-Zufallszahlen `#include <random>`, zur Vertiefung[22].
- Benötigt die Compileroption `-std=c++11` (Standard)

10. Trennung in Header- und Source-Files (Deklaration vs. Definition)
Signum-Fkt. im Beispiel v_4c[23]: [clevere sgn.Funktion[24]]
Um mehrfaches Einbinden desselben Headerfiles in einem Sourcefile zu verhindern: Voranstellen von

`#pragma once`[25]

Diese *preprocessor directive* ist nicht im Standard inkludiert wird aber von allen(?) Compilern untestützt und ersetzt das ältere *header guarding*
`#ifndef FKT_H_INCLUDED #define FKT_H_INCLUDED ... #endif`

11. **Signatur** einer Funktion:
Dient der eindeutigen Zuordnung einer Funktion beim Aufruf.

```
namespace::classname::functionname (inputParamList, outputParamList)
```

- Return Type einer Funktion ist für deren Signatur irrelevant.
- Übergabe eines Parameters via Copy oder via Referenz erzeugt die identische Signatur, da die zu übergebenden Parameter identisch sind.
- Eine Signatur ist der globale, eindeutige Bezeichner Ihrer Funktion.
- Sollen Funktionen auf die Übersetzungseinheit beschränkt bleiben (also noch global), dann ist der Funktionsdeklaration ein `static` voranzustellen.

Literatur

[Haase] Gundolf Haase: Einführung in die Programmierung mit C++, *www*[26].

[Stroustrup10] Bjarne Stroustrup: Einführung in die Programmierung mit C++. Pearson Studium, München (2010).

[22] https://en.cppreference.com/w/cpp/header/random
[23] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS26/v_4c.zip
[24] https://stackoverflow.com/questions/1903954/is-there-a-standard-sign-function-signum-sgn-in-c-c
[26] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf