

Prozedurale Programmierung - TU Freiberg

https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/

andre-dietrich

JayTee42

SebastianZug
Lorcc

galinarudolf
Anjuschenka

DkPepper

Lalelele

Inhaltsverzeichnis

1	Einführung	7
1.1	Umfrage	7
1.2	Wie arbeitet ein Rechner eigentlich?	7
1.2.1	Programmierung	10
1.2.2	Einordnung von C und C++	11
1.3	Erstes C++ Programm	11
1.3.1	“Hello World”	11
1.3.2	Ein Wort zu den Formalien	12
1.3.3	Gute Kommentare	12
1.3.4	Schlechte Kommentare	13
1.3.5	Was tun, wenn es schief geht?	14
1.3.6	Compilerfehlermeldungen	14
1.3.7	Und wenn das Kompilieren gut geht?	14
1.4	Warum dann C++?	15
1.5	Beispiele der Woche	15
2	Grundlagen der Sprache C	17
2.1	Variablen	17
2.1.1	Zulässige Variablennamen	18
2.1.2	Datentypen	19
2.1.3	Wertspezifikation	25
2.1.4	Adressen	26
2.1.5	Sichtbarkeit und Lebensdauer von Variablen	26
2.1.6	Definition vs. Deklaration vs. Initialisierung	27
2.1.7	Typische Fehler	27
2.2	Ein- und Ausgabe	28
2.2.1	Ausgabe	29
2.2.2	Eingabe	31
2.2.3	Beispiel der Woche	32
3	Operatoren & Kontrollstrukturen	33
3.1	Operatoren	33
4	Unterscheidungsmerkmale	35
4.0.1	Zuweisungsoperator	35
4.0.2	Inkrement und Dekrement	36
4.0.3	Arithmetische Operatoren	36
4.0.4	Vergleichsoperatoren	37
4.0.5	Logische Operatoren	37
4.0.6	sizeof - Operator	38
4.1	Vorrangregeln	38
4.2	... und mal praktisch	39
5	Kontrollfluss	41
5.0.1	Verzweigungen	42
5.0.2	Schleifen	47
5.0.3	Kontrolliertes Verlassen der Anweisungen	50
5.1	Beispiel des Tages	51

6	Grundlagen der Sprache C++	53
6.1	Wie weit waren wir gekommen?	53
6.2	Arrays	54
6.2.1	Deklaration, Definition, Initialisierung, Zugriff	54
6.2.2	Fehlerquelle Nummer 1 - out of range	55
6.2.3	Anwendung eines eindimensionalen Arrays	55
6.2.4	Mehrdimensionale Arrays	55
6.2.5	Anwendung eines zweidimensionalen Arrays	56
6.3	Sonderfall Zeichenketten / Strings	57
6.4	Grundkonzept Zeiger	58
6.4.1	Definition von Zeigern	58
6.4.2	Initialisierung	59
6.4.3	Fehlerquellen	60
6.4.4	Dynamische Datenobjekte	61
6.5	Referenz	61
6.6	Beispiel der Woche	62
7	Grundlagen der Sprache C++	65
7.1	Einschub - Klausurhinweise	65
7.2	Motivation	66
7.2.1	Prozedurale Programmierung Ideen und Konzepte	66
7.2.2	Anwendung	67
7.3	C++ Funktionen	67
7.3.1	Funktionsdefinition	67
7.3.2	Beispiele für Funktionsdefinitionen	68
7.3.3	Aufruf der Funktion	68
7.3.4	Fehler	69
7.3.5	Funktionsdeklaration	70
7.3.6	Parameterübergabe und Rückgabewerte	71
7.3.7	Zeiger und Referenzen als Rückgabewerte	73
7.3.8	Besonderheit Arrays	74
7.3.9	main-Funktion	74
7.4	Beispiel des Tages	75
8	Objektorientierte Programmierung mit C++	77
8.1	Motivation	77
8.2	C++ - Entwicklung	79
8.3	... das kennen wir doch schon	80
8.4	Definieren von Klassen und Objekten	81
8.4.1	Objekte in Objekten	82
8.4.2	Datenkapselung	83
8.4.3	Memberfunktionen	84
8.4.4	Modularisierung unter C++	85
8.4.5	Überladung von Methoden	86
8.5	Ein Wort zur Ausgabe	87
8.6	Initialisierung/Zerstören eines Objektes	88
8.6.1	Konstruktoren	89
8.6.2	Destruktoren	91
8.7	Beispiel des Tages	91
8.8	Anwendung	92
9	Objektorientierte Programmierung mit C++	93
9.1	Rückblick	93
9.2	Operatorenüberladung	95
9.3	Motivation	95
9.3.1	Konzept	96
9.3.2	Anwendung	97
9.4	Vererbung	99
9.4.1	Motivation	99
9.4.2	Implementierung in C++	100
9.4.3	Vererbungsattribute	101

9.4.4	Überschreiben von Methoden der Basisklasse	103
9.5	Anwendungsfall	104
10	Softwareentwicklung für Microcontroller	105
10.1	Microcontroller als Datensammler	105
10.1.1	Sensoren / Aktoren	106
10.1.2	Programmiervorgang	107
10.1.3	Besonderheiten	108
10.2	Arduino Konzept	108
10.2.1	Hardware	108
10.2.2	Programmierung	109
10.2.3	Bibliotheken	109
10.2.4	Entwicklungsumgebung	110
10.2.5	Wo ist unser main()?	110
10.3	Exkurs: Serielle Schnittstelle	111
10.3.1	Schreiben	111
10.3.2	Lesen	111
10.4	Seriellen Daten in der Arduino IDE	112
10.5	Unser Controller	114
10.6	Anwendungsfall	116

Kapitel 1

Einführung

Parameter	Kursinformationen
Veranstaltung:	Vorlesung Prozedurale Programmierung / Einführung in die Informatik
Semester:	Wintersemester 2022/23
Hochschule:	Technische Universität Freiberg
Inhalte:	Vorstellung des Arbeitsprozesses
Link auf	https://github.com/TUBAF-Iff-
Repository:	LiaScript/VL_ProzeduraleProgrammierung/blob/master/00_Einfuehrung.md
Autoren	@author

Fragen an die heutige Veranstaltung ...

- Welche Aufgabe erfüllt eine Programmiersprache?
 - Erklären Sie die Begriffe Compiler, Editor, Programm, Hochsprache!
 - Was passiert beim Kompilieren eines Programmes?
 - Warum sind Kommentare von zentraler Bedeutung?
 - Worin unterscheiden sich ein konventionelles C++ Programm und eine Anwendung, die mit dem Arduino-Framework geschrieben wurde?
-

1.1 Umfrage

Hat Sie die letztwöchige Vorstellung der Ziele der Lehrveranstaltung überzeugt?

- [(ja)] Ja, ich gehe davon aus, viel nützliches zu erfahren.
- [(schau'n wir mal)] Ich bin noch nicht sicher. Fragen Sie in einigen Wochen noch mal.
- [(nein)] Nein, ich bin nur hier, weil ich muss.

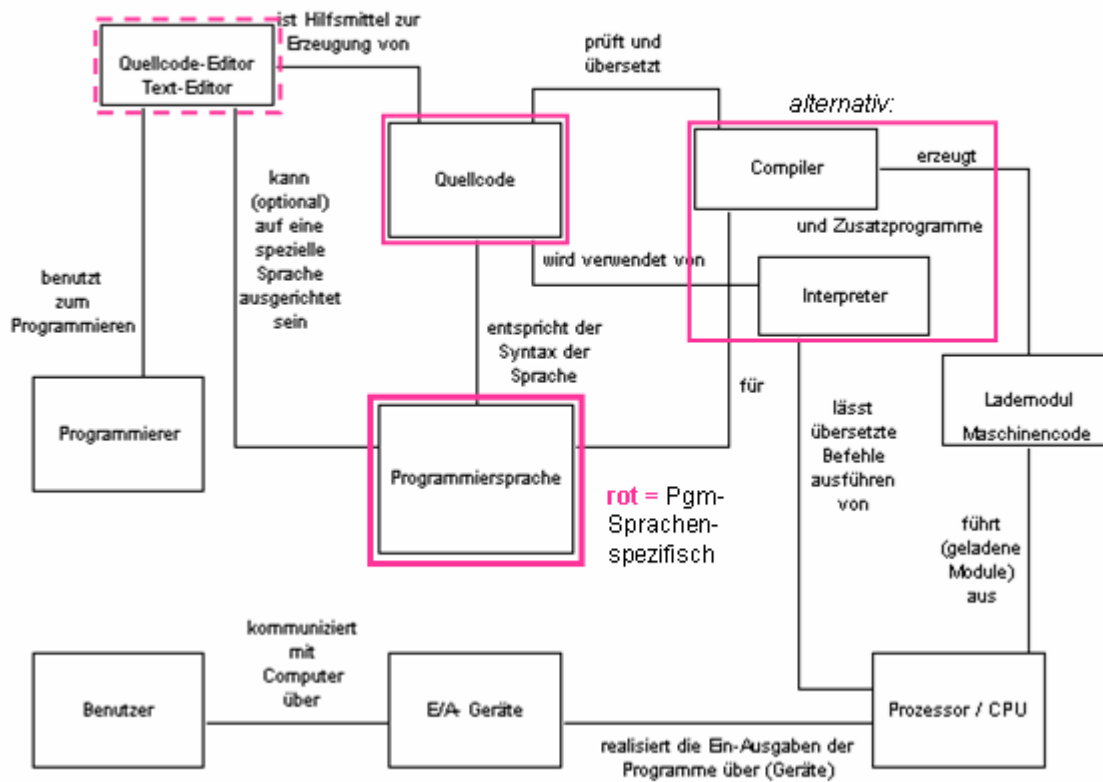
1.2 Wie arbeitet ein Rechner eigentlich?

Programme sind Anweisungslisten, die vom Menschen erdacht, auf einem Rechner zur Ausführung kommen. Eine zentrale Hürde ist dabei die Kluft zwischen menschlicher Vorstellungskraft und Logik, die **implizite Annahmen und Erfahrungen** einschließt und der **“stupiden” Abarbeitung von Befehlsfolgen** in einem Rechner.

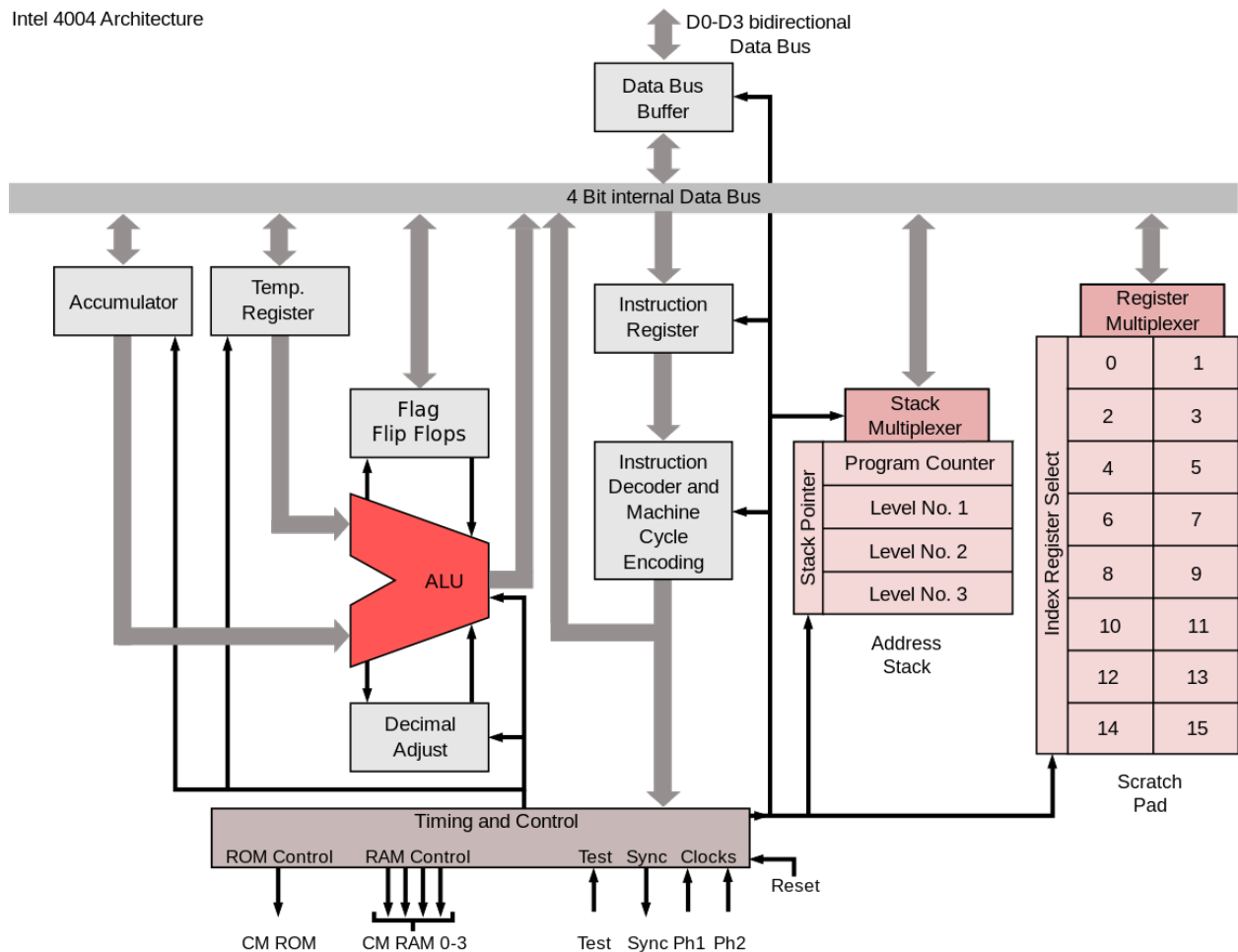
Programmiersprachen bemühen sich diese Lücke zu schließen und werden dabei von einer Vielzahl von Tools begleitet, diesen **Transformationsprozess** unterstützen sollen.

Um das Verständnis für diesen Vorgang zu entwickeln werden zunächst die Vorgänge in einem Rechner bei der Abarbeitung von Programmen beleuchtet, um dann die Realisierung eines Programmes mit C++ zu adressieren.

Programmiersprache: Vom Quellcode zur Ausführung im Prozessor



Intel 4004 Architecture



Jeder Rechner hat einen spezifischen Satz von Befehlen, die durch "0" und "1" ausgedrückt werden, die er überhaupt abarbeiten kann.

Speicherauszug den Intel 4004:

Adresse	Speicherinhalt	OpCode	Mnemonik
0010	1101 0101	1101 DDDD	LD \$5
0012	1111 0010	1111 0010	IAC

Unterstützung für die Interpretation aus dem Nutzerhandbuch, dass das *Instruction Set* beschreibt:

4004 Instruction Set

BASIC INSTRUCTIONS (* = 2 Word Instructions)

Hex Code	MNEMONIC	OPR D ₃ D ₂ D ₁ D ₀	OPA D ₃ D ₂ D ₁ D ₀	DESCRIPTION OF OPERATION
00	NOP	0 0 0 0	0 0 0 0	No operation.
1 - ..	*JCN	0 0 0 1 A ₂ A ₂ A ₂ A ₂	C ₁ C ₂ C ₃ C ₄ A ₁ A ₁ A ₁ A ₁	Jump to ROM address A ₂ A ₂ A ₂ A ₂ , A ₁ A ₁ A ₁ A ₁ (within the same ROM that contains this JCN instruction) if condition C ₁ C ₂ C ₃ C ₄ is true, otherwise go to the next instruction in sequence.
2 - ..	*FIM	0 0 1 0 D ₂ D ₂ D ₂ D ₂	R R R 0 D ₁ D ₁ D ₁ D ₁	Fetch immediate (direct) from ROM Data D ₂ D ₂ D ₂ D ₂ D ₁ D ₁ D ₁ D ₁ to index register pair location RRR.
■ ■ ■				
8 -	ADD	1 0 0 0	R R R R	Add contents of register RRRR to accumulator with carry.
9 -	SUB	1 0 0 1	R R R R	Subtract contents of register RRRR to accumulator with borrow.
A -	LD	1 0 1 0	R R R R	Load contents of register RRRR to accumulator.
B -	XCH	1 0 1 1	R R R R	Exchange contents of index register RRRR and accumulator.
C -	BBL	1 1 0 0	D D D D	Branch back (down 1 level in stack) and load data DDDD to accumulator.
D -	LDM	1 1 0 1	D D D D	Load data DDDD to accumulator.
F0	CLB	1 1 1 1	0 0 0 0	Clear both. (Accumulator and carry)
F1	CLC	1 1 1 1	0 0 0 1	Clear carry.
F2	IAC	1 1 1 1	0 0 1 0	Increment accumulator.

Quelle: [Intel 4004 Assembler](#)

1.2.1 Programmierung

Möchte man so Programme schreiben?

Vorteil:

- ggf. sehr effizienter Code (Größe, Ausführungsdauer), der gut auf die Hardware abgestimmt ist

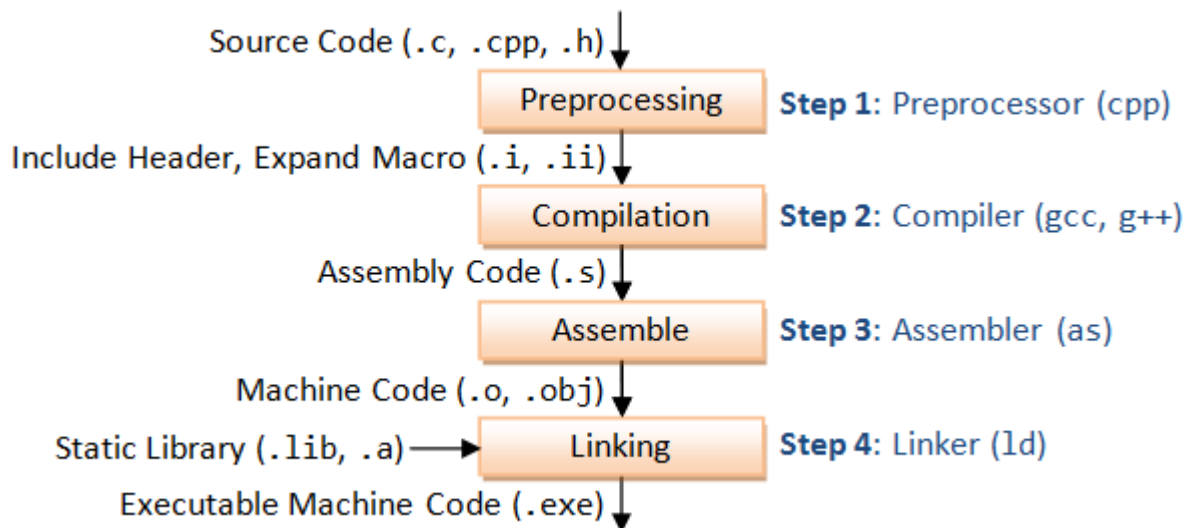
Nachteile:

- systemspezifische Realisierung
- geringer Abstraktionsgrad, bereits einfache Konstrukte benötigen viele Codezeilen
- weitgehende semantische Analysen möglich

Eine höhere Programmiersprache ist eine Programmiersprache zur Abfassung eines Computerprogramms, die in **Abstraktion und Komplexität** von der Ebene der Maschinensprachen deutlich entfernt ist. Die Befehle müssen durch **Interpreter oder Compiler** in Maschinensprache übersetzt werden.

Ein **Compiler** (auch Kompiler; von englisch für zusammentragen bzw. lateinisch compilare ‚aufhäufen‘) ist ein Computerprogramm, das Quellcodes einer bestimmten Programmiersprache in eine Form übersetzt, die von einem Computer (direkter) ausgeführt werden kann.

Stufen des Compile-Vorganges:



1.2.2 Einordnung von C und C++

- Adressiert Hochsprachenaspekte und Hardwarenähe -> Hohe Geschwindigkeit bei geringer Programmgröße
- Imperative Programmiersprache

imperative (befehlsorientierte) Programmiersprachen: Ein Programm besteht aus einer Folge von Befehlen an den Computer. Das Programm beschreibt den Lösungsweg für ein Problem (C, Python, Java, LabView, Matlab, ...).

deklarative Programmiersprachen: Ein Programm beschreibt die allgemeinen Eigenschaften von Objekten und ihre Beziehungen untereinander. Das Programm beschreibt zunächst nur das Wissen zur Lösung des Problems (Prolog, Haskell, SQL, ...).

- Wenige Schlüsselwörter als Sprachumfang

Schlüsselwort Reserviertes Wort, das der Compiler verwendet, um ein Programm zu parsen (z.B. if, def oder while). Schlüsselwörter dürfen nicht als Name für eine Variable gewählt werden

- Große Mächtigkeit

Je "höher" und komfortabler die Sprache, desto mehr ist der Programmierer daran gebunden, die in ihr vorgesehenen Wege zu beschreiten.

1.3 Erstes C++ Programm

1.3.1 "Hello World"

```

1 // That's my first C program
2 // Karl Klammer, Oct. 2022
3
4 #include <iostream>
5
6 int main() {
7     std::cout << "Hello World!";
8     return 0;
9 }
  
```

Zeile	Bedeutung
1 - 2	Kommentar (wird vom Präprozessor entfernt)
4	Voraussetzung für das Einbinden von Befehlen der Standardbibliothek hier <code>std::cout()</code>
6	Einsprungstelle für den Beginn des Programmes
6 - 9	Ausführungsbereich der <code>main</code> -Funktion

Zeile	Bedeutung
7	Anwendung eines Operators << hier zur Ausgabe auf dem Bildschirm
8	Definition eines Rückgabewertes für das Betriebssystem

Halt! Unsere C++ Arduino Programme sahen doch ganz anders aus?

```
1 void setup() {
2   pinMode(LED_BUILTIN, OUTPUT);
3 }
4
5 void loop() {
6   digitalWrite(LED_BUILTIN, HIGH);
7   delay(1000);
8   digitalWrite(LED_BUILTIN, LOW);
9   delay(1000);
10 }
```

@AVR8js.sketch

Noch mal Halt! Das klappt ja offenbar alles im Browserfenster, aber wenn ich ein Programm auf meinem Rechner kompilieren möchte, was ist dann zu tun?

1.3.2 Ein Wort zu den Formalien

```
1 // Karl Klammer
2 // Print Hello World drei mal
3
4 #include <iostream>
5
6 int main() {
7   int zahl;
8   for (zahl=0; zahl<3; zahl++){
9     std::cout << "Hello World! ";
10  }
11   return 0;
12 }
```

```
1 #include <iostream>
2 int main() {int zahl; for (zahl=0; zahl<3; zahl++){ std::cout << "Hello World! ";} return 0;}
```

- Das *systematische Einrücken* verbessert die Lesbarkeit und senkt damit die Fehleranfälligkeit Ihres Codes!
- Wählen sie *selbsterklärende Variablen- und Funktionsnamen!*
- Nutzen Sie ein *Versionsmanagementsystem*, wenn Sie ihren Code entwickeln!
- Kommentieren Sie Ihr Vorgehen trotz “Good code is self-documenting”

1.3.3 Gute Kommentare

1. Kommentare als Pseudocode

```
1 /* loop backwards through all elements returned by the server
2 (they should be processed chronologically)*/
3 for (i = (numElementsReturned - 1); i >= 0; i--){
4   /* process each element's data */
5   updatePattern(i, returnedElements[i]);
6 }
```

2. Kommentare zur Datei

```
1 // This is the mars rover control application
2 //
3 // Karl Klammer, Oct. 2018
4 // Version 109.1.12
```

```

5
6 int main(){...}

```

3. Beschreibung eines Algorithmus

```

1 /* Function: approx_pi
2  * -----
3  * computes an approximation of pi using:
4  *    $\pi/6 = 1/2 + (1/2 \times 3/4) 1/5 (1/2)^3 + (1/2 \times 3/4 \times 5/6) 1/7 (1/2)^5 +$ 
5  *
6  * n: number of terms in the series to sum
7  *
8  * returns: the approximate value of pi obtained by summing the first n terms
9  *           in the above series
10 *           returns zero on error (if n is non-positive)
11 */
12
13 double approx_pi(int n);

```

In realen Projekten werden Sie für diese Aufgaben Dokumentationstools verwenden, die die Generierung von Webseite, Handbüchern auf der Basis eigener Schlüsselworte in den Kommentaren unterstützen -> [doxygen](#).

4. Debugging

```

1 int main(){
2     ...
3     preProcessedData = filter1(rawData);
4     // printf('Filter1 finished ... \n');
5     // printf('Output %d \n', preProcessedData);
6     result=complexCalculation(preProcessedData);
7     ...
8 }

```

1.3.4 Schlechte Kommentare

1. Überkommentierung von Code

```

1 x = x + 1; /* increment the value of x */
2 std::cout << "Hello World! "; // displays Hello world

```

“... over-commenting your code can be as bad as under-commenting it”

Quelle: [C Code Style Guidelines](#)

2. “Merkwürdige Kommentare”

```

1 //When I wrote this, only God and I understood what I was doing
2 //Now, God only knows
3
4 // sometimes I believe compiler ignores all my comments
5
6 // Magic. Do not touch.
7 Hello World !Hello World !Hello World !Hello World !Hello World !Hello World !Hello World
8 // I am not responsible of this code.
9
10 try {
11
12 } catch(e) {
13
14 } finally { // should never happen }

```

[Sammlung von Kommentaren](#)

1.3.5 Was tun, wenn es schief geht?

```

1 // Karl Klammer
2 // Print Hello World drei mal
3
4 #include <iostream>
5
6 int main() {
7     for (zahl=0; zahl<3; zahl++){
8         std::cout << "Hello World! "
9     }
10    return 0;

```

Methodisches Vorgehen:

- **** RUHE BEWAHREN ****
- Lesen der Fehlermeldung
- Verstehen der Fehlermeldung / Aufstellen von Hypothesen
- Systematische Evaluation der Thesen
- Seien Sie im Austausch mit anderen (Kommilitonen, Forenbesucher, usw.) konkret

1.3.6 Compilerfehlermeldungen

Beispiel 1

```

1 #include <iostream>
2
3 int mani() {
4     std::cout << "Hello World!";
5     return 0;
6 }

```

Beispiel 2

```

1 #include <iostream>
2
3 int main()
4     std::cout << "Hello World!";
5     return 0;
6 }

```

Manchmal muss man sehr genau hinschauen, um zu verstehen, warum ein Programm nicht funktioniert. Versuchen Sie es!

```

1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello World!";
5     std::cout << "Wo liegt der Fehler?";
6     return 0;
7 }

```

1.3.7 Und wenn das Kompilieren gut geht?

... dann bedeutet es noch immer nicht, dass Ihr Programm wie erwartet funktioniert.

```

1 #include <iostream>
2
3 int main() {
4     char zahl;
5     for (zahl=250; zahl<256; zahl++){
6         std::cout << "Hello World!";
7     }
8     return 0;
9 }

```

Hinweis: Die Datentypen werden wir in der nächsten Woche besprechen.

1.4 Warum dann C++?

Zwei Varianten der Umsetzung ... C++ vs. Python

```
1 #include <iostream>
2
3 int main() {
4     char zahl;
5     for (int zahl=0; zahl<3; zahl++){
6         std::cout << "Hello World! " << zahl << "\n";
7     }
8     return 0;
9 }
```

```
1 for i in range(3):
2     print("Hallo World ", i)
```

1.5 Beispiele der Woche

Gültiger C++ Code

```
1 #include <iostream>
2
3 int main() {
4     int i = 5;
5     int j = 4;
6     i = i + j + 2;
7     std::cout << "Hello World ";
8     std::cout << i << "!";
9     return 0;
10 }
```

Umfrage: Welche Ausgabe erwarten Sie für folgendes Code-Schnippelchen?

- [(Hello World5)] Hello World7
- [(Hello World11)] Hello World11
- [(Hello World 11!)] Hello World 11!
- [(Hello World 11 !)] Hello World 11 !
- [(Hello World 5 !)] Hello World 5 !

Algorithmisches Denken

Aufgabe: Ändern Sie den Code so ab, dass das die LED zwei mal mit 1 Hz blinkt und dann ausgeschaltet bleibt.

```
1 void setup() {
2     pinMode(LED_BUILTIN, OUTPUT);
3 }
4
5 void loop() {
6     digitalWrite(LED_BUILTIN, HIGH);
7     delay(1000);
8     digitalWrite(LED_BUILTIN, LOW);
9     delay(1000);
10 }
```

@AVR8js.sketch

Kapitel 2

Grundlagen der Sprache C

Parameter	Kursinformationen
-----------	-------------------

Veranstaltung: Einführung in das wissenschaftliche Programmieren

Semester: Wintersemester 2022/23

Hochschule: Technische Universität Freiberg

Inhalte: Ein- und Ausgabe / Variablen

Link auf [https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/](https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/01_EingabeAusgabeDatentypen.md)

Repository: 01_EingabeAusgabeDatentypen.md

Autoren @author

Fragen an die heutige Veranstaltung ...

- Durch welche Parameter ist eine Variable definiert?
- Erklären Sie die Begriffe Deklaration, Definition und Initialisierung im Hinblick auf eine Variable!
- Worin unterscheidet sich die Zahldarstellung von ganzen Zahlen (`int`) von den Gleitkommazahlen (`float`).
- Welche Datentypen kennt die Sprache C++?
- Erläutern Sie für `char` und `int` welche maximalen und minimalen Zahlenwerte sich damit angeben lassen.
- Ist `printf` ein Schlüsselwort der Programmiersprache C++?
- Welche Beschränkung hat `getchar`

Vorwarnung: Man kann Variablen nicht ohne Ausgaben und Ausgaben nicht ohne Variablen erklären. Deshalb wird im folgenden immer wieder auf einzelne Aspekte vorgegriffen. Nach der Vorlesung sollte sich dann aber ein Gesamtbild ergeben.

2.1 Variablen

Lassen sie uns den Rechner als Rechner benutzen ... und die Lösungen einer quadratischen Gleichung bestimmen:

$$y = 3x^2 + 4x + 8$$

```
1 #include<iostream>
2
3 int main() {
4     // Variante 1 - ganz schlecht
5     std::cout <<"f("<<x<<" = "<<3*5*5 + 4*5 + 8<<" \n";
6
7     // Variante 2 - besser
8     int x = 9;
```

```

9  std::cout <<"f("<<x<<" ) = "<<3*x*x + 4*x + 8<<" \n";
10  return 0;
11 }

```

Unbefriedigende Lösung, jede neue Berechnung muss in den Source-Code integriert und dieser dann kompiliert werden. Ein Taschenrechner wäre die bessere Lösung!

Ein Programm manipuliert Daten, die in Variablen organisiert werden.

Eine Variable ist ein **abstrakter Behälter** für Inhalte, welche im Verlauf eines Rechenprozesses benutzt werden. Im Normalfall wird eine Variable im Quelltext durch einen Namen bezeichnet, der die Adresse im Speicher repräsentiert. Alle Variablen müssen vor Gebrauch vereinbart werden.

Kennzeichen einer Variable:

1. Name
2. Datentyp
3. Wert
4. Adresse
5. Gültigkeitsraum

Mit `const` kann bei einer Vereinbarung der Variable festgelegt werden, dass ihr Wert sich nicht ändert.

```

1  const double e = 2.71828182845905;

```

Ein weiterer Typqualifikator ist `volatile`. Er gibt an, dass der Wert der Variable sich jederzeit z. B. durch andere Prozesse ändern kann.

2.1.1 Zulässige Variablennamen

Der Name kann Zeichen, Ziffern und den Unterstrich enthalten. Dabei ist zu beachten:

- Das erste Zeichen muss ein Buchstabe sein, der Unterstrich ist auch möglich.
- C++ betrachte Groß- und Kleinschreibung - `Zahl` und `zahl` sind also unterschiedliche Variablennamen.
- Schlüsselworte (`class`, `for`, `return`, etc.) sind als Namen unzulässig.

