

## **Prozedurale Programmierung - TU Freiberg**

[https://github.com/TUBAF-IfI-LiaScript/VL\\_ProzeduraleProgrammierung/](https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/)

andre-dietrich

JayTee42

SebastianZug  
Lorcc

galinarudolf  
Anjuschenka

DkPepper

Lalelele



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>7</b>
1.1	Wie arbeitet ein Rechner eigentlich?	7
1.1.1	Programmierung	9
1.1.2	Einordnung von C und C++	10
1.2	Erstes C Programm	11
1.2.1	“Hello World”	11
1.2.2	Ein Wort zu den Formalien	11
1.2.3	Gute Kommentare	12
1.2.4	Schlechte Kommentare	13
1.2.5	Was tun, wenn es schief geht?	13
1.2.6	Compilermessages	13
1.2.7	Und wenn das Kompilieren gut geht?	14
1.3	Warum dann C?	14
1.4	Und wo kommt der Begriff des Algorithmus vor?	14
<b>2</b>	<b>Grundlagen der Sprache C</b>	<b>15</b>
2.1	Variablen	15
2.1.1	Zulässige Variablennamen	16
2.1.2	Datentypen	17
2.1.3	Wertspezifikation	23
2.1.4	Adressen	24
2.1.5	Sichtbarkeit und Lebensdauer von Variablen	24
2.1.6	Definition vs. Deklaration vs. Initialisierung	25
2.1.7	Typische Fehler	25
2.2	Input-/ Output Operationen	26
2.2.1	<code>printf</code>	26
2.2.2	<code>getchar</code>	28
2.2.3	<code>scanf</code>	29
2.3	Ausblick	29
<b>3</b>	<b>Operatoren</b>	<b>31</b>
3.1	Überblick	32
3.2	Zuweisungsoperator	32
3.3	Inkrement und Dekrement	33
3.4	Arithmetische Operatoren	33
3.5	Vergleichsoperatoren	35
3.6	Logische Operatoren	35
3.7	OPTIONAL: Bit-Operationen	36
3.8	OPTIONAL: Shift Operatoren	38
3.9	OPTIONAL: Bedingungsoperator	38
3.10	<code>sizeof</code> - Operator	39
3.11	Vorrangregeln	39
3.12	OPTIONAL: Beispiel des Tages	40
3.13	... und mal praktisch	40
<b>4</b>	<b>Kontrollstrukturen</b>	<b>43</b>
4.1	Cast-Operatoren	44
4.1.1	Implizite Datentypumwandlung	44
4.1.2	Explizite Datentypumwandlung	45

4.2	Kontrollfluss . . . . .	46
4.2.1	Verzweigungen . . . . .	46
4.2.2	Schleifen . . . . .	52
4.2.3	Kontrolliertes Verlassen der Anweisungen . . . . .	55
4.2.4	GoTo or not GoTo? . . . . .	55
4.3	Darstellung von Algorithmen . . . . .	56
4.4	Beispiel des Tages . . . . .	57
<b>5</b>	<b>Zeiger und Arrays</b>	<b>59</b>
5.1	Wie weit waren wir gekommen? . . . . .	59
5.2	Grundkonzept Zeiger . . . . .	60
5.2.1	Definition von Zeigern . . . . .	60
5.2.2	Initialisierung . . . . .	61
5.2.3	Fehlerquellen . . . . .	62
5.3	Arrays . . . . .	62
5.3.1	Deklaration, Definition, Initialisierung, Zugriff . . . . .	63
5.3.2	Fehlerquelle Nummer 1 - out of range . . . . .	64
5.3.3	Anwendung eines eindimensionalen Arrays . . . . .	64
5.3.4	Mehrdimensionale Arrays . . . . .	64
5.3.5	Anwendung eines zweidimensionalen Arrays . . . . .	65
5.3.6	Strings/Zeichenketten . . . . .	66
5.3.7	Anwendung von Zeichenketten . . . . .	66
5.3.8	Fehlerquellen . . . . .	68
5.4	Zeigerarithmetik . . . . .	68
5.5	Beispiel der Woche . . . . .	69
<b>6</b>	<b>Funktionen</b>	<b>71</b>
6.1	Feedback von Ihrer Seite . . . . .	71
6.2	Motivation . . . . .	71
6.2.1	Funktionsdefinition . . . . .	73
6.2.2	Beispiele für Funktionsdefinitionen . . . . .	74
6.2.3	Aufruf der Funktion . . . . .	74
6.2.4	Fehler . . . . .	75
6.2.5	asdfas . . . . .	76
6.2.6	Funktionsdeklaration . . . . .	76
6.2.7	Parameterübergabe und Rückgabewerte . . . . .	77
6.2.8	Zeiger als Rückgabewerte . . . . .	79
6.2.9	main-Funktion . . . . .	80
6.2.10	inline-Funktionen . . . . .	80
6.3	Lebensdauer und Sichtbarkeit von Variablen . . . . .	80
6.4	Beispiel des Tages . . . . .	81
<b>7</b>	<b>Zusammengesetzte Datentypen</b>	<b>83</b>
7.1	Exkurs: Präcompiler Direktiven . . . . .	84
7.1.1	#include . . . . .	84
7.1.2	#define . . . . .	85
7.2	Aufzählungen . . . . .	87
7.3	Typdefinition mit typedef . . . . .	88
7.4	Strukturen . . . . .	89
7.4.1	Deklaration, Definition, Initialisierung und Zugriff . . . . .	90
7.4.2	Vergleich von struct-Variablen . . . . .	90
7.4.3	Structs und Funktionen . . . . .	91
7.4.4	Arrays von Strukturen . . . . .	91
7.5	Beispiel des Tages . . . . .	92
<b>8</b>	<b>Standardalgorithmen in C</b>	<b>95</b>
8.1	Rekursion . . . . .	96
8.2	Algorithmusbegriff . . . . .	97
8.3	Suche des Maximums . . . . .	99
8.4	Sortieren . . . . .	102
8.4.1	BubbleSort . . . . .	103

8.4.2	Quicksort . . . . .	104
8.5	Suchen . . . . .	106
8.6	Implementierung in der Standardbibliothek . . . . .	107
8.7	Beispiel des Tages . . . . .	109



# Kapitel 1

## Einführung

Parameter	Kursinformationen
Veranstaltung	Vorlesung Prozedurale Programmierung
Semester	Wintersemester 2021/22
Hochschule:	Technische Universität Freiberg
Inhalte:	Vorstellung des Arbeitsprozesses
Link auf	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/00_Einfuehrung.md">https://github.com/TUBAF-IfI-</a>
Repository:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/00_Einfuehrung.md">LiaScript/VL_ProzeduraleProgrammierung/blob/master/00_Einfuehrung.md</a>
Autoren	@author

---

### Fragen an die heutige Veranstaltung ...

- Welche Aufgabe erfüllt eine Programmiersprache?
  - Erklären Sie die Begriffe Compiler, Editor, Programm, Hochsprache!
- 

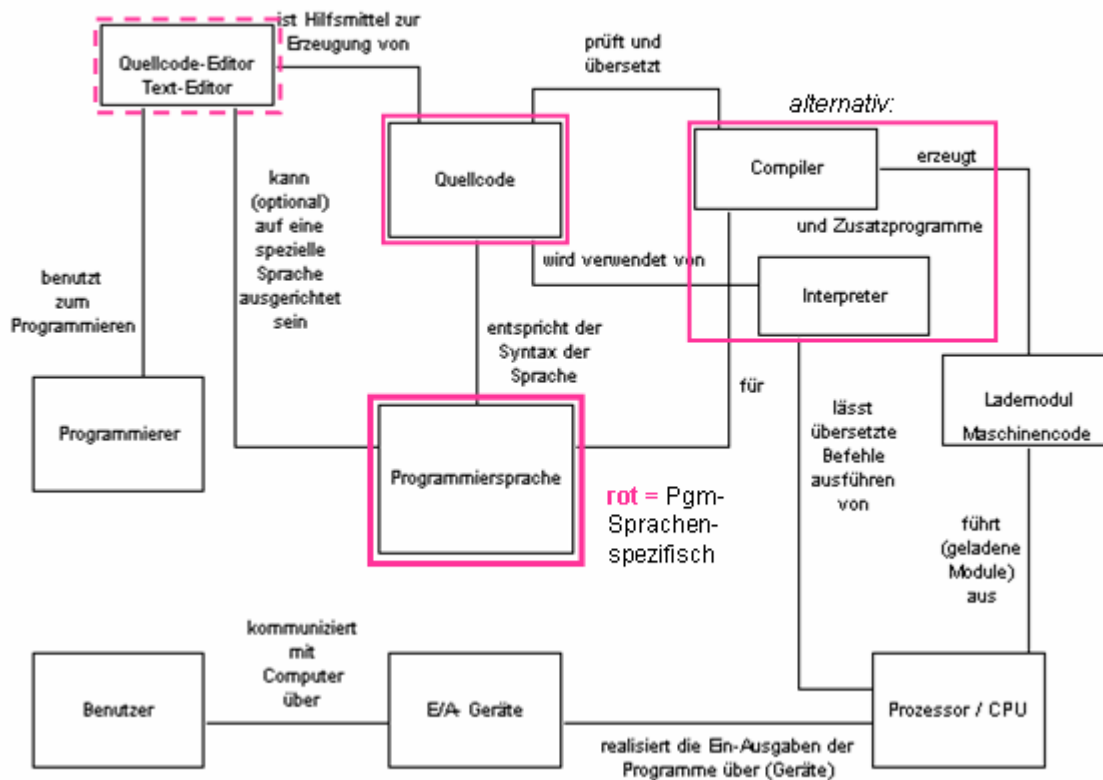
### 1.1 Wie arbeitet ein Rechner eigentlich?

