Draft der C++-Vorlesung vom 20.März 2020

1. Zyklen: In Beispiel Loops[1] 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`!

2. Programmierstil: Die Beispiele Loops_BadStyle1.cpp[2] (keine Kommentare) und Loops_BadStyle2.cpp[3] (zusätzlich keine Einrückungen) zeigen auf wie sich nicht programmiere sollen.
   C++ erlaubt auch, wie in Loops_BadStyle3.cpp[4], 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.

3. Vektoren: Nachzulesen in [Haase20, §5.1] und Beispielen v_3b[5] und v_3c[6].
   - 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)

[1] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/Loops.cpp
[2] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/Loops_BadStyle1.cpp
[3] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/Loops_BadStyle2.cpp
[4] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/Loops_BadStyle3.cpp
[5] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_3b.zip
[6] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_3c.zip

- 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[7] (html[8]) und [Haase20, §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 [Haase20, §5.1.2].

4. Parameterübergabe an Funktionen: Sie können Variablen auf drei Arten in der Parameterliste einer Funktion anführen [Haase20, §7.2]:
   - per Value (= Kopie): `vector<double>`
   - per Referenz (= Original mit anderem Bezeichner): `vector<double>&`
   - per Pointer: `vector<double>*` **Verboten in dieser LV!**

   Damit können Sie Input-Parameter
   - als Kopie (`vector<double>`) oder
   - als konstante Referenz (`vector<double> const&`)

   übergeben. Reine Output-Parameter und InOut-Parameter werden
   - als veränderbare Referenz (`vector<double>&`)

   gehandhabt.

5. Signatur einer Funktion: Siehe [Haase20, §7.1] und Beispiel v_3_signatur[9].
   Damit eine Funktion beim Linken (der bereits kompilierten Programmteile) ihren Aufrufen in anderen Programmteilen zugeordnet werden kann, ist eine eindeutige Kennung (=*Signatur*) nötig.
   - Signatur := Funktionsname + Datentypen der Parameterliste, z.B.
     ```
     double square(double x)                          // main:6
     int square(int x)                                // main:13
     void print_vek(vector<float> const & v)          // v_3c/main:59
     ```
   - Beachte beim letzten Bsp., daß
     `void print_vek(vector<float> & v)`
     eine neue Signatur darstellt, da das `const` in den Datentyp eingeht.

[7] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_3c.zip
[8] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_3c/html
[9] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_3_signatur.zip

6. Header- und Sourcefiles: Bei Variablen unterscheiden wir zwischen der Deklaration `float g;` und der Definition `g=-1.234;` .
   Genauso unterscheiden wir bei Funktionen zwischen der
   - Deklaration (=Ankündigung) einer Funktion (Strichpunkt ; beachten!)
     `void copy_vek(vector<int> const& a, vector<int> const& b);`
   - der Definition (=Realisierung) an anderer Stelle

     ```
     void copy_vek(vector<int> const& a, vector<int>& b)
     {
         b.resize(a.size());
         for (unsigned int k = 0; k < a.size(); ++k)  b[k] = a[k];
         return;
     }
     ```
   - Damit ergibt sich in Beispiel v_4c[10] die natürliche Aufteilung
     - Deklarationen ⟶ Headerfile `v_4c/fkt.h:29`
     - Definitionen ⟶ Sourcefile `v_4c/fkt.cpp:48`
     - Die Deklarationen werden in die anderen Sourcefiles (und weitere Headerfiles) via
       `#include "fkt.h"`
       inkludiert, siehe `v_4c/main.cpp:4` und `v_4c/fkt.cpp:1`.
   - Die Dokumentation der Funktion gehört als *Beschreibung des Interfaces* in das Headerfile (`v_4c/fkt.h:25`). Hieraus generiert *doxygen* die Dokumentation im gewünschten Outputformat (html, LaTeX, ...).
   - Um mehrfaches Einbinden desselben Headerfiles in einem Sourcefile zu verhindern, wird das *header guarding* angewandt (`v_4c/fkt.h`):

     ```
     #ifndef FKT_H_INCLUDED
        #define FKT_H_INCLUDED
        ...                                   // Unsere Deklarationen
     #endif
     ```

     Dies wird durch das **global eindeutigen** Macro `FKT_H_INCLUDED` garantiert.
     Eine weitere C++-Lösungsmöglichkeit im Voranstellen von `#pragme once`[11].