Name	Zulässigkeit
<code>gcc</code>	erlaubt
<code>a234a_xwd3</code>	erlaubt
<code>speed_m_per_s</code>	erlaubt
<code>double</code>	nicht zulässig (Schlüsselwort)
<code>robot.speed</code>	nicht zulässig (. im Namen)
<code>3thName</code>	nicht zulässig (Ziffer als erstes Zeichen)
<code>x y</code>	nicht zulässig (Leerzeichen im Variablennamen)

```

1  #include<iostream>
2
3  int main() {
4      int x = 5;
5      std::cout<<"Unsere Variable hat den Wert "<<x<<" \n";
6      return 0;
7  }

```

Vergeben Sie die Variablennamen mit Sorgfalt. Für jemanden der Ihren Code liest, sind diese wichtige Informationsquellen! [Link](#)

Neben der Namensgebung selbst unterstützt auch eine entsprechende Notationen die Lesbarkeit. In Programmen sollte ein Format konsistent verwendet werden.

Bezeichnung	denkbare Variablennamen
CamelCase (upperCamel)	<code>YouLikeCamelCase</code> , <code>HumanDetectionSuccessfull</code>
(lowerCamel)	<code>youLikeCamelCase</code> , <code>humanDetectionSuccessfull</code>
underscores	<code>I_hate_Camel_Case</code> , <code>human_detection_successfull</code>

In der Vergangenheit wurden die Konventionen (zum Beispiel durch Microsoft “Ungarische Notation”) verpflichtend durchgesetzt. Heute dienen eher die generellen Richtlinien des Clean Code in Bezug auf die Namensgebung.

2.1.2 Datentypen

Welche Informationen lassen sich mit Blick auf einen Speicherauszug im Hinblick auf die Daten extrahieren?

Adresse | Speicherinhalt |

| binär |

0010 | 0000 1100 |

0011 | 1111 1101 |

0012 | 0001 0000 |

0013 | 1000 0000 |

Adresse | Speicherinhalt | Zahlenwert |

| (Byte) |

0010 | 0000 1100 | 12 |

0011 | 1111 1101 | 253 (-3) |

0012 | 0001 0000 | 16 |

0013 | 1000 0000 | 128 (-128) |

Adresse | Speicherinhalt | Zahlenwert | Zahlenwert | Zahlenwert |

| (Byte) | (2 Byte) | (4 Byte) |

0010 | 0000 1100 | 12 | |

0011 | 1111 1101 | 253 (-3) | 3325 | |

0012 | 0001 0000 | 16 | |

0013 | 1000 0000 | 128 (-128) | 4224 | 217911424 |

Der dargestellte Speicherauszug kann aber auch eine Kommazahl (Floating Point) umfassen und repräsentiert dann den Wert 3.8990753E-31

Folglich bedarf es eines expliziten Wissens um den Charakter der Zahl, um eine korrekte Interpretation zu ermöglichen. Dabei erfolgt die Einteilung nach:

- Wertebereichen (größte und kleinste Zahl)
- ggf. vorzeichenbehaftet Zahlen
- ggf. gebrochene Werte

2.1.2.1 Ganze Zahlen, char und bool

Ganzzahlen sind Zahlen ohne Nachkommastellen mit und ohne Vorzeichen. In C/C++ gibt es folgende Typen für Ganzzahlen:

Schlüsselwort	Benutzung	Mindestgröße
<code>char</code>	1 Byte bzw. 1 Zeichen	1 Byte (min/max)
<code>short int</code>	Ganzzahl (ggf. mit Vorzeichen)	2 Byte
<code>int</code>	Ganzzahl (ggf. mit Vorzeichen)	“natürliche Größe”
<code>long int</code>	Ganzzahl (ggf. mit Vorzeichen)	
<code>long long int</code>	Ganzzahl (ggf. mit Vorzeichen)	
<code>bool</code>	boolsche Variable	1 Byte

```
1 signed char <= short <= int <= long <= long long
```

Gängige Zuschnitte für `char` oder `int`

Schlüsselwort	Wertebereich
<code>signed char</code>	-128 bis 127
<code>char</code>	0 bis 255 (0xFF)
<code>signed int</code>	-32768 bis 32767
<code>int</code>	65536 (0xFFFF)

Wenn die Typ-Spezifizierer (`long` oder `short`) vorhanden sind kann auf die `int` Typangabe verzichtet werden.

```
1 short int a; // entspricht short a;  
2 long int b; // äquivalent zu long b;
```

Standardmäßig wird von vorzeichenbehafteten Zahlenwerten ausgegangen. Somit wird das Schlüsselwort `signed` eigentlich nicht benötigt

```
1 int a; // signed int a;  
2 unsigned long long int b;
```

2.1.2.2 Sonderfall char

Für den Typ `char` ist der mögliche Gebrauch und damit auch die Vorzeichenregel zwiespältig:

- Wird `char` dazu verwendet einen **numerischen Wert** zu speichern und die Variable nicht explizit als vorzeichenbehaftet oder vorzeichenlos vereinbart, dann ist es implementierungsabhängig, ob `char` vorzeichenbehaftet ist oder nicht.
- Wenn ein Zeichen gespeichert wird, so garantiert der Standard, dass der gespeicherte Wert der nicht negativen Codierung im **Zeichensatz** entspricht.

```
1 char c = 'M'; // = 1001101 (ASCII Zeichensatz)  
2 char c = 77; // = 1001101  
3 char s[] = "Eine kurze Zeichenkette";
```

Achtung: Anders als bei einigen anderen Programmiersprachen unterscheidet C/C++ zwischen den verschiedenen Anführungsstrichen.

Scan- code	ASCII hex dez	Zeichen	Scan- code	ASCII hex dez	Zch.	Scan- code	ASCII hex dez	Zch.	Scan- code	ASCII hex dez	Zch.
	00 0	NUL ^@		20 32	SP		40 64	@	0D	60 96	`
	01 1	SOH ^A	02	21 33	!	1E	41 65	A	1E	61 97	a
	02 2	STX ^B	03	22 34	"	30	42 66	B	30	62 98	b
	03 3	ETX ^C	29	23 35	#	2E	43 67	C	2E	63 99	c
	04 4	EOT ^D	05	24 36	\$	20	44 68	D	20	64 100	d
	05 5	ENQ ^E	06	25 37	%	12	45 69	E	12	65 101	e
	06 6	ACK ^F	07	26 38	&	21	46 70	F	21	66 102	f
	07 7	BEL ^G	0D	27 39	'	22	47 71	G	22	67 103	g
0E	08 8	BS ^H	09	28 40	(23	48 72	H	23	68 104	h
0F	09 9	TAB ^I	0A	29 41)	17	49 73	I	17	69 105	i
	0A 10	LF ^J	1B	2A 42	*	24	4A 74	J	24	6A 106	j
	0B 11	VT ^K	1B	2B 43	+	25	4B 75	K	25	6B 107	k
	0C 12	FF ^L	33	2C 44	,	26	4C 76	L	26	6C 108	l
1C	0D 13	CR ^M	35	2D 45	-	32	4D 77	M	32	6D 109	m
	0E 14	SO ^N	34	2E 46	.	31	4E 78	N	31	6E 110	n
	0F 15	SI ^O	08	2F 47	/	18	4F 79	O	18	6F 111	o
	10 16	DLE ^P	0B	30 48	0	19	50 80	P	19	70 112	p
	11 17	DC1 ^Q	02	31 49	1	10	51 81	Q	10	71 113	q
	12 18	DC2 ^R	03	32 50	2	13	52 82	R	13	72 114	r
	13 19	DC3 ^S	04	33 51	3	1F	53 83	S	1F	73 115	s
	14 20	DC4 ^T	05	34 52	4	14	54 84	T	14	74 116	t
	15 21	NAK ^U	06	35 53	5	16	55 85	U	16	75 117	u
	16 22	SYN ^V	07	36 54	6	2F	56 86	V	2F	76 118	v
	17 23	ETB ^W	08	37 55	7	11	57 87	W	11	77 119	w
	18 24	CAN ^X	09	38 56	8	2D	58 88	X	2D	78 120	x
	19 25	EM ^Y	0A	39 57	9	2C	59 89	Y	2C	79 121	y
	1A 26	SUB ^Z	34	3A 58	:	15	5A 90	Z	15	7A 122	z
01	1B 27	Esc ^[33	3B 59	;		5B 91	[7B 123	{
	1C 28	FS ^\	2B	3C 60	<		5C 92	\		7C 124	
	1D 29	GS ^]	0B	3D 61	=		5D 93]		7D 125	}
	1E 30	RS ^^	2B	3E 62	>	29	5E 94	^		7E 126	~
	1F 31	US ^_	0C	3F 63	?	35	5F 95	_	53	7F 127	DEL

2.1.2.3 Sonderfall bool

Auf die Variablen von Datentyp `bool` können Werte `true` (1) und `false` (0) gespeichert werden. Eine implizite Umwandlung der ganzen Zahlen zu den Werten 0 und 1 ist ebenfalls möglich.

```

1 #include <iostream>
2
3 int main() {
4     bool a = true;
5     bool b = false;
6     bool c = 45;
7
8     std::cout<<"a = "<<a<<" b = "<<b<<" c = "<<c<<"\n";
9     return 0;
10 }
```

Sinnvoll sind boolsche Variablen insbesondere im Kontext von logischen Ausdrücken. Diese werden zum späteren Zeitpunkt eingeführt.

2.1.2.4 Architekturspezifische Ausprägung (Integer Datentypen)

Der Operator `sizeof` gibt Auskunft über die Größe eines Datentyps oder einer Variablen in Byte.

```

1 #include <iostream>
2
3 int main(void)
4 {
5     int x;
6     std::cout<<"x umfasst " <<sizeof(x)<<" Byte.";
7     return 0;
8 }

1 #include <iostream>
2 #include <limits.h>    /* INT_MIN und INT_MAX */
3
4 int main(void) {
5     std::cout<<"int size: "<< sizeof(int)<<" Byte\n";
6     std::cout<<"Wertebereich von "<< INT_MIN<<" bis "<< INT_MAX<<"\n";
7     std::cout<<"char size : "<< sizeof(char) <<" Byte\n";
8     std::cout<<"Wertebereich von "<< CHAR_MIN<<" bis "<<CHAR_MAX<<"\n";
9     return 0;
10 }

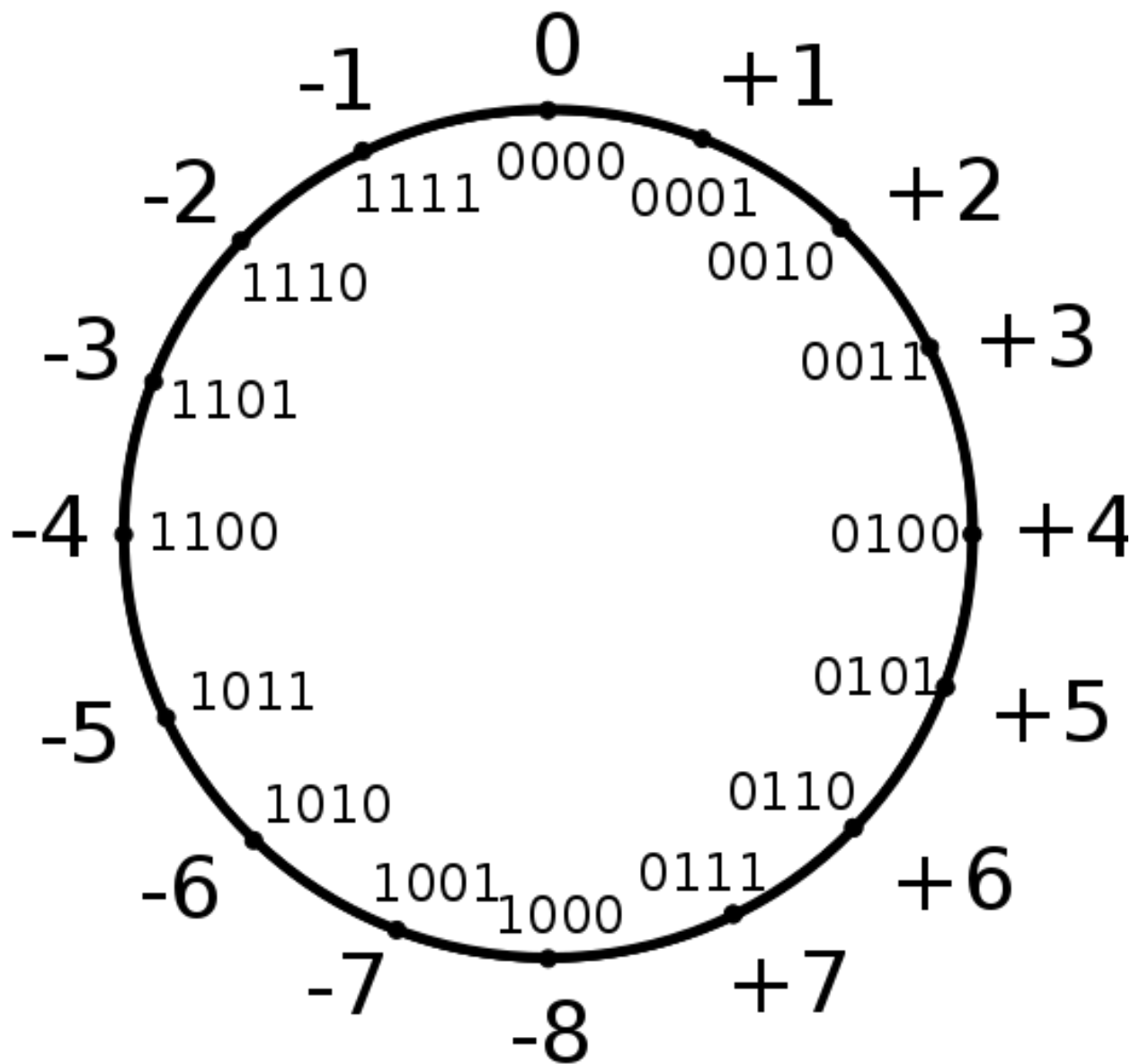
```

Die implementierungsspezifische Werte, wie die Grenzen des Wertebereichs der ganzzahligen Datentypen sind in `limits.h` definiert, z.B.

Makro	Wert
CHAR_MIN	-128
CHAR_MAX	+127
SHRT_MIN	-32768
SHRT_MAX	+32767
INT_MIN	-2147483648
INT_MAX	+2147483647
LONG_MIN	-9223372036854775808
LONG_MAX	+9223372036854775807

2.1.2.5 Was passiert bei der Überschreitung des Wertebereiches

Der Arithmetische Überlauf (arithmetic overflow) tritt auf, wenn das Ergebnis einer Berechnung für den gültigen Zahlenbereich zu groß ist, um noch richtig interpretiert werden zu können.



Quelle: [Arithmetischer Überlauf](#) (Autor: WissensDürster)

```

1 #include <iostream>
2 #include <limits.h>    /* SHRT_MIN und SHRT_MAX */
3
4 int main(){
5     short a = 30000;
6
7     std::cout<<"Berechnung von 30000+3000 mit:\n\n";
8
9     signed short c;    // -32768 bis 32767
10    std::cout<<"(signed) short c - Wertebereich von "<<SHRT_MIN<<" bis "<<SHRT_MAX<<"\n";
11    c = 3000 + a;      // ÜBERLAUF!
12    std::cout<<"c="<<c<<"\n";
13
14    unsigned short d;  //      0 bis 65535
15    std::cout<<"unsigned short d - Wertebereich von "<<0<<" bis "<<USHRT_MAX<<"\n";
16
17    d = 3000 + a;
18    std::cout<<"d="<<d<<"\n";
19 }

```

Ganzzahlüberläufe in der fehlerhaften Bestimmung der Größe eines Puffers oder in der Adressierung eines Feldes können es einem Angreifer ermöglichen den Stack zu überschreiben.

2.1.2.6 Fließkommazahlen

Fließkommazahlen sind Zahlen mit Nachkommastellen (reelle Zahlen). Im Gegensatz zu Ganzzahlen gibt es bei den Fließkommazahlen keinen Unterschied zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen. Alle Fließkommazahlen sind in C/C++ immer vorzeichenbehaftet.

In C/C++ gibt es zur Darstellung reeller Zahlen folgende Typen:

Schlüsselwort	Mindestgröße
<code>float</code>	4 Byte
<code>double</code>	8 Byte
<code>long double</code>	je nach Implementierung

```
1 float <= double <= long double
```

Gleitpunktzahlen werden halb logarithmisch dargestellt. Die Darstellung basiert auf die Zerlegung in drei Teile: ein Vorzeichen, eine Mantisse und einen Exponenten zur Basis 2.

Zur Darstellung von Fließkommazahlen sagt der C/C++-Standard nichts aus. Zur konkreten Realisierung ist die Headerdatei `float.h` auszuwerten.

	<code>float</code>	<code>double</code>
kleinste positive Zahl	1.1754943508e-38	2.2250738585072014E-308
Wertebereich	$\pm 3.4028234664e+38$	$\pm 1.7976931348623157E+308$

Achtung: Fließkommazahlen bringen einen weiteren Faktor mit - die Unsicherheit

```
1 #include<iostream>
2 #include<float.h>
3
4 int main(void) {
5     std::cout<<"float Genauigkeit : "<<FLT_DIG<<" \n";
6     std::cout<<"double Genauigkeit : "<<DBL_DIG<<" \n";
7     float x = 0.1;
8     if (x == 0.1) { // <- das ist ein double "0.1"
9         //if (x == 0.1f) { // <- das ist ein float "0.1"
10        std::cout<<"Gleich\n";
11    }else{
12        std::cout<<"Ungleich\n";
13    }
14    return 0;
15 }
```

Potenzen von 2 (zum Beispiel $2^{-3} = 0.125$) können im Unterschied zu 0.1 präzise im Speicher abgebildet werden. Können Sie erklären?

2.1.2.7 Datentyp void

`void` wird als „unvollständiger Typ“ bezeichnet, umfasst eine leere Wertemenge und wird verwendet überall dort, wo kein Wert vorhanden oder benötigt wird.

Anwendungsbeispiele:

- Rückgabewert einer Funktion
- Parameter einer Funktion
- anonymer Zeigertyp `void*`

```
1 int main(void) {
2     //Anweisungen
```



```

3  return 0;
4  }

1  void funktion(void) {
2      //Anweisungen
3  }

```

2.1.3 Wertspezifikation

Zahlenliterale können in C/C++ mehr als Ziffern umfassen!

Gruppe	zulässige Zeichen
<i>decimal-digits</i>	0 1 2 3 4 5 6 7 8 9
<i>octal-prefix</i>	0
<i>octal-digits</i>	0 1 2 3 4 5 6 7
<i>hexadecimal-prefix</i>	0x 0X
<i>hexadecimal-digits</i>	0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<i>unsigned-suffix</i>	u U
<i>long-suffix</i>	l L
<i>long-long-suffix</i>	ll LL
<i>fractional-constant</i>	.
<i>exponent-part</i>	e E
<i>binary-exponent-part</i>	p P
<i>sign</i>	+ -
<i>floating-suffix</i>	f l F L

Zahlentyp	Dezimal	Oktal	Hexadezimal
Eingabe	x	x	x
Ausgabe	x	x	x
Beispiel	12	011	0x12
	0.123		0X1a
	123e-2		0xC.68p+2
	1.23F		

Erkennen Sie jetzt die Bedeutung der Compilerfehlermeldung `error: invalid suffix "abc" on integer constant` aus dem ersten Beispiel der Vorlesung?

Variable = (Vorzeichen) (Zahlensystem) [Wert] (Typ);

Literal	Bedeutung
12	Ganzzahl vom Typ <code>int</code>
-234L	Ganzzahl vom Typ <code>signed long</code>
100000000000	Ganzzahl vom Typ <code>long</code>
011	Ganzzahl also oktale Zahl (Wert 9_d)
0x12	Ganzzahl (18_d)
1.23F	Fließkommazahl vom Typ <code>float</code>
0.132	Fließkommazahl vom Typ <code>double</code>
123e-2	Fließkommazahl vom Typ <code>double</code>
0xC.68p+2	hexadizimale Fließkommazahl vom Typ <code>double</code>

```

1  #include<iostream>
2
3  int main(void)
4  {
5      int x=020;

```

```

6  int y=0x20;
7  std::cout<<"x = "<<x<<"\n";
8  std::cout<<"y = "<<y<<"\n";
9  std::cout<<"Rechnen mit Oct und Hex x + y = "<< x + y;
10 return 0;
11 }

```

2.1.4 Adressen

Merke: Einige Anweisungen in C/C++ verwenden Adressen von Variablen.

Jeder Variable in C++ wird eine bestimmten Position im Hauptspeicher zugeordnet. Diese Position nennt man Speicheradresse. Solange eine Variable gültig ist, bleibt sie an dieser Stelle im Speicher. Um einen Zugriff auf die Adresse einer Variablen zu haben, kann man den Operator & nutzen.

```

1 #include <iostream>
2
3 int main(void)
4 {
5     int x=020;
6     std::cout<<&x<<"\n";
7     return 0;
8 }

```

2.1.5 Sichtbarkeit und Lebensdauer von Variablen

Lokale Variablen

Variablen *leben* innerhalb einer Funktion, einer Schleife oder einfach nur innerhalb eines durch geschwungene Klammern begrenzten Blocks von Anweisungen von der Stelle ihrer Definition bis zum Ende des Blocks. Beachten Sie, dass die Variable vor der ersten Benutzung vereinbart werden muss.

Wird eine Variable/Konstante z. B. im Kopf einer Schleife vereinbart, gehört sie zu dem Block, in dem auch der Code der Schleife steht. Folgender Codeausschnitt soll das verdeutlichen:

```

1 #include<iostream>
2
3 int main(void)
4 {
5     int v = 1;
6     int w = 5;
7     {
8         int v;
9         v = 2;
10        std::cout<<v<<"\n";
11        std::cout<<w<<"\n";
12    }
13    std::cout<<v<<"\n";
14    return 0;
15 }

```

Globale Variablen

Muss eine Variable immer innerhalb von `main` definiert werden? Nein, allerdings sollten globale Variablen vermieden werden.

```

1 #include<iostream>
2
3 int v = 1; /*globale Variable*/
4
5 int main(void)
6 {
7     std::cout<<v<<"\n";
8     return 0;

```

```
9 }
```

Sichtbarkeit und Lebensdauer spielen beim Definieren neuer Funktionen eine wesentliche Rolle und werden in einer weiteren Vorlesung in diesem Zusammenhang nochmals behandelt.

2.1.6 Definition vs. Deklaration vs. Initialisierung

... oder andere Frage, wie kommen Name, Datentyp, Adresse usw. zusammen?

Deklaration ist nur die Vergabe eines Namens und eines Typs für die Variable. Definition ist die Reservierung des Speicherplatzes. Initialisierung ist die Zuweisung eines ersten Wertes.

Merke: Jede Definition ist gleichzeitig eine Deklaration aber nicht umgekehrt!

```
1 extern int a;           // Deklaration
2 int i;                  // Definition + Deklaration
3 int a,b,c;
4 int i = 5;              // Definition + Deklaration + Initialisierung
```

Das Schlüsselwort `extern` in obigem Beispiel besagt, dass die Definition der Variablen `a` irgendwo in einem anderen Modul des Programms liegt. So deklariert man Variablen, die später beim Binden (Linken) aufgelöst werden. Da in diesem Fall kein Speicherplatz reserviert wurde, handelt es sich um keine Definition.

2.1.7 Typische Fehler

Fehlende Initialisierung

```
1 #include<iostream>
2
3 int main(void) {
4     int x = 5;
5     std::cout<<"x="<<x<<"\n";
6     int y;           // <- Fehlende Initialisierung
7     std::cout<<"y="<<y<<"\n";
8     return 0;
9 }
```

Redeklaration

```
1 #include<iostream>
2
3 int main(void) {
4     int x;
5     int x;
6     return 0;
7 }
```

Falsche Zahlenlitterale

```
1 #include<iostream>
2
3 int main(void) {
4     float a=1,5;      /* FALSCH */
5     float b=1.5;      /* RICHTIG */
6     return 0;
7 }
```

Was passiert wenn der Wert zu groß ist?

```
1 #include<iostream>
2
3 int main(void) {
4     short a;
5     a = 0xFFFF + 2;
6     std::cout<<"Schaun wir mal ... "<<a<<"\n";
```

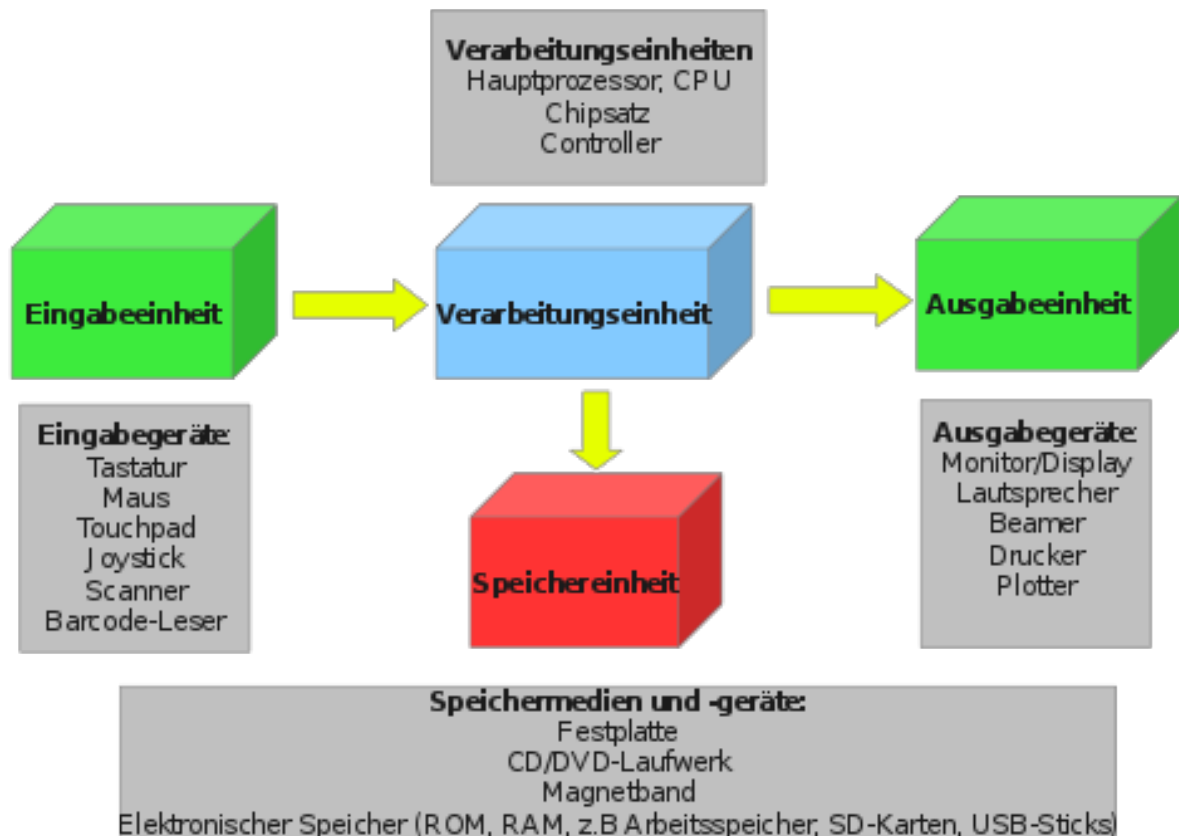
```

7  return 0;
8  }

```

2.2 Ein- und Ausgabe

Ausgabefunktionen wurden bisher genutzt, um den Status unserer Programme zu dokumentieren. Nun soll dieser Mechanismus systematisiert und erweitert werden.



Quelle: EVA-Prinzip (Autor: Deadlyhappen)

Für Ein- und Ausgabe stellt C++ das Konzept der Streams bereit, dass nicht nur für elementare Datentypen gilt, sondern auch auf die neu definierten Datentypen (Klassen) erweitert werden kann. Unter Stream wird eine Folge von Bytes verstanden.

Als Standard werden verwendet:

- `std::cin` für die Standardeingabe (Tastatur),
- `std::cout` für die Standardausgabe (Console) und
- `std::cerr` für die Standardfehlerausgabe (Console)

Achtung: Das `std::` ist ein zusätzlicher Indikator für eine bestimmte Implementierung, ein sogenannter Namespace. Um sicherzustellen, dass eine spezifische Funktion, Datentyp etc. genutzt wird stellt man diese Bezeichnung dem verwendeten Element zuvor. Mit `using namespace std;` kann man die permanente Nennung umgehen.

Stream-Objekte werden durch `#include <iostream>` bekannt gegeben. Definiert werden sie als Komponente der Standard Template Library (STL) im Namensraum `std`.

Mit Namensräumen können Deklarationen und Definitionen unter einem Namen zusammengefasst und gegen andere Namen abgegrenzt werden.

```

1  #include <iostream>
2
3  int main(void) {
4      char hanna[]="Hanna";

```

```

5 char anna[]="Anna";
6 std::cout << "C++ stream: " << "Hallo " << hanna << ", " << anna <<std::endl;
7 return 0;
8 }

```

2.2.1 Ausgabe

Der Ausgabeoperator << formt automatisch die Werte der Variablen in die Textdarstellung der benötigten Weite um. Der Rückgabewert des Operators ist selbst ein Stream-Objekt (Referenz), so dass ein weiterer Ausgabeoperator darauf angewendet werden kann. Damit ist die Hintereinanderschaltung von Ausgabeoperatoren möglich.

```
1 std::cout<<55<<"55"<<55.5<<true;
```

Welche Formatierungsmöglichkeiten bietet der Ausgabeoperator noch?

Mit Hilfe von in <iomanip> definierten Manipulatoren können besondere Ausgabeformatierungen erreicht werden.

Manipulator	Bedeutung
<code>setbase(int B)</code>	Basis 8, 10 oder 16 definieren
<code>setfill(char c)</code>	Füllzeichen festlegen
<code>setprecision(int n)</code>	Flieskommaprezession
<code>setw(int w)</code>	Breite setzen

```

1 #include <iostream>
2 #include <iomanip>
3
4 int main(){
5     std::cout<<std::setbase(16)<< std::fixed<<55<<std::endl;
6     std::cout<<std::setbase(10)<< std::fixed<<55<<std::endl;
7     return 0;
8 }

```

Achtung: Die Manipulatoren wirken auf alle darauf folgenden Ausgaben.

2.2.1.1 Feldbreite

Die Feldbreite definiert die Anzahl der nutzbaren Zeichen, sofern diese nicht einen größeren Platz beanspruchen.

Der Manipulator `right` sorgt im Beispiel für eine rechtsbündige Ausrichtung der Ausgabe, wegen `setw(5)` ist die Ausgabe fünf Zeichen breit, wegen `setfill('0')` werden nicht benutzte Stellen mit dem Zeichen 0 aufgefüllt, `endl` bewirkt die Ausgabe eines Zeilenumbruchs.

```

1 #include <iostream>
2 #include <iomanip>
3 int main(){
4
5     std::cout<<std::right<< std::setw(5)<<55<<std::endl;
6     std::cout<<std::right<< std::setfill('0')<<std::setw(5)<<55<<std::endl;
7     std::cout<<std::left<< std::fixed<<std::setw(5)<<55<<std::endl;
8     std::cout<<std::setw(5)<<"Zu klein gedacht: "<<234534535<<std::endl;
9     return 0;
10 }

```

2.2.1.2 Genauigkeit

```

1 #include <iostream>
2 #include <iomanip>
3 #include <math.h>
4

```

```

5 int main() {
6     for (int i = 12; i > 1; i -=3) {
7         std::cout << std::setprecision(i) << std::fixed << M_PI << std::endl;
8     }
9 }

```

2.2.1.3 Escape-Sequenzen

Sequenz	Bedeutung
\n	newline
\b	backspace
\r	carriage return
\t	horizontal tab
\\	backslash
\'	single quotation mark
\"	double quotation mark

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "123456789\r";
6     cout << "ABCD\n\n";
7     cout << "Vorname \t Name \t\t Alter \n";
8     cout << "Andreas \t Mustermann\t 42 \n\n";
9     cout << "Manchmal braucht man auch ein \"\\\"\"";
10    return 0;
11 }

```

Beispiele

Newline erschafft eine neue Zeile in der weitergeschrieben wird.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Dieser Text steht in der 1. Zeile.\nDieser Text steht in der 2. Zeile.";
6     return 0;
7 }

```

```

1 Dieser Text steht in der 1. Zeile.
2 Dieser text steht in der 2. Zeile.