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



Beispiel: Intel 4004-Architektur (1971)

intel

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 <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	OPA D <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	DESCRIPTION OF OPERATION
00	NOP	0 0 0 0	0 0 0 0	No operation.
1 - ..	*JCN	0 0 0 1 A <sub>2</sub> A <sub>2</sub> A <sub>2</sub> A <sub>2</sub>	C <sub>1</sub> C <sub>2</sub> C <sub>3</sub> C <sub>4</sub> A <sub>1</sub> A <sub>1</sub> A <sub>1</sub> A <sub>1</sub>	Jump to ROM address A <sub>2</sub> A <sub>2</sub> A <sub>2</sub> A <sub>2</sub> , A <sub>1</sub> A <sub>1</sub> A <sub>1</sub> A <sub>1</sub> (within the same ROM that contains this JCN instruction) if condition C <sub>1</sub> C <sub>2</sub> C <sub>3</sub> C <sub>4</sub> is true, otherwise go to the next instruction in sequence.
2 - ..	*FIM	0 0 1 0 D <sub>2</sub> D <sub>2</sub> D <sub>2</sub> D <sub>2</sub>	R R R 0 D <sub>1</sub> D <sub>1</sub> D <sub>1</sub> D <sub>1</sub>	Fetch immediate (direct) from ROM Data D <sub>2</sub> D <sub>2</sub> D <sub>2</sub> D <sub>2</sub> D <sub>1</sub> D <sub>1</sub> D <sub>1</sub> D <sub>1</sub> 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](#)

### 1.1.1 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

**Beispiel**

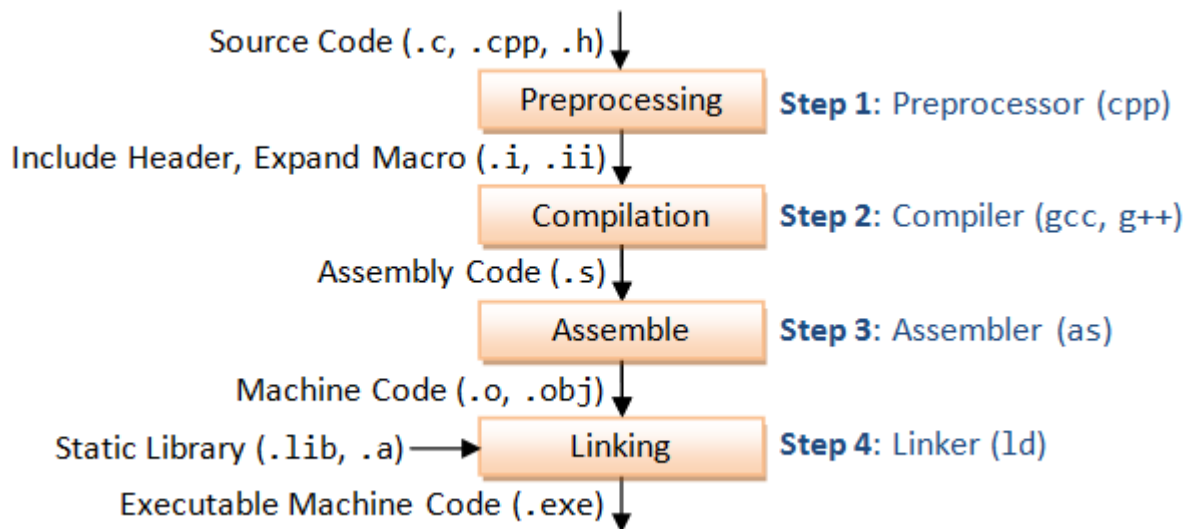
1	Assembler Code		Fortran
2			
3	.START ST		A:=2;
4	ST: MOV R1, #2		FOR I:=1 TO 20 LOOP
5	MOV R2, #1		A:=A*I;
6	M1: CMP R2, #20		END LOOP;
7	BGT M2		PRINT(A);
8	MUL R1, R2		
9	INI R2		
10	JMP M1		
11	M2: JSR PRINT		
12	.END		

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:



### 1.1.2 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.

## 1.2 Erstes C Programm

We will assume that you are familiar with the mysteries of creating files, text editing, and the like in the operating system you run on, and that you have programmed in some language before.

## 2. A Simple C Program

```
main( ) {
    printf("hello, world");
}
```

A C program consists of one or more functions, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. `main` is such a function, and in fact all C programs must have a `main`. Execution of the program begins at the first statement of `main`. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here `main` is a function of no arguments, indicated by `( )`. The `{ }` enclose the statements of the function. Individual statements end with a semicolon but are otherwise free-format.

`printf` is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A function is invoked by naming it, followed by a list of arguments in parentheses. There is no `CALL` statement as in Fortran or PL/I.

Quelle: Brian\_Kernighan, Programming in C: A Tutorial 1974

### 1.2.1 “Hello World”

```
1 // That's my first C program
2 // Karl Klammer, Oct. 2018
3
4 #include<stdio.h>
5
6 int main(void) {    // alternativ "int main()"
7     printf("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>printf()</code>
6	Einsprungstelle für den Beginn des Programmes
6 - 9	Ausführungsblock der <code>main</code> -Funktion
7	Anwendung einer Funktion ... <code>name(Parameterliste)</code> ... hier zur Ausgabe auf dem Bildschirm
8	Definition eines Rückgabewertes für das Betriebssystem

### 1.2.2 Ein Wort zu den Formalien

```
1 // Karl Klammer
2 // Print Hello World drei mal
3
4 #include <stdio.h>
5
```

```

6 int main() {
7     int zahl;
8     for (zahl=0; zahl<3; zahl++){
9         printf("Hello World! ");
10    }
11    return 0;
12 }

```

```

1 #include <stdio.h>
2 int main() {int zahl; for (zahl=0; zahl<3; zahl++){ printf("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”

### 1.2.3 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 x 3/4) 1/5 (1/2)^3 + (1/2 x 3/4 x 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 suming 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 }

```

### 1.2.4 Schlechte Kommentare

#### 1. Überkommentierung von Code

```
1 x = x + 1;  /* increment the value of x */
2 printf("Hello World\n"); // 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
8 // I am not responsible of this code.
9
10 try {
11
12 } catch(e) {
13
14 } finally { // should never happen }
```

[Sammlung von Kommentaren](#)

### 1.2.5 Was tun, wenn es schief geht?

```
1 #include<stdio.h>
2
3 int main() {
4     for (zahl=0; zahl<3; zahl++){
5         printf("Hello World ! ")
6     }
7     return 0;
8 }
```

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

### 1.2.6 Compilermessages

#### Beispiel 1

```
1 #include <stdio.h>
2
3 int mani() {
4     printf("Hello World");
5     return 0;
6 }
```

#### Beispiel 2

```
1 #include <stdio.h>
2
3 int main()
```

```

4   printf("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 <stdio.h>
2
3 int main() {
4     printf("Hello World\n");
5     printf("Wo liegt der Fehler?")
6     return 0;
7 }

```

### 1.2.7 Und wenn das Kompilieren gut geht?

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

```

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

```

## 1.3 Warum dann C?

Zwei Varianten der Umsetzung ... C vs. Python

```

1 #include <stdio.h>
2
3 int main() {
4     char zahl;
5     for (int zahl=0; zahl<3; zahl++){
6         printf("%d Hello World\n", zahl);
7     }
8     return 0;
9 }

```

```

1 import sys
2
3 for i in range(3):
4     print(i, "Hello World")
5
6 #sys.exit()

```

@Pyodide.eval

## 1.4 Und wo kommt der Begriff des Algorithmus vor?

... Nehmen wir mal an ...

# Kapitel 2

## Grundlagen der Sprache C

---

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

---

Veranstaltung	Vorlesung Prozedurale Programmierung
---------------	--------------------------------------

Semester	Wintersemester 2021/22
----------	------------------------

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

Inhalte:	Elemente der Programmiersprache
----------	---------------------------------

Link auf	<a href="https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/01_EingabeAusgabeDatentypen.md">https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/</a>
----------	---

Reposi-	01_EingabeAusgabeDatentypen.md
---------	--------------------------------

tory:	
-------	--

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

*Ein Rechner ist eigentlich ziemlich dumm, dass aber viele Millionen mal pro Sekunde*

Zitat - Quelle gesucht!

```
1 #include<stdio.h>
2
3 int main(void) {
4     printf("%d",43 + 17);
5     return 0;
6 }
```

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

```

1 #include<stdio.h>
2
3 int main(void) {
4     int x;
5     x = 5;
6     printf("f(%d) = %d \n",x, 3*x*x + 4*x + 8);
7     x = 9;
8     printf("f(%d) = %d ",x, 3*x*x + 4*x + 8);
9     return 0;
10 }

```

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 (`auto`, `goto`, `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<stdio.h>
2
3 int main(void) {
4     int int = 5;
5     printf("Unsere Variable hat den Wert %d \n", x);
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)	YouLikeCamelCase, HumanDetectionSuccessfull
(lowerCamel)	youLikeCamelCase, humanDetectionSuccessfull
underscores	I_hate_Camel_Case, human_detection_successfull

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 gibt es folgende Typen für Ganzzahlen:

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

```
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 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

Seit dem C99 Standard existiert ein spezieller Datentyp `_Bool` für binäre Variablen. Zuvor konnte das Wertepaar `true` (für wahr) und `false` (für falsch) verwendet werden, die in der Headerdatei `<stdbool.h>` mit der Konstante 1 und 0 definiert sind.

```

1 #include <stdio.h>
2 #include <stdbool.h>
3
4 int main() {
5     _Bool a = true;
6     _Bool b = false;
7     _Bool c = 45;
8
9     printf("a = %i, b = %i, c = %i\n", a, b, c);
10    return 0;
11 }
```

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 <stdio.h>
2
3 int main(void)
4 {
5     int x;
6     printf("x umfasst %d Byte.", (unsigned int)sizeof x);
7     return 0;
8 }
```

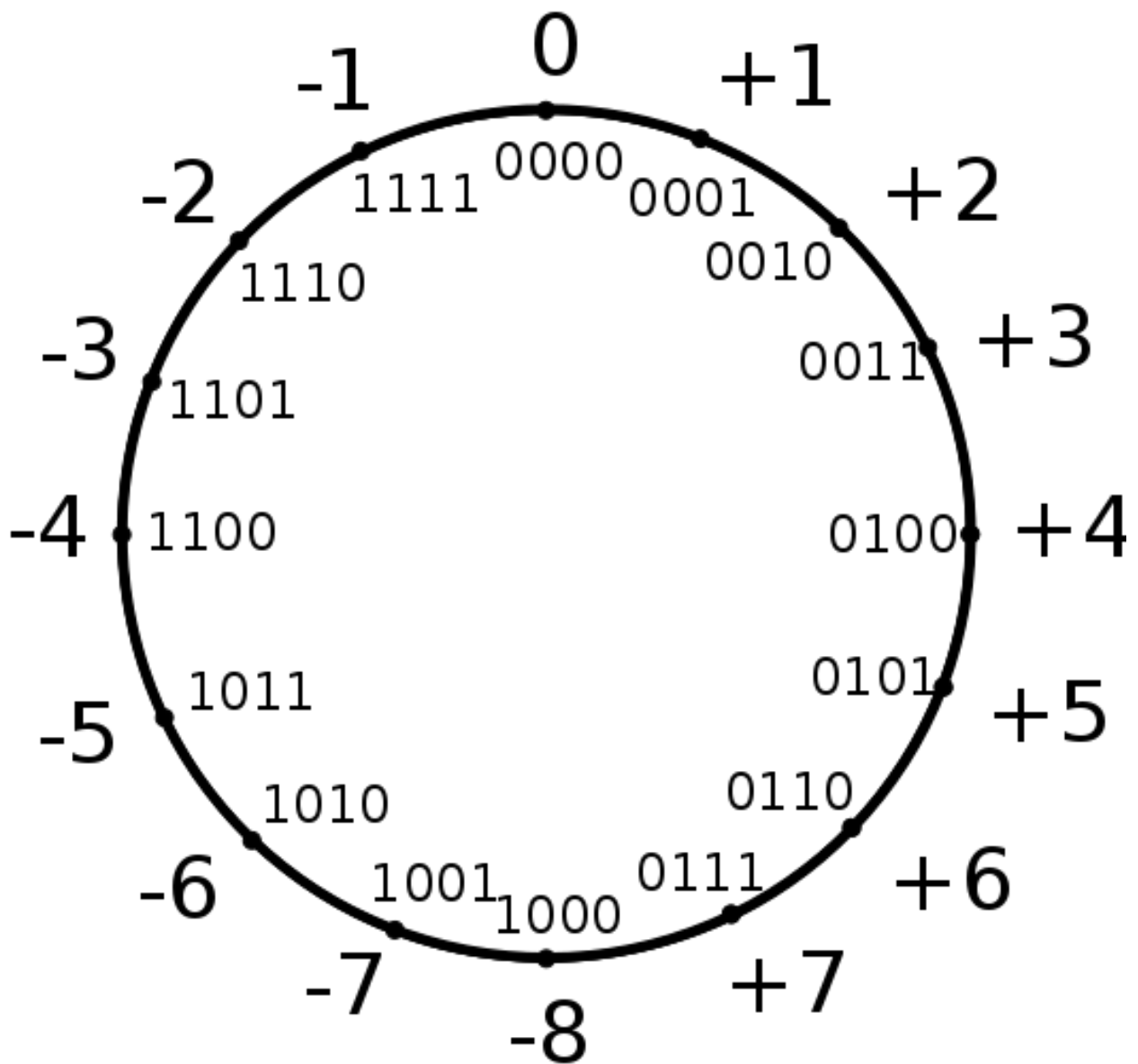
```
1 #include <stdio.h>
2 #include <limits.h>    /* INT_MIN und INT_MAX */
3
4 int main(void) {
5     printf("int size : %d Byte\n", (unsigned int) sizeof( int ) );
6     printf("Wertebereich von %d bis %d\n", INT_MIN, INT_MAX);
7     printf("char size : %d Byte\n", (unsigned int) sizeof( char ) );
8     printf("Wertebereich von %d bis %d\n", CHAR_MIN, CHAR_MAX);
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 <stdio.h>
2 #include <limits.h>    /* SHRT_MIN und SHRT_MAX */
3
4 int main(){
5     short a = 30000;
6
7     signed short c;    // -32768 bis 32767
8     printf("unsigned short - Wertebereich von %d bis %d\n", 0, USHRT_MAX);
9     c = 3000 + a;      // ÜBERLAUF!
10    printf("c=%d\n", c);
11
12    unsigned short d;   // 0 bis 65535
13    printf("short - Wertebereich von %d bis %d\n", SHRT_MIN, SHRT_MAX);
14    d = 3000 + a;
15    printf("d=%d\n", d);
16 }

```

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 immer vorzeichenbehaftet.

In 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-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<stdio.h>
2 #include<float.h>
3
4 int main(void) {
5     printf("float Genauigkeit  :%d \n", FLT_DIG);
6     printf("double Genauigkeit :%d \n", DBL_DIG);
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        printf("Gleich\n");
11    }else{
12        printf("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 im C-Standard 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 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 <stdio.h>
2
3 int main(void)
4 {
5     int x=020;
6     int y=0x20;
7     printf("x = %d\n", x);
8     printf("y = %d\n", y);