[10] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_4c.zip`
[11] `https://de.wikipedia.org/wiki/Include-Guard`

7. IO-System und File-IO: Die Ein- und Ausgabe von Daten in Programmen erfolgt über Datenströme (*stream*). Sie kennen bereits die Datenströme `cout` und `cin` mit ihren zugehörigen Operatoren `<<` und `>>`. Zusätzlich haben wir die Fehlerausgabe über `cerr` welche ebenfalls mit `#include <iostream>` inkludiert wird.
   Diese Datenströme lassen sich aus/in Files umlenken [Haase20, §8], was bei größeren Datenmengen natürlich einen enormen Vorteil darstellt.

   - Statt mit `cout << d` geben wir eine double-Variable `d` in das File *out.txt* aus, siehe dazu Beispiel v_5a[12]:
     ```
     #include <fstream>                                    // ofstream
     #include <iostream>
     using namespace std;
     ...
     ofstream out_file("out.txt");                  // ASCII-File als Ausgabefile
     out_file << d;
     ```
   - Wir haben ein Text/ASCII[13]-file benutzt, Binärfiles sind ebenfalls möglich[14].
   - Statt zwei String-Variablen `c1`, `c2` mit `cin>>c1>>c2` über die Tastatur einzugeben, lesen wir diese vom ASCII-File *my_in.txt* ein.
     ```
     #include <fstream>                                    // ofstream
     #include <iostream>
     using namespace std;
     ...
     ifstream in_file("my_in.txt");                 // ASCII-File als Eingabefile
     in_file >> c1 >> c2;
     ```
   - Siehe auch das ausführliche Beispiel Simple_FileIO[15].
   - Wenn die Operatoren `<<` und `>>` für einen Datentyp definiert sind, so sind obige IO-Operation mit cin/cout oder via Files möglich. Anderfalls müssen diese beiden Operatoren für den neuen Datentyp definiert werden [Haase20, §9.8].

8. Einlesen eines Vektors vom ASCII-File: Das Beispiel file_io[16] demonstriert das Lesen/Schreiben eines double-Vektors via ASCII-Files.

   - Das kurze Hauptprogramm demonstriert die Anwendung der Leserfunktion (`main.cpp:16`) und der Schreibfunktion (`main.cpp:24`).
   - Die Lesefunktion `read_vector_from_file` (`file_io.cpp:30`) öffnet das File und bricht mit einer Fehlermeldung ab, falls das Inputfile nicht gefunden wird.
   - In obigem Erfolgsfalle wird der Vektor in der Funktion `fill_vector` solange mit Daten gefüllt (und dabei dynamisch verlängert, `file_io.cpp:14`) bis das Fileende erreicht ist. Zeilen `file_io.cpp:15-24` dienen nur der möglichen Fehlerbehandlung [Stroustrup10, p.364] und `file_io.cpp:25` verkürzt den Vektor auf die nötige Länge.
   - Die Schreibfunktion `write_vector_to_file` ist selbserklärend.

Literatur

[Haase20] Gundolf Haase: Einführung in die Programmierung mit C++ (2020), *www*[17].

[12] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/v_5a.zip`
[13] `https://de.wikipedia.org/wiki/Textdatei`
[14] `http://www.cplusplus.com/forum/general/21018/`
[15] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/Simple_FileIO.zip`
[16] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS20/file_io.zip`
[17] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf`

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