```

Backspace setzt den Cursor um eins zurück und ermöglicht es das Symbol zu überschreiben.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Dieser Text steht in der 9\b1. Zeile.\nDieser Text steht in der 2. Zeile.";
6     return 0;
7 }

```

```

1 Dieser Text steht in der 1. Zeile.
2 Dieser text steht in der 2. Zeile.

```

Carriage return setzt den Cursor auf den Anfang der Zeile zurück und ermöglicht es Text zu überschreiben.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Dieser Text steht in der 9\b2. Zeile. Dies steht noch am Ende.\rDieser Text steht in
        der 1. Zeile.";
6     return 0;
7 }

```

```

1 Dieser Text steht in der 1. Zeile. Dies steht noch am Ende.

```

Horizontal tab erzeugt einen Tabulator. Damit ist eine saubere Formattierung möglich.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Name\tAlter\n";
6     cout << "Peter\t18\n";
7     cout << "Frank\t25\n";
8     cout << "Xi\t22\n";
9     return 0;
10 }

```

```

1 Name  Alter
2 Peter 18
3 Frank 25
4 Xi    22

```

Escape characters ermöglichen auch das Ausgeben von escape characters und Anführungszeichen.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "A\\B und \"C\" und 'D'\n";
6     return 0;
7 }

```

```

1 A\B und "C" und 'D'

```

2.2.2 Eingabe

Für die Eingabe stellt iostream den Eingabeoperator >> zur Verfügung. Der Rückgabewert des Operators ist ebenfalls eine Referenz auf ein Stream-Objekt (Referenz), so dass auch hier eine Hintereinanderschaltung von Operatoren möglich ist.

```

1 #include <iostream>
2
3 int main()
4 {
5     char b;
6     float a;
7     int i;
8     std::cout<<"Bitte Werte eingeben [char float int] : ";
9     std::cin>>b>>a>>i;
10    std::cout<<"char - " <<b<< " float - " <<a<< " int - " <<i;
11    return 0;
12 }

```

2.2.3 Beispiel der Woche

Implementieren Sie einen programmierbaren Taschenrechner für quadratische Funktionen.

```
1 #include<iostream>
2
3 int main() {
4     // Variante 1 - ganz schlecht
5     std::cout <<"f("<<x<<" ) = "<<3*5*5 + 4*5 + 8<<" \n";
6
7     // Variante 2 - besser
8     int x = 9;
9     std::cout <<"f("<<x<<" ) = "<<3*x*x + 4*x + 8<<" \n";
10    return 0;
11 }
```


Kapitel 3

Operatoren & Kontrollstrukturen

Parameter	Kursinformationen
<hr/>	
Veranstaltung:	Einführung in das wissenschaftliche Programmieren
Semester	Wintersemester 2022/23
Hochschule:	Technische Universität Freiberg
Inhalte:	Operatoren / Kontrollstrukturen
Link auf	https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/
Repository:	02_OperatorenKontrollstrukturen.md
<hr/>	
Autoren	@author

Fragen an die heutige Veranstaltung ...

- Wonach lassen sich Operatoren unterscheiden?
 - Welche unterschiedliche Bedeutung haben `x++` und `++x`?
 - Erläutern Sie den Begriff unärer, binärer und tertiärer Operator.
 - Unterscheiden Sie Zuweisung und Anweisung.
 - Wie lassen sich Kontrollflüsse grafisch darstellen?
 - Welche Konfigurationen erlaubt die `for`-Schleife?
 - In welchen Funktionen (Verzweigungen, Schleifen) ist Ihnen das Schlüsselwort `break` bekannt?
 - Worin liegt der zentrale Unterschied der `while` und `do-while` Schleife?
 - Recherchieren Sie Beispiele, in denen `goto`-Anweisungen Bugs generierten.
-

3.1 Operatoren

Kapitel 4

Unterscheidungsmerkmale

Ein Ausdruck ist eine Kombination aus Variablen, Konstanten, Operatoren und Rückgabewerten von Funktionen. Die Auswertung eines Ausdrucks ergibt einen Wert.

Zahl der beteiligten Operationen

Man unterscheidet in der Sprache C/C++ *unäre*, *binäre* und *ternäre* Operatoren

Operator	Operanden	Beispiel	Anwendung
Unäre Operatoren	1	& Adressoperator sizeof Größenoperator	sizeof(b); b=-a;
Binäre Operatoren	2	+, -, %	b=a-2;
Ternäre Operatoren	3	? Bedingungsoperator	b=(3 > 4 ? 0 : 1);

Es gibt auch Operatoren, die, je nachdem wo sie stehen, entweder unär oder binär sind. Ein Beispiel dafür ist der --Operator.

Position

Des Weiteren wird unterschieden, welche Position der Operator einnimmt:

- *Infix* – der Operator steht zwischen den Operanden.
- *Präfix* – der Operator steht vor den Operanden.
- *Postfix* – der Operator steht hinter den Operanden.

+ und - können alle drei Rollen einnehmen:

```
1 a = b + c; // Infix
2 a = -b;    // Präfix
3 a = b++;   // Postfix
```

Funktion des Operators

- Zuweisung
- Arithmetische Operatoren
- Logische Operatoren
- Bit-Operationen
- Bedingungsoperator

Weitere Unterscheidungsmerkmale ergeben sich zum Beispiel aus der [Assoziativität der Operatoren](#).

Achtung: Die nachvollgende Aufzählung erhebt nicht den Anspruch auf Vollständigkeit! Es werden bei weitem nicht alle Varianten der Operatoren dargestellt - vielmehr liegt der Fokus auf den für die Erreichung der didaktischen Ziele notwendigen Grundlagen.

4.0.1 Zuweisungsoperator

Der Zuweisungsoperator = ist von seiner mathematischen Bedeutung zu trennen - einer Variablen wird ein Wert zugeordnet. Damit macht dann auch `x=x+1` Sinn.

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int zahl1 = 10;
6     int zahl2 = 20;
7     int ergebn = 0;
8     // Zuweisung des Ausdrucks 'zahl1 + zahl2'
9     ergebn = zahl1 + zahl2;
10
11     cout<<zahl1<<" + "<<zahl2<<" = "<<ergebnis<<"\n";
12     return 0;
13 }

```

Achtung: Verwechseln Sie nicht den Zuweisungsoperator = mit dem Vergleichsoperator ==. Der Compiler kann die Fehlerhaftigkeit kaum erkennen und generiert Code, der ein entsprechendes Fehlverhalten zeigt.

4.0.2 Inkrement und Dekrement

Mit den ++ und -- Operatoren kann ein L-Wert um eins erhöht bzw. um eins vermindert werden. Man bezeichnet die Erhöhung um eins auch als Inkrement, die Verminderung um eins als Dekrement. Ein Inkrement einer Variable x entspricht $x = x + 1$, ein Dekrement einer Variable x entspricht $x = x - 1$.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int x, result;
6     x = 5;
7     result = 2 * ++x;    // Gebrauch als Präfix
8     cout<<"x="<<x<<" und result="<<result<<"\n";
9     result = 2 * x++;    // Gebrauch als Postfix
10    cout<<"x="<<x<<" und result="<<result<<"\n";
11    return 0;
12 }

```

4.0.3 Arithmetische Operatoren

Operator	Bedeutung	Ganzzahlen	Gleitkommazahlen
+	Addition	x	x
-	Subtraktion	x	x
*	Multiplikation	x	x
/	Division	x	x
%	Modulo (Rest einer Division)	x	

Achtung: Divisionsoperationen werden für Ganzzahlen und Gleitkommazahlen unterschiedlich realisiert.

- Wenn zwei Ganzzahlen wie z. B. 4/3 dividiert werden, erhalten wir das Ergebnis 1 zurück, der nicht ganzzahlige Anteil der Lösung bleibt unbeachtet.
- Für Fließkommazahlen wird die Division wie erwartet realisiert.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int timestamp, minuten;
6
7     timestamp = 345; // [s]
8     cout<<"Zeitstempel "<<timestamp<<" [s]\n";

```

```

9   minuten=timestamp/60;
10  cout<<timestamp<<" [s] entsprechen "<<minuten<<" Minuten\n";
11  return 0;
12 }

```

Die Modulo Operation generiert den Rest einer Divisionsoperation bei ganzen Zahlen.

```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int timestamp, sekunden, minuten;
6
7      timestamp = 345; //[s]
8      cout<<"Zeitstempel " <<timestamp<<" [s]\n";
9      minuten=timestamp/60;
10     sekunden=timestamp%60;
11     cout<<"Besser lesbar = " <<minuten<<" min. " <<sekunden<<" sek.\n";
12     return 0;
13 }

```

4.0.4 Vergleichsoperatoren

Kern der Logik sind Aussagen, die wahr oder falsch sein können.

Operation	Bedeutung
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich
==	gleich
!=	ungleich

```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int x = 15;
6      cout<<"x = " <<x<<" \n";
7      cout<<boolalpha<<"Aussage x > 5 ist " << (x>5) << " \n";
8      cout<<boolalpha<<"Aussage x == 5 ist " << (x==15) << " \n";
9      return 0;
10 }

```

Merke: Der Rückgabewert einer Vergleichsoperation ist `bool`. Dabei bedeutet `false` eine ungültige und `true` eine gültige Aussage. Vor 1993 wurde ein logischer Datentyp in C++ durch `int` simuliert. Aus den Gründen der Kompatibilität wird `bool` überall, wo wie hier nicht ausdrücklich `bool` verlangt wird in `int` (Werte 0 und 1) umgewandelt.

Mit dem `boolalpha` Parameter kann man `cout` überreden zumindest `true` und `false` auszugeben.

4.0.5 Logische Operatoren

Und wie lassen sich logische Aussagen verknüpfen? Nehmen wir an, dass wir aus den Messdaten zweier Sensoren ein Alarmsignal generieren wollen. Nur wenn die Temperatur *und* die Luftfeuchte in einem bestimmten Fenster liegen, soll dies nicht passieren.

Operation	Bedeutung
&&	UND
	ODER

Operation	Bedeutung
!	NICHT

Das ODER wird durch senkrechte Striche repräsentiert (Altgr+< Taste) und nicht durch große I!

Nehmen wir an, sie wollen Messdaten evaluieren. Ihr Sensor funktioniert nur dann wenn die Temperatur ein Wert zwischen -10 und -20 Grad annimmt und die Luftfeuchte zwischen 40 bis 60 Prozent beträgt.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     float Temperatur = -30;    // Das sind unsere Probewerte
6     float Feuchte = 65;
7
8     // Vergleichsoperationen und Logische Operationen
9     bool TempErgebnis = .... // Hier sind Sie gefragt!
10
11     // Ausgabe
12     if ... {
13         cout<<"Die Messwerte kannst Du vergessen!";
14     }
15     return 0;
16 }
```

Anmerkung: C++ bietet für logische Operatoren und Bit-Operatoren Synonyme **and**, **or**, **xor**. Die Synonyme sind Schlüsselwörter, wenn Compiler-Einstellungen `/permissive-` oder `/Za` (Spracherweiterungen deaktivieren) angegeben werden. Sie sind keine Schlüsselwörter, wenn Microsoft-Erweiterungen aktiviert sind. Die Verwendung der Synonyme kann die Lesbarkeit deutlich erhöhen.

4.0.6 sizeof - Operator

Der Operator `sizeof` ermittelt die Größe eines Datentyps (in Byte) zur Kompiliertzeit.

- `sizeof` ist keine Funktion, sondern ein Operator.
- `sizeof` wird häufig zur dynamischen Speicherreservierung verwendet.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     double wert=0.0;
6     cout<<sizeof(0)<<" "<<sizeof(double)<<" "<<sizeof(wert);
7     return 0;
8 }
```

4.1 Vorrangregeln

Konsequenterweise bildet auch die Programmiersprache C/C++ eigene Vorrangregeln ab, die grundlegende mathematische Definitionen "Punktrechnung vor Strichrechnung" realisieren. Die Liste der unterschiedlichen Operatoren macht aber weitere Festlegungen notwendig.

Prioritäten

In welcher Reihung erfolgt beispielsweise die Abarbeitung des folgenden Ausdrucks?

```
1 c = sizeof(x) + ++a / 3;
```

Für jeden Operator wurde eine Priorität definiert, die die Reihung der Ausführung regelt.

[Liste der Vorrangregeln](#)

Im Beispiel bedeutet dies:

```

1 c = sizeof(x) + ++a / 3;
2 //      |      | | |
3 //      |      | | |--- Priorität 13
4 //      |      | |--- Priorität 14
5 //      |      |--- Priorität 12
6 //      |--- Priorität 14
7
8 c = (sizeof(x)) + ((++a) / 3);

```

Assoziativität

Für Operatoren mit der gleichen Priorität ist für die Reihenfolge der Auswertung die Assoziativität das zweite Kriterium.

```

1 a = 4 / 2 / 2;
2
3 // von rechts nach links (FALSCH)
4 // 4 / (2 / 2) // ergibt 4
5
6 // von links nach rechts ausgewertet
7 // (4 / 2) / 2 // ergibt 1

```

Merke: Setzen Sie Klammern, um alle Zweifel auszuräumen

4.2 ... und mal praktisch

Folgender Code nutzt die heute besprochenen Operatoren um die Eingaben von zwei Buttons auf eine LED abzubilden. Nur wenn beide Taster gedrückt werden, beleuchte das rote Licht für 3 Sekunden.

```

1 const int button_A_pin = 10;
2 const int button_B_pin = 11;
3 const int led_pin = 13;
4
5 int buttonAState;
6 int buttonBState;
7
8 void setup(){
9   pinMode(button_A_pin, INPUT);
10  pinMode(button_B_pin, INPUT);
11  pinMode(led_pin, OUTPUT);
12  Serial.begin(9600);
13 }
14
15 void loop() {
16   Serial.println("Wait one second for A ");
17   delay(1000);
18   buttonAState = digitalRead(button_A_pin);
19   Serial.println ("... and for B");
20   delay(1000);
21   buttonBState = digitalRead(button_B_pin);
22
23   if ( buttonAState && buttonBState){
24     digitalWrite(led_pin, HIGH);
25     delay(3000);
26   }
27   else
28   {
29     digitalWrite(led_pin, LOW);
30   }
31 }

```

4.2.0.1 Logische Operatoren

Wofür steht der logische Operator `&&?` `[[und]]`

```
let input = "@input".trim().toLowerCase()
```

```
input == "und" || input == "UND" || input == "Und"
```

Wofür steht der logische Operator `||?` `[[oder]]`

```
let input = "@input".trim().toLowerCase()
```

```
input == "oder" || input == "ODER" || input == "Oder"
```

Wofür steht der logische Operator `!?` `[[nicht]]`

```
let input = "@input".trim().toLowerCase()
```

```
input == "nicht" || input == "NICHT" || input == "Nicht"
```


Kapitel 5

Kontrollfluss

Bisher haben wir Programme entworfen, die eine sequenzielle Abfolge von Anweisungen enthielt.

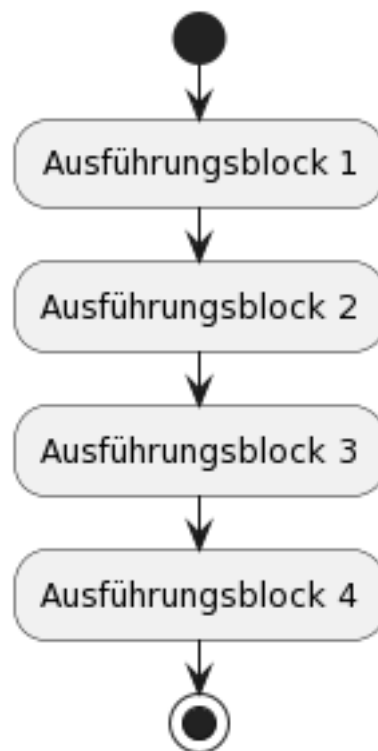


Abbildung 5.1: Modelle

Diese Einschränkung wollen wir nun mit Hilfe weiterer Anweisungen überwinden:

1. **Verzweigungen (Selektion):** In Abhängigkeit von einer Bedingung wird der Programmfluss an unterschiedlichen Stellen fortgesetzt.

Beispiel: Wenn bei einer Flächenberechnung ein Ergebnis kleiner Null generiert wird, erfolgt eine Fehlerausgabe. Sonst wird im Programm fortgefahren.

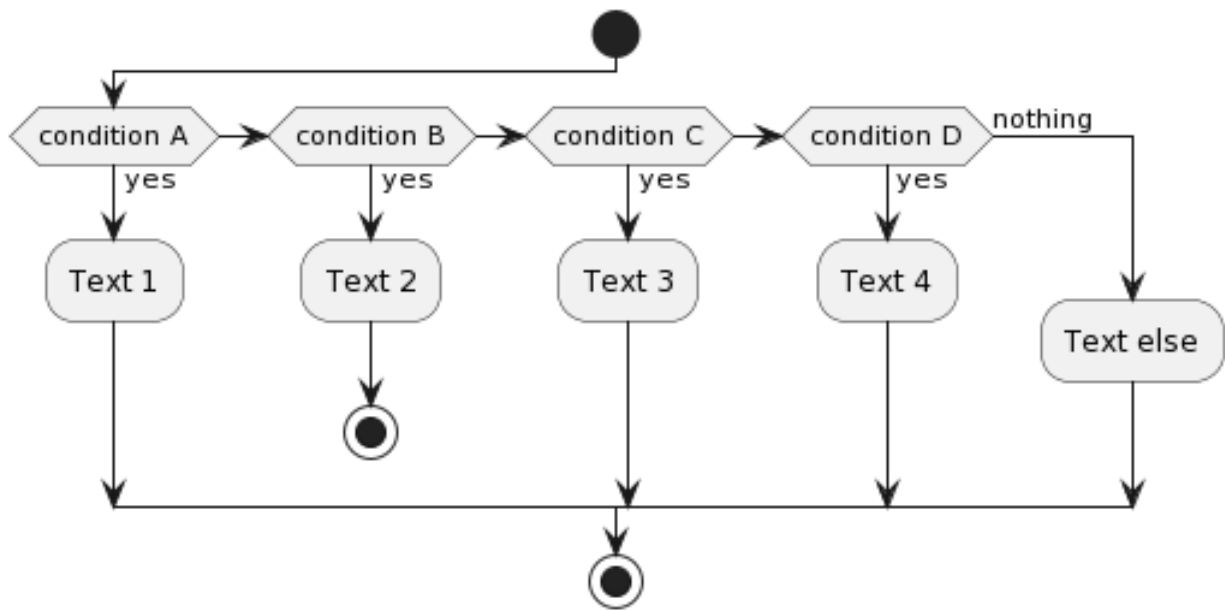
2. **Schleifen (Iteration):** Ein Anweisungsblock wird so oft wiederholt, bis eine Abbruchbedingung erfüllt wird.

Beispiel: Ein Datensatz wird durchlaufen um die Gesamtsumme einer Spalte zu bestimmen. Wenn der letzte Eintrag erreicht ist, wird der Durchlauf abgebrochen und das Ergebnis ausgegeben.

3. Des Weiteren verfügt C/C++ über **Sprünge**: die Programmausführung wird mit Hilfe von Sprungmarken an einer anderen Position fortgesetzt. Formal sind sie jedoch nicht notwendig. Statt die nächste Anweisung auszuführen, wird (zunächst) an eine ganz andere Stelle im Code gesprungen.

5.0.1 Verzweigungen

Verzweigungen entfalten mehrere mögliche Pfade für die Ausführung des Programms.



5.0.1.1 if-Anweisungen

Im einfachsten Fall enthält die `if`-Anweisung eine einzelne bedingte Anweisung oder einen Anweisungsblock. Sie kann mit `else` um eine Alternative erweitert werden.

Zum Anweisungsblock werden die Anweisungen mit geschweiften Klammern (`{` und `}`) zusammengefasst.

```

1 if(Bedingung) Anweisung; // <- Einzelne Anweisung
2
3 if(Bedingung){           // <- Beginn Anweisungsblock
4   Anweisung;
5   Anweisung;
6 }                         // <- Ende Anweisungsblock
  
```

Optional kann eine alternative Anweisung angegeben werden, wenn die Bedingung nicht erfüllt wird:

```

1 if(Bedingung){
2   Anweisung;
3 }else{
4   Anweisung;
5 }
  
```

Mehrere Fälle können verschachtelt abgefragt werden:

```

1 if(Bedingung)
2   Anweisung;
3 else
4   if(Bedingung)
5     Anweisung;
6   else
7     Anweisung;
8   Anweisung; //!!!
  
```

Merke: An diesem Beispiel wird deutlich, dass die Klammern für die Zuordnung elementar wichtig sind. Die letzte Anweisung gehört NICHT zum zweiten `else` Zweig und auch nicht zum ersten. Diese Anweisung wird immer ausgeführt!

Weitere Beispiele für Bedingungen

Die Bedingungen können als logische UND arithmetische Ausdrücke formuliert werden.

Ausdruck	Bedeutung
<code>if (a != 0)</code>	$a \neq 0$
<code>if (a == 0)</code>	$a = 0$
<code>if (!(a <= b))</code>	$\overline{(a \leq b)}$ oder $a > b$
<code>if (a != b)</code>	$a \neq b$
<code>if (a b)</code>	$a > 0$ oder $b > 0$

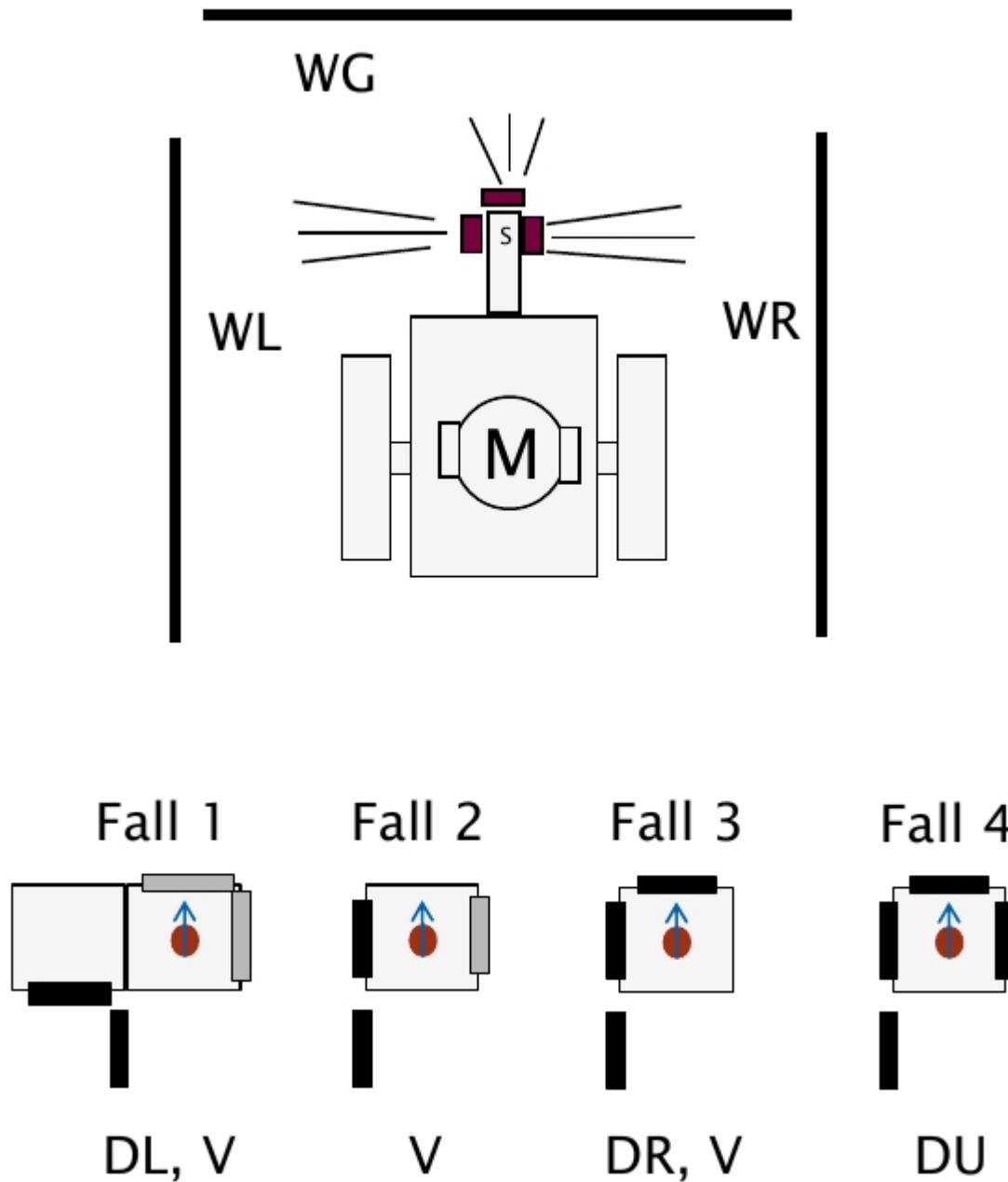
Mögliche Fehlerquellen

1. Zuweisungs- statt Vergleichsoperator in der Bedingung (kein Compilerfehler)
2. Bedingung ohne Klammern (Compilerfehler)
3. ; hinter der Bedingung (kein Compilerfehler)
4. Multiple Anweisungen ohne Anweisungsblock
5. Komplexität der Statements

5.0.1.2 Beispiel

Nehmen wir an, dass wir einen kleinen Roboter aus einem Labyrinth fahren lassen wollen. Dazu gehen wir davon aus, dass er bereits an einer Wand steht. Dieser soll er mit der “Linke-Hand-Regel” folgen. Dabei wird von einem einfach zusammenhängenden Labyrinth ausgegangen.

Die nachfolgende Grafik illustriert den Aufbau des Roboters und die vier möglichen Konfigurationen des Labyrinths, nachdem ein neues Feld betreten wurde.



Fall	Bedeutung
1.	Die Wand knickt nach links weg. Unabhängig von WG und WR folgt der Roboter diesem Verlauf.
2.	Der Roboter folgt der linksseitigen Wand.
3.	Die Wand blockiert die Fahrt. Der Roboter dreht sich nach rechts, damit liegt diese Wandelement nun wieder zu seiner linken Hand.
4.	Der Roboter folgt dem Verlauf nach einer Drehung um 180 Grad.

WL	WG	WR	Fall	Verhalten
0	0	0	1	Drehung Links, Vorwärts
0	0	1	1	Drehung Links, Vorwärts
0	1	0	1	Drehung Links, Vorwärts
0	1	1	1	Drehung Links, Vorwärts
1	0	0	2	Vorwärts
1	0	1	2	Vorwärts
1	1	0	3	Drehung Rechts, Vorwärts
1	1	1	4	Drehung 180 Grad

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int WL, WG, WR;
6     WL = 0; WG = 1; WR =1;
7     if (!WL)                                // Fall 1
8         cout<<"Drehung Links\n";
9     if ((WL) && (!WG))                        // Fall 2
10        cout<<"Vorwärts\n";
11     if ((WL) && (WG) && (!WR))                // Fall 3
12        cout<<"Drehung Rechts\n";
13     if ((WL) && (WG) && (WR))                  // Fall 4
14        cout<<"Drehung 180 Grad\n";
15     return 0;
16 }

```

Sehen Sie mögliche Vereinfachungen des Codes?*

5.0.1.3 Zwischenfrage

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int Punkte = 45;
7     int Zusatzpunkte = 15;
8     if (Punkte + Zusatzpunkte >= 50)
9     {
10        cout<<"Test ist bestanden!\n";
11        if (Zusatzpunkte >= 15)
12        {
13            cout<<"Alle Zusatzpunkte geholt!\n";
14        }else{
15            if(Zusatzpunkte > 8) {
16                cout<<"Respektable Leistung\n";
17            }
18        }
19    }else{
20        cout<<"Leider durchgefallen!\n";
21    }
22    return 0;
23 }

```

- [(Test ist bestanden)] Test ist bestanden
- [(Alle Zusatzpunkte geholt)] Alle Zusatzpunkte geholt
- [(Leider durchgefallen!)] Leider durchgefallen!
- [(Test ist bestanden!+Alle Zusatzpunkte geholt!)] Test ist bestanden!+Alle Zusatzpunkte geholt!
- [(Test ist bestanden!+Respektable Leistung)] Test ist bestanden!+Respektable Leistung

5.0.1.4 switch-Anweisungen

Too many ifs - I think I switch

Berndt Wischniewski

Eine übersichtlichere Art der Verzweigung für viele, sich ausschließende Bedingungen wird durch die **switch**-Anweisung bereitgestellt. Sie wird in der Regel verwendet, wenn eine oder einige unter vielen Bedingungen ausgewählt werden sollen. Das Ergebnis der "expression"-Auswertung soll eine Ganzzahl (oder **char**-Wert) sein. Stimmt es mit einem "const_expr"-Wert überein, wird die Ausführung an dem entsprechenden **case**-Zweig fortgesetzt. Trifft keine der Bedingungen zu, wird der **default**-Fall aktiviert.

```

1 switch(expression)
2 {
3     case const-expr: Anweisung break;
4     case const-expr:
5         Anweisungen
6         break;
7     case const-expr: Anweisungen break;
8     default: Anweisungen
9 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a=50, b=60;
6     char op;
7     cout<<"Bitte Operator definieren (+,-,*,/): ";
8     cin>>op;
9
10    switch(op) {
11        case '+':
12            cout<<a<<" + "<<b<<" = "<<a+b<<" \n";
13            break;
14        case '-':
15            cout<<a<<" - "<<b<<" = "<<a-b<<" \n";
16            break;
17        case '*':
18            cout<<a<<" * "<<b<<" = "<<a*b<<" \n";
19            break;
20        case '/':
21            cout<<a<<" / "<<b<<" = "<<a/b<<" \n";
22            break;
23        default:
24            cout<<op<<"? kein Rechenoperator \n";
25    }
26    return 0;
27 }

```

Im Unterschied zu einer if-Abfrage wird in den unterschiedlichen Fällen immer nur auf Gleichheit geprüft! Eine abgefragte Konstante darf zudem nur einmal abgefragt werden und muss ganzzahlig oder `char` sein.

```

1 // Fehlerhafte case Blöcke
2 switch(x)
3 {
4     case x < 100: // das ist ein Fehler
5         y = 1000;
6         break;
7
8     case 100.1: // das ist genauso falsch
9         y = 5000;
10        z = 3000;
11        break;
12 }

```

Und wozu brauche ich das `break`? Ohne das `break` am Ende eines Falls werden alle darauf folgenden Fälle bis zum Ende des `switch` oder dem nächsten `break` zwingend ausgeführt.

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a=5;

```

```

6
7  switch(a) {
8      case 5:    // Multiple Konstanten
9      case 6:
10     case 7:
11         cout<<"Der Wert liegt zwischen 4 und 8\n";
12     case 3:
13         cout<<"Der Wert ist 3 \n";
14         break;
15     case 0:
16         cout<<"Der Wert ist 0 \n";
17     default: cout<<"Wert in keiner Kategorie\n";}
18
19     return 0;
20 }

```

Unter Ausnutzung von `break` können Kategorien definiert werden, die aufeinander aufbauen und dann übergreifend “aktiviert” werden.