```

```

9  printf("Rechnen mit Oct und Hex z = %d", x + y);
10 return 0;
11 }

```

### 2.1.4 Adressen

**Merke:** Einige Anweisungen, wie z.B. `scanf`, verwenden Adressen von Variablen.

Um einen Zugriff auf die Adresse einer Variablen zu haben, kann man den Operator `&` nutzen. Gegenwärtig ist noch nicht klar, warum dieser Zugriff erforderlich ist, wird aber in einer der nächsten Veranstaltungen verdeutlicht.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x=020;
6      printf("%p\n",&x);
7      return 0;
8  }

```

### 2.1.5 Sichtbarkeit und Lebensdauer von Variablen

#### Lokale Variablen

Bis C99 musste eine Variable immer am Anfang eines Anweisungsblocks vereinbart werden. Nun genügt es, die Variable unmittelbar vor der ersten Benutzung zu vereinbaren.

Variablen *leben* innerhalb einer Funktion, einer Schleife oder einfach nur innerhalb eines durch geschwungene Klammern begrenzten Blocks von Anweisungen.

Wird eine Variable/Konstante z. B. im Kopf einer Schleife vereinbart, gehört sie laut C99-Standard zu dem Block, in dem auch der Code der Schleife steht. Folgender Codeausschnitt soll das verdeutlichen:

```

1  #include<stdio.h>
2
3  int main(void)
4  {
5      int v = 1;
6      int w = 5;
7      {
8          int v;
9          v = 2;
10         printf("%d\n", v);
11         printf("%d\n", w);
12     }
13     printf("%d\n", v);
14     return 0;
15 }

```

#### Globale Variablen

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

```

1  #include<stdio.h>
2
3  int v = 1; /*globale Variable*/
4
5  int main(void)
6  {
7      printf("%d\n", v);
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 i = 5;                  // 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<stdio.h>
2
3 void foo() {
4     int a;           // <- Fehlende Initialisierung
5     printf("a=%d ", a);
6 }
7
8 int main(void) {
9     int x = 5;
10    printf("x=%d ", x);
11    int y;           // <- Fehlende Initialisierung
12    printf("y=%d ", y);
13    foo();
14    return 0;
15 }
```

#### Redeklaration

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

#### Falsche Zahlenlitterale

```
1 #include<stdio.h>
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?

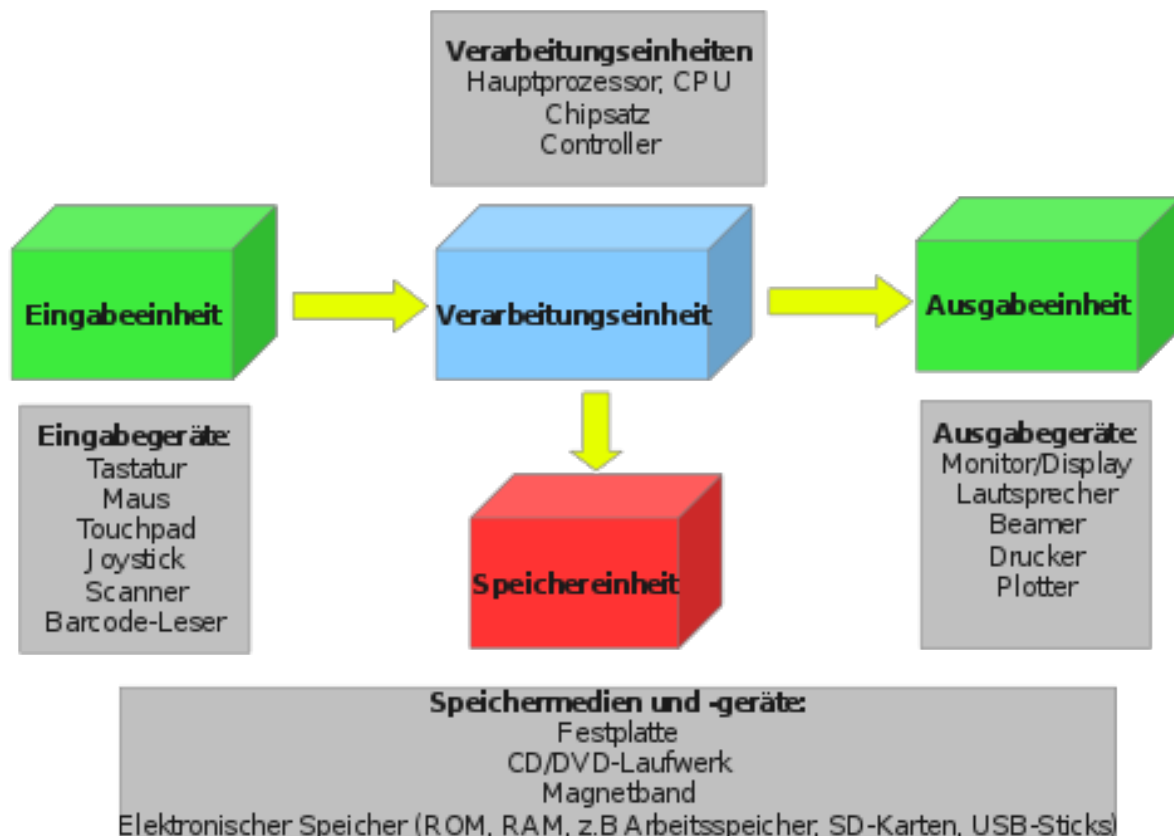
```

1 #include<stdio.h>
2
3 int main(void) {
4     short a;
5     a = 0xFFFF + 2;
6     printf("Schaun wir mal ... %hi\n", a);
7     return 0;
8 }

```

## 2.2 Input-/ Output Operationen

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



Quelle: EVA-Prinzip (Autor: Deadlyhappen)

### 2.2.1 printf

Die Bibliotheksfunktion `printf` dient dazu, eine Zeichenkette (engl. *String*) auf der Standardausgabe auszugeben. In der Regel ist die Standardausgabe der Bildschirm. Über den Rückgabewert liefert `printf` die Anzahl der ausgegebenen Zeichen. Wenn bei der Ausgabe ein Fehler aufgetreten ist, wird ein negativer Wert zurückgegeben.

Als erstes Argument von `printf` sind **nur Zeichenkette** erlaubt. Bei folgender Zeile gibt der Compiler beim Übersetzen deshalb eine Warnung oder einen Fehler aus:

```

1 printf(55);           // Falsch
2 printf("55");         // Korrekt

```

Dabei kann die Zeichenkette um entsprechende Parameter erweitert werden. Jeder Parameter nach der Zeichenkette wird durch einen Platzhalter in dem selben repräsentiert.

```

1 #include <stdio.h>
2

```

```

3 int main(){
4     //
5     //      |-----|
6     printf("%i plus %2.2f ist gleich %s.\n", 3, 2.0, "Fuenf");
7     //      <----->
8     //      String mit Referenzen
9     //      auf Parameter
10    return 0;
11 }

```

Zeichen	Umwandlung
%d oder %i	int
%c	einzelnes Zeichen
%e oder %E	double im Format [-]d.ddd e±dd bzw. [-]d.ddd E±dd
%f	double im Format [-]ddd.ddd
%o	int als Oktalzahl ausgeben
%p	die Adresse eines Zeigers
%s	Zeichenkette ausgeben
%u	unsigned int
%x oder %X	int als Hexadezimalzahl ausgeben
%%	Prozentzeichen

Welche Formatierungsmöglichkeiten bietet `printf` noch?