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     char ch;
6     cout<<"Geben Sie ein Zeichen ein : ";
7     cin>>ch;
8
9     switch(ch)
10    {
11        case 'a':
12        case 'A':
13        case 'e':
14        case 'E':
15        case 'i':
16        case 'I':
17        case 'o':
18        case 'O':
19        case 'u':
20        case 'U':
21            cout<<"\n\n"<<ch<<" ist ein Vokal.\n\n";
22            break;
23        default:
24            cout<<ch<<" ist ein Konsonant.\n\n";
25    }
26    return 0;
27 }

```

5.0.2 Schleifen

Schleifen dienen der Wiederholung von Anweisungsblöcken – dem sogenannten Schleifenrumpf oder Schleifenkörper – solange die Schleifenbedingung als Laufbedingung gültig bleibt bzw. als Abbruchbedingung nicht eintritt. Schleifen, deren Schleifenbedingung immer zur Fortsetzung führt oder die keine Schleifenbedingung haben, sind *Endlosschleifen*.

Schleifen können verschachtelt werden, d.h. innerhalb eines Schleifenkörpers können weitere Schleifen erzeugt und ausgeführt werden. Zur Beschleunigung des Programmablaufs werden Schleifen oft durch den Compiler entrollt (*Enrollment*).

Grafisch lassen sich die wichtigsten Formen in mit der Nassi-Shneiderman Diagrammen wie folgt darstellen:

- Iterationssymbol

```

1  +-----+

```

```

2 | | | | |
3 | | zähle [Variable] von [Startwert] bis [Endwert], mit [Schrittweite] |
4 | +-----+
5 | | | | |
6 | | | Anweisungsblock 1 |
7 | +-----+

```

- Wiederholungsstruktur mit vorausgehender Bedingungsprüfung

```

1 +-----+
2 | | | | |
3 | | solange Bedingung wahr |
4 | +-----+
5 | | | | |
6 | | | Anweisungsblock 1 |
7 | +-----+

```

- Wiederholungsstruktur mit nachfolgender Bedingungsprüfung

```

1 +-----+
2 | | | | |
3 | | | Anweisungsblock 1 |
4 | +-----+
5 | | | | |
6 | | | solange Bedingung wahr |
7 | +-----+

```

Die Programmiersprache C/C++ kennt diese drei Formen über die Schleifenkonstrukte `for`, `while` und `do while`.

5.0.2.1 for-Schleife

Der Parametersatz der `for`-Schleife besteht aus zwei Anweisungsblöcken und einer Bedingung, die durch Semikolons getrennt werden. Mit diesen wird ein **Schleifenzähler** initiiert, dessen Manipulation spezifiziert und das Abbruchkriterium festgelegt. Häufig wird die Variable mit jedem Durchgang inkrementiert oder dekrementiert, um dann anhand eines Ausdrucks evaluiert zu werden. Es wird überprüft, ob die Schleife fortgesetzt oder abgebrochen werden soll. Letzterer Fall tritt ein, wenn dieser den Wert `false` (falsch) annimmt.

```

1 // generisches Format der for-Schleife
2 for(Initialisierung; Bedingung; Reinitialisierung) {
3     // Anweisungen
4 }
5
6 // for-Schleife als Endlosschleife
7 for(;;){
8     // Anweisungen
9 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int i;
6     for (i = 1; i<10; i++)
7         cout<<i<<" ";
8
9     cout<<"\nNach der Schleife hat i den Wert "<<i<<"\n";
10    return 0;
11 }

```

Beliebte Fehlerquellen

- Semikolon hinter der schließenden Klammer von `for`

- Kommas anstatt Semikolons zwischen den Parametern von `for`
- fehlerhafte Konfiguration von Zählschleifen
- Nichtberücksichtigung der Tatsache, dass die Zählvariable nach dem Ende der Schleife über dem Abbruchkriterium liegt

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int i;
6     for (i = 1; i<10; i++);
7         cout<<i<<" ";
8
9     cout<<"Das ging jetzt aber sehr schnell ... \n"<<i;
10    return 0;
11 }

```

5.0.2.2 while-Schleife

Während bei der `for`-Schleife auf ein n-maliges Durchlaufen Anweisungsfolge konfiguriert wird, definiert die `while`-Schleife nur eine Bedingung für den Fortführung/Abbruch.

```

1 // generisches Format der while-Schleife
2 while (Bedingung)
3     Anweisungen;
4
5 while (Bedingung){
6     Anweisungen;
7     Anweisungen;
8 }

```

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     char c;
6     int zaehler = 0;
7     cout<<"Pluszeichenzähler - zum Beenden \"_\" [Enter]\n";
8     cin>>c;
9     while(c != '_')
10    {
11        if(c == '+')
12            zaehler++;
13        cin>>c;
14    }
15    cout<<"Anzahl der Pluszeichen: "<<zaehler<<"\n";
16    return 0;
17 }

```

Dabei soll erwähnt werden, dass eine `while`-Schleife eine `for`-Schleife ersetzen kann.

```

1 // generisches Format der while-Schleife
2 i = 0;
3 while (i<10){
4     // Anweisungen;
5     i++;
6 }
7
8 for (i=0; i<10; i++){
9     // Anweisungen;
10 }

```

5.0.2.3 do-while-Schleife

Im Gegensatz zur `while`-Schleife führt die `do-while`-Schleife die Überprüfung des Abbruchkriteriums erst am Schleifenende aus.

```
1 // generisches Format der while-Schleife
2 do
3     Anweisung;
4 while (Bedingung);
```

Welche Konsequenz hat das? Die `do-while`-Schleife wird in jedem Fall einmal ausgeführt.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     char c;
6     int zaehler = 0;
7     cout<<"Pluszeichenzähler - zum Beenden \"_\" [Enter]\\n";
8     do
9     {
10         cin>>c;
11         if(c == '+')
12             zaehler++;
13     }while(c != '_');
14     cout<<"Anzahl der Pluszeichen: "<<zaehler<<"\\n";
15     return 0;
16 }
```

5.0.3 Kontrolliertes Verlassen der Anweisungen

Bei allen drei Arten der Schleifen kann zum vorzeitigen Verlassen der Schleife `break` benutzt werden. Damit wird aber nur die unmittelbar umgebende Schleife beendet!

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int i;
6     for (i = 1; i<10; i++){
7         if (i == 5) break;
8         cout<<i<<" ";
9     }
10    cout<<"\\nUnd vorbei ... i ist jetzt "<<i<<"\\n";
11    return 0;
12 }
```

Eine weitere wichtige Eingriffsmöglichkeit für Schleifenkonstrukte bietet `continue`. Damit wird nicht die Schleife insgesamt, sondern nur der aktuelle Durchgang gestoppt.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int i;
6     for (i = -5; i<6; i++){
7         if (i == 0) continue;
8         cout<<12. / i<<"\\n";
9     }
10    return 0;
11 }
```

Durch `return`-Anweisung wird das Verlassen einer Funktion veranlasst (genaues in der Vorlesung zu Funktionen).

5.1 Beispiel des Tages

Das Codebeispiel des Tages führt die Berechnung eines sogenannten magischen Quadrates vor.

Das Lösungsbeispiel stammt von der Webseite <https://rosettacode.org>, die für das Problem [magic square](#) und viele andere “Standardprobleme” Lösungen in unterschiedlichen Sprachen präsentiert. Sehr lesenswerte Sammlung!

```
1 #include <iostream>
2 using namespace std;
3
4 int f(int n, int x, int y)
5 {
6     return (x + y*2 + 1) % n;
7 }
8
9 int main() {
10     int i, j, n;
11
12     //Input must be odd and not less than 3.
13     n = 5;
14     if (n < 3 || (n % 2) == 0) return 2;
15
16     for (i = 0; i < n; i++) {
17         for (j = 0; j < n; j++){
18             cout<<f(n, n - j - 1, i)*n + f(n, j, i) + 1<<"\t";
19             //fflush(stdout);
20         }
21         cout<<"\n";
22     }
23     cout<<"\nMagic Constant: "<<(n*n+1)/2*n<<".\n";
24
25     return 0;
26 }
```


Kapitel 6

Grundlagen der Sprache C++

Parameter	Kursinformationen
-----------	-------------------

Veranstaltung	Prozedurale Programmierung / Einführung in die Informatik
---------------	---

Semester	Wintersemester 2022/23
----------	------------------------

Hochschule	Technische Universität Freiberg
------------	---------------------------------

Inhalte:	Array, Zeiger und Referenzen
----------	------------------------------

Link auf	https://github.com/TUBAF-Iff-
----------	---

Repository:	LiaScript/VL_ProzeduraleProgrammierung/blob/master/03_ArrayZeigerReferenzen.md
-------------	--

Autoren	@author
---------	---------

Fragen an die heutige Veranstaltung ...

- Was ist ein Array?
 - Wie können zwei Arrays verglichen werden?
 - Erklären Sie die Idee des Zeigers in der Programmiersprache C/C++.
 - Welche Gefahr besteht bei der Initialisierung von Zeigern?
 - Was ist ein NULL-Zeiger und wozu wird er verwendet?
-

6.1 Wie weit waren wir gekommen?

Aufgabe: Die LED blinkt im Beispiel 10 mal. Integrieren Sie eine Abbruchbedingung für diese Schleife, wenn der rote Button gedrückt wird. Welches Problem sehen Sie?

```
1 void setup() {
2   pinMode(2, INPUT);
3   pinMode(3, INPUT);
4   pinMode(13, OUTPUT);
5 }
6
7 void loop() {
8   bool a = digitalRead(2);
9   if (a){
10    for (int i = 0; i<10; i++){
11      digitalWrite(13, HIGH);
12      delay(250);
13      digitalWrite(13, LOW);
14      delay(250);
15    }
16  }
```

```
17 }
```

```
@AVR8js.sketch
```

6.2 Arrays

Bisher umfassten unsere Variablen einzelne Skalare. Arrays erweitern das Spektrum um Folgen von Werten, die in n-Dimensionen aufgestellt werden können. Array ist eine geordnete Folge von Werten des gleichen Datentyps. Die Deklaration erfolgt in folgender Anweisung:

```
1 Datentyp Variablenname[Anzahl_der_Elemente];
```

```
1 int a[6];
```

```
a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |
```

```
1 Datentyp Variablenname[Anzahl_der_Elemente_Dim0][Anzahl_der_Elemente_Dim1];
```

```
1 int a[3][5];
```

	Spalten				
Zeilen	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Achtung 1: Im hier beschriebenen Format muss zum Zeitpunkt der Übersetzung die Größe des Arrays (Anzahl_der_Elemente) bekannt sein.

Achtung 2: Der Variablenname steht nunmehr nicht für einen Wert sondern für die Speicheradresse (Pointer) des ersten Elementes!

6.2.1 Deklaration, Definition, Initialisierung, Zugriff

Initialisierung und genereller Zugriff auf die einzelnen Elemente des Arrays sind über einen Index möglich.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int a[3];           // Array aus 3 int Werten
6     a[0] = -2;
7     a[1] = 5;
8     a[2] = 99;
9     for (int i=0; i<3; i++)
10         cout<<a[i]<<" ";
11     cout<<"\nNur zur Info "<< sizeof(a);
12     cout<<"\nZahl der Elemente "<< sizeof(a) / sizeof(int);
13     return 0;
14 }
```

Wie können Arrays noch initialisiert werden:

- vollständig (alle Elemente werden mit einem spezifischen Wert belegt)
- anteilig (einzelne Elemente werden mit spezifischen Werten gefüllt, der rest mit 0)

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int a[] = {5, 2, 2, 5, 6};
6     float b[5] = {1.0};
7     for (int i=0; i<5; i++){
```

```

8     cout<<a[i]<<" "<<b[i]<<"\n";
9 }
10 return 0;
11 }

```

Und wie bestimme ich den erforderlichen Speicherbedarf bzw. die Größe des Arrays?

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int a[3];
6     cout<<"\nNur zur Speicherplatz [Byte] "<<sizeof(a);
7     cout<<"\nZahl der Elemente "<<sizeof(a)/sizeof(int)<<"\n";
8     return 0;
9 }

```

6.2.2 Fehlerquelle Nummer 1 - out of range

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int a[] = {-2, 5, 99};
6     for (int i=0; i<=3; i++)
7         cout<<a[i]<<" ";
8     return 0;
9 }

```

6.2.3 Anwendung eines eindimensionalen Arrays

Schreiben Sie ein Programm, das zwei Vektoren miteinander vergleicht. Warum ist die intuitive Lösung `a == b` nicht korrekt, wenn `a` und `b` arrays sind?

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int a[] = {0, 1, 2, 4, 3, 5, 6, 7, 8, 9};
6     int b[10];
7     for (int i=0; i<10; i++)
8         b[i]=i;
9     for (int i=0; i<10; i++)
10         if (a[i]!=b[i])
11             cout<<"An Stelle "<<i<<" unterscheiden sich die Vektoren \n";
12     return 0;
13 }

```

Welche Verbesserungsmöglichkeiten sehen Sie bei dem Programm?

6.2.4 Mehrdimensionale Arrays

Deklaration:

```

1 int Matrix[4][5];    /* Zweidimensional - 4 Zeilen x 5 Spalten */

```

Deklaration mit einer sofortigen Initialisierung aller bzw. einiger Elemente:

```

1 int Matrix[4][5] = { {1,2,3,4,5},
2                     {6,7,8,9,10},
3                     {11,12,13,14,15},
4                     {16,17,18,19,20}};
5

```

```

6 int Matrix[4][4] = { {1,},
7                     {1,1},
8                     {1,1,1},
9                     {1,1,1,1}};
10
11 int Matrix[4][4] = {1,2,3,4,5,6,7,8};

```

Initialisierung eines n-dimensionalen Arrays:

	0	1	2	3	4	5	6	7
0								
1								
2		1						
3						3		
4			2					
5								
6								4
7								

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     // Initialisierung
6     int brett[8][8] = {0};
7     // Zuweisung
8     brett[2][1] = 1;
9     brett[4][2] = 2;
10    brett[3][5] = 3;
11    brett[6][7] = 4;
12    // Ausgabe
13    int i, j;
14    // Schleife fuer Zeilen, Y-Achse
15    for(i=0; i<8; i++) {
16        // Schleife fuer Spalten, X-Achse
17        for(j=0; j<8; j++) {
18            cout<<brett[i][j]<<" ";
19        }
20        cout<<"\n";
21    }
22    return 0;
23 }

```

6.2.5 Anwendung eines zweidimensionalen Arrays

Elementweise Addition zweier Matrizen

```

1 #include <iostream>

```



```

2 using namespace std;
3
4 int main(void)
5 {
6     int C[2][3];
7     int i,j;
8     for (i=0;i<2;i++)
9         for (j=0;j<3;j++)
10            C[i][j]=A[i][j]+B[i][j];
11     for (i=0;i<2;i++)
12     {
13         for (j=0;j<3;j++)
14             cout<<C[i][j]<<"\t";
15         cout<<"\n";
16     }
17     return 0;
18 }

```

Weiteres Beispiel: Lösung eines Gleichungssystem mit dem Gausschen Eliminationsverfahren [Link](#)

6.3 Sonderfall Zeichenketten / Strings

Folgen von Zeichen, die sogenannten *Strings* werden in C/C++ durch Arrays mit Elementen vom Datentyp `char` repräsentiert. Die Zeichenfolgen werden mit `\0` abgeschlossen.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     cout<<"Diese Form eines Strings haben wir bereits mehrfach benutzt!\n";
6     //////////////////////////////////////
7
8     char a[] = "Ich bin ein char Array!"; // Der Compiler fügt das \0 automatisch ein!
9     if (a[23] == '\0'){
10         cout<<"char Array Abschluss in a gefunden!";
11     }
12
13     cout<<"->"<<a<<"<-\n";
14     char b[] = { 'H', 'a', 'l', 'l', 'o', ' ',
15                 'F', 'r', 'e', 'i', 'b', 'e', 'r', 'g', '\0' };
16     cout<<"->"<<b<<"<-\n";
17     char c[] = "Noch eine \0Möglichkeit";
18     cout<<"->"<<c<<"<-\n";
19     char d[] = { 80, 114, 111, 122, 80, 114, 111, 103, 32, 50, 48, 50, 50, 0 };
20     cout<<"->"<<d<<"<-\n";
21     return 0;
22 }

```

C++ implementiert einen separaten string-Datentyp (Klasse), die ein deutliche komfortablen Umgang mit Texten erlaubt. Beim Anlegen eines solchen muss nicht angegeben werden, wie viele Zeichen reserviert werden sollen. Zudem können Strings einfach zuweisen und vergleichen werden, wie es für andere Datentypen üblich ist. Die C `const char *` Mechanismen funktionieren aber auch hier.

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(void) {
6     string hanna = "Hanna";
7     string anna = "Anna";
8     string alleBeide = anna + " + " + hanna;

```

```

9  cout<<"Hallo: "<<alleBeide<<std::endl;
10
11  int res = anna.compare(hanna);
12
13  if (res == 0)
14      cout << "\nBoth the input strings are equal." << endl;
15  else if(res < 0)
16      cout << "String 1 is smaller as compared to String 2\n.";
17  else
18      cout<<"String 1 is greater as compared to String 2\n.";
19
20  return EXIT_SUCCESS;
21 }

```

6.4 Grundkonzept Zeiger

Bisher umfassten unserer Variablen als Datencontainer Zahlen oder Buchstaben. Das Konzept des Zeigers (englisch Pointer) erweitert das Spektrum der Inhalte auf Adressen.

An dieser Adresse können entweder Daten, wie Variablen oder Objekte, aber auch Programmcodes (Anweisungen) stehen. Durch Dereferenzierung des Zeigers ist es möglich, auf die Daten oder den Code zuzugreifen.

1	Variablen-	Speicher-	Inhalt	
2	name	adresse		
3			+-----+	
4		0000		
5			+-----+	
6		0001		
7			+-----+	
8	a ----->	0002	+--- 00001007	Adresse
9			z +-----+	
10		0003	e	
11			i +-----+	
12		g	
13			t +-----+	
14		1005		
15			a +-----+	
16		1006	u	
17			f +-----+	
18	b ----->	1007	<--+ 00001101	Wert = 13
19			+-----+	
20		1008		
21			+-----+	
22	

Welche Vorteile ergeben sich aus der Nutzung von Zeigern, bzw. welche Programmier Techniken lassen sich realisieren:

- dynamische Verwaltung von Speicherbereichen,
- Übergabe von Datenobjekten an Funktionen via “call-by-reference”,
- Übergabe von Funktionen als Argumente an andere Funktionen,
- Umsetzung rekursiver Datenstrukturen wie Listen und Bäume.

An dieser Stelle sei erwähnt, dass die Übergabe der “call-by-reference”-Parameter via Reference ist ebenfalls möglich und einfacher in der Handhabung.

6.4.1 Definition von Zeigern

Die Definition eines Zeigers besteht aus dem Datentyp des Zeigers und dem gewünschten Zeigernamen. Der Datentyp eines Zeigers besteht wiederum aus dem Datentyp des Werts auf den gezeigt wird sowie aus einem Asterisk. Ein Datentyp eines Zeigers wäre also z. B. `double*`.

```

1  /* kann eine Adresse aufnehmen, die auf einen Wert vom Typ Integer zeigt */
2  int* zeiger1;
3  /* das Leerzeichen kann sich vor oder nach dem Stern befinden */
4  float *zeiger2;
5  /* ebenfalls möglich */
6  char * zeiger3;
7  /* Definition von zwei Zeigern */
8  int *zeiger4, *zeiger5;
9  /* Definition eines Zeigers und einer Variablen vom Typ Integer */
10 int *zeiger6, ganzzahl;

```

6.4.2 Initialisierung

Merke: Zeiger müssen vor der Verwendung initialisiert werden.

Der Zeiger kann initialisiert werden durch die Zuweisung: * der Adresse einer Variable, wobei die Adresse mit Hilfe des Adressoperators & ermittelt wird, * eines Arrays, * eines weiteren Zeigers oder * des Wertes von NULL.

```

1  #include <iostream>
2  using namespace std;
3
4  int main(void)
5  {
6      int a = 0;
7      int * ptr_a = &a;          /* mit Adressoperator */
8
9      int feld[10];
10     int * ptr_feld = feld; /* mit Array */
11
12     int * ptr_b = ptr_a;      /* mit weiterem Zeiger */
13
14     int * ptr_Null = NULL;   /* mit NULL */
15
16     cout<<"Pointer ptr_a      "<<ptr_a<<"\n";
17     cout<<"Pointer ptr_feld "<<ptr_feld<<"\n";
18     cout<<"Pointer ptr_b      "<<ptr_b<<"\n";
19     cout<<"Pointer ptr_Null "<<ptr_Null<<"\n";
20     return 0;
21 }

```

Die konkrete Zuordnung einer Variablen im Speicher wird durch den Compiler und das Betriebssystem bestimmt. Entsprechend kann die Adresse einer Variablen nicht durch den Programmierer festgelegt werden. Ohne Manipulationen ist die Adresse einer Variablen über die gesamte Laufzeit des Programms unveränderlich, ist aber bei mehrmaligen Programmstarts unterschiedlich.

In den Ausgaben von Pointer wird dann eine hexadezimale Adresse ausgegeben.

Zeiger können mit dem "Wert" NULL als ungültig markiert werden. Eine Dereferenzierung führt dann meistens zu einem Laufzeitfehler nebst Programmabbruch. NULL ist ein Macro und wird in mehreren Header-Dateien definiert (mindestens in <stddef> (stddef.h)). Die Definition ist vom Standard implementierungsabhängig vorgegeben und vom Compilerhersteller passend implementiert, z. B.

```

1  #define NULL 0
2  #define NULL 0L
3  #define NULL (void *) 0

```

Und umgekehrt, wie erhalten wir den Wert, auf den der Pointer zeigt? Hierfür benötigen wir den *Inhaltsoperator* *.

```

1  #include <iostream>
2  using namespace std;
3
4  int main(void)
5  {

```

```

6  int a = 15;
7  int * ptr_a = &a;
8  cout<<"Wert von a" <<a<<"\n";
9  cout<<"Pointer ptr_a" <<ptr_a<<"\n";
10 cout<<"Wert hinter dem Pointer ptr_a" <<*ptr_a<<"\n";
11 *ptr_a = 10;
12 cout<<"Wert von a" <<a<<"\n";
13 cout<<"Wert hinter dem Pointer ptr_a" <<*ptr_a<<"\n";
14 return 0;
15 }

```

6.4.3 Fehlerquellen

Fehlender Adressoperator bei der Zuweisung

```

1  #include <iostream>
2  using namespace std;
3
4  int main(void)
5  {
6      int a = 5;
7      int * ptr_a;
8      ptr_a = a;
9      cout<<"Pointer ptr_a" <<ptr_a<<"\n";
10     cout<<"Wert hinter dem Pointer ptr_a" <<*ptr_a<<"\n";
11     cout<<"Aus Maus!\n";
12     return 0;
13 }

```

Fehlender Dereferenzierungsoperator beim Zugriff

```

1  #include <iostream>
2  using namespace std;
3
4  int main(void)
5  {
6      int a = 5;
7      int * ptr_a = &a;
8      cout<<"Pointer ptr_a" <<((void*)ptr_a)<<"\n";
9      cout<<"Wert hinter dem Pointer ptr_a" <<*ptr_a<<"\n";
10     cout<<"Aus Maus!\n";
11     return 0;
12 }

```

Uninitialisierte Pointer zeigen "irgendwo ins nirgendwo"!

```

1  #include <iostream>
2  using namespace std;
3
4  int main(void)
5  {
6      int * ptr_a;
7      *ptr_a = 10;
8      // korrekte Initialisierung
9      // int * ptr_a = NULL;
10     // Prüfung auf gültige Adresse
11     // if (ptr_a != NULL) *ptr_a = 10;
12     cout<<"Pointer ptr_a" <<ptr_a<<"\n";
13     cout<<"Wert hinter dem Pointer ptr_a" <<*ptr_a<<"\n";
14     cout<<"Aus Maus!\n";
15     return 0;
16 }

```

6.4.4 Dynamische Datenobjekte

C++ bietet die Möglichkeit den Speicherplatz für eine Variable zur Laufzeit zur Verfügung zu stellen. Mit `new`-Operator wird der Speicherplatz bereit gestellt und mit `delete`-Operator (`delete[]`) wieder freigegeben.

`new` erkennt die benötigte Speichermenge am angegebenen Datentyp und reserviert für die Variable auf dem Heap die entsprechende Byte-Menge.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     int * ptr_a;
7     ptr_a=new int;
8     *ptr_a = 10;
9     cout<<"Pointer ptr_a          "<<ptr_a<<"\n";
10    cout<<"Wert hinter dem Pointer ptr_a  "<<*ptr_a<<"\n";
11    cout<<"Aus Maus!\n";
12    delete ptr_a;
13    return 0;
14 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     int * ptr_a;
7     ptr_a=new int[3];
8     ptr_a[0] = ptr_a[1] = ptr_a[2] = 42;
9     cout<<"Werte hinter dem Pointer ptr_a:  ";
10    for (int i=0;i<3;i++) cout<<ptr_a[i]<<" ";
11    cout<<"\n";
12    cout<<"Aus Maus!\n";
13    delete[] ptr_a;
14    return 0;
15 }
```

- `delete` darf nur einmal auf ein Objekt angewendet werden
- `delete` darf ausschließlich auf mit `new` angelegte Objekte oder `NULL`-Pointer angewandt werden
- Nach der Verwendung von `delete` ist das Objekt *undefiniert* (nicht gleich `NULL`)

Merke: Die Verwendung von Zeigern kann zur unerwünschten Speicherfragmentierung und die Programmierfehler zu den Programmabstürzen und Speicherlecks führen. *Intelligente* Zeiger stellen sicher, dass Programme frei von Arbeitsspeicher- und Ressourcenverlusten sind.

6.5 Referenz

Eine Referenz ist eine Datentyp, der Verweis (Aliasnamen) auf ein Objekt liefert und ist genau wie eine Variable zu benutzen ist. Der Vorteil der Referenzen gegenüber den Zeigern besteht in der einfachen Nutzung:

- Dereferenzierung ist nicht notwendig, der Compiler löst die Referenz selbst auf
- Freigabe ist ebenfalls nicht notwendig

Merke: Auch Referenzen müssen vor der Verwendung initialisiert werden. Eine Referenz bezieht sich immer auf ein existierendes Objekt, sie kann nie `NULL` sein

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     int a = 1;  // Variable
```

```

7 int &r = a; // Referenz auf die Variable a
8
9 std::cout << "a: " << a << " r: " << r << std::endl;
10 std::cout << "a: " << &a << " r: " << &r << std::endl;
11 }

```

Die Referenzen werden verwendet:

- zur “call bei reference”-Parameterübergabe
- zur Optimierung des Programms, um Kopien von Objekten zu vermeiden
- in speziellen Memberfunktionen, wie Copy-Konstruktor und Zuweisungsoperator
- als sogenannte universelle Referenz (engl.: universal reference), die bei Templates einen beliebigen Parametertyp repräsentiert.

Achtung: Zur dynamischen Verwaltung von Speicherbereichen sind Referenzen nicht geeignet.

6.6 Beispiel der Woche

Gegeben ist ein Array, das eine sortierte Reihung von Ganzzahlen umfasst. Geben Sie alle Paare von Einträgen zurück, die in der Summe 18 ergeben.

Die intuitive Lösung entwirft einen kreuzweisen Vergleich aller sinnvollen Kombinationen der n Einträge im Array. Dafür müssen wir $(n-1)^2/2$ Kombinationen bilden.

	1	2	5	7	9	10	12	13	16	17	18	21	25
1	x										18		
2	x	x								18			
5	x	x	x					18					
7	x	x	x	x									
9	x	x	x	x	x								
10	x	x	x	x	x	x							
12	x	x	x	x	x	x	x						
13	x	x	x	x	x	x	x	x					
16	x	x	x	x	x	x	x	x	x				
17	x	x	x	x	x	x	x	x	x	x			
18	x	x	x	x	x	x	x	x	x	x	x		
21	x	x	x	x	x	x	x	x	x	x	x	x	
25	x	x	x	x	x	x	x	x	x	x	x	x	x

Haben Sie eine bessere Idee?

```

1 #include <iostream>
2 using namespace std;
3 #define ZIELWERT 18
4
5 int main(void)
6 {
7     int a[] = {1, 2, 5, 7, 9, 10, 12, 13, 16, 17, 18, 21, 25};
8     int i_left=0;
9     int i_right=12;
10    cout<<"Value left "<<a[i_left]<<" right "<<a[i_right]<<"\n-----\n";
11    do{
12        cout<<"Value left "<<a[i_left]<<" right "<<a[i_right];
13        if (a[i_right] + a[i_left] == ZIELWERT){
14            cout<<" -> TREFFER";
15        }
16        cout<<"\n";
17        if (a[i_right] + a[i_left] >= ZIELWERT) i_right--;
18        else i_left++;
19    }while (i_right != i_left);
20    return 0;s

```

21 }



Kapitel 7

Grundlagen der Sprache C++

Parameter Kursinformationen

Veranstaltung: Prozedurale Programmierung / Einführung in die Informatik

Semester: Wintersemester 2022/23

Hochschule: Technische Universität Freiberg

Inhalte: Funktionen

Link auf: [https://github.com/TUBAF-Iff-](https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md)

Repository: [LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md](https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md)

Autoren: @author

Fragen an die heutige Veranstaltung ...

- Welche Komponenten beschreiben Definition einer Funktion?
- Wozu werden in der Funktion die Parameter gebraucht?
- Wann ist es sinnvoll Referenzen-Parameter zu verwenden?
- Warum ist es sinnvoll Funktionen in Look-Up-Tables abzubilden, letztendlich kostet das Ganze doch Speicherplatz?

7.1 Einschub - Klausurhinweise

- Während der Klausur können Sie “alle Hilfsmittel aus Papier” verwenden!
- Im OPAL finden sich Klausurbeispiele.

Beispielhafte Klausuraufgabe aus dem vergangenen Jahr

Die Zustimmung (in Prozent) für die Verwendung der künstlichen Intelligenz im Pflegebereich unter der Bevölkerung von Mauritius und Réunion soll vergleichend betrachtet werden. Die Ergebnisse der Umfragen für die Jahre 2010 bis 2020 (je ein Wert pro Jahr) in zwei Arrays erfasst werden (je ein Array pro Insel) und in einem Programm ausgewertet werden.

- Für beide Inseln soll aus den in Arrays erfassten Werten je ein Mittelwert berechnet werden. Schreiben Sie dazu eine Funktion, die ein Array übergeben bekommt und einen Mittelwert als ein Ergebnis an die main-Funktion zurück liefert. Rufen Sie die Funktion in der main-Funktion für jedes beider Arrays auf und geben Sie die Mittelwerte in der main-Funktion aus.
- Schreiben Sie eine weitere Funktion, die die korrespondierenden Werte beider Arrays miteinander vergleicht. Geben Sie für jedes Jahr aus, auf welcher Insel die Zustimmung größer war, bei den gleichen Werte ist eine entsprechende Meldung auszugeben. Rufen Sie die Funktion in der main-Funktion auf.
- In der main()-Funktion sind die Werte von der Console einzulesen und in die Arrays zu speichern.

7.2 Motivation

Erklären Sie die Idee hinter folgendem Code.

```

1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 #define VALUECOUNT 17
6
7 int main(void) {
8     int a [] = {1,2,3,3,4,2,3,4,5,6,7,8,9,1,2,3,4};
9
10    // Ergebnis Histogramm
11    int hist[10] = {0,0,0,0,0,0,0,0,0,0};
12    // Ergebnis Mittelwert
13    int summe = 0;
14    // Ergebnis Standardabweichung
15    float abweichung = 0;
16    for (int i=0; i<VALUECOUNT; i++){
17        hist[a[i]]++;
18        summe += a[i];
19    }
20    float mittelwert = summe / (float)VALUECOUNT;
21    for (int i=0; i<VALUECOUNT; i++){
22        abweichung += pow((a[i]-mittelwert),2.);
23    }
24    // Ausgabe
25    for (int i=0; i<10; i++){
26        printf("%d - %d\n", i, hist[i]);
27    }
28    // Ausgabe Mittelwert
29    cout<<"Die Summe betraegt "<<summe<<"", der Mittelwert "<<mittelwert<<"\n";
30    // Ausgabe Standardabweichung
31    float stdabw = sqrt(abweichung / VALUECOUNT);
32    cout<<"Die Standardabweichung der Grundgesamtheit betraegt "<<stdabw<<"\n";
33    return 0;
34 }
```

Ihre Aufgabe besteht nun darin ein neues Programm zu schreiben, das Ihre Implementierung der Mittelwertbestimmung integriert. Wie gehen Sie vor? Was sind die Herausforderungen dabei?

Stellen Sie das Programm so um, dass es aus einzelnen Bereichen besteht und überlegen Sie, welche Variablen wo gebraucht werden.

7.2.1 Prozedurale Programmierung Ideen und Konzepte

Bessere Lesbarkeit

Der Quellcode eines Programms kann schnell mehrere tausend Zeilen umfassen. Beim Linux Kernel sind es sogar über 15 Millionen Zeilen und Windows, das ebenfalls zum Großteil in C geschrieben wurde, umfasst schätzungsweise auch mehrere Millionen Zeilen. Um dennoch die Lesbarkeit des Programms zu gewährleisten, ist die Modularisierung unerlässlich.

Wiederverwendbarkeit

In fast jedem Programm tauchen die gleichen Problemstellungen mehrmals auf. Oft gilt dies auch für unterschiedliche Applikationen. Da nur Parameter und Rückgabetyt für die Benutzung einer Funktion bekannt sein müssen, erleichtert dies die Wiederverwendbarkeit. Um die Implementierungsdetails muss sich der Entwickler dann nicht mehr kümmern.

Wartbarkeit

Fehler lassen sich durch die Modularisierung leichter finden und beheben. Darüber hinaus ist es leichter, weitere Funktionalitäten hinzuzufügen oder zu ändern.

In allen 3 Aspekten ist der Vorteil in der Kapselung der Funktionalität zu suchen.

7.2.2 Anwendung

Funktionen sind Unterprogramme, die ein Ausgangsproblem in kleine, möglicherweise wiederverwendbare Codeelemente zerlegen.

```

1 #include <iostream>
2
3 using namespace std;
4 // Funktion für den Mittelwert
5 // Mittelwert = f_Mittelwert(daten)
6
7 // Funktion für die Standardabweichung
8 // Standardabweichung = f_Standardabweichung(daten)
9
10 // Funktion für die Histogrammgenerierung
11 // Histogramm = f_Histogramm(daten)
12
13 // Funktion für die Ausgabe
14 // f_Ausgabe(daten, {Mittelwert, Standardabweichung, Histogramm})
15
16 int main(void) {
17     int a[] = {3,4,5,6,2,3,2,5,6,7,8,10};
18     // b = f_Mittelwert(a) ...
19     // c = f_Standardabweichung(a) ...
20     // d = f_Histogramm(a) ...
21     // f_Ausgabe(a, b, c, d) ...
22     return 0;
23 }
```

Wie findet sich diese Idee in großen Projekten wieder?

Write Short Functions

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

[Google Style Guide für C++ Projekte](#)

7.3 C++ Funktionen

7.3.1 Funktionsdefinition

```

1 Rückgabedatentyp Funktionsname([Parameterliste]) {
2     /* Anweisungsblock mit Anweisungen */
3     [return Rückgabewert]
4 }
```

- **Rückgabedatentyp** - Welchen Datentyp hat der Rückgabewert?

Eine Funktion ohne Rückgabewert wird vom Programmierer als `void` deklariert. Sollten Sie keinen Rückgabetyt angeben, so wird automatisch eine Funktion mit Rückgabewert vom Datentyp `int` erzeugt.

- **Funktionsname** - Dieser Bestandteil der Funktionsdefinition ist eine eindeutige Bezeichnung, die für den Aufruf der Funktion verwendet wird.

Es gelten die gleichen Regeln für die Namensvergabe wie für Variablen.

- **Parameterliste** - Parameter sind Variablen (oder Pointer bzw. Referenzen darauf) die durch einen Datentyp und einen Namen spezifiziert werden. Mehrere Parameter werden durch Kommas getrennt.

Parameterliste ist optional, die Klammern jedoch nicht. Alternative zur fehlenden Parameterliste ist die Liste aus einem Parameter vom Datentyp `void` ohne Angabe des Namen.

- **Anweisungsblock** - Der Anweisungsblock umfasst die im Rahmen der Funktion auszuführenden Anweisungen und Deklarationen. Er wird durch geschweifte Klammern gekapselt.

Für die Funktionen gelten die gleichen Gültigkeits- und Sichtbarkeitsregeln wie für die Variablen.

7.3.2 Beispiele für Funktionsdefinitionen

```

1 int main (void) {
2     /* Anweisungsblock mit Anweisungen */
3 }

1 double pow (double base, double exponent){
2     /* Anweisungsblock mit Anweisungen */
3 }
4
5 //int y = pow(25.0,0.5));

1 void tauschen(int &var1,int &var2){
2     /* Anweisungsblock mit Anweisungen */
3 }

1 int mittelwert(int * array){
2     /* Anweisungsblock mit Anweisungen */
3 }
```

7.3.3 Aufruf der Funktion

Merke: Die Funktion (mit der Ausnahme der `main`-Funktion) wird erst ausgeführt, wenn sie aufgerufen wird. Vor dem Aufruf muss die Funktion definiert oder deklariert werden.

Der Funktionsaufruf einer Funktionen mit dem Rückgabewert kann Teil einer Anweisung, z.B. einer Zuweisung oder einer Ausgabeanweisung.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void info(){
6     cout<<"Dieses Programm rundet Zahlenwerte.\n";
7     cout<<"-----\n";
8 }
9
10 int runden(float a){
11     if (a < 0)
12         return (int)(a - 0.5);
13     else
14         return (int)(a + 0.5);
15 }
16
17 float rundenf(float a, int nachkomma){
18     float shifted= a* pow(10, nachkomma);
19     int result=0;
20     if (shifted < 0)
21         result = int(shifted -0.5);
```

```

22     else
23         result = int(shifted + 0.5);
24     return (float)result * pow(10, -nachkomma);
25 }
26
27 int main(void){
28     info();
29     float input = -8.4565;
30     cout<<"Eingabewert "<<input<<" - Ausgabewert "<<runden(input)<<"\n";
31     cout<<"Eingabewert "<<input<<" - Ausgabewert "<<rundenf(input,1)<<"\n";
32     return 0;
33 }

```

Die Funktion `runden` nutzt die Funktionalität des Cast-Operators `int` aus.

- Wenn N eine positive Zahl ist, wird 0.5 addiert
 - $15.2 + 0.5 = 15.7$ `int(15.7) = 15`
 - $15.7 + 0.5 = 16.2$ `int(16.2) = 16`
- Wenn N eine negative Zahl ist, wird 0.5 subtrahiert
 - $-15.2 - 0.5 = -15.7$ `int(-15.7) = -15`
 - $-15.7 - 0.5 = -16.2$ `int(-16.2) = -16`

Welche Verbesserungsmöglichkeit sehen Sie bei dem Programm? Tipp: Wie können wir den redundanten Code eliminieren?

Hinweis: C++ unterstützt gleiche Codenamen bei unterschiedlichen Parametern. Der Compiler “sucht sich” die passende Funktion aus. Der Mechanismus wird als *Funktionsüberladung* bezeichnet.

7.3.4 Fehler

Rückgabewert ohne Rückgabedefinition

```

1 void foo()
2 {
3     /* Code */
4     return 5; /* Fehler */
5 }
6
7 int main(void)
8 {
9     foo()
10    return 0;
11 }

```

Erwartung eines Rückgabewertes

```

1 #include <iostream>
2 using namespace std;
3
4 void foo(){
5     cout<<"Ausgabe";
6 }
7
8 int main(void) {
9     int i = foo();
10    return 0;
11 }

```

Falscher Rückgabebetyp

```

1 #include <iostream>
2 using namespace std;

```

```

3
4 float foo(){
5     return 3.123f;
6 }
7
8 int main(void) {
9     int i = foo();
10    cout<<i<<"\n";
11    return 0;
12 }

```

Parameterübergabe ohne entsprechende Spezifikation

```

1 #include <iostream>
2
3 int foo(void){           // <- Die Funktion erwartet explizit keine Parameter
4     return 3;
5 }
6
7 int main(void) {
8     int i = foo(5);
9     return 0;
10 }

```

Anweisungen nach dem return-Schlüsselwort

```

1 int foo()
2 {
3     return 5;
4     /* Code */           // Wird nie erreicht!
5 }

```

Falsche Reihenfolgen der Parameter

```

1 #include <iostream>
2 using namespace std;
3
4 void foo(int index, float wert){
5     cout<<"Index   - Wert\n";
6     cout<<index<<"   - "<<wert<<"\n\n";
7 }
8
9 int main(void) {
10    foo(4, 6.5);
11    foo(6.5, 4);
12    return 0;
13 }

```

7.3.5 Funktionsdeklaration

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void) {
5     int i = foo();           // <- Aufruf der Funktion
6     cout<<"i="<<i<<"\n";
7     return 0;
8 }
9
10 int foo(void){              // <- Definition der Funktion
11     return 3;
12 }

```

Damit der Compiler überhaupt von einer Funktion Kenntnis nimmt, muss diese vor ihrem Aufruf bekannt gegeben werden. Im vorangegangenen Beispiel wird die die Funktion erst nach dem Aufruf definiert. Der Compiler zeigt dies an.

Eine explizite Deklaration zeigt folgendes Beispiel:

```
1 #include <iostream>
2 using namespace std;
3
4 int foo(void); // Explizite Einführung der Funktion foo()
5
6 int main(void) {
7     int i = foo(); // <- Aufruf der Funktion
8     cout << "i=" << i << "\n";
9     return 0;
10 }
11
12 int foo(void) { // <- Definition der Funktion foo()
13     return 3;
14 }
```

Das Ganze wird dann relevant, wenn Funktionen aus anderen Quellcodedateien eingefügt werden sollen. Die Deklaration macht den Compiler mit dem Aussehen der Funktion bekannt. Diese werden mit dem Schlüsselwort `extern` markiert.

```
1 extern float berechneFlaeche(float breite, float hoehe);
```

7.3.6 Parameterübergabe und Rückgabewerte

Bisher wurden Funktionen betrachtet, die skalare Werte als Parameter erhielten und ebenfalls einen skalaren Wert als einen Rückgabewert lieferten. Allerdings ist diese Möglichkeit sehr einschränkend.

Es wird in vielen Programmiersprachen, darunter in C/C++, zwei Konzepte der Parameterübergabe realisiert.

call-by-value

In allen Beispielen bis jetzt wurden Parameter an die Funktionen *call-by-value*, übergeben. Das bedeutet, dass innerhalb der aufgerufenen Funktion mit einer Kopie der Variable gearbeitet wird und die Änderungen sich nicht auf den ursprünglichen Wert auswirken.

```
1 #include <iostream>
2 using namespace std;
3
4 // Definitionsteil
5 void doSomething(int a) { cout << ++a << " a in der Schleife\n"; }
6
7 int main(void) {
8     int a = 5;
9     cout << a << " a in main\n";
10    doSomething(a);
11    cout << a << " a in main\n";
12    return 0;
13 }
```

call-by-reference

Bei einer Übergabe als Referenz wirken sich Änderungen an den Parametern auf die ursprünglichen Werte aus, es werden keine Kopien von Parametern angelegt. *Call-by-reference* wird unbedingt notwendig, wenn eine Funktion mehrere Rückgabewerte hat.

In C++ wird die "call-by-reference"- Parameterübergabe mit Hilfe der Referenzen realisiert. In der Liste der formalen Parameter wird eine Referenz eines passenden Typs definiert. Beim Funktionsaufruf wird als Argument eine Variable des gleichen Datentyps übergeben.

```
1 #include <iostream>
2 using namespace std;
```

```

3
4 void inkrementieren(int &variable){
5     variable++;
6 }
7
8 int main(void) {
9     int a=0;
10    inkrementieren(a);
11    cout<<"a = "<<a<<"\n";
12    inkrementieren(a);
13    cout<<"a = "<<a<<"\n";
14    return 0;
15 }

```

Der Vorteil der Verwendung der Referenzen als Parameter besteht darin, dass in der Funktion mehrere Variablen auf eine elegante Weise verändert werden können. Die Funktion hat somit quasi mehrere Ergebnisse.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void tauschen(char &anna, char &hanna){
6     char aux=anna;
7     anna=hanna;
8     hanna=aux;
9 }
10
11 int main(void) {
12     char anna='A',hanna='H';
13     cout<<anna<<" und "<<hanna<<"\n";
14     tauschen(anna,hanna);
15     cout<<anna<<" und "<<hanna<<"\n";
16     return 0;
17 }

```

Es besteht ebenfalls die Möglichkeit, die bereits in C zur Verfügung stand, “call-by-reference”- Parameterübergabe mit Hilfe der Zeiger (Pointer) zu realisieren. Im allgemeinen ist die Verwendung von Referenzen vorzuziehen, bei Übergabe der Array-Parameter wird jedoch der Zeiger verwendet.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 double sinussatz(double *lookup_sin, int angle, double opositeSide){
6     return opositeSide*lookup_sin[angle];
7 }
8 /*
9 double sinussatz(double lookup_sin[], int angle, double opositeSide){
10     return opositeSide*lookup_sin[angle];
11 }
12 */
13
14 int main(void) {
15     double sin_values[360] = {0};
16     for(int i=0; i<360; i++) {
17         sin_values[i] = sin(i*M_PI/180);
18     }
19     cout<<"Größe des Arrays "<<sizeof(sin_values)<<"\n";
20     cout<<"Result = "<<sinussatz(sin_values, 30, 20)<<" \n";
21     return 0;
22 }

```


Statt Zeiger kann die Notation als Array undefinierter (definierter) Größe verwendet werden. Unabhängig von der Notation entspricht die Größe des übergebenen Arrays der Definition in der aufrufenden Funktion (hier main-Funktion).

Auch bei der Verwendung von Zeigern und Referenzen werden keine Kopien von Paramern angelegt, sondern die Parameter selbst direkt verändert. Falls keine Veränderung angestrebt wird, aber das Anlegen von Kopien vermieden werden soll, können konstante Zeiger bzw. Referenzen verwendet werden.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 double sinussatz(const double *lookup_sin, int angle, double oppositeSide){
6     return oppositeSide*lookup_sin[angle];
7 }
8
9 int main(void) {
10     double sin_values[360] = {0};
11     for(int i=0; i<360; i++) {
12         sin_values[i] = sin(i*M_PI/180);
13     }
14     cout<<"Größe des Arrays "<<sizeof(sin_values)<<"\n";
15     cout<<"Result = "<<sinussatz(sin_values, 30, 20)<<" \n";
16     return 0;
17 }

```

7.3.7 Zeiger und Referenzen als Rückgabewerte

Analog zur Bereitstellung von Parametern entsprechend dem “call-by-reference” Konzept können auch Rückgabewerte als Pointer oder Referenz vorgesehen sein. Allerdings sollen Sie dabei aufpassen ...

```

1 #include <iostream>
2 using namespace std;
3
4 int& doCalc(int &wert) {
5     int a = wert + 5;
6     return a;
7 }
8
9 int main(void) {
10     int b = 5;
11     cout<<"Irgendwas stimmt nicht "<<doCalc(b);
12     return 0;
13 }

```

Mit dem Beenden der Funktion werden deren lokale Variablen vom Stack gelöscht. Um diese Situation zu handhaben können Sie zwei Lösungsansätze realisieren.

Variante 1 Sie übergeben den Rückgabewert in der Parameterliste.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void kreisflaeche(double durchmesser, double &flaeche) {
6     flaeche = M_PI * pow(durchmesser / 2, 2);
7     // Hier steht kein return !
8 }
9
10 int main(void) {
11     double wert = 5.0;
12     double flaeche = 0;
13     kreisflaeche(wert, flaeche);

```

```

14  cout<<"Die Kreisfläche beträgt für d="<<wert<<"[m] "<<flaeche<<"[m²] \n";
15  return 0;
16 }

```

Variante 2 Für den Rückgabezeiger wird der Speicherplatz mit `new` dynamisch angelegt, aber Achtung: zu jedem `new` gehört ein `delete`.

```

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  double* kreisflaeche(double durchmesser) {
6      double *flaeche=new double;
7      *flaeche = M_PI * pow(durchmesser / 2, 2);
8      return flaeche;
9  }
10
11 int main(void) {
12     double wert = 5.0;
13     double *flaeche;
14     flaeche=kreisflaeche(wert);
15     cout<<"Die Kreisfläche beträgt für d="<<wert<<"[m] "<<*flaeche<<"[m²] \n";
16     delete flaeche;
17     return 0;
18 }

```

7.3.8 Besonderheit Arrays

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  void printSizeOf(int intArray[])
6  {
7      cout<<"sizeof of parameter: "<<sizeof(intArray)<<"\n";
8  }
9
10 void printLength(int intArray[])
11 {
12     cout<<"Length of parameter: "<<sizeof(intArray) / sizeof(intArray[0])<<"\n";
13 }
14
15 int main()
16 {
17     int array[] = { 0, 1, 2, 3, 4, 5, 6 };
18
19     cout<<"sizeof of array: "<<sizeof(array)<<"\n";
20     printSizeOf(array);
21
22     cout<<"Length of array: "<< sizeof(array) / sizeof(array[0])<<"\n";
23     printLength(array);
24 }

```

7.3.9 main-Funktion

In jedem Programm muss und darf nur eine `main`-Funktion geben. Diese Funktion wird beim Programmstart automatisch ausgeführt.

Definition der `main`-Funktion:

```

1  int main(void) {
2      /*Anweisungen*/
3  }

```


28 }

Aufgabe: Wie könnten wir den Code abwandeln, um eine Laufschrift umzusetzen?

```
1 void setup() {
2   pinMode(LED_BUILTIN, OUTPUT);
3   Serial.begin(9600);
4   String text = "Das ist ein Test";
5   Serial.println(text);
6   String output = "";
7   int signs_per_line = 5;
8   for (int i=0; i<text.length(); i++){
9     if (i <= text.length()-signs_per_line)
10      output = text.substring(i, i+signs_per_line);
11     else
12      output = text.substring(i, text.length()) +
13      text.substring(0, signs_per_line-(text.length()-i));
14     Serial.println(output);
15   }
16 }
17
18 void loop() {
19 }
```

@AVR8js.sketch

Kapitel 8

Objektorientierte Programmierung mit C++

Parameter	Kursinformationen
Veranstaltung	Prozedurale Programmierung / Einführung in die Informatik
Semester	Wintersemester 2022/23
Hochschule:	Technische Universität Freiberg
Inhalte:	Klassen und Objekte
Link auf	https://github.com/TUBAF-IfL-
Repository:	LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md
Autoren	@author

Fragen an die heutige Veranstaltung ...

- Wie strukturieren wir unseren Code?
- Was sind Objekte?
- Welche Aufgabe erfüllt ein Konstruktor?
- Was geschieht, wenn kein expliziter Konstruktor durch den Entwickler vorgesehen wurde?
- Wann ist ein Dekonstruktor erforderlich?
- Was bedeutet das “Überladen” von Funktionen?
- Nach welchen Kriterien werden überladene Funktionen differenziert?

8.1 Motivation

Aufgabe: Statistische Untersuchung der Gehaltsentwicklung in Abhängigkeit vom Alter.

Welche zwei Elemente machen eine solche Untersuchung aus?

Daten	Funktionen
Name	Alter bestimmen
Geburtsdatum	Daten ausgeben
Gehalt	
...	

Wir entwerfen also eine ganze Sammlung von Funktionen wie zum Beispiel für `alterbestimmen()`:

```
1 int alterbestimmen(int tag, int monat, int jahr,  
2                     int akt_tag, int akt_monat, int akt_jahr)  
3 {
```

```

4  //TODO
5  }
6
7  int main(){
8      int tag, monat, jahr;
9      int akt_tag, akt_monat, akt_jahr;
10     int alter=alterbestimmen(tag,monat,jahr,akt_tag,akt_monat,akt_jahr);
11 }

```

Was gefällt ihnen an diesem Ansatz nicht?

- lange Parameterliste bei der Funktionen
- viele Variablen
- der inhaltliche Zusammenhang der Daten schwer zu erkennen

Lösungsansatz 1: Wir fassen die Daten zusammen in einer übergreifenden Datenstruktur **struct**.

Ein struct vereint unterschiedliche Aspekte eines Datensets in einer Variablen.

```

1  struct Datum{ // hier wird ein neuer Datentyp definiert
2      int tag, monat, jahr;
3  };
4
5  int alterbestimmen(Datum geb_datum, Datum akt_datum)
6  {
7      //TODO
8      return 0;
9  }
10
11 int main(){
12     Datum geburtsdatum;
13     Datum akt_datum; // ... und hier wird eine Variable des
14                     // Typs angelegt
15     geburtsdatum.tag = 5; // ... und "befüllt"
16     geburtsdatum.monat = 10;
17     geburtsdatum.jahr = 1923;
18     int alter=alterbestimmen(geburtsdatum, akt_datum);
19 }

```

Mit **struct Datum** wurde ein Datentyp definiert, **Datum geburtsdatum**; definiert eine Variable (besser gesagt ein Objekt).

Was gefällt ihnen an diesem Ansatz nicht?

- die Funktionen sind von dem neuen Datentyp abhängig gehören aber trotzdem nicht zusammen
- es fehlt eine Überprüfung der Einträge für die Gültigkeit unserer Datumsangaben

Die objektorientierte Programmierung (OOP) ist ein Programmierparadigma, das auf dem Konzept der "Objekte" basiert, die Daten und Code enthalten können: Daten in Form von Feldern (oft als Attribute oder Eigenschaften bekannt) und Code in Form von Prozeduren (oft als Methoden bekannt).

```

1  struct Datum{
2      int tag,monat,jahr;
3      int alterbestimmen(Datum akt_datum)
4      {
5          //hier sind tag, monat und jahr bereits bekannt
6          //TODO
7      }
8  }
9
10 int main(){
11     Datum geburtsdatum;
12     Datum akt_datum;
13     int alter=geburtsdatum.alterbestimmen(akt_datum);
14 }

```

C++ sieht vor als Datentyp für Objekte **struct**- und **class** - Definitionen. Der Unterschied wird später geklärt, vorerst verwenden wird nur die **class** - Definitionen.

```

1 class Datum{
2 public:
3     int tag,monat,jahr;
4     int alterbestimmen(Datum akt_datum)
5     {
6         //hier sind tag, monat und jahr bereits bekannt
7         //TODO
8     }
9 }
10
11 int main(){
12     Datum geburtsdatum;
13     Datum akt_datum;
14     int alter=geburtsdatum.alterbestimmen(akt_datum);
15 }
```

Ein Merkmal von Objekten ist, dass die eigenen Prozeduren eines Objekts auf die Datenfelder seiner selbst zugreifen und diese oft verändern können - Objekte haben somit eine Vorstellung davon oder von sich selbst :-).

Ein OOP-Computerprogramm kombiniert Objekt und lässt sie interagieren. Viele der am weitesten verbreiteten Programmiersprachen (wie C++, Java, Python usw.) sind Multi-Paradigma-Sprachen und unterstützen mehr oder weniger objektorientierte Programmierung, typischerweise in Kombination mit imperativer, prozeduraler Programmierung.

1 main	Farm	Animal
2 +-----+	+-----+	+-----+
3 Farm JohnsFarm;	-> Animal myAnimals[];	-> string name;
4 Farm PetersFarm;	checkAnimalsPosition();	...
5 ...	feedAnimals();	
6	getAnimalStatistics();	Building
7	...	+-----+
8	Farmbuilding buildings[];	-> int purpose
9		startScanning()
10		...

Wir erzeugen ausgehend von unserem Bauplan verschiedene Instanzen / Objekte vom Typ **Animals**. Jede hat den gleichen Funktionsumfang, aber unterschiedliche Daten.

Merke: Unter einer Klasse versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.

8.2 C++ - Entwicklung

C++ eine von der ISO genormte Programmiersprache. Sie wurde ab 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt. Bezeichnenderweise trug C++ zunächst den Namen "**C with classes**". C++ erweitert die Abstraktionsmöglichkeiten erlaubt aber trotzdem eine maschinennahe Programmierung. Der Standard definiert auch eine Standardbibliothek, zu der wiederum verschiedene Implementierungen existieren.

Jahr	Entwicklung
197?	Anfang der 70er Jahre entsteht die Programmiersprache C
...	
1979	„C with Classes“ - Klassenkonzept mit Datenkapselung, abgeleitete Klassen, ein strenges Typsystem, Inline-Funktionen und Standard-Argumente
1983	“C++” - Überladen von Funktionsnamen und Operatoren, virtuelle Funktionen, Referenzen, Konstanten, eine änderbare Freispeicherverwaltung
1985	Referenzversion
1989	Version 2.0 - Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen

Jahr	Entwicklung
1998	ISO/IEC 14882:1998 oder C++98
...	Kontinuierliche Erweiterungen C++11, C++20, ...

C++ kombiniert die Effizienz von C mit den Abstraktionsmöglichkeiten der objektorientierten Programmierung. C++ Compiler können C Code überwiegend kompilieren, umgekehrt funktioniert das nicht.

8.3 ... das kennen wir doch schon

Ein- und Ausgabe

Bereits häufig verwendete `std::cin` und `std::cout` sind Objekte:

- `std::cin` ist ein Objekt der Klasse `istream`
- `std::cout` ist ein Objekt der Klasse `ostream`

```
1 extern std::istream cin;
```

```
1 extern std::ostream cout;
```

Sie werden in `<iostream>` als Komponente der Standard Template Library (STL) im Namensraum `std` definiert und durch `#include <iostream>` bekannt gegeben.

Die Streams lassen sich nicht nur für die Standard Ein- und Ausgabe verwenden, sondern auch mit Dateien oder Zeichenketten.

Mehr zu `iostream` unter <https://www.c-plusplus.net/forum/topic/152915/ein-und-ausgabe-in-c-io-streams>

Klasse `string`

C++ implementiert eine separate Klasse `string`-Datentyp (Klasse). In Programmen müssen nur Objekte dieser Klasse angelegt werden. Beim Anlegen eines Objektes muss nicht angegeben werden, wie viele Zeichen darin enthalten werden sollen, eine einfache Zuweisung reicht aus.

```
1 #include <iostream>
2 #include <string>
3
4 int main(void) {
5     std::string hallo;
6     hallo="Hallo";
7     std::string hanna = "Hanna";
8     std::string anna("Anna");
9     std::string alleBeide = anna + " + " + hanna;
10    std::cout<<hallo<<" "<<alleBeide<<std::endl;
11    return 0;
12 }
```

Arduino Klassen

Die Implementierungen für unseren Mikrocontroller sind auch objektorientiert. Klassen repräsentieren unserer Hardwarekomponenten und sorgen für deren einfache Verwendung.

```
1 #include <OledDisplay.h>
2 ...
3 Screen.init();
4 Screen.print("This is a very small display including only 4 lines", true);
5 Screen.draw(0, 0, 128, 8, BMP);
6 Screen.clean();
7 ...
```


8.4 Definieren von Klassen und Objekten

Eine Klasse wird in C++ mit dem Schlüsselwort `class` definiert und enthält Daten (member variables, Attribute) und Funktionen (member functions, Methoden).

Klassen verschmelzen Daten und Methoden in einem “Objekt” und deklarieren den individuellen Zugriff. Die wichtigste Eigenschaft einer Klasse ist, dass es sich um einen Typ handelt!

```

1 class class_name {
2     access_specifier_1:
3         member1;
4     access_specifier_2:
5         member2;
6     ...
7 }; // <- Das Semikolon wird gern vergessen
8
9 class_name instance_name;
```

Bezeichnung	Bedeutung
<code>class_name</code>	Bezeichner für die Klasse - Typ
<code>instance_name</code>	Objekte der Klasse <code>class_name</code> - Instanz
<code>access_specifier</code>	Zugriffsberechtigung

Der Hauptteil der Deklaration kann *member* enthalten, die entweder Daten- oder Funktionsdeklarationen sein können und jeweils einem Zugriffsbezeichner . Ein Zugriffsbezeichner ist eines der folgenden drei Schlüsselwörter: `private`, `public` oder `protected`. Diese Bezeichner ändern die Zugriffsrechte für die *member*, die ihnen nachfolgen:

- `private member` einer Klasse sind nur von anderen *members* derselben Klasse (oder von ihren “Freunden”) aus zugänglich.
- `protected member` sind von anderen *member* derselben Klasse (oder von ihren “Freunden”) aus zugänglich, aber auch von Mitgliedern ihrer abgeleiteten Klassen.
- `public member` sind öffentliche *member* von überall her zugänglich, wo das Objekt sichtbar ist.

Standardmäßig sind alle Members in der Klasse `private`!

Merke: Klassen und Strukturen unterscheiden sich unter C++ durch die Default-Zugriffsrechte und die Möglichkeit der Vererbung. Anders als bei `class` sind die *member* von `struct` per Default `public`. Die Veränderung der Zugriffsrechte über die oben genannten Zugriffsbezeichner ist aber ebenfalls möglich.

Im Folgenden fokussieren die Ausführungen Klassen, eine analoge Anwendung mit Strukturen ist aber zumeist möglich.

```

1 #include <iostream>
2
3 class Datum
4 {
5     public:
6         int tag;
7         int monat;
8         int jahr;
9
10        void print(){
11            std::cout << tag << "." << monat << "." << jahr << std::endl;
12        }
13 };
14
15 int main()
16 {
17     // 1. Instanz der Klasse, Objekt myDatumA
18     Datum myDatumA;
```

```

19 myDatumA.tag = 12;
20 myDatumA.monat = 12;
21 myDatumA.jahr = 2000;
22 myDatumA.print();
23
24 // 2. Instanz der Klasse, Objekt myDatumB
25 // alternative Initialisierung
26 Datum myDatumB = {.tag = 12, .monat = 3, .jahr = 1920};
27 myDatumB.print();
28 return 0;
29 }

```

Und wie können wir weitere Methoden zu unserer Klasse hinzufügen?

```

1 #include <iostream>
2
3 class Datum{
4 public:
5     int tag;
6     int monat;
7     int jahr;
8
9     const int monthDays[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
10
11     void print(){
12         std::cout << tag << "." << monat << "." << jahr << std::endl;
13     }
14
15     int istSchaltJahr(){
16         if (((jahr % 4 == 0) && (jahr % 100 != 0)) || (jahr % 400 == 0)) return 1;
17         else return 0;
18     }
19
20     int nterTagimJahr(){
21         int n = tag;
22         for (int i=0; i<monat - 1; i++){
23             n += monthDays[i];
24         }
25         if (monat > 2) n += istSchaltJahr();
26         return n;
27     }
28 };
29
30 int main()
31 {
32     Datum myDatumA;
33     myDatumA.tag = 31;
34     myDatumA.monat = 12;
35     myDatumA.jahr = 2000;
36     myDatumA.print();
37     std::cout << myDatumA.ntenTagimJahr() << "ter Tag im Jahr " << myDatumA.jahr << std::endl;
38     return 0;
39 }

```

8.4.1 Objekte in Objekten

Natürlich lassen sich Klassen beliebig miteinander kombinieren, was folgende Beispiel demonstriert.