- die Feldbreite
- ein Flag
- durch einen Punkt getrennt die Anzahl der Nachkommastellen (Längenangabe) und an letzter Stelle schließlich
- das Umwandlungszeichen selbst (siehe Tabelle oben)

### 2.2.1.1 Feldbreite

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

```

1 #include <stdio.h>
2
3 int main(){
4     printf("rechtsbuendig      : %5d, %5d, %5d\n",34, 343, 3343);
5     printf("linksbuendig       : %-5d, %-5d, %-5d\n",34, 343, 3343);
6     printf("Zu klein gedacht    : %5d\n", 234534535);
7     //printf("Ohnehin besser    : d", 12, 234534535);return 0;

```

### 2.2.1.2 Formatierungsflags

Flag	Bedeutung
-	linksbündig justieren
+	Vorzeichen ausgeben
Leerzeichen	Leerzeichen
0	numerische Ausgabe mit 0 aufgefüllt
#	für %x wird ein x in die Hex-Zahl eingefügt, für %e oder %f ein Dezimaltrenner (.)

```

1 #include <stdio.h>
2
3 int main(){
4     printf("012345678901234567890\n");
5     printf("Integer: %+11d\n", 42);
6     printf("Integer: %+11d\n", -4242);

```

```

7  printf("Integer: %-e\n", 42.2345);
8  printf("Integer: %-g\n", 42.2345);
9  printf("Integer: %#x\n", 42);
10 return 0;
11 }

```

### 2.2.1.3 Genauigkeit

Bei `%f` werden standardmäßig 6 Nachkommastellen ausgegeben. Mit `%a.bf` kann eine genauere Spezifikation erfolgen. `a` steht für die Feldbreite, `b` für die Nachkommastellen.

```

1 #include <stdio.h>
2
3 int main(){
4     float a = 142.23443513452352;
5     printf("Float Wert: %f\n", a);
6     printf("Float Wert: %9.3f\n", a);
7     return 0;
8 }

```

### 2.2.1.4 Escape-Sequenzen

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

```

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

```

## 2.2.2 getchar

Die Funktion `getchar()` liest Zeichen für Zeichen aus einem Puffer. Dies geschieht aber erst, wenn die Enter-Taste gedrückt wird.

```

1 #include <stdio.h>
2
3 int main(){
4     char c;
5     printf("Mit welchem Buchstaben beginnt ihr Vorname? ");
6     c = getchar();
7     printf("\nIch weiss jetzt, dass Ihr Vorname mit '%c' beginnt.\n", c);
8     return 0;
9 }

```

Problem: Es lassen sich immer nur einzelne Zeichen erfassen, die durch Enter-Taste bestätigt wurden. Als flexiblere Lösung lässt sich `scanf` anwenden.

### 2.2.3 scanf

Format-String der `scanf`-Anweisung kann folgendes enthalten:

- % specifier
- count of input characters
- length modifier (hh h l ll)
- conversion specifier

```
1 #include <stdio.h>
2
3 int main(){
4     int i;
5     float a;
6     char b;
7     printf("Bitte folgende Werte eingeben %%c %%f %%d: ");
8     int n=scanf("%c %f %d", &b, &a, &i);
9     printf("\n%%c %%f %%d\n", b, a, i);
10    return 0;
11 }
```

## 2.3 Ausblick

```
1 #include<stdio.h>
2
3 int main() {
4     printf("... \t bis \n\t\t zum \n\t\t\t");
5     printf("naechsten \n\t\t\t\t\t mal! \n");
6     return 0;
7 }
```



# Kapitel 3

## Operatoren

---

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

---

Veranstaltung	Vorlesung Prozedurale Programmierung
---------------	--------------------------------------

Semester	Wintersemester 2021/22
----------	------------------------

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

Inhalte:	Nutzung von Operatoren
----------	------------------------

Link auf	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/02_Operatoren.md">https://github.com/TUBAF-IfI-</a>
----------	--

Repository:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/02_Operatoren.md">LiaScript/VL_ProzeduraleProgrammierung/blob/master/02_Operatoren.md</a>
-------------	--

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.
- Was bedeutet rechtsassoziativ und linksassoziativ?
- Welche Funktion erfüllen Bit-Operationen?
- Wie werden Shift-Operatoren genutzt?
- Wie können boolsche Variablen in C ausgedrückt werden?

---

### Wie weit waren wir gekommen?

```
1 #include <stdio.h>
2
3 int main(){
4     int zahl1, zahl2, ergeb;
5     printf("Bitte geben Sie zwei Zahlen ein : ");
6     scanf("%d %d",&zahl1, &zahl2);
7     printf("\n  %8d\n", zahl1);
8     printf("+  %8d\n", zahl2);
9     printf("-----\n");
10    ergeb=zahl1+zahl2;
11    printf("  %8d\n", ergeb);
12    return 0;
13 }
```

### 3.1 Überblick

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 *unäre*, *binäre* und *ternäre* Operatoren

Operator	Operanden	Beispiel
Unäre Operatoren	1	& Adressoperator sizeof Größenoperator
Binäre Operatoren	2	+, -, %
Ternäre Operatoren	3	? Bedingungsoperator

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

```
1 b = -a; // ein Vorzeichen (unär)
2 b = a-2; // arithmetische Operation (binär)
```

#### 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
```

#### Assoziativität

Linksassoziativ	Rechtsassoziativ
Auswertung von links nach rechts	Auswertung von rechts nach links

#### Funktion des Operators

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

### 3.2 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 <stdio.h>
2
3 int main() {
4     int zahl1 = 10;
5     int zahl2 = 20;
6     int ergeb;
7     // Zuweisung des Ausdrucks 'zahl1 + zahl2'
8     ergeb = zahl1 + zahl2;
9 }
```



```

10 printf("%d + %d = %d\n", zahl1, zahl2, ergebn);
11 return 0;
12 }

```

Die Zuweisungsoperation ist rechtsassoziativ. Der Ausdruck wird von rechts nach links ausgewertet.

```

1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     a = 5;
6     a = b = a + 1;    // kein guter Stil!
7     printf("a = %d, b = %d\n", a, b);
8 }

```

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

```

1 #include <stdio.h>
2
3 int main(){
4     int x, y;
5     x = 15;           // Zuweisungsoperator
6     y = x = x + 5;
7     printf("x=%d und y=%d\n",x, y);
8
9     y = x == 20;      // Gleichheitsoperator
10    printf("x=%d und y=%d\n",x, y);
11    return 0;
12 }

```

### 3.3 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 <stdio.h>
2
3 int main(){
4     int x, result;
5     x = 5;
6     result = 2 * ++x;    // Gebrauch als Präfix
7     printf("x=%d und result=%d\n",x, result);
8     result = 2 * x++;    // Gebrauch als Postfix
9     printf("x=%d und result=%d\n",x, result);
10    return 0;
11 }

```

### 3.4 Arithmetische Operatoren

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

```

1 #include <stdio.h>
2
3 int main(){
4     int zahl1,zahl2;
5     int ergebnis;
6     zahl1=10;
7     zahl2=20;
8     printf("Zahl 1= %d\n",zahl1);
9     printf("Zahl 2= %d\n",zahl2);
10
11     // Moeglichkeit 1: zuerst Berechnung, dann Ausgabe
12     ergebnis=zahl1+zahl2;
13     printf("Summe der Zahlen:%d\n",ergebnis);
14     // Moeglichkeit 2: Berechnung direkt für die Ausgabe
15     printf("%d + %d = %d\n", zahl1, zahl2, zahl1+zahl2);
16     return 0;
17 }

```

**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 <stdio.h>
2
3 int main(){
4     int timestamp, minuten;
5
6     timestamp = 345; //[s]
7     printf("Zeitstempel %d [s]\n", timestamp);
8     minuten=timestamp/60;
9     printf("%d [s] entsprechen %d Minuten\n", timestamp, minuten);
10    return 0;
11 }

```

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

```

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

```

**Achtung:** Der Rechenoperator Modulo mit dem % Zeichen hat nichts mit dem Formatierungszeichen in printf("%d") und scanf("%f") zu tun.

Die arithmetischen Operatoren lassen sich in verkürzter Schreibweise wie folgt darstellen:

Operation	Bedeutung
+=	a+=b äquivalent zu a=a+b
-=	a-=b äquivalent zu a=a-b
*=	a*=b äquivalent zu a=a*b
/=	a/=b äquivalent zu a=a/b
%=	a%=b äquivalent zu a=a%b

```

1 #include <stdio.h>
2
3 int main() {
4     int x=2, y=4, z=6;
5     printf("x=%d y=%d z=%d\n", x, y, z);
6
7     printf("x = x+y =%d\n", x+=y);
8     printf("z = z+y =%d\n", z+=y);
9     printf("z = z+x =%d\n", z+=x);
10    printf("x=%d y=%d z=%d\n", x, y, z);
11    return 0;
12 }

```

**Achtung:** Verlieren Sie bei langen Berechnungsketten nicht den Überblick! Insbesondere die verkürzte Schreibweise forciert dies.

## 3.5 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 <stdio.h>
2 // #include <stdbool.h>
3
4 int main(){
5     int x = 15;
6     printf("x = %d \n", x);
7     printf("Aussage x > 5 ist %d \n", x > 5);
8     printf("Aussage x == 5 ist %d \n", x == -15);
9     return 0;
10 }

```

**Merke:** Der Rückgabewert einer Vergleichsoperation ist `int`! Dabei bedeutet 0 eine ungültige und 1 eine gültige Aussage.

Warum wird dafür nicht der `_Bool` Datentyp verwendet?

## 3.6 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
!	NICHT

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

Wir wollen (willkürlich) festlegen, dass für die Temperatur ein Wert zwischen -10 und -20 Grad toleriert wird

und die Luftfeuchte mit 40 bis 60 Prozent.

```

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

Anmerkung: Seit dem C99-Standard finden Sie in der Headerdatei `<iso646.h>` einige Makros `and`, `or`, `xor`, die Sie als alternative Schreibweise für logische Operatoren und Bit-Operatoren nutzen können.

Darüber hinaus können aber auch die Erweiterungen des C99 Standard die Lesbarkeit deutlich erhöhen. Nehmen wir an, dass wir die Wertetabelle einer boolschen Funktion umsetzen wollen. Diese kann zum Beispiel die Aktivierung eines Alarmzustandes generieren. `a`, `b` und `c` sind unsere Eingangsgrößen, `f` die Ausgangsgröße. Nur für zwei Fälle soll der Alarm ausgelöst werden:

a	b	c	f
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	1
1	0	1	
1	1	0	1
1	1	1	

```

1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <iso646.h>
4
5 int main(){
6     _Bool a = true;
7     _Bool b = false;
8     _Bool c = false;
9
10    _Bool f = (a and b and !c) or (a and !b and !c);
11    printf("Ergebnis der boolschen Funktion: %d", f);
12    return 0;
13 }
```

### 3.7 OPTIONAL: Bit-Operationen

Bitoperatoren verknüpfen logische Aussagen oder manipulieren einzelne Bits in binären Zahlendarstellungen.

Operation	Bedeutung
<code>&amp;</code> , <code>&amp;=</code>	bitweises und
<code> </code> , <code> =</code>	bitweises oder
<code>^</code> , <code>^=</code>	bitweises xor
<code>~</code>	bitweises Komplement

Bei der hardwarenahen Programmierung gilt es häufig Konfigurationen von Komponenten des Prozessors über einzelne Bits zu setzen oder auszulesen.