```

1 #include <iostream>
2 #include <ctime>
3 #include <string>
4

```

```

5 class Datum{
6     public:
7         int tag;
8         int monat;
9         int jahr;
10
11     void print(){
12         std::cout << tag << "." << monat << "." << jahr <<std::endl;
13     }
14 };
15
16 class Person{
17     public:
18         Datum geburtstag;
19         std::string name;
20
21     void print(){
22         std::cout << name << ": ";
23         geburtstag.print();
24         std::cout <<std::endl;
25     }
26
27     int zumGeburtstagAnrufen() {
28         time_t t = time(NULL);
29         tm* tPtr = localtime(&t);
30         if ((geburtstag.tag == tPtr->tm_mday) &&
31             (geburtstag.monat == (tPtr->tm_mon + 1))) {
32             std::cout << "\"Weil heute Dein ... \"" <<std::endl;
33             return 1;
34         }
35         else return -1;
36     }
37 };
38
39 int main()
40 {
41     Person freundA = {.geburtstag = {1, 12, 2022}, .name = "Peter"};
42     freundA.print();
43     if (freundA.zumGeburtstagAnrufen() == 1){
44         std::cout << "Zum Geburtstag gratuliert!" <<std::endl;
45     }
46     else{
47         std::cout << freundA.name << " hat heute nicht Geburtstag" <<std::endl;
48     }
49     return 0;
50 }

```

8.4.2 Datenkapselung

Als Datenkapselung bezeichnet man das Verbergen von Implementierungsdetails einer Klasse. Auf die interne Daten kann nicht direkt zugegriffen werden, sondern nur über definierte Schnittstelle, die durch `public`-Methoden repräsentiert wird.

- get- und set-Methoden
- andere Methoden

```

1 #include <iostream>
2 using namespace std;
3 class Datum
4 {
5     private:
6         int tag;

```

```

7     int monat;
8     int jahr;
9 public:
10    void setTag(int _tag){
11        tag=_tag;
12    }
13    void setMonat(int _monat){
14        monat=_monat;
15    }
16    void setJahr(int _jahr){
17        jahr=_jahr;
18    }
19    int getTag(){
20        return tag;
21    }
22    //analog monat und jahr
23    void ausgabe()
24    {
25        cout<<tag<<"."<<monat<<"."<<jahr<<endl;
26    }
27 };
28
29 int main()
30 {
31     Datum datum;
32     datum.setTag(31);datum.setMonat(12);datum.setJahr(1999);
33     datum.ausgabe();//jetzt geht es
34 }

```

8.4.3 Memberfunktionen

Mit der Integration einer Funktion in eine Klasse wird diese zur *Methode* oder *Memberfunktion*. Der Mechanismus der Nutzung bleibt der gleiche, es erfolgt der Aufruf, ggf mit Parametern, die Abarbeitung realisiert Berechnungen, Ausgaben usw. und ein optionaler Rückgabewert bzw. geänderte Parameter (bei Call-by-Referenz Aufrufen) werden zurückgegeben.

Worin liegt der technische Unterschied?

```

1 #include <iostream>
2
3 class Student{
4 public:
5     std::string name;    // "-"
6     int alter;
7     std::string ort;
8
9     void ausgabeMethode(){
10        std::cout << name << " " << ort << " " << alter << std::endl;
11    }
12 };
13
14 void ausgabeFunktion(Student studentA){
15     std::cout << studentA.name << " " << studentA.ort << " " << studentA.alter << std::endl;
16 }
17
18 int main()
19 {
20     Student bernhard {"Cotta", 25, "Zillbach"};
21     bernhard.ausgabeMethode();
22
23     ausgabeFunktion(bernhard);
24 }

```

```

25  return 0;
26 }

```

Methoden können auch von der Klassendefinition getrennt werden.

```

1  #include <iostream>
2
3  class Student{
4  public:
5      std::string name;  // "-"
6      int alter;
7      std::string ort;
8
9      void ausgabeMethode();    // Deklaration der Methode
10 };
11
12 // Implementierung der Methode
13 void Student::ausgabeMethode(){
14     std::cout << name << " " << ort << " " << alter << std::endl;
15 }
16
17 int main()
18 {
19     Student bernhard {"Cotta", 25, "Zillbach"};
20     bernhard.ausgabeMethode();
21     return 0;
22 }

```

Diesen Ansatz kann man weiter treiben und die Aufteilung auf einzelne Dateien realisieren.

8.4.4 Modularisierung unter C++

Der Konzept der Modularisierung lässt sich unter C++ durch die Aufteilung der Klassen auf verschiedene Dateien umsetzen. Unser kleines Beispiel umfasst Klassen, die von einer `main.cpp` aufgerufen werden.

```

1  #include <iostream>
2
3  #ifndef DATUM_H_INCLUDED
4  #define DATUM_H_INCLUDED
5  /* ^^ these are the include guards */
6
7  class Datum
8  {
9  public:
10     int tag;
11     int monat;
12     int jahr;
13
14     void ausgabeMethode(){
15         std::cout << tag << "." << monat << "." << jahr << std::endl;
16     }
17 };
18
19 #endif

```

```

1  #ifndef STUDENT_H_INCLUDED
2  #define STUDENT_H_INCLUDED
3
4  #include <iostream>
5  #include <string>
6  #include "Datum.h"
7

```

```

8 class Student{
9     public:
10         std::string name;    // "-"
11         Datum geburtsdatum;
12         std::string ort;
13
14         void ausgabeMethode();    // Deklaration der Methode
15 };
16
17 #endif

```

```

1 #include "Student.h"
2
3 void Student::ausgabeMethode(){
4     std::cout << name << " " << ort << " ";
5     geburtsdatum.ausgabeMethode();
6     std::cout << std::endl;
7 }

```

```

1 #include <iostream>
2 #include "Student.h"
3
4 int main()
5 {
6     Datum datum{1,1,2000};
7     Student bernhard {"Cotta", datum, "Zillbach"};
8     bernhard.ausgabeMethode();
9     return 0;
10 }

```

```
1 g++ -Wall main.cpp Student.cpp -o a.out
```

Definitionen der Klassen erfolgen in den Header-Dateien (.h), wobei für die meisten member-Funktionen nur die Deklarationen angegeben werden. In den Implementierungsdateien (.cpp) werden die Klassendefinitionen mit include-Anweisungen bekannt gemacht und die noch nicht implementierten member-Funktionen implementiert.

Achtung: Die außerhalb der Klasse implementierte Funktionen erhalten einen Namen, der den Klassenname einschließt.

8.4.5 Überladung von Methoden

C++ verbietet für die Variablen und Objekte die gleichen Namen, erlaubt jedoch die variable Verwendung von Funktionen in Abhängigkeit von der Signatur der Funktion. Dieser Mechanismus heißt “Überladen von Funktionen” und ist sowohl an die global definierten Funktionen als auch an die Methoden der Klasse anwendbar.

Merke: Der Rückgabedatentyp trägt nicht zur Unterscheidung der Methoden bei. Unterscheidet sich die Signatur nur in diesem Punkt, “meckert” der Compiler.

```

1 #include <iostream>
2 #include <fstream>
3
4 class Seminar{
5     public:
6         std::string name;
7         bool passed;
8 };
9
10 class Lecture{
11     public:
12         std::string name;
13         float mark;
14 };
15

```

```

16 class Student{
17     public:
18         std::string name; // "-"
19         int alter;
20         std::string ort;
21
22         void printCertificate(Seminar sem){
23             std::string comment = " nicht bestanden";
24             if (sem.passed)
25                 comment = " bestanden!";
26             std::cout << "\n" << name << " hat das Seminar " << sem.name
27                 << comment;
28         }
29
30         void printCertificate(Lecture lect){
31             std::cout << "\n" << name << " hat in der Vorlesung " <<
32                 lect.name << " die Note " << lect.mark << " erreicht";
33         }
34 };
35
36 int main()
37 {
38     Student bernhard {"Cotta", 25, "Zillbach"};
39     Seminar roboticSeminar {"Robotik-Seminar", false};
40     Lecture ProzProg {"Prozedurale Programmierung", 1.3};
41
42     bernhard.printCertificate(roboticSeminar);
43     bernhard.printCertificate(ProzProg);
44
45     return 0;
46 }

```

Achtung: Im Beispiel erfolgt die Ausgabe nicht auf die Console, sondern in ein ostream-Objekt, dessen Referenz an die print-Methoden als Parameter übergeben wird. Das ermöglicht eine flexible Ausgabe, z.B. in eine Datei, auf den Drucker etc.

8.5 Ein Wort zur Ausgabe

```

1 #include <iostream>
2 #include <fstream>
3
4 class Seminar{
5     public:
6         std::string name;
7         bool passed;
8 };
9
10 class Lecture{
11     public:
12         std::string name;
13         float mark;
14 };
15
16 class Student{
17     public:
18         std::string name; // "-"
19         int alter;
20         std::string ort;
21
22         void printCertificate(std::ostream& os, Seminar sem);

```

```

23     void printCertificate(std::ostream& os, Lecture sem);
24 };
25
26 void Student::printCertificate(std::ostream& os, Seminar sem){
27     std::string comment = " nicht bestanden";
28     if (sem.passed)
29         comment = " bestanden!";
30     os << "\n" << name << " hat das Seminar " << sem.name << comment;
31 }
32
33 void Student::printCertificate(std::ostream& os, Lecture lect){
34     os << "\n" << name << " hat in der Vorlesung " << lect.name << " die Note " << lect.mark << "
        erreicht";
35 }
36
37 int main()
38 {
39     Student bernhard {"Cotta", 25, "Zillbach"};
40     Seminar roboticSeminar {"Robotik-Seminar", false};
41     Lecture ProzProg {"Prozedurale Programmierung", 1.3};
42
43     bernhard.printCertificate(std::cout, roboticSeminar);
44     bernhard.printCertificate(std::cout, ProzProg);
45
46     return 0;
47 }

```

8.6 Initialisierung/Zerstören eines Objektes

Die Klasse spezifiziert unter anderem (!) welche Daten in den Instanzen/Objekten zusammenfasst werden. Wie aber erfolgt die Initialisierung? Bisher haben wir die Daten bei der Erzeugung der Instanz übergeben.

```

1  #include <iostream>
2
3  class Student{
4  public:
5      std::string name;
6      int alter;
7      std::string ort;
8
9      void ausgabeMethode(std::ostream& os); // Deklaration der Methode
10 };
11
12 // Implementierung der Methode
13 void Student::ausgabeMethode(std::ostream& os){
14     os << name << " " << ort << " " << alter << "\n";
15 }
16
17 int main()
18 {
19     Student bernhard {"Cotta", 25, "Zillbach"};
20     bernhard.ausgabeMethode(std::cout);
21
22     Student alexander { .name = "Humboldt" , .ort = "Berlin" };
23     alexander.ausgabeMethode(std::cout);
24
25     Student unbekannt;
26     unbekannt.ausgabeMethode(std::cout);
27
28     return 0;
29 }

```


Es entstehen 3 Instanzen der Klasse `Student`, die sich im Variablennamen `bernhard`, `alexander` und `unbekannt` und den Daten unterscheiden.

Im Grunde können wir unsere drei Datenfelder im Beispiel in vier Kombinationen initialisieren:

```
1 {name, alter, ort}
2 {name, alter}
3 {name}
4 {}
```

Elementinitialisierung beim Aufruf:

Umsetzung	Beispiel
vollständige Liste in absteigender Folge (uniforme Initialisierung)	<code>Student Bernhard {"Cotta", 25, "Zillbach"};</code>
unvollständige Liste (die fehlenden Werte werden durch Standard Defaultwerte ersetzt)	<code>Student Bernhard {"Cotta", 25};</code>
vollständig leere Liste, die zum Setzen von Defaultwerten führt	<code>Student Bernhard {};</code>
Aggregierende Initialisierung (C++20)	<code>Student alexander = { .ort = "unknown"};</code>

Wie können wir aber:

- erzwingen, dass eine bestimmte Membervariable in jedem Fall gesetzt wird (weil die Klasse sonst keinen Sinn macht)
- prüfen, ob die Werte einem bestimmten Muster entsprechen ("Die PLZ kann keine negativen Werte umfassen")
- automatische weitere Einträge setzen (einen Zeitstempel, der die Initialisierung festhält)
- ... ?

8.6.1 Konstruktoren

Konstruktoren dienen der Koordination der Initialisierung der Instanz einer Klasse. Sie werden entweder implizit über den Compiler erzeugt oder explizit durch den Programmierer angelegt.

```
1 class class_name {
2     access_specifier_1:
3         typ member1;
4     access_specifier_2:
5         typ member2;
6     memberfunktionA(...)
7
8     class_name (...) {           // <- Konstruktor
9         // Initialisierungscode
10    }
11 };
12
13 class_name instance_name (...);
```

Merke: Ein Konstruktor hat keinen Rückgabetyt!

Beim Aufruf `Student bernhard {"Cotta", 25, "Zillbach"};` erzeugt der Compiler eine Methode `Student::Student(std::string, int, std::string)`, die die Initialisierungsparameter entgegennimmt und diese der Reihenfolge nach an die Membervariablen übergibt. Sobald wir nur einen expliziten Konstruktor integrieren, weist der Compiler diese Verantwortung von sich.

Entfernen Sie den Kommentar in Zeile 11 und der Compiler macht Sie darauf aufmerksam.

```
1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name; // "-"
```

```

6   int alter;
7   std::string ort;
8
9   void ausgabeMethode(std::ostream& os){
10      os << name << " " << ort << " " << alter;
11  }
12
13      //Student();
14 };
15
16 // Implementierung der Methode
17
18
19 int main()
20 {
21     Student bernhard {"Cotta", 25, "Zillbach"};
22     bernhard.ausgabeMethode(std::cout);
23     return 0;
24 }

```

Dabei sind innerhalb des Konstruktors zwei Schreibweisen möglich:

```

1 //Initalisierung
2 Student(std::string name, int alter, std::string ort): name(name), alter(alter), ort(ort)
3 {
4 }
5
6 // Zuweisung innerhalb des Konstruktors
7 Student(std::string name, int alter, std::string ort):
8     this->name = name;
9     this->alter = alter;
10    this->ort = ort;
11 }

```

Die zuvor beschriebene Methodenüberladung kann auch auf die Konstruktoren angewandt werden. Entsprechend stehen dann eigene Aufrufmethoden für verschiedene Datenkonfigurationen zur Verfügung. In diesem Fall können wir auf drei verschiedenen Wegen Default-Werte setzen:

- ohne spezifische Vorgabe wird der Standardinitialisierungswert verwendet (Ganzzahlen 0, Gleitkomma 0.0, Strings)
- die Vorgabe eines individuellen Default-Wertes (vgl. Zeile 5)

```

1 #include <iostream>
2
3 class Student{
4 public:
5     std::string name;
6     int alter;
7     std::string ort; // = "Freiberg"
8
9     void ausgabeMethode(std::ostream& os){
10        os << name << " " << ort << " " << alter << "\n";
11    }
12
13    Student(std::string name, int alter, std::string ort): name(name), alter(alter), ort(ort)
14    {
15    }
16
17    Student(std::string name): name(name)
18    {
19    }
20 };

```

```

21
22 int main()
23 {
24     Student bernhard {"Cotta", 25, "Zillbach"};
25     bernhard.ausgabeMethode(std::cout);
26
27     Student alexander = Student("Humboldt");
28     alexander.ausgabeMethode(std::cout);
29
30     return 0;
31 }

```

Delegierende Konstruktoren rufen einen weiteren Konstruktor für die teilweise Initialisierung auf. Damit lassen sich Codeduplikationen, die sich aus der Kombination aller Parameter ergeben, minimieren.

```

1 Student(std::string n, int a, std::string o): name{n}, alter{a}, ort{o} { }
2 Student(std::string n) : Student (n, 18, "Freiberg") {};
3 Student(int a, std::string o): Student ("unknown", a, o) {};

```

8.6.2 Destruktoren

```

1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9         Student(std::string n, int a, std::string o);
10        ~Student();
11 };
12
13 Student::Student(std::string n, int a, std::string o): name{n}, alter{a}, ort{o} {}
14
15 Student::~~Student(){
16     std::cout << "Destructing object of type 'Student' with name = '" << this->name << "'\n";
17 }
18
19 int main()
20 {
21     Student max {"Maier", 19, "Dresden"};
22     std::cout << "End...\n";
23     return 0;
24 }

```

Destruktoren werden aufgerufen, wenn eines der folgenden Ereignisse eintritt:

- Das Programm verlässt den Gültigkeitsbereich (*Scope*, d.h. einen Bereich der mit {...} umschlossen ist) eines lokalen Objektes.
- Ein Objekt, das `new`-erzeugt wurde, wird mithilfe von `delete` explizit aufgehoben (Speicherung auf dem Heap)
- Ein Programm endet und es sind globale oder statische Objekte vorhanden.
- Der Destruktor wird unter Verwendung des vollqualifizierten Namens der Funktion explizit aufgerufen.

Einen Destruktor explizit aufzurufen, ist selten notwendig (oder gar eine gute Idee!).

8.7 Beispiel des Tages

Aufgabe: Erweitern Sie das Beispiel um zusätzliche Funktionen, wie die Berechnung des Umfanges. Überwachen Sie die Eingaben der Höhe und der Breite. Sofern diese negativ sind, sollte die Eingabe zurückgewiesen werden.

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle {
5     private:
6         int width, height;
7     public:
8         void set_values (int,int);           // Deklaration
9         int area() {return width*height;}    // Deklaration und Defintion
10 };
11
12 void Rectangle::set_values (int x, int y) {
13     width = x;
14     height = y;
15 }
16
17 int main () {
18     Rectangle rect;
19     rect.set_values (3,4);
20     cout << "area: " << rect.area();
21     return 0;
22 }
```

8.8 Anwendung

1. Ansteuern einer mehrfarbigen LED

Die Auflistung der Memberfunktionen der entsprechenden Klasse finden Sie unter [Link](#)

2. Abfragen eines Sensors

Die Auflistung der Memberfunktionen der entsprechenden Klassen finden Sie unter [Link](#)

Der Beispielcode für die Anwendungen ist in den `examples` Ordnern des Projektes enthalten.

Kapitel 9

Objektorientierte Programmierung mit C++

Parameter Kursinformationen

Veranstaltung: Prozedurale Programmierung / Einführung in die Informatik

Semester: Wintersemester 2022/23

Hochschule: Technische Universität Freiberg

Inhalte: Operatorenüberladung / Vererbung

Link auf <https://github.com/TUBAF-Ifl->

Repository: [LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md](https://github.com/TUBAF-Ifl-LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_Funktionen.md)

Autoren @author

Fragen an die heutige Veranstaltung ...

- Was sind Operatoren?
 - Warum werden eigene Operatoren für individuelle Klassen benötigt?
 - Wann spricht man von Vererbung und warum wird sie angewendet?
 - Welche Zugriffsattribute kennen Sie im Zusammenhang mit der Vererbung?
-

9.1 Rückblick

Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.

Und was bedeutet das angewandt auf unsere Vision mit dem Mikrocontroller Daten zu erheben?

```
1 @startuml
2 ditaa
3
4 +-----+ +-----+
5 | API Implementierung des Herstellers | | Eigene Klassenimplementierungen |
6 | * Led-Klasse                        | | * Filter-Klasse                |
7 | * Drucksensor-Klasse                | | * System-Monitor-Klasse         |
8 | * Serial-Klasse                     | | * ....                          |
9 | * ....                              cCCB | | * ....                      cBFB |
10 +-----+ +-----+
11 |                                     |
12 |                                     |
13 +-----+ +-----+
```

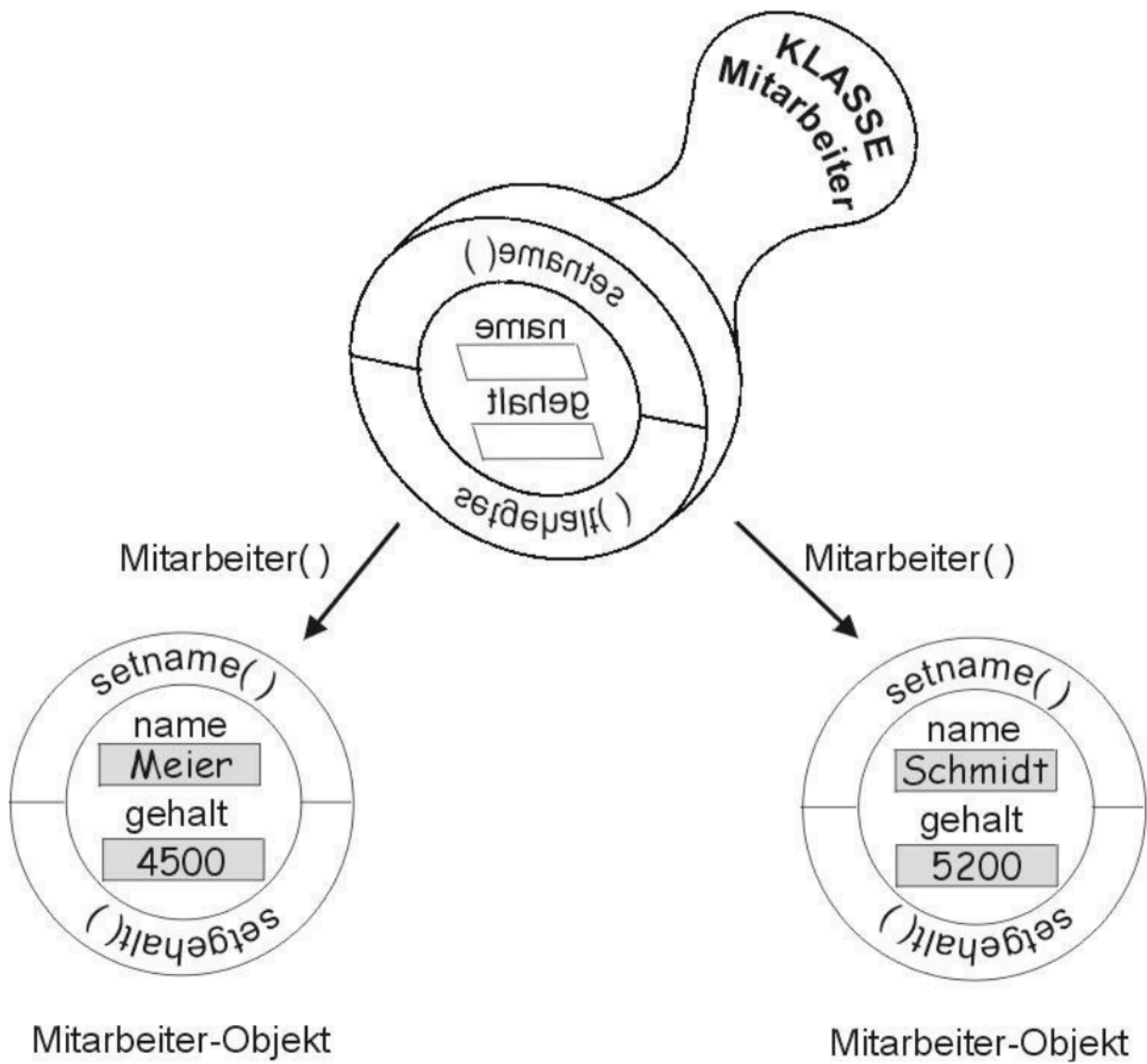


Abbildung 9.1: Prinzipdarstellung

```

14          |
15          v
16          +-----+
17          | // Mein Programm          |
18          | void setup{                |
19          |   RGB_LED rgbLed(red, green, blue); |
20          | }                          |
21          |                            |
22          | void setup{                |
23          |   rgbLed.setColor(255, 0, 0); |
24          | }                          | {d}cBFB
25          +-----+

```

Für die Implementierung einer Ausgabe auf dem Display des MXCHIP Boards nutzen wir die Klassenimplementierung der API.

[Link](#)

9.2 Operatorenüberladung

9.3 Motivation

Folgendes Beispiel illustriert den erreichten Status unserer C++ Implementierungen. Unsere Klasse **Student** besteht aus:

- 3 Membervariablen (Zeile 5-7)
- 2 Konstruktoren (Zeile 9-10)
- 1 Memberfunktion (Zeile 12)

Alle sind als `public` markiert.

```

1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9         Student(std::string n);
10        Student(std::string n, int a, std::string o);
11
12        void ausgabeMethode(std::ostream& os); // Deklaration der Methode
13 };
14
15 Student::Student(std::string n): name(n), alter(8), ort("Freiberg"){
16
17 Student::Student(std::string n, int a, std::string o): name(n), alter(a), ort(o) {}
18
19 void Student::ausgabeMethode(std::ostream& os){
20     os << name << " " << ort << " " << alter << "\n";
21 }
22
23 int main()
24 {
25     Student gustav = Student("Zeuner", 27, "Chemnitz");
26     //Student gustav {"Zeuner", 27, "Chemnitz"};
27     //Student gustav("Zeuner", 27, "Chemnitz");
28     gustav.ausgabeMethode(std::cout);
29
30     Student bernhard {"Cotta", 18, "Zillbach"};
31     bernhard.ausgabeMethode(std::cout);

```

```

32
33 Student nochmalBernhrd {"Cotta", 18, "Zillbach"};
34 }

```

Aufgabe: Schreiben Sie

- eine Funktion `int vergleich(Student, Student)` und
- eine Methode `int Student::vergleich(Student, Student)`,

die zwei Studenten miteinander vergleicht!

Frage: Was ist der Nachteil unserer Lösung?

9.3.1 Konzept

Das Überladen von Operatoren erlaubt die flexible klassenspezifische Nutzung von Arithmetischen- und Vergleichs-Symbolen wie `+`, `-`, `*`, `==`. Damit kann deren Bedeutung für selbstdefinierter Klassen mit einer neuen Bedeutung versehen werden. Ausnahmen bilden spezieller Operatoren, die nicht überladen werden dürfen (`?:`, `::`, `.`, `.*`, `typeid`, `sizeof` und die Cast-Operatoren).

```

1 Matrix a, b;
2 Matrix c = a + b;    \\ hier wird mit dem Plus eine Matrixoperation ausgeführt
3
4 String a, b;
5 String c = a + b;    \\ hier werden mit dem Plus zwei Strings konkateniert

```

Operatorüberladung ist Funktionsüberladung, wobei die Funktionen durch eine spezielle Namensgebung gekennzeichnet sind. Diese beginnen mit dem Schlüsselwort `operator`, das von dem Token für den jeweiligen Operator gefolgt wird.

```

1 class Matrix{
2     public:
3         Matrix operator+(Matrix zweiterOperand){ ... }
4         Matrix operator/(Matrix zweiterOperand){ ... }
5         Matrix operator*(Matrix zweiterOperand){ ... }
6 }
7
8 class String{
9     public:
10        String operator+(String zweiterString){ ... }
11 }

```

Operatoren können entweder als Methoden der Klasse oder als globale Funktionen überladen werden. Die Methodenbeispiele sind zuvor dargestellt, analoge Funktionen ergeben sich zu:

```

1 class Matrix{
2     public:
3         ...
4 }
5
6 Matrix operator+(Matrix ersterOperand, Matrix zweiterOperand){ ... }
7 Matrix operator/(Matrix ersterOperand, Matrix zweiterOperand){ ... }
8 Matrix operator*(Matrix ersterOperand, Matrix zweiterOperand){ ... }

```

Merke: Funktion oder Methode - welche Version sollte wann zum Einsatz kommen? Einstellige Operatoren `++` sollten Sie als Methode, zweistellige Operatoren ohne Manipulation der Operanden als Funktion implementieren. Für zweistellige Operatoren, die einen der Operanden verändern (`+=`), sollte als Methode realisiert werden.

Als Beispiel betrachten wir eine Klasse Rechteck und implementieren zwei Operatorüberladungen:

- eine Vergleichsoperation
- eine Additionsoperation die `A = A + B` oder abgekürzt `A+=B`

implementiert.


```

1 #include <iostream>
2
3 class Rectangle {
4     private:
5         float width, height;
6     public:
7         Rectangle(int w, int h): width{w}, height{h} {}
8         float area() {return width*height;}
9         Rectangle operator+=(Rectangle offset) {
10             float ratio = (offset.area() + this->area()) / this->area();
11             this->width = ratio * this->width;
12             return *this;
13         }
14 };
15
16 bool operator>(Rectangle a, Rectangle b){
17     if (a.area() > b.area()) return 1;
18     else return 0;
19 }
20
21 int main () {
22     Rectangle rect_a(3,4);
23     Rectangle rect_b(5,7);
24     std::cout << "Vergleich: " << (rect_a > rect_b) << "\n";
25
26     std::cout << "Fläche a : " << rect_a.area() << "\n";
27     std::cout << "Fläche b : " << rect_b.area() << "\n";
28     rect_a += rect_b;
29     std::cout << "Summe      : " << rect_a.area();
30
31     return 0;
32 }

```

Merke: Üblicherweise werden die Operanden bei der Operatorüberladung als Referenzen übergeben. Damit wird eine Kopie vermieden. In Kombination mit dem Schlüsselwort `const` kann dem Compiler angezeigt werden, dass keine Veränderung an den Daten vorgenommen wird. Sie müssen also nicht gespeichert werden.

```

1 bool operator>(const Rectangle& a, const Rectangle& b){
2     if (a.area() > b.area()) return 1;
3     else return 0;
4 }

```

Stellen wir die Abläufe nochmals grafisch dar:

- Aufruf über Call-by-Value [Pythontutor](#)
- Aufruf über Referenz [Pythontutor](#)

9.3.2 Anwendung

Im folgenden Beispiel wird der Vergleichsoperator `==` überladen. Dabei sehen wir den Abgleich des Namens und des Alters als ausreichend an.

```

1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9         Student(std::string n);
10        Student(std::string n, int a, std::string o);

```

```

11
12     void ausgabeMethode(std::ostream& os); // Deklaration der Methode
13     bool operator==(const Student&);
14 };
15
16 Student::Student(std::string n): name(n), alter(8), ort("Freiberg"){
17
18 Student::Student(std::string n, int a, std::string o): name(n), alter(a), ort(o) {}
19
20 void Student::ausgabeMethode(std::ostream& os){
21     os << name << " " << ort << " " << alter << "\n";
22 }
23
24 bool Student::operator==(const Student& other){
25     if ((this->name == other.name) && (this->alter == other.alter)){
26         return true;
27     }else{
28         return false;
29     }
30 }
31
32 int main()
33 {
34     Student gustav = Student("Zeuner", 27, "Chemnitz");
35     gustav.ausgabeMethode(std::cout);
36
37     Student bernhard {"Cotta", 18, "Zillbach"};
38     bernhard.ausgabeMethode(std::cout);
39
40     Student NochMalBernhard {"Cotta", 18, "Zillbach"};
41     NochMalBernhard.ausgabeMethode(std::cout);
42
43     if (bernhard == NochMalBernhard){
44         std::cout << "Identische Studenten \n";
45     }else{
46         std::cout << "Ungleiche Identitäten \n";
47     }
48 }

```

Eine besondere Form der Operatorüberladung ist der <<, mit dem die Ausgabe auf ein Streamobjekt realisiert werden kann.

```

1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9         Student(const Student&);
10        Student(std::string n);
11        Student(std::string n, int a, std::string o);
12
13        void ausgabeMethode(std::ostream& os); // Deklaration der Methode
14
15        bool operator==(const Student&);
16 };
17
18 Student::Student(std::string n, int a, std::string o): name{n}, alter{a}, ort{o} {}
19

```

```

20 std::ostream& operator<<(std::ostream& os, const Student& student)
21 {
22     os << student.name << '/' << student.alter << '/' << student.ort;
23     return os;
24 }
25
26 int main()
27 {
28     Student gustav = Student("Zeuner", 27, "Chemnitz");
29     Student bernhard = Student("Cotta", 18, "Zillbach");
30     std::cout << gustav;
31 }

```

Eine umfangreiche Diskussion zur Operatorüberladung finden Sie unter <https://www.c-plusplus.net/forum/topic/232010/%C3%9C-von-operatoren-in-c-teil-1/2>

9.4 Vererbung

```

1 #include <iostream>
2
3 class Student{
4     public:
5         std::string name;
6         std::string ort;
7         std::string studiengang;
8
9         Student(std::string n, std::string o, std::string sg): name{n}, ort{o}, studiengang{sg} {};
10        void printCertificate(std::ostream& os){
11            os << "Studentendatensatz: " << name << " " << ort << " " << studiengang << "\n";
12        }
13 };
14
15 int main()
16 {
17     Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
18     gustav.printCertificate(std::cout);
19
20     //Professor winkler = Professor("Winkler", "Freiberg");
21     //winkler.printCertificate(std::cout);
22 }

```

Aufgabe: Implementieren Sie eine neue Klasse **Professor**, die aber auf die Membervariable **Studiengang** verzichtet, aber eine neue Variable **Fakultät** einführt.