### Auslesen

Im Beispiel sehen Sie ein fiktives Konfigurationsregister eines Analog-Digital-Wandlers

7	6	5	4	3	2	1	0
XXXX	8Bit Ausgabe	10Bit Ausgabe	Loop Modus	XXXX	Start Wandlung	Wandlung Fertig	XXXX

Bit	Bedeutung
7	XXXX
6	8 Bit Ausgabe
5	10 Bit Ausgabe
4	Loop Modus
3	XXXX
2	Start Wandlung
1	Wandlung Fertig
0	XXXX

Wie können wir die Wandlung starten, sprich das 3. Bit setzen?

```

1 #include <stdio.h>
2
3 int main(){
4     int x = 64;  //'0b01000000'
5     // Here we need some magic ...
6     if (x == 68)
7         printf("Wandlung gestartet!\n");
8     else
9         printf("Nein, das passte nicht!\n");
10    return 0;
11 }
```

Der Analog-Digital-Wandler wurde gestartet, nun wollen wir prüfen, ob die Wandlung abgeschlossen ist. Wie können wir auslesen, ob das 2. Bit (Wandlung Fertig) gesetzt ist?

```

1 #include <stdio.h>
2
3 int main(){
4     int x = 66;  //'0b01000010'
5     printf("Pruefe ADC-Status ... \n");
6     if (0) // Here we need some magic ...
7         printf("Wandlung beendet!\n");
8     else
9         printf("Irgendwas stimmt nicht!\n");
10    return 0;
11 }
```

Anmerkung: Üblicherweise würde man keine “festen” Werte für die set und test Methoden verwenden. Vielmehr werden dafür durch die Hersteller entsprechende Makros bereitgestellt, die eine Portierbarkeit erlauben.

### Unterschied zu den logischen Operatoren

```

1 #include <stdio.h>
2
3 int main() {
4     if (1 && 2) printf("Aussage 1 ist wahr\n");
5     else printf("Aussage 1 ist falsch\n");
6 }
```

```

6  if (1 & 2) printf("Aussage 2 ist wahr\n");
7  else printf("Aussage 2 ist falsch\n");
8  return 0;
9 }

```

Operator	Bedeutung	Rückgabewert
<code>&amp;&amp;</code>	logischer <i>und</i> - Operator Ganzzahlen ungleich "0" werden als <i>wahr</i> interpretiert, "0" als <i>falsch</i>	{0,1}
<code>&amp;</code>	bitweiser <i>und</i> - Operator Sofern eine Übereinstimmung an einer Stelle auftritt, ergibt sich die zugehörige Ganzzahl als Ergebnis.	<code>int</code>

### 3.8 OPTIONAL: Shift Operatoren

Die Operatoren `<<` und `>>` dienen dazu, den Inhalt einer Variablen bitweise um einige Stellen nach links bzw. nach rechts zu verschieben.

```

1  #include <stdio.h>
2
3  int main(){
4      printf("%d \n", 15 << 1);
5      return 0;
6  }

```

Und wozu braucht man das? Zum einen beim Handling einzelner Bits, wie es heute beim *Beispiel des Tages* gezeigt wird. Zum anderen zur Realisierung von Multiplikations- und Divisionsoperationen mit Faktoren/Divisoren  $2^n$ .

```

1  int Bit_Test(BYTE val, BYTE bit) {
2      BYTE test_val = 0x01;    /* dezimal 1 / binär 0000 0001 */
3      /* Bit an entsprechende Pos. schieben */
4      test_val = (test_val << bit);
5      /* 0=Bit nicht gesetzt; 1=Bit gesetzt */
6      if ((val & test_val) == 0)
7          return 0;    /* nicht gesetzt */
8      else
9          return 1;    /* gesetzt */
10 }

```

### 3.9 OPTIONAL: Bedingungsoperator

Der Bedingungsoperator liefert in Abhängigkeit von der Gültigkeit einer logischen Aussage einen von zwei möglichen Ergebniswerten zurück. Folglich hat er drei Operanden:

- die Bedingung
- den Wert für eine gültige Aussage
- den Wert für eine falsche Aussage

```

1  bedingung ? wert_wenn_wahr : wert_wenn_falsch

```

Die damit formulierbaren Anweisungen sind äquivalent zu `if`-Statements.

```

1  // Falls a größer als b ist, wird a zurückgegeben, ansonsten b.
2  if (a > b)
3      return a;
4  else
5      return b;
6

```

```

7 // analog
8 return (a > b) ? a : b;

```

Aussagen mit dem Bedingungsoperator sind nicht verkürzte Schreibweise für `if-else` Anweisungen. Es sind zwei offenkundige Unterschiede zu berücksichtigen:

- Der Bedingungsoperator generiert einen Ergebniswert und kann daher z.B. in Formeln und Funktionsaufrufen verwendet werden.
- Bei `if`-Anweisungen kann der `else`-Teil entfallen, der Bedingungsoperator verlangt stets eine Angabe von beiden Ergebniswerten.

Wofür nutzen wir den Bedingungsoperator? Zum Beispiel für die hübschere Ausgabe von `_Bool` Variablen. Es gibt keine Formatierungsoption für diesen Datentyp, daher müssen wir selbst eine Ausgabe von `true` und `false` umsetzen.

```

1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <iso646.h>
4
5 int main(){
6     _Bool a = true;
7     printf("Der Ausdruck a ist %s\n", a ? "wahr" : "falsch");
8     return 0;
9 }

```

## 3.10 `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 <stdio.h>
2
3 int main(){
4     double wert=0.0;
5     printf("%ld %ld %ld\n", sizeof(0), sizeof(double), sizeof(wert));
6     return 0;
7 }

```

## 3.11 Vorrangregeln

Konsequenterweise bildet auch die Programmiersprache 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 Ausdruckes?

```

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

## 3.12 OPTIONAL: Beispiel des Tages

Das folgende Codebeispiel realisiert die binäre Ausgabe einer Zahl in der Konsole. Für die bessere Handhabung wurde die eigentliche Ausgabe in einer eigenen Funktion `int2binOutput` organisiert. Dieser Funktion wird der darzustellende Wert als Parameter `n` übergeben.

Zunächst wird ermittelt, wieviele Bits die Zahl umfasst. Natürlich ließe sich hier auch ein fester Wert hinterlegen, garantiert aber eine weitgehende Unabhängigkeit von der konkreten Architektur. Die binäre Zahlendarstellung muss nun von vorn beginnend durchlaufen werden und geprüft werden, ob an dieser Stelle eine 1 oder eine 0 steht. Dazu wird der Zahlenwert `i` mal nach rechts geschoben und das dann niederwertigste Bit mit einer 1 verglichen (und-Operator `&`). Sofern das Bit gleich eins ist, wird eine "1" ausgegeben, sonst eine "0". Dieser Vorgang wird kontinuierlich für einen immer kleiner werdende Index `i` ausgeführt, bis dieser 0 erreicht.

Um die Lesbarkeit zu steigern wird nach 8 Bit ein Leerzeichen eingefügt. Dazu wird in Zeile 10 geprüft, ob die Division mit 8 einen Rest generiert.

```

1 #include<stdio.h>
2
3 void int2binOutput(int n) {
4     int i;
5     for(i= (sizeof n) * 8 -1; i>=0; i--) {
6         if ((n >> i) & 1)
7             printf("1");
8         else
9             printf("0");
10        if (i % 8 == 0)
11            printf(" ");
12    }
13    printf("\n");
14 }
15
16 int main(){
17     int x = 254;
18     int2binOutput(x);
19     return 0;
20 }

```

## 3.13 ... 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 }
```

@AVR8js.sketch

Wie würden Sie den Code so erweitern, dass die LED bei einem einzelnen Tastendruck (A ODER B) für 1 und bei beiden synchron betätigten Tastern für 5 Sekunden aktiv wird?



# Kapitel 4

## Kontrollstrukturen

---

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

---

Veranstaltung	Vorlesung Prozedurale Programmierung
---------------	--------------------------------------

Semester	Wintersemester 2021/22
----------	------------------------

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

Inhalte:	Einführung von Basis-Kontrollstrukturen
----------	---

Link auf	<a href="https://github.com/TUBAF-IfI-">https://github.com/TUBAF-IfI-</a>
----------	---

Repository:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/03_Kontrollstrukturen.md">LiaScript/VL_ProzeduraleProgrammierung/blob/master/03_Kontrollstrukturen.md</a>
-------------	--

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

---

---

### Fragen an die heutige Veranstaltung ...

- Welche Fallstricke lauern bei expliziter und impliziter Typumwandlung?
- Wann sollte man eine explizite Umwandlung vornehmen?
- 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.

---

### Wie weit waren wir gekommen?

```
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 }

```

@AVR8js.sketch

## 4.1 Cast-Operatoren

*Casting* beschreibt die Konvertierung eines Datentypen in einen anderen. Dies kann entweder automatisch durch den Compiler vorgenommen oder durch den Programmierer angefordert werden.

Im erst genannten Fall spricht man von

- impliziten Typumwandlung, im zweiten von
- expliziter Typumwandlung.

Es wird bei Methoden vorausgesetzt, dass der Compiler eine Typumwandlung auch wirklich unterstützt. Eine Typumwandlung kann einschränkend oder erweiternd sein!

### 4.1.1 Implizite Datentypumwandlung

Operanden dürfen in C eine Variable mit unterschiedlichen Datentyp verknüpfen. Die implizite Typumwandlung generiert einen gemeinsamen Datentyp, der in einer Rangfolge am weitesten oben steht. Das Ergebnis ist ebenfalls von diesem Typ.

1. char -> short -> int -> long -> long long / float -> double -> long double
2. Die Rangfolge bei ganzzahligen Typen ist unabhängig vom Vorzeichen.
3. Standarddatentypen haben einen höheren Rang als erweiterte Ganzzahl-Typen aus `<stdint.h>` wie `int_least32_t`, obwohl beide die gleiche Breite besitzen.

Dabei sind einschränkende Konvertierungskonfigurationen kritisch zu sehen:

- Bei der Umwandlung von höherwertigen Datentypen in niederwertigere Datentypen kann es zu Informationsverlust kommen.
- Der Vergleich von `signed`- und `unsigned`-Typen kann zum falschen Ergebnis führen. So kann beispielsweise `-1 > 1U` wahr sein.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int i = -1;
6     unsigned int limit = 200;
7
8     if ( i > limit ){
9         printf(" i > limit!!! \n");
10    }
11    else{
12        printf("Kein Problem!\n");
13    }
14    return 0;
15 }

```

**Konvertierung zu \_Bool**

\*6.3.1.2 When any scalar value is converted to `_Bool`, the result is 0 if the value compares equal to 0; otherwise, the result is 1\*

C99 Standard

**Konvertierung zum int-Typ**

Beim Konvertieren des Wertes von einem in das andere `int`-Typ

- bleibt der Wert unverändert, wenn die Darstellung im Zieltyp möglich ist,
- anderenfalls ist das Ergebnis implementierungsabhängig bzw. führt zu einer Fehlermeldung

**Vermischen von Ganzzahl und Gleitkommawerten**

- Die Division zweier `int`-Werte gibt immer nur einen Ganzzahlanteil zurück. Hier findet keine automatische Konvertierung in eine Gleitpunktzahl statt.
- Bei der Umwandlung von ganz großen Zahlen (beispielsweise `long long`) in einen Gleitpunkttyp kann es passieren, dass die Zahl nicht mehr darstellbar ist.

*6.3.1.4 Real floating and integer - When a finite value of real floating type is converted to an integer type other than `_Bool`, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.*

C99 Standard

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float f=3.14;
6     int i=f;
7     printf("float value = %f / Integer-Anteil %d \n", f, i);
8     return 0;
9 }
```

**Achtung:** Implizite Typumwandlungen bergen erhebliche Risiken in sich!

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float f = -3.14;
6     unsigned int i = f;
7     printf("float value = %f / Integer-Anteil %u \n", f, i);
8     printf("float value = %f / Integer-Anteil %d \n", f, i);
9     return 0;
10 }
```

Die Headerdatei `<fenv.h>` definiert verschiedene Einstellungen für das Rechnen mit Gleitpunktzahlen. Unter anderem können Sie das Rundungsverhalten von Gleitpunkt-Arithmetiken über entsprechende Makros anpassen.