9.4.1 Motivation

Merke: Eine unserer Hauptmotivationen bei der “ordentlichen” Entwicklung von Code ist die Vermeidung von Codedopplungen!

In unserem Code entstehen Dopplungen, weil bestimmte Variablen oder Memberfunktionen usw. mehrfach für individuelle Klassen implementiert werden. Dies wäre für viele Szenarien analog der Fall:

Basisklasse	abgeleitete Klassen	Gemeinsamkeiten
Fahrzeug	Flugzeug, Boot, Automobil	Position, Geschwindigkeit, Zulassungsnummer, Führerscheinpflicht
Datei	Foto, Textdokument, Datenbankauszug	Dateiname, Dateigröße, Speicherort
Nachricht	Email, SMS, Chatmessage	Adressat, Inhalt, Datum der Versendung

Merke: Die *Vererbung* ermöglicht die Erstellung neuer Klassen, die ein in existierenden Klassen

definiertes Verhalten wieder verwenden, erweitern und ändern. Die Klasse, deren Member vererbt werden, wird Basisklasse genannt, die erbende Klasse als abgeleitete Klasse bezeichnet.

9.4.2 Implementierung in C++

In C++ werden Vererbungsmechanismen folgendermaßen abgebildet:

```

1 class Fahrzeug{
2     public:
3         int aktuellePosition[2];    // lat, long Position auf der Erde
4         std::string Zulassungsnummer;
5         Bool Fuehrerscheinpflichtig
6         ...
7 };
8
9 class Flugzeug: public Fahrzeug{
10     public:
11         int Flughoehe;
12         void fliegen();
13         ...
14 };
15
16 class Boot: public Fahrzeug{
17     public:
18         void schwimmen();
19         ...
20 };

```

Die generellere Klasse **Fahrzeug** liefert einen Bauplan für die spezifischeren, die die Vorgaben weiter ergänzen. Folglich müssen wir uns die Frage stellen, welche Daten oder Funktionalität übergreifend abgebildet werden soll und welche individuell realisiert werden sollen.

Dabei können ganze Ketten von Vererbungen entstehen, wenn aus einem sehr allgemeinen Objekt über mehrere Stufen ein spezifischeres Set von Membern umgesetzt wird.

```

1 class Fahrzeug{
2     public:
3         int aktuellePosition[2];    // lat, long Position auf der Erde
4         std::string Zulassungsnummer;
5         Bool Fuehrerscheinpflichtig
6         ...
7 };
8
9 class Automobil: public Fahrzeug{
10     public:
11         void fahren();
12         int ZahlderRaeder;
13         int Sitze;
14         ...
15 };
16
17 class Hybrid: public Automobil{
18     public:
19         void fahreElektrisch();
20         ...
21 };

```

Was bedeutet das für unsere Implementierung von Studenten und Professoren?

```

1 #include <iostream>
2
3 class Student{
4     public:

```

```

5     std::string name;
6     std::string ort;
7     std::string studiengang;
8
9     Student(std::string n, std::string o, std::string sg): name{n}, ort{o}, studiengang{sg} {};
10    void printCertificate(std::ostream& os){
11        os << "Studentendatensatz: " << name << " " << ort << " " << studiengang << "\n";
12    }
13 };
14
15 int main()
16 {
17     Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
18     gustav.printCertificate(std::cout);
19
20     //Professor winkler = Professor("Winkler", "Freiberg");
21     //winkler.printCertificate(std::cout);
22 }

```

Ein weiteres Beispiel greift den Klassiker der Einführung objektorientierter Programmierung auf, den Kanon der Haustiere :-). Das Beispiel zeigt die Initialisierung der Membervariablen :

- der Basisklasse beim Aufruf des Konstruktors der erbenden Klasse
- der Member der erbenden Klasse wie gewohnt

```

1 #include <iostream>
2
3 class Animal {
4 public:
5     Animal(): name{"Animal"}, weight{0.0} {};
6     Animal(std::string _name, double _weight): name{_name}, weight{_weight} {};
7     void sleep () {
8         std::cout << name << " is sleeping!" << std::endl;
9     }
10    std::string name;
11    double weight;
12 };
13
14 class Dog : public Animal {
15 public:
16     Dog(std::string name, double weight, int id): Animal(name, weight), id{id} {};
17     void bark() {
18         std::cout << "woof woof" << std::endl;
19     }
20     double top_speed() {
21         return (weight < 40) ? 15.5 : (weight < 90) ? 17.0 : 16.2;
22     }
23     int id;
24 };
25
26 int main(){
27     Dog dog = Dog("Rufus", 50.0, 2342);
28     dog.sleep();
29     dog.bark();
30     std::cout << dog.top_speed() << std::endl;
31 }

```

9.4.3 Vererbungsattribute

Die Zugriffsattribute `public` und `private` kennen Sie bereits. Damit können wir Elemente unserer Implementierung vor dem Zugriff von außen schützen.

Aufgabe: Verhindern Sie, dass die Einträge von `id` im Nachhinein geändert werden können! Welche zusätzlichen Methoden benötigen Sie dann?

```

1 #include <iostream>
2
3 class Animal {
4 public:
5     std::string name;
6     int id;
7     Animal(std::string name, int id): name{name}, id{id} {};
8 };
9
10 int main(){
11     Animal fish = Animal("Nemo", 234242343);
12     std::cout << fish.id << std::endl;
13 }
```

Wie wirkt sich das Ganze aber auf die Vererbung aus? Hierbei muss neben dem individuellen Zugriffsattribut auch der Status der Vererbung beachtet werden. Damit ergibt sich dann folgendes Bild:

```

1 class A
2 {
3 public:
4     int x;
5 protected:
6     int y;
7 private:
8     int z;
9 };
10
11 class B : public A
12 {
13     // x is public
14     // y is protected
15     // z is not accessible from B
16 };
17
18 class C : protected A
19 {
20     // x is protected
21     // y is protected
22     // z is not accessible from C
23 };
24
25 class D : private A    // 'private' is default for classes
26 {
27     // x is private
28     // y is private
29     // z is not accessible from D
30 };
```

Das Zugriffsattribut `protected` spielt nur bei der Vererbung eine Rolle. Innerhalb einer Klasse ist `protected` gleichbedeutend mit `private`. In der Basisklasse ist also ein Member geschützt und nicht von außen zugreifbar. Bei der Vererbung wird der Unterschied zwischen `private` und `protected` deutlich: Während `private` Member in erbenden Klassen nicht direkt verfügbar sind, kann auf die als `protected` deklariert zugegriffen werden.

Entsprechend muss man auch die Vererbungskonstellation berücksichtigen, wenn man festlegen möchte ob ein Member gar nicht (`private`), immer (`public`) oder nur im Vererbungsverlauf verfügbar sein (`protected`) soll.

```

1 #include <iostream>
2
3 class Animal {
4 public:
```

```

5   Animal(): name{"Animal"}, weight{0.0} {};
6   Animal(std::string _name, double _weight): name{_name}, weight{_weight} {};
7   void sleep () {
8       std::cout << name << " is sleeping!" << std::endl;
9   }
10  std::string name;
11  double weight;
12 };
13
14 class Dog : public Animal {
15 public:
16     Dog(std::string name, double weight, int id): Animal(name, weight), id{id} {};
17     void bark() {
18         std::cout << "woof woof" << std::endl;
19     }
20     double top_speed() {
21         return (weight < 40) ? 15.5 : (weight < 90) ? 17.0 : 16.2;
22     }
23     int id;
24 };
25
26 int main(){
27     Dog dog = Dog("Rufus", 50.0, 2342);
28     dog.sleep();
29     dog.bark();
30     std::cout << dog.top_speed() << std::endl;
31 }

```

9.4.4 Überschreiben von Methoden der Basisklasse

Die grundsätzlicher Idee bezieht sich auf die Implementierung "eigener" Methoden gleicher Signatur in den abgeleiteten Klassen. Diese implementieren dann das spezifische Verhalten der jeweiligen Objekte.

```

1  #include <iostream>
2
3  class Person{
4  public:
5      std::string name;
6      std::string ort;
7
8      Person(std::string n, std::string o): name{n}, ort{o} {};
9      void printData(std::ostream& os){
10         os << "Datensatz: " << name << " " << ort << "\n";
11     }
12 };
13
14 class Student : public Person{
15 public:
16     Student(std::string n, std::string o, std::string sg): Person(n, o), studiengang{sg}{};
17     std::string studiengang;
18     void printCertificate(std::ostream& os){
19         os << "Studentendatensatz: " << name << " " << ort << " " << studiengang << "\n";
20     }
21 };
22
23 class Professor : public Person{
24 public:
25     Professor(std::string n, std::string o, int id): Person(n, o), id{id}{};
26     int id;
27 };
28

```

```

29 int main()
30 {
31     Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
32     gustav.printData(std::cout);
33
34     Professor winkler = Professor("Winkler", "Freiberg", 234234);
35     winkler.printData(std::cout);
36 }

```

Die Polymorphie (griechisch “Vielgestaltigkeit”) der objektorientierten Programmierung ist eine Eigenschaft, die in Zusammenhang mit Vererbung einhergeht. Eine Methode ist genau dann polymorph, wenn sie von verschiedenen Klassen unterschiedlich genutzt wird. Wenn Sie mehr darüber wissen wollen, sind Sie herzlich zur Vorlesung Softwareentwicklung im Sommersemester eingeladen!

Dabei untersuchen wir unter anderem Konzepte, wie wir die ererbenden Methoden zwingen können eine bestimmte Methode zu implementieren. Mit der Notation `virtual void printData(std::ostream& os)= 0;` wird aus unserer Implementierung eine abstrakte Methode, die in jedem Fall in den ererbenden Klassen implementiert sein muss.

9.5 Anwendungsfall

Entwerfen Sie eine Klasse, die das Verhalten einer Ampel mit den notwendigen Zuständen modelliert. Welche Methoden sollten zusätzlich in die Klasse aufgenommen werden?

```

1 class Ampel {
2 private:
3     int redPin, yellowPin, greenPin;
4     int state = 0;
5
6 public:
7     Ampel(int red, int yellow, int green): redPin{red}, yellowPin{yellow}, greenPin{green} {
8         pinMode(red, OUTPUT);
9         pinMode(yellow, OUTPUT);
10        pinMode(green, OUTPUT);
11    };
12    void activateRed() {
13        digitalWrite(redPin, HIGH);
14    }
15    void startOnePeriod(int waitms) {
16        digitalWrite(redPin, HIGH);
17        delay(waitms);
18        digitalWrite(yellowPin, HIGH);
19        delay(waitms);
20        digitalWrite(redPin, LOW);
21        digitalWrite(yellowPin, LOW);
22        digitalWrite(greenPin, HIGH);
23    }
24 };
25
26 void setup() {
27     Ampel trafficLight = Ampel(13, 12, 11);
28     trafficLight.activateRed();
29     trafficLight.startOnePeriod(1000);
30 }
31
32 void loop() {
33     delay(100);
34 }

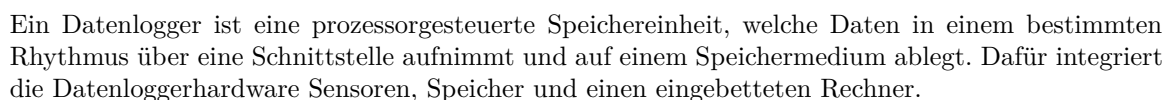
```

@AVR8js.sketch

Softwareentwicklung für Microcontroller

Fragen an die heutige Veranstaltung ...

- ## 10.1 Microcontroller als Datensammler



- 105

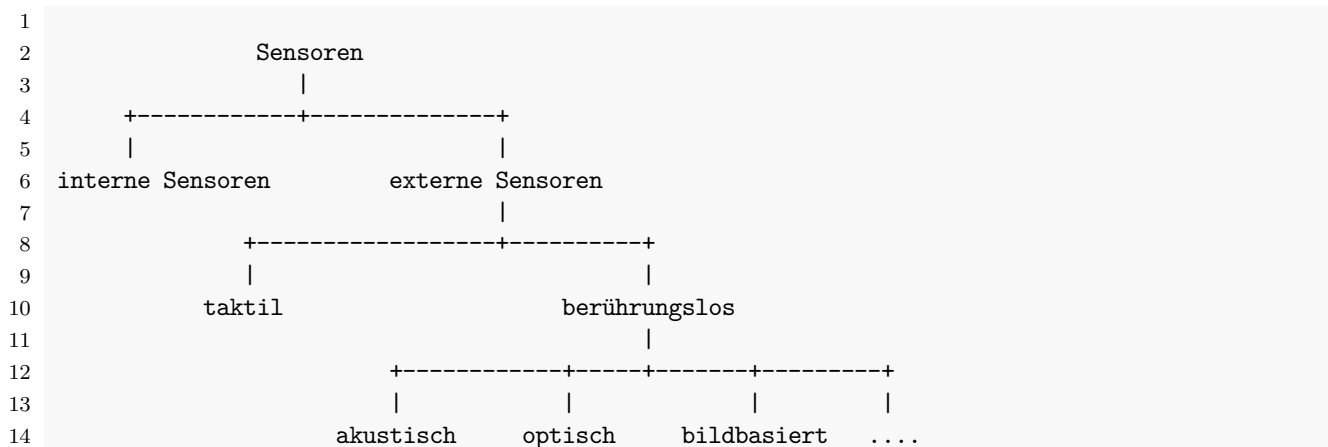
- Soll die Messdatenaufnahme periodisch oder eventgetrieben erfolgen?
- Wie lang ist der intendierte Messzeitraum - reichen Datenspeicher und Batteriekapazität?
- Braucht unser Logger ein Display oder eine andere Möglichkeit um seinen Zustand anzuzeigen?

10.1.1 Sensoren / Aktoren

Ein Sensor (von lateinisch *sentire*, „fühlen“ oder „empfinden“), auch als Detektor, (Messgrößen- oder Mess-)Aufnehmer oder (Mess-)Fühler bezeichnet, ist ein technisches Bauteil, das Umgebungsparameter als Messgröße quantitativ erfassen kann. Dabei werden physikalische Eigenschaften wie Temperatur, Feuchtigkeit, Druck, Helligkeit, Beschleunigung oder chemische z. B. pH-Wert, elektrochemisches Potential usw. als weiterverarbeitbares elektrisches Signal bereitgestellt.

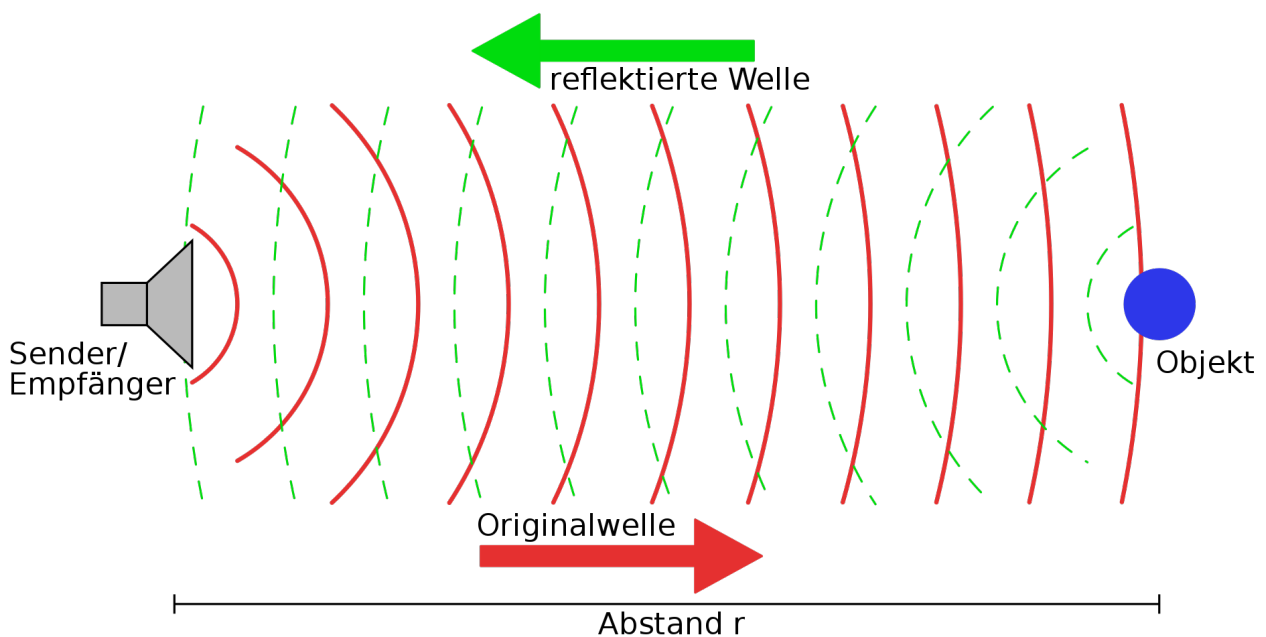
Klassifikation von Sensoren

- intern/extern ... bezogen auf den Messgegenstand (Radencoder vs. Kamera)
- aktiv/passiv () ... mit und ohne Beeinflussung der Umgebung (Ultraschall vs. Kamera)
- Ergebnisdimension ... 1, 2, 2.5, 3D
- Modalitäten ... physikalische Messgröße



Beispiel: Ultraschallsensor

... Was war das noch mal, “Schallgeschwindigkeit”



Erkenntnis 1: Wir messen unter Umständen die eigentliche Messgröße nicht direkt.

Für eine gleichförmige Bewegung können wir den Weg als Produkt aus dem Messintervall und der halben Laufzeit abbilden.

$$s = v \cdot \frac{t}{2}$$

Erkenntnis 2: Die Genauigkeit dieses Sensors wird über die Zeitaufösung des Mikrocontrollers definiert.

Leider gibt es ein Problem, die Schallgeschwindigkeit in Luft ist nicht konstant ist. Sie ist eine Funktion der Dichte ρ und des (adiabatischen) Kompressionsmoduls K und hängt damit vom Standort, der Temperatur usw. ab. Annäherungsweise gilt

$$v(m/s) = 331.3 + (0.606 \times t)$$

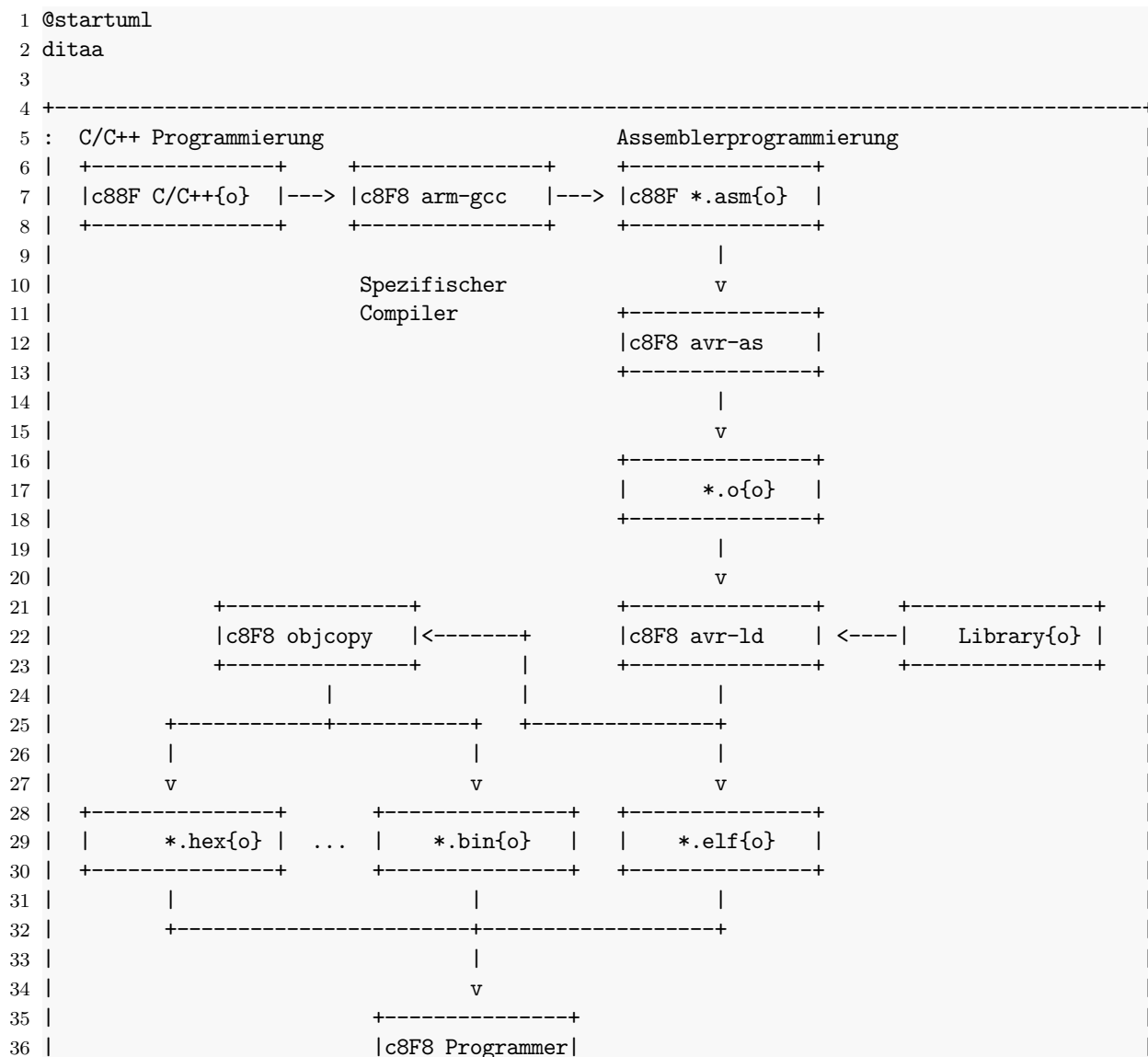
Versuchen wir eine kleine Fehlerabschätzung, wenn wir den Temperatureinfluss ignorieren.

```
“ python @PyScript.repl def calcUSspeed(T): return 331.3 + (0.606 * T)
print(calcUSspeed(25) / calcUSspeed(0)) “
```

Erkenntnis 3: Abhängigkeiten beeinflussen den Messprozess

10.1.2 Programmiervorgang

Der Programmiervorgang für einen Mikrocontroller unterscheidet sich in einem Punkt signifikant von Ihren bisherigen C/C++ Aufgaben - die erstellten Programme sind auf dem Entwicklungssystem nicht ausführbar. Wir tauschen also den Compiler mit der Hardware aus. Dadurch “verstehen” der Entwicklungsrechner die Anweisungen aber auch nicht.



```

37 |          +-----+
38 +-----+
39 |          |
40 |          +-----+
41 |          |          |
42 |          v          v
43 +-----+
44 : +-----+ +-----+ +-----+
45 | |c2F8 SRAM |<----- |c2F8 Flash | |c2F8 EEPROM |
46 | +-----+ +-----+ +-----+
47 | Mikrocontroller
48 +-----+
49 @enduml

```

Dabei zeigt sich aber auch der Vorteil der Hochsprachen C und C++, die grundsätzlichen Sprachinhalte sind prozessorunabhängig!

10.1.3 Besonderheiten

Standardbibliothek nicht vollständig umgesetzt

10.2 Arduino Konzept

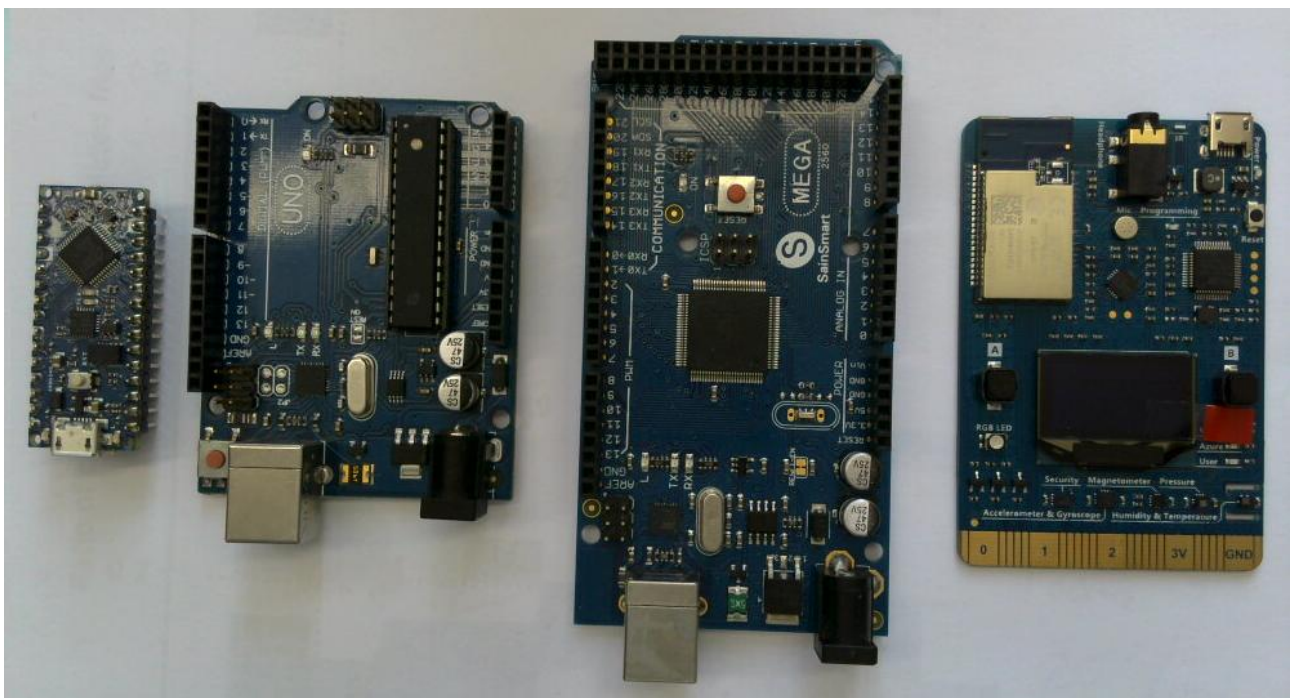
Das Arduino-Projekt wurde am *Interaction Design Institute Ivrea* (IDII) in Ivrea, Italien, ins Leben gerufen. Ausgangspunkt war die Suche nach einem preiswerten, einfach zu handhabenen Mikrocontroller der insbesondere für die Ausbildung von Nicht-Informatikern geeignet ist. Das anfängliche Arduino-Kernteam bestand aus Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino und David Mellis.

Nach der Fertigstellung der Plattform wurden leichtere und preiswertere Versionen in der Open-Source-Community verbreitet. Mitte 2011 wurde geschätzt, dass über 300.000 offizielle Arduinos kommerziell produziert worden waren, zwischenzeitlich wurden mehrere Millionen produziert.

10.2.1 Hardware

Das Arduino Projekt hat eine Vielzahl von unterschiedlichen Boards hervorgebracht, die eine unterschiedliche Leistungsfähigkeit und Ausstattung kennzeichnen. Das Spektrum reicht von einfachen 8-Bit Controllern bis hin zu leistungsstarken ARM Controllern, die ein eingebettetes Linux ausführen.

Merke: Es gibt nicht **den** Arduino Controller, sondern eine Vielzahl von verschiedenen Boards.



Unser Controller, ein 32 Bit System, auf den im nachfolgenden eingegangen wird, liegt im mittleren Segment der Leistungsfähigkeit.

Erweitert werden die Boards durch zusätzliche **Shields**, die den Funktionsumfang erweitern.

10.2.2 Programmierung

Jedes Arduino-Programm umfasst zwei zentrale Funktionen - **setup** und **loop**. Erstgenannte wird einmalig ausgeführt und dient der Konfiguration, die zweite wird kontinuierlich umgesetzt.

```

1 #define LED_PIN 13           // Pin number attached to LED.
2 //const int led_pin_red = 13;
3
4 void setup() {
5     pinMode(LED_PIN, OUTPUT); // Configure pin 13 to be a digital output.
6 }
7
8 void loop() {
9     digitalWrite(LED_PIN, HIGH); // Turn on the LED.
10    delay(1000);                 // Wait 1 second (1000 milliseconds).
11    digitalWrite(LED_PIN, LOW);  // Turn off the LED.
12    delay(1000);                 // Wait 1 second.
13 }
```

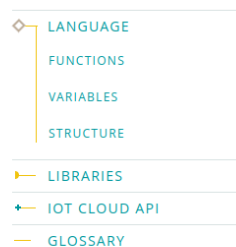
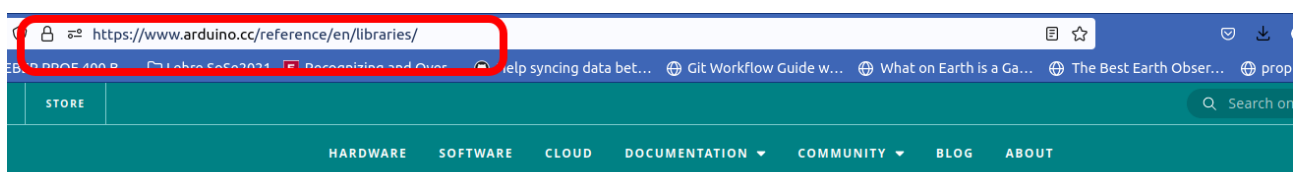
@AVR8js.sketch

Befehl	Bedeutung
<code>pinMode(pin_id, direction)</code>	Festlegung der Konfiguration eines Pins als Input / Output (INPUT, OUTPUT)
<code>digitalWrite(pin_id, state)</code>	Schreiben eines Pins, daraufhin liegen entweder (ungefähr) 3.3 V HIGH oder 0V LOW an

Eine Allgemeine Übersicht zu den Arduinobefehlen finden Sie unter folgendem [Link](#).

10.2.3 Bibliotheken

Darüber hinaus existiert eine Vielzahl von Bibliotheken, die die Arbeit mit verschiedenen Sensoren/Aktoren vereinfachen und bei der Entwicklung von Anwendungslogik unterstützen.



The Arduino Reference text is licensed under a [Creative Commons Attribution-Share Alike 3.0 License](#).

Find anything that can be improved? Suggest corrections and new documentation via [GitHub](#).

Doubts on how to use Github? Learn everything you need to know in [this tutorial](#).

Libraries

The Arduino environment can be extended through the use of libraries, just like most programming platforms.