**4.1.2 Explizite Datentypumwandlung**

Anders als bei der impliziten Typumwandlung wird bei der expliziten Typumwandlung der Zieldatentyp konkret im Code angegeben.

1 (Zieltyp) ausdrück;

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i = 3;
6     int j = 4;
7     printf("int i / int j    = %d \n", i / j);
```

```

8  printf("float(i / j)    = %f \n", (float)(i / j));
9  printf("float i / int j = %f \n", (float) i / j);
10 return 0;
11 }

```

Im Beispiel wird in der ersten `printf`-Anweisung das Ergebnis der ganzzahligen Division `i/j` in `float` konvertiert, in der zweiten `printf`-Anweisung die Variable `i`.

## 4.2 Kontrollfluss

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

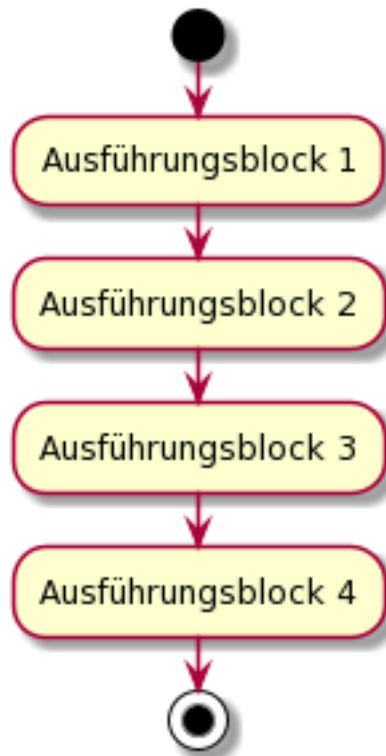


Abbildung 4.1: Modelle

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.

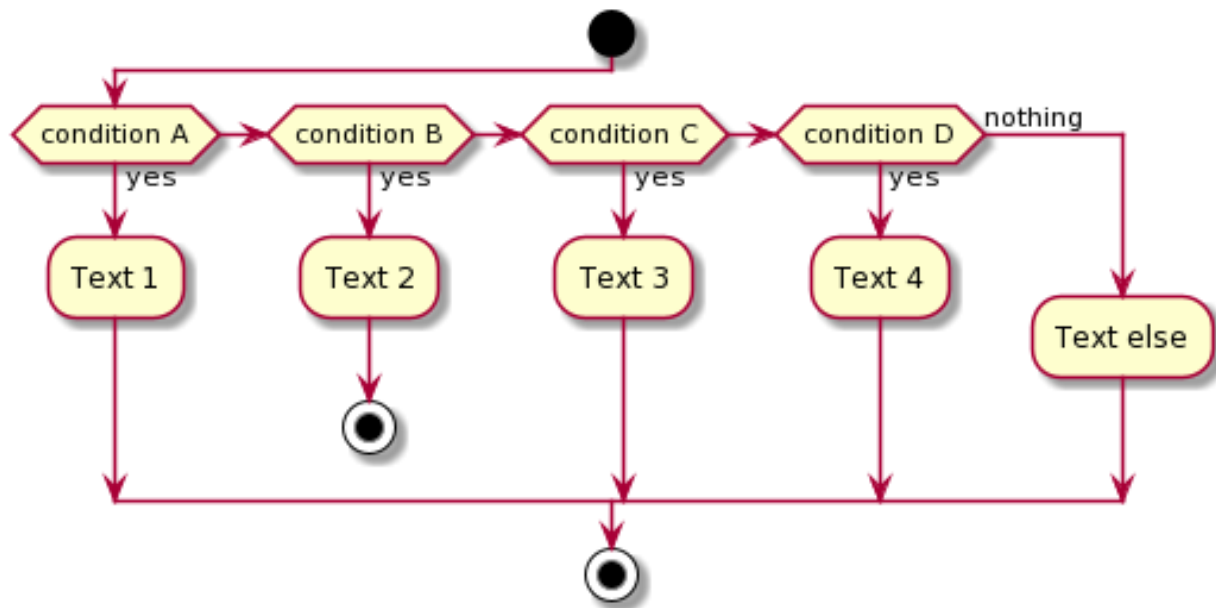
Beispiel: Ein Datensatz wird durchlaufen um die Gesamtsumme einer Spalte zu bestimmen. Wenn der letzte Eintrag erreicht ist, wird der Durchlauf abgebrochen und das Ergebnis ausgegeben.

3. Des Weiteren verfügt C 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.

### 4.2.1 Verzweigungen

Verzweigungen entfalten mehrere mögliche Pfade für die Ausführung des Programms.

Darstellungsbeispiele für mehrstufige Verzweigungen (`switch`)



Eingabe(a, operator, b)				
operator				
'+'	'-'	'*'	'/'	Sonst
ergebnis=a+b	ergebnis=a-b	ergebnis=a*b	ergebnis=a/b	Fehlermeldung

Abbildung 4.2: instruction-set

#### 4.2.1.1 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 sondern zum ersten!

### Weitere Beispiele für Bedingungen

Die Bedingungen können als logische UND arithmetische Ausdrücke formuliert werden.

Ausdruck	Bedeutung
<code>if (a)</code>	<code>if (a != 0)</code>
<code>if (!a)</code>	<code>if (a == 0)</code>
<code>if (a &gt; b)</code>	<code>if (!(a &lt;= b))</code>
<code>if ((a-b))</code>	<code>if (a != b)</code>
<code>if (a &amp; b)</code>	$a > 0, b > 0$ , mindestens ein $i$ mit $a_i == b_i$

### Mögliche Fehlerquellen

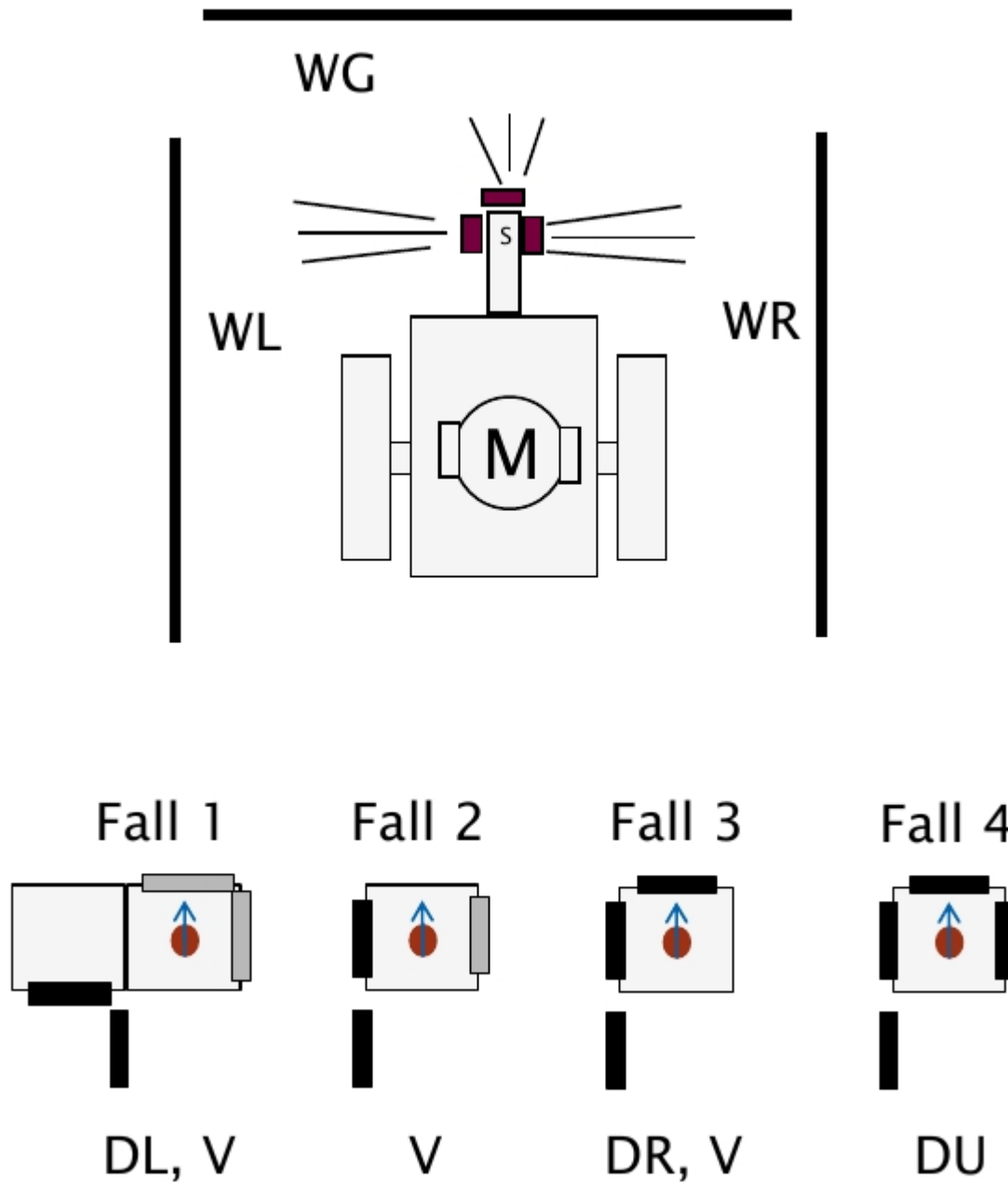
1. Zuweisungs- statt Vergleichsoperator in der Bedingung (kein Compilerfehler)
2. Bedingung ohne Klammern (Compilerfehler)
3. ; hinter der Bedingung (kein Compilerfehler)
4. Multiple Anweisungen ohne Anweisungsblock
5. Komplexität der Statements

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

```

Sehen Sie mögliche Vereinfachungen des Codes?

#### 4.2.1.2 Zwischenfrage

```

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

```

#### 4.2.1.3 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 <stdio.h>
2
3 int main() {
4     int a=50, b=60;
5     char operator;
6     printf("Bitte Operator definieren (+,-,*,/): ");
7     operator = getchar();
8
9     switch(operator) {
10         case '+':
11             printf("%d + %d = %d \n", a, b, a+b);
12             break;
13         case '-':
14             printf("%d - %d = %d \n", a, b, a-b);
15             break;
16         case '*':
17             printf("%d * %d = %d \n", a, b, a*b);
18             break;
19         case '/':
20             printf("%d / %d = %d \n", a, b, a/b);
21             break;
22         default:
23             printf("%c? kein Rechenoperator \n", operator);
24     }
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 <stdio.h>
2
3 int main() {
4     int a=5;
5
6     switch(a) {
7         case 5: // Multiple Konstanten
8         case 6:
9         case 7:
10            printf("Der Wert liegt zwischen 4 und 8\n");
11        case 3:
12            printf("Der Wert ist 3 \n");
13            break;

```

```

14     case 0:
15         printf("Der Wert ist 0 \n");
16     default: printf("Wert in keiner Kategorie\n");}
17
18     return 0;
19 }

```

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

```

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

```

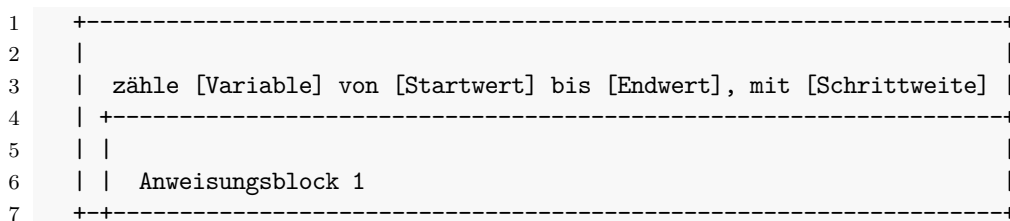
## 4.2.2 Schleifen

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*.

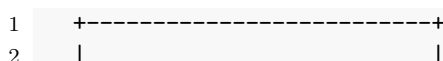
Schleifen können verschachtelt werden, d.h. innerhalb eines Schleifenkörpers können weitere Schleifen erzeugt und ausgeführt werden. Zur Beschleunigung des Programmablaufs werden Schleifen oft durch den Compiler entrollt (*Enrollment*).

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

- Iterationssymbol



- Wiederholungsstruktur mit vorausgehender Bedingungsprüfung



```

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 kennt diese drei Formen über die Schleifenkonstrukte `for`, `while` und `do while`.

#### 4.2.2.1 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 0 annimmt – also der Ausdruck `false` (falsch) ist.

```

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 <stdio.h>
2
3 int main(){
4     int i;
5     for (i = 1; i<10; i++)
6         printf("%d ", i);
7
8     printf("\nNach der Schleife hat i den Wert %d\n", i);
9     return 0;
10 }

```

#### 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 <stdio.h>
2
3 int main(){
4     int i;
5     for (i = 1; i<10; i++);
6     printf("%d ", i);
7

```

```

8  printf("Das ging jetzt aber sehr schnell ... \n %d" ,i);
9  return 0;
10 }

```

#### 4.2.2.2 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 <stdio.h>
2
3 int main(){
4     int c;
5     int zaehler = 0;
6     printf("Leerzeichenzähler - zum Beenden \"_\" [Enter]\n");
7     while((c = getchar()) != '_')
8     {
9         if(c == ' ')
10            zaehler++;
11     }
12     printf("Anzahl der Leerzeichen: %d\n", zaehler);
13     return 0;
14 }

```

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 }

```

#### 4.2.2.3 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 <stdio.h>
2
3 int main(){
4     int c;
5     int zaehler = 0;

```

```

6  printf("Leerzeichenzähler - zum Beenden \"_\" [Enter]\n");
7  do
8  {
9      c = getchar();
10     if(c == ' ')
11         zaehler++;
12 }
13 while(c != '_');
14 printf("Anzahl der Leerzeichen: %d\n", zaehler);
15 return 0;
16 }

```

### 4.2.3 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 <stdio.h>
2
3  int main(){
4      int i;
5      for (i = 1; i<10; i++){
6          if (i == 5) break;
7          printf("%d ", i);
8      }
9      printf("\nUnd vorbei ... i ist jetzt %d\n", i);
10     return 0;
11 }

```

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

```

1  #include <stdio.h>
2
3  int main(){
4      int i;
5      for (i = -5; i<6; i++){
6          if (i == 0) continue;
7          printf("%5.1f \n", 12. / i);
8      }
9      return 0;
10 }

```

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

### 4.2.4 GoTo or not GoTo?

**Goto** erlaubt es Sprungmarken (Labels) zu definieren und bei der Anweisung, die diese referenziert, die Ausführung fortzusetzen. Konsequenterweise ist damit aber eine nahezu beliebige Auflösung der Ordnung des Codes möglich.

*Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a \* **goto** can be rewritten to avoid them.\**

Tutorialspoint

```

1  #include <stdio.h>
2
3  int main(){
4      int i;
5      label:

```

```
6     printf("%d ", i);
7     i++;
8     if (i <= 10) goto label;
9     return 0;
10 }
```

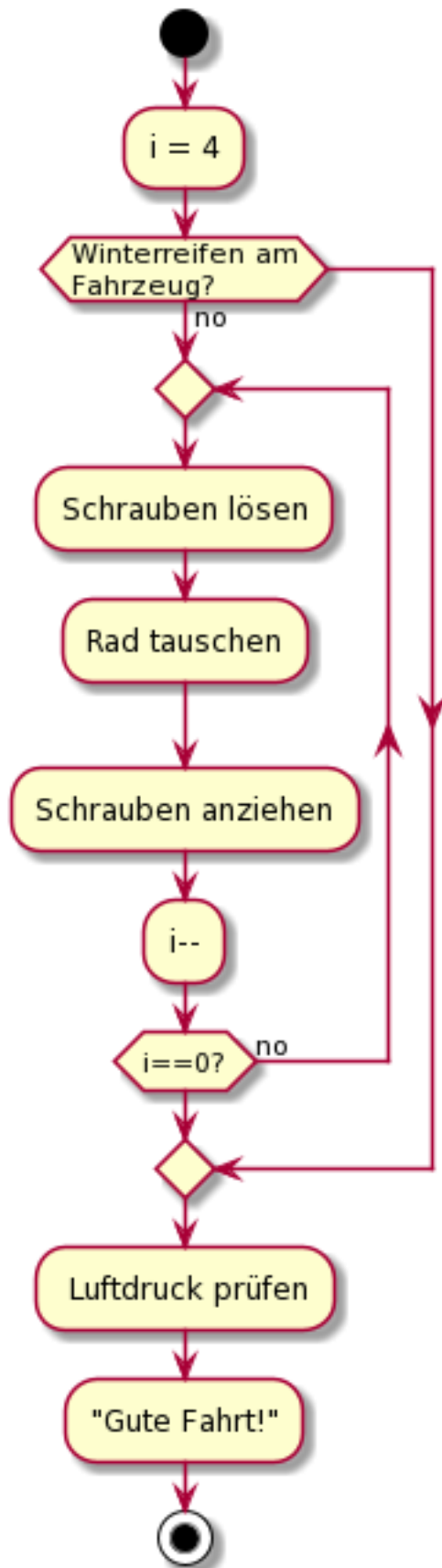
Ein wichtiger Fehler, der häufig immer mit `goto` in Verbindung gebracht wird, hat aber eigentlich nichts damit zu tun [Apple-SSL Bug](#)

## 4.3 Darstellung von Algorithmen

- Nassi-Shneiderman-Diagramme
- [Flußdiagramme](#)

Beispiel - Wechseln der Räder





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

# Kapitel 5

## Zeiger und Arrays

---

Parameter    Kursinformationen

---

**Veranstaltung:** Vorlesung Prozedurale Programmierung

**Semester:** Wintersemester 2021/22

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Darstellung der Verwendung von Zeigern und Arrays

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

**Repository:** [LiaScript/VL\\_ProzeduraleProgrammierung/blob/master/04\\_ZeigerUndArrays.md](https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/04_ZeigerUndArrays.md)

**Autoren**    @author

---

---

### Fragen an die heutige Veranstaltung ...

- Erklären Sie die Idee des Zeigers in der Programmiersprache C.
  - Welche Vorteile ergeben sich, wenn eine Variable nicht mit dem gti Wert sondern über die Adresse übergeben wird?
  - Welche Funktion hat der Adressoperator `&`?
  - Welche Gefahr besteht bei der Initialisierung von Zeigern?
  - Was ist ein NULL-Zeiger und wozu wird er verwendet?
  - Wie gibt man die Adresse, auf die ein Zeiger gerichtet ist, mit `printf` aus?
  - Erläutern Sie die mehrfache Nutzung von `*` im Zusammenhang mit der Arbeit von Zeigern.
  - In welchem Kontext ist die Typisierung von Zeigern von Bedeutung?
- 

### 5.1 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

## 5.2 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 Datenobjekte an Funktionen via “call-by-reference”,
- Übergabe von Funktionen als Argumente an andere Funktionen,
- Umsetzung rekursiver Datenstrukturen wie Listen und Bäume.

Der Vollständigkeit halber sei erwähnt, dass C anders als C++ keine Referenzen im eigentlichen Sinne kennt. Hier ist die Übergabe der Adresse einer Variablen als Parameter gemeint und nicht das Konstrukt “Reference”.

### 5.2.1 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;

```

### 5.2.2 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 (folgt gleich im zweiten Teil der Vorlesung), \* eines weiteren Zeigers oder \* des Wertes von NULL.

```

1  #include <stdio.h>
2  #include <stdlib.h>
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     printf("Pointer ptr_a      %p\n", ptr_a);
17     printf("Pointer ptr_feld %p\n", ptr_feld);
18     printf("Pointer ptr_b      %p\n", ptr_b);
19     printf("Pointer ptr_Null %p\n", ptr_Null);
20     return EXIT_SUCCESS;
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.

Ausgaben von Pointer erfolgen mit `printf("%p", ptr)`, es 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.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 <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int a = 15;
7      int * ptr_a = &a;
8      printf("Wert von a           %d\n", a);
9      printf("Pointer ptr_a        %p\n", ptr_a);
10     printf("Wert hinter dem Pointer ptr_a %d\n", *ptr_a);

```

```

11 *ptr_a = 10;
12 printf("Wert von a          %d\n", a);
13 printf("Wert hinter dem Pointer ptr_a  %d\n", *ptr_a);
14 return EXIT_SUCCESS;
15 }

```

### 5.2.3 Fehlerquellen

Fehlender Adressoperator bei der Zuweisung

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5;
7     int * ptr_a;
8     ptr_a = a;
9     printf("Pointer ptr_a          %p\n", ptr_a);
10    printf("Wert hinter dem Pointer ptr_a  %d\n", *ptr_a);
11    printf("Aus Maus!\n");
12    return EXIT_SUCCESS;
13 }

```

Fehlender Dereferenzierungsoperator beim Zugriff

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5;
7     int * ptr_a = &a;
8     printf("Pointer ptr_a          %p\n", (void*)ptr_a);
9     printf("Wert hinter dem Pointer ptr_a  %d\n", ptr_a);
10    printf("Aus Maus!\n");
11    return EXIT_SUCCESS;
12 }

```

Uninitialisierte Pointer zeigen "irgendwo ins nirgendwo"!

```

1 #include <stdio.h>
2 #include <stdlib.h>
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    printf("Pointer ptr_a          %p\n", ptr_a);
13    printf("Wert hinter dem Pointer ptr_a  %d\n", *ptr_a);
14    return EXIT_SUCCESS;
15 }

```

## 5.3 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 Feldes!

### 5.3.1 Deklaration, Definition, Initialisierung, Zugriff

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

```

1 #include <stdio.h>
2
3 int main(void) {
4     int a[3];           // Array aus 3 int Werten
5     a[0] = -2;
6     a[1] = 5;
7     a[2] = 99;
8     for (int i=0; i<3; i++)
9         printf("%d ", a[i]);
10    printf("\nNur zur Info %ld", sizeof(a));
11    printf("\nZahl der Elemente %ld", sizeof(a) / sizeof(int));
12    printf("\nAnwendung des Adressoperators auf das Array %d", *a);
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 <stdio.h>
2
3 int main(void) {
4     int a[] = {5, 2, 2, 5, 6};
5     float b[5] = {1.0};
6     int c[5] = {[2] = 5, [1] = 2, [4] = 9}; // ISO C99
7     for (int i=0; i<5; i++){
8         printf("%5d %f %5d\n", a[i], b[i], c[i]);
9     }
10    return 0;
11 }

```

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

```

1 #include <stdio.h>
2
3 int main(void) {

```

```

4  int a[3];
5  printf("\nNur zur Speicherplatz [Byte] %ld", sizeof(a));
6  printf("\nZahl der Elemente %ld\n", sizeof(a)/sizeof(int));
7  return 0;
8  }

```

### 5.3.2 Fehlerquelle Nummer 1 - out of range

```

1  #include <stdio.h>
2
3  int main(void) {
4      int a[] = {-2, 5, 99};
5      for (int i=0; i<=3; i++)
6          printf("%d ", a[i]);
7      return 0;
8  }

```

### 5.3.3 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 <stdio.h>
2
3  int main(void) {
4      int a[] = {0, 1, 2, 4, 3, 5, 6, 7, 8, 9};
5      int b[10];
6      for (int i=0; i<10; i++)
7          b[i]=i;
8      for (int i=0; i<10; i++)
9          if (a[i]!=b[i])
10             printf("An Stelle %d unterscheiden sich die Vektoren \n", i);
11      return 0;
12  }

```

Welche Verbesserungsmöglichkeiten sehen Sie bei dem Programm?

### 5.3.4 Mehrdimensionale Arrays

Deklaration:

```

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

```

Deklaration mit einer sofortigen Initialisierung aller bzw. einiger Elemente:

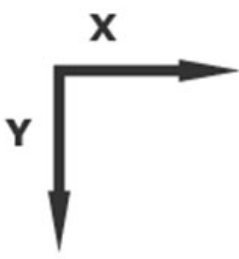
```

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

```

Quelle: [C-Kurs](#)

### 5.3.5 Anwendung eines zweidimensionalen Arrays

Elementweise Addition zweier Matrizen

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int C[2][3];
6     int i,j;
7     for (i=0;i<2;i++)
8         for (j=0;j<3;j++)
9             C[i][j]=A[i][j]+B[i][j];

```

```

10  for (i=0;i<2;i++)
11  {
12      for (j=0;j<3;j++)
13          printf("%d\t",C[i][j]);
14      printf("\n");
15  }
16  return 0;
17 }

```

### Multiplikation zweier Matrizen

## 5.3.6 Strings/Zeichenketten

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

```

1  #include <stdio.h>
2
3  int main(void) {
4      printf("Diese Form eines Strings haben wir bereits mehrfach benutzt!\n");
5      ///////////////////////////////////////////////////
6
7      char a[] = "Ich bin ein char Array!"; // Der Compiler fügt das \0 automatisch ein!
8      if (a[23] == '\0'){
9          printf("char Array Abschluss in a gefunden!");
10     }
11
12     printf("->%s<-\\n", a);
13     char b[] = { 'H', 'a', 'l', 'l', 'o', ' ',
14                 'F', 'r', 'e', 'i', 'b', 'e', 'r', 'g', '\0' };
15     printf("->%s<-\\n", b);
16     char c[] = "Noch eine \0Möglichkeit";
17     printf("->%s<-\\n", c);
18     char d[] = { 80, 114, 111, 122, 80, 114, 111, 103, 32, 50, 48, 50, 49, 0 };
19     printf("->%s<-\\n", d);
20     return 0;
21 }

```

Wie kopiere ich die Inhalte in einem Array?

```

1  #include <stdio.h>
2  #include <string.h> // notwendig für strcpy
3
4  int main(void) {
5      char a[] = "012345678901234567890123456789";
6      strcpy(a, "Das ist ein neuer Text");
7      printf("%s\\n",a);
8      return 0;
9  }

```

## 5.3.7 Anwendung von Zeichenketten

Schreiben Sie ein Programm, dass in einem Text groß geschriebene Buchstaben durch klein geschriebene ersetzt und umgekehrt.

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

Da Variablen des Datentyps `char` genau ein Byte benötigen, liefert `sizeof`-Operator im folgenden Beispiel die Anzahl der Elemente des Arrays.

```

1 #include <stdio.h>
2
3 int main() {
4     char a[] = "Das ist ein beispielhafter Text.";
5     char b[sizeof a];
6     for (int i=0; i< sizeof a; i++){
7         b[i] = a[i];
8         if ((a[i]>=65) && (a[i]<=90))
9             b[i] = a[i] + 32;
10        if ((a[i]>=97) && (a[i]<=122))
11            b[i] = a[i] - 32;
12    }
13    printf("%s\n", a);
14    printf("%s\n", b);
15    return 0;

```

```
16 }
```

### 5.3.8 Fehlerquellen

Bitte unterscheiden Sie die Initialisierungsphase von normalen Zuweisungen, bei denen Sie nur auf einzelne Elemente zugreifen können.

```
1 #include <stdio.h>
2 #include <string.h>      // notwendig für strcpy
3
4 int main(void) {
5     char a[] = "Das ist der Originaltext";
6     a = "Das ist ein neuer Text"; // Compiler Error
7     //strcpy(a, "Das ist ein neuer Text");
8     a[0]='X';
9     printf("%s\n",a);
10    return 0;
11 }
```

Auf die umfangreiche Funktionssammlung der `string.h` zur Manipulation von Strings wird in einer folgenden Vorlesung eingegangen.

## 5.4 Zeigerarithmetik

Zeiger können manipuliert werden, um variabel auf Inhalte im Speicher zuzugreifen. Wie groß ist aber eigentlich ein Zeiger und warum muss er typisiert werden?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     char    *v;
6     int     *w;
7     float   *x;
8     double  *y;
9     void    *z;
10
11    printf("char\t int\t float\t double\t void\n");
12    printf("%lu\t %lu\t %lu\t %lu\t %lu \n",
13          sizeof(v), sizeof(w), sizeof(x), sizeof(y), sizeof(z));
14    printf("%lu\t %lu\t %lu\t %lu\t %lu \n",
15          sizeof(*v), sizeof(*w), sizeof(*x), sizeof(*y), sizeof(*z));
16    return EXIT_SUCCESS;
17 }
```

Die Zeigerarithmetik erlaubt:

- Ganzzahl-Additionen
- Ganzzahl-Subtraktionen
- Inkrementierungen `ptr_i--`;
- Dekrementierungen `ptr_i++`;

Der Compiler wertet dabei den Typ der Variablen aus und inkrementiert bzw. dekrementiert die Adresse entsprechend.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a[] = {0,1,2,3,4,5};
7     int *ptr_a = a;
```

```

8  printf("Pointer ptr_a          %p\n", ptr_a);
9  int *ptr_b;
10 ptr_b = ptr_a + 1;
11 ptr_b++;
12 printf("Pointer ptr_b          %p\n", ptr_b);
13 printf("Differenz ptr_b - ptr_a %ld\n", (long)(ptr_b - ptr_a));
14 printf("Differenz ptr_b - ptr_a %ld\n", (long)ptr_b - (long)ptr_a);
15
16 printf("Wert hinter Pointer ptr_b  '%d'\n", *ptr_b);
17
18 return EXIT_SUCCESS;
19 }

```

Was bedeutet das im Umkehrschluss? Eine falsche Deklaration bewirkt ein falsches “Bewegungsmuster” über dem Speicher.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int a[] = {6,7,8,9};
7      char * ptr_a = a;
8      for (int i=0; i<sizeof(a)/sizeof(int)*4; i++){
9          printf("ptr_a %p -> ", ptr_a);
10         printf("%d\n", *ptr_a);
11         ptr_a++;
12     }
13     return EXIT_SUCCESS;
14 }

```

Pointer können natürlich nicht nur manipuliert sondern auch verglichen werden. Dabei sei noch mal darauf verwiesen, dass dabei die Adressen und nicht die Werte evaluiert werden.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int a[] = {6,7,6,9};
7      int * ptr_a = a;
8      int * ptr_b = &a[2];
9      printf("ptr_a %p -> %d \n", (void*)ptr_a, *ptr_a);
10     printf("ptr_b %p -> %d \n", (void*)ptr_b, *ptr_b);
11     if (*ptr_a == *ptr_b) printf("Werte sind gleich!\n");
12     // Im Unterschied dazu
13     if (ptr_a == ptr_b) printf("Adressen sind gleich!\n");
14     else printf("Adressen sind ungleich!\n");
15     ptr_a += 2;
16     printf("Nun zeigt ptr_a auf %p\n", (void*)ptr_a);
17     if (ptr_a == ptr_b) printf("Jetzt sind die Adressen gleich!\n");
18     else printf("Adressen sind ungleich!\n");
19     return EXIT_SUCCESS;
20 }

```

## 5.5 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 <stdio.h>
2 #include <stdlib.h>
3
4 #define ZIELWERT 18
5
6 int main(void)
7 {
8     int a[] = {1, 2, 5, 7, 9, 10, 12, 13, 16, 17, 18, 21, 25};
9     int *ptr_left = a;
10    int *ptr_right = (int *)(&a + 1) - 1;
11    printf("Value left %3d right %d\n-----\n", *ptr_left, * ptr_right);
12    do{
13        printf("Value left %3d right %d", *ptr_left, * ptr_right);
14        if (*ptr_right + *ptr_left == ZIELWERT){
15            printf(" -> TREFFER");
16        }
17        printf("\n");
18        if (*ptr_right + *ptr_left >= ZIELWERT) ptr_right--;
19        else ptr_left++;
20    }while (ptr_right != ptr_left);
21    return EXIT_SUCCESS;
22 }

```

Schauen wir uns das Ganze noch in der Ausführung mit Pythontutor an!

# Kapitel 6

## Funktionen

---

Parameter    Kursinformationen

---

**Veranstaltung:** Vorlesung Prozedurale Programmierung

**Semester:** Wintersemester 2021/22

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Motivation, Definition und Verwendung von Funktionen in C-Programmen

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

**Repository:** [LiaScript/VL\\_ProzeduraleProgrammierung/blob/master/05\\_Funktionen.md](https://github.com/TUBAF-IFI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/05_Funktionen.md)

**Autoren**    @author

---

---

### Fragen an die heutige Veranstaltung ...

- Nennen Sie Vorteile prozeduraler Programmierung!
  - Welche Komponenten beschreiben Definition einer Funktion?
  - Welche unterschiedlichen Bedeutungen kann das Schlüsselwort **static** ausfüllen?
  - Beschreiben Sie Gefahren bei der impliziten Typkonvertierung.
  - Erläutern Sie die Begriffe Sichtbarkeit und Lebensdauer von Variablen.
  - Welche kritischen Punkte sind bei der Verwendung globaler Variablen zu beachten.
  - Warum ist es sinnvoll Funktionen in Look-Up-Tables abzubilden, letztendlich kostet das