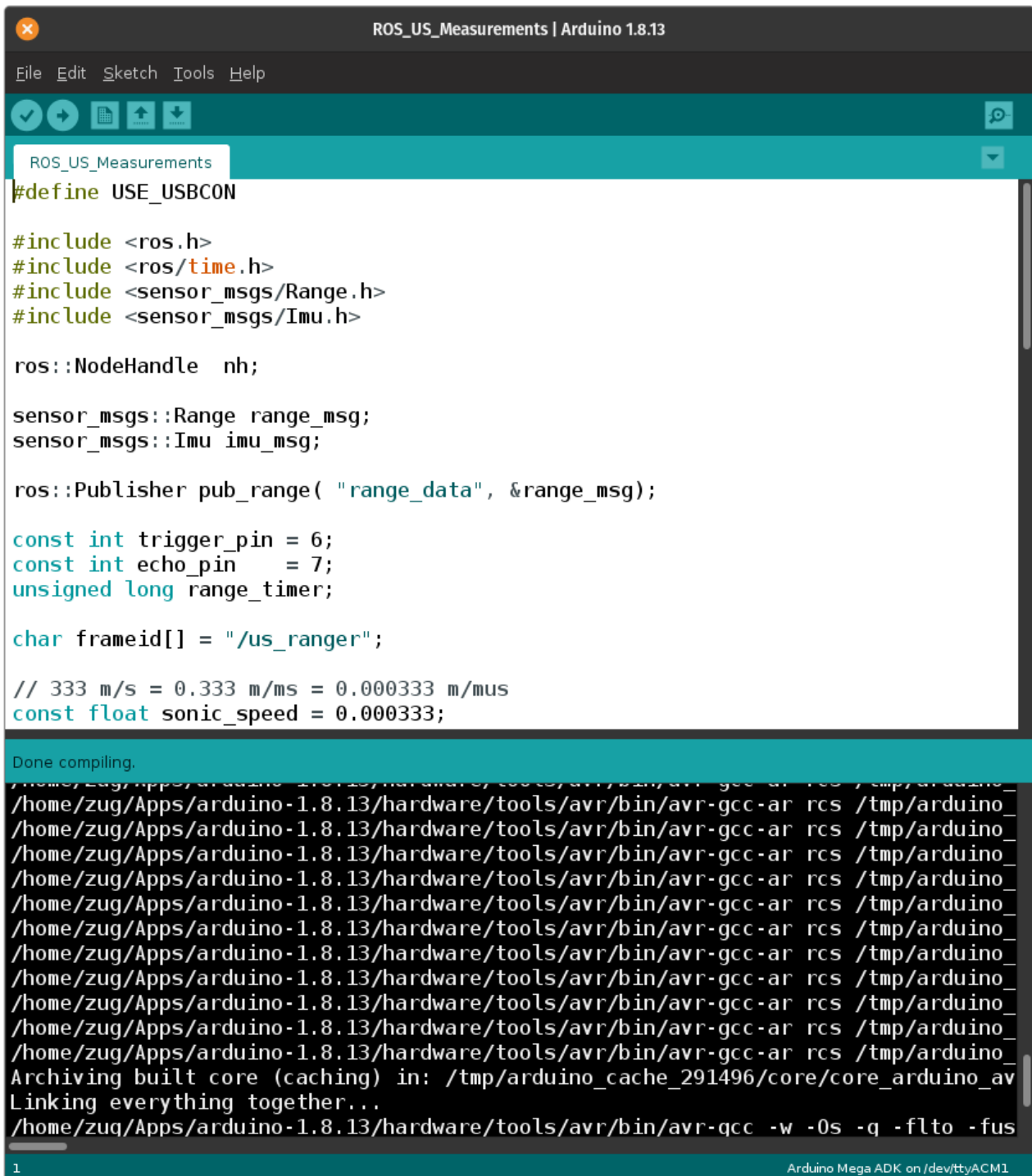
Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data. To use a library in a sketch, select it from **Sketch > Import Library**.

A number of libraries come installed with the IDE, but you can also download or create your own. See [these instructions](#) for details on installing libraries. There is also a [tutorial on writing your own libraries](#). See the [API Style Guide](#) for information on making a good Arduino-style API for your library.

- [Communication](#) (1179)
- [Data Processing](#) (299)
- [Data Storage](#) (148)
- [Device Control](#) (950)
- [Display](#) (467)
- [Other](#) (435)
- [Sensors](#) (1109)
- [Signal Input/Output](#) (410)
- [Timing](#) (219)
- [Uncategorized](#) (199)

10.2.4 Entwicklungsumgebung

Die Entwicklungsumgebung fasst grundsätzliche Entwicklungswerkzeuge zusammen und richtet sich an Einsteiger.



```

ROS_US_Measurements | Arduino 1.8.13
File Edit Sketch Tools Help

ROS_US_Measurements
#define USE_USBCON

#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/Range.h>
#include <sensor_msgs/Imu.h>

ros::NodeHandle nh;

sensor_msgs::Range range_msg;
sensor_msgs::Imu imu_msg;

ros::Publisher pub_range( "range_data", &range_msg);

const int trigger_pin = 6;
const int echo_pin = 7;
unsigned long range_timer;

char frameid[] = "/us_ranger";

// 333 m/s = 0.333 m/ms = 0.000333 m/mus
const float sonic_speed = 0.000333;

Done compiling.
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc-ar rcs /tmp/arduino_
Archiving built core (caching) in: /tmp/arduino_cache_291496/core/core_arduino_av
Linking everything together...
/home/zug/Apps/arduino-1.8.13/hardware/tools/avr/bin/avr-gcc -w -Os -q -flto -fus
1 Arduino Mega ADK on /dev/ttyACM1

```

Wir verwenden die Visual Studio Code Umgebung für die Entwicklung von Mikrocontroller Code.

10.2.5 Wo ist unser main()?

Oha, wo ist denn unsere main() Methode geblieben? Diese sollte doch zwingender Bestandteil eines jeden C++ Programmes sein?

```

1 # define LED_PIN 13 // Pin number attached to LED.
2
3 int main() {
4     // Das ist unser Setup-Bereich

```

```

5  init();    // Aufruf einer
6  pinMode(LED_PIN, OUTPUT);
7
8  // Endlosschleife als Entsprechung für Loop
9  while(true){
10     digitalWrite(LED_PIN, HIGH);
11     delay(1000);
12     digitalWrite(LED_PIN, LOW);
13     delay(1000);
14 }
15 }

```

@AVR8js.sketch

10.3 Exkurs: Serielle Schnittstelle

Die serielle Schnittstelle ist eine umgangssprachliche Bezeichnung für einen Übertragungsmechanismus zur Datenübertragung zwischen zwei Geräten, bei denen einzelne Bits zeitlich nacheinander ausgetauscht werden. Die Bezeichnung bezieht sich in der umgangssprachlichen Verwendung:

- das Wirkprinzip generell, das dann verschiedenste Kommunikationsprotokolle meinen kann (CAN, I2C, usw.) oder
- die als EIA-RS-232 bezeichnete Schnittstellendefinition.

Für Mikrocontroller werden die zugehörigen Bauteile als *Universal Asynchronous Receiver Transmitter* (UART) bezeichnet.

Mögliche Anwendungen des UART:

- Debug-Schnittstelle
- Mensch-Maschine Schnittstelle - Konfiguration eines Parameters, Statusabfragen
- Übertragen von gespeicherten Werten für ein Langzeit-Logging von Daten
- Anschluss von Geräten - Sensoren, GNSS-Empfängern, Funkmodems
- Implementierung von "Feldbussen" auf RS485/RS422-Basis

10.3.1 Schreiben

Für das Schreiben auf der Seriellen Schnittstelle stehen in der Arduino Welt drei Funktionen bereit `println`, `print` und `write`. Diese können mit zusätzlichen Parametern versehen werden, um eine eingeschränkte Formatierung vorzunehmen.

```

1  void setup() {
2     Serial.begin(9600);
3     Serial.println("Los geht's!");
4     Serial.println(5);
5     Serial.println(5, BIN);
6     Serial.println(5.34543, 2);
7     Serial.println("Fertig!");
8  }
9
10 void loop() {
11 }

```

@AVR8js.sketch

10.3.2 Lesen

Die Umkehr der Kommunikationsrichtung ermöglicht es, Daten an den Mikrocontroller zu senden und damit bestimmte Einstellungen vorzunehmen.

```

1  int incomingByte = 0; // for incoming serial data
2
3  void setup() {
4     Serial.begin(9600);

```

```
5 pinMode(13, OUTPUT); // LED Pin als Output
6 }
7
8 void loop() {
9   // any data received?
10  if (Serial.available() > 0) {
11    // read the incoming byte:
12    incomingByte = Serial.read();
13
14    // say what you got:
15    Serial.print("I received: ");
16    Serial.println(incomingByte, DEC);
17  }
18 }
```

@AVR8js.sketch

Aufgabe: Schalten Sie die LED die mit Pin 13 Verbunden ist mit einem A an und mit einem B aus. Dabei soll die LED mindestens 3s angeschaltet bleiben.

Aufgabe: Erweitern Sie das Programm, so dass mit ‘AN’ und ‘AUS’ die Aktivierung umgesetzt werden kann. Gehen Sie davon aus, dass der Nutzer auch kleine Buchstaben verwenden kann. (Hilfen [String Klasse](#), [Arduino Reference](#))

10.4 Seriellen Daten in der Arduino IDE

Der in der Arduino IDE eingebettete Serial Monitor ist eine Möglichkeit die über die Serielle Schnittstelle empfangenen Daten auszulesen. Mit der richtigen Konfiguration von

- USB Port (in der Entwicklungsumgebung selbst) und
- Baudrate (im unteren Teil des Monitors)

werden die Texte sichtbar. Sie können die Informationen speichern, indem Sie diese Markieren und in eine Datei kopieren. Sofern Sie auf eine gleiche Zahl von Einträgen pro Zeile achten, können die Daten dann als [csv](#) (*Comma-separated values*) Datei zum Beispiel mit Tabellenkalkulationsprogrammen geöffnet werden.


```

SensorData | Arduino 1.8.13
File Edit Sketch Tools Help

int gAxesData[3];

void setup()
{
  Screen.init();
  Screen.print(0);
  Screen.print(1);
  Serial.begin(115200);

  // Initialize t
  Screen.print(2);
  accelgyroSensor
  accelgyroSensor
}

void loop()
{
  accelgyroSensor
  //Serial.print
  Serial.printf("
  delay(50);
}

Done uploading.
xPSR: 0x61000000
wrote 344064 byte
** Programming Fi
** Verify Started
target halted due
xPSR: 0x61000000
verified 226244 b
** Verified OK **

09:41:03.890 -> -24 7 1030
09:41:03.957 -> -23 7 1030
09:41:04.023 -> -25 7 1031
09:41:04.056 -> -24 7 1031
09:41:04.122 -> -24 6 1031
09:41:04.155 -> -24 7 1031
09:41:04.221 -> -24 7 1030
09:41:04.255 -> -25 7 1030
09:41:04.321 -> -24 7 1029
09:41:04.354 -> -24 8 1030
09:41:04.420 -> -24 8 1031
09:41:04.453 -> -24 6 1030
09:41:04.519 -> -24 7 1030
09:41:04.586 -> -24 7 1030
09:41:04.619 -> -24 7 1030
09:41:04.685 -> -24 8 1030
09:41:04.718 -> -25 8 1031
09:41:04.784 -> -25 7 1031
09:41:04.818 -> -24 7 1030
09:41:04.884 -> -24 7 1031
09:41:04.917 -> -24 6 1032
09:41:04.983 -> -24 7 1030
09:41:05.016 -> -23 7 1031
09:41:05.082 -> -25 8 1030

xAxesData[2]);

.955 KiB/s)

Autoscroll Show timestamp
Newline 115200 baud Clear output

MXCHIP AZ3166 on /dev/ttyACM0

```

Merke: Die Einblendung des Zeitstempels garantiert die Zuordenbarkeit der kommunizierten Daten. Der Zeitstempel wird dabei vom Entwicklungsrechner generiert.

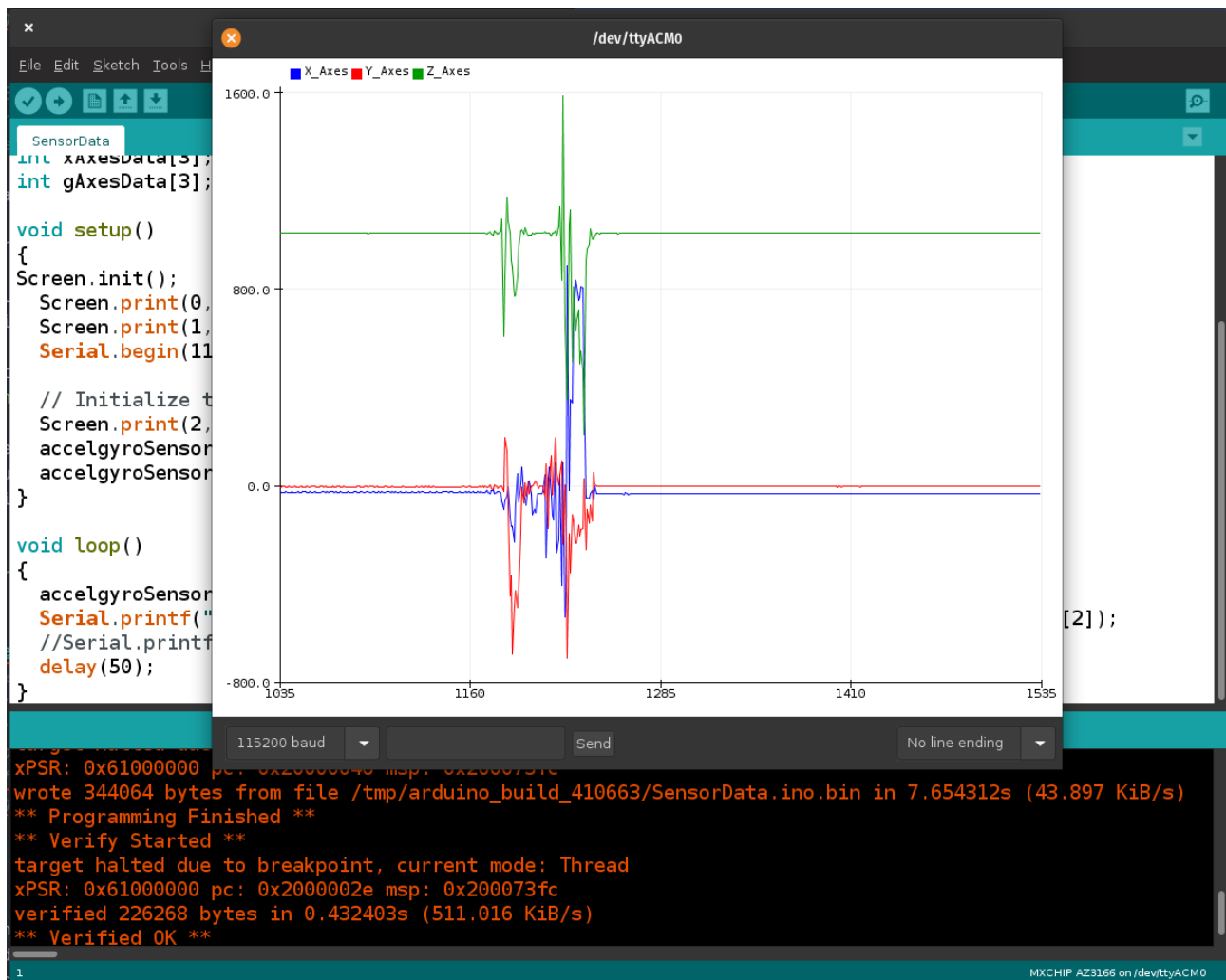
Die Eingabe von Daten, die an den erfolgt in der Zeile unter dem Ausgabefeld. Erst mit dem Betätigen von Enter, werden die Daten tatsächlich übertragen.

Der Plotter ermöglicht die unmittelbare grafische Darstellung der Daten, die über die serielle Schnittstelle ausgetauscht werden.

Ein Datensatz besteht aus einer einzelnen Zeile, die durch einen Zeilenumbruch (`\n` oder `Serial.println`) begrenzt wird.

Neben dem einfachen plotten von Zahlen können diese auch beschriftet werden. Beschriftungen können entweder einmalig im Setup gesetzt werden ODER sie können mit jeder Zeile übergeben werden.

```
1 Sensor1:30, Sensor2:45\n.
```



Anmerkungen:

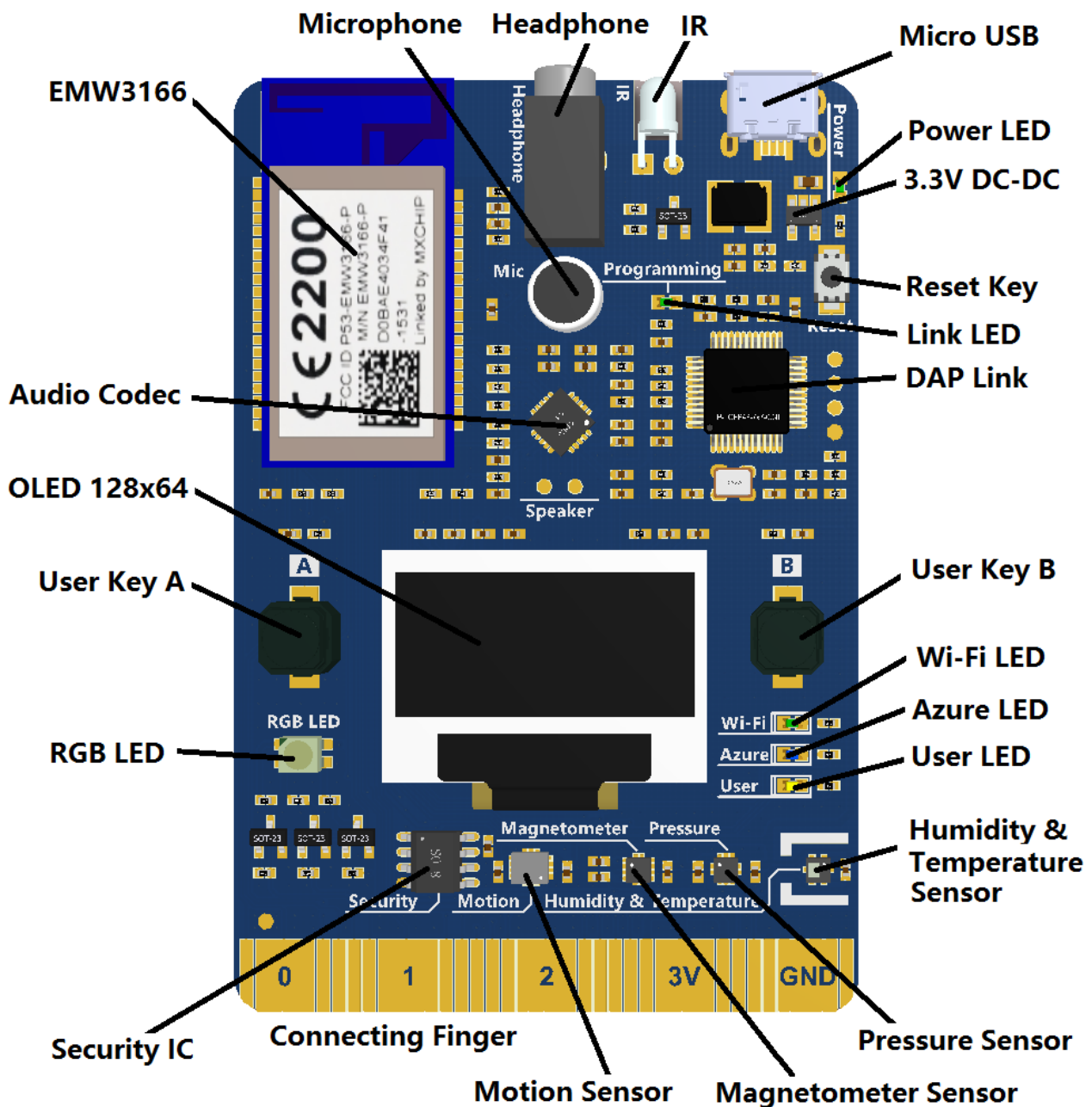
1. Unser Board generiert automatisch eine Ausgabe zur Version, Hersteller usw., die den Plotter "irritiert". Entsprechend zeigte sich die Darstellung der Legende in der zeilenbasierten Darstellung als stabiler.
2. Wenn Sie die zeilenbasierten Beschriftungen verwenden, fügen Sie kein Leerzeichen nach dem ":" ein!

Merke: Die Serielle Schnittstelle kann aus allen Programmiersprachen heraus genutzt werden. Sie können also in C, C++, Python usw. eigene Programme für das Auslesen und die Interpretation der Daten schreiben.

10.5 Unser Controller

Kategorie	Features laut Webseite	Bedeutung
Controller	<i>EMW3166 Wifi module</i>	Eingebetteter Controller mit WLAN Schnittstelle
Peripherie	<i>Audio codec chip</i>	Hardware Audioverarbeitungseinheit
	<i>Security encryption chip</i>	Verschlüsselungsfeatures in einem separaten Controller
Aktoren	<i>2 user button</i>	
	<i>1 RGB light</i>	
	<i>3 working status indicator</i>	WiFi, Azure, User Leds auf der rechten Seite
	<i>Infrared emitter</i>	Infrarot Sender für die Nutzung als Fernbedienung
	<i>OLED, 128×64</i>	Display mit einer Auflösung von 128 x 64 Pixeln
Sensoren	<i>Motion sensor</i>	Interialmesssystem aus Beschleunigungssensor und Gyroskop
	<i>Magnetometer sensor</i>	
	<i>Atmospheric pressure sensor</i>	
	<i>Temperature and humidity sensor</i>	

Kategorie	Features laut Webseite	Bedeutung
	<i>Microphone</i>	
Anschlüsse	<i>Connecting finger extension interface</i>	



1

Worin unterscheidet sich der Controller von einem willkürlich ausgewählten Laptop?

Parameter	AZ3166	Laptop
Bandbreite	32Bit	64Bit
Taktrate	100Mhz	3.5Ghz
Arbeitsspeicher	256kBytes	8 GBytes
Programmspeicher	1 MByte	
Energieaufnahme	0.x Watt	100 Watt
Kosten	40 Euro	> 500 Euro
Einsatz	Spezifisches eingebettetes System	Vielfältige Einsatzmöglichkeiten

¹Produktwebseite Firma MXChip, AZ3166 Product Details [Link](#)

Zum Vergleich, eine Schreibmaschinenseite umfasst etwa 2KBytes.

Die Dokumentation der zugehörige *Application Programming Interface* (API) finden Sie unter [Link](#). Hier stehen Klassen bereit, die Einbettung der Sensoren, LEDs, des Displays kapseln und die Verwendung vereinfachen.

10.6 Anwendungsfall

Bestimmen Sie die Periodendauer eines Fadenpendels bzw. die Erdbeschleunigung bei einer bekannten Fadenlänge!

$$T_0 = 2\pi\sqrt{\frac{l}{g}}$$

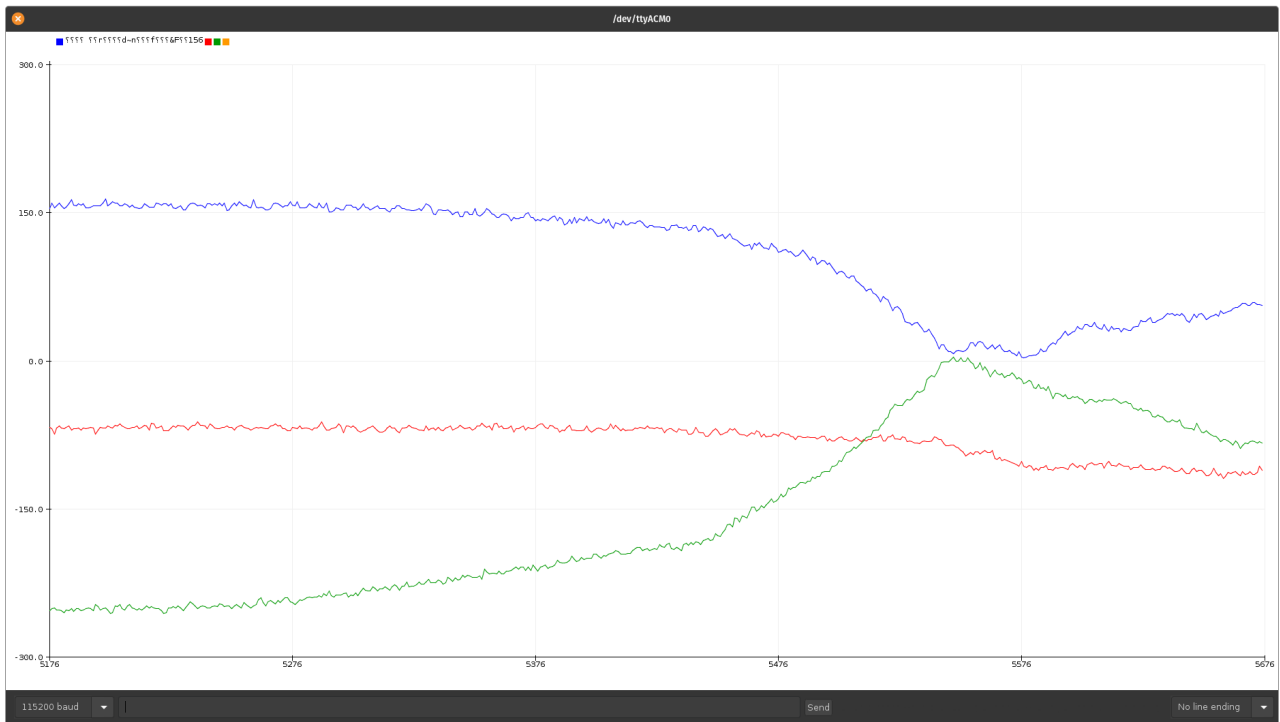
Frage: Welche Sensoren können wir dafür nutzen?

Unser Ansatz unterscheidet sich davon, wir nutzen den Magnetkompass unseres Boards, der auf die Präsenz von magnetoresistiven Materialien in der Umgebung reagiert [Link](#).

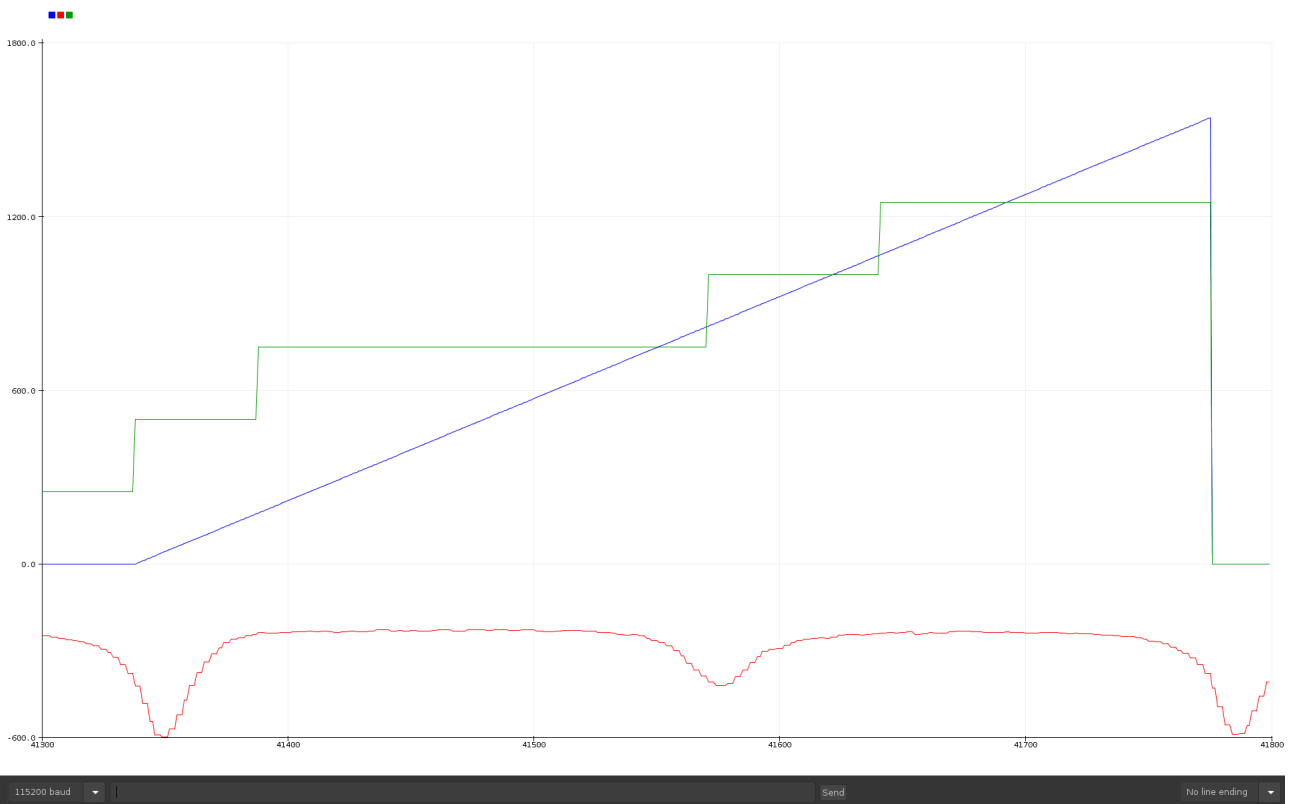
Der Sensor ist über den I2C Bus an unser Board angeknüpft. Dieser überträgt die Messdaten serialisiert an den Controller. Wir müssen also, um die Daten nutzen zu können sowohl eine Instanz der Sensorklasse `LIS2MDLSensor` als auch eine Instanz der Busklasse `DevI2C` erzeugen. Dem Sensor wird dabei das erzeugte Busobjekt im Konstruktor übergeben.

```
1 #include "LIS2MDLSensor.h"
2
3 DevI2C ext_i2c(D14,D15);
4 LIS2MDLSensor lis2mdl(ext_i2c);
5
6 int axes[3];
7
8 void setup(){
9     lis2mdl.init(NULL);
10    Serial.begin(115200);
11 }
12
13 void loop(){
14     lis2mdl.getMAxes(axes);
15     Serial.printf("%d, %d, %d\n", axes[0], axes[1], axes[2]);
16     delay(10);
17 }
```

Bringen wir einen magnetoresistives Objekt in die Nähe des Sensors, so kann er dessen Auswirkung auf das Magnetfeld der Erde messen.



Für unser Pendel sieht der Verlauf dann entsprechend wie folgt aus:



Eine Lösung für die Extraktion der Periodendauer finden Sie in unserem Vorlesungsverzeichnis [Link](#). Gelingt es Ihnen eine bessere Lösung zu entwickeln?

```
python @PyScript.env - matplotlib - numpy
```

```
“ python @PyScript.repl import numpy as np import matplotlib.pyplot as plt
```

```
periods = np.array([1524, 1541, 1541, 1541, 1541, 1541, 1531, 1541, 1541, 1541, 1541, 1537, 1531, 1541, 1541,
1541, 1541, 1531, 1530, 1530, 1541, 1541, 1542, 1537, 1531, 1541, 1541, 1531, 1541, 1531, 1541, 1531, 1541, 1531,
1542, 1531, 1542, 1541, 1531, 1537, 1531, 1542, 1531, 1541, 1531, 1531, 1542, 1541, 1531, 1541, 1531, 1531, 1531,
1531, 1531, 1541, 1535, 1531, 1541, 1531, 1545, 1541, 1541, 1541, 1541, 1541, 1541, 1541, 1541, 1541, 1537, 1541, 1541,
```

```
1541, 1531, 1541, 1542, 1541, 1541, 1541, 1541, 1531, 1542, 1541, 1541, 1541, 1541, 1531, 1541, 1541, 1542, 1541,
1541, 1542, 1531, 1541, 1541, 1531, 1538, 1530, 1531, 1530, 1541, 1530, 1541, 1534, 1541, 1531, 1531, 1541, 1541,
1530, 1541, 1531, 1542, 1531, 1541, 1531, 1534, 1541, 1531, 1541, 1534, 1541, 1542, 1531, 1541, 1531, 1541, 1541,
1541, 1541, 1531, 1531, 1530, 1541, 1531, 1530, 1532, 1531, 1541, 1541, 1541, 1534, 1545, 1541, 1531, 1531, 1541,
1542, 1541, 1534, 1530, 1542, 1531, 1531, 1542, 1531, 1541, 1541, 1530, 1535, 1541, 1541])
```

```
periods = periods / 1000 # ms -> s length = (periods / ( 2 * np.pi))**2 * 9.81 # vgl. Gleichung Fadenpendel
```

```
fig, ax = plt.subplots() ax.hist(length) plt.title("Gaussian Histogram") plt.xlabel("Periodendauer in ms")
plt.ylabel("Häufigkeit")
```

```
fig # notwendig für die Ausgabe in LiaScript sonst plt.show() ““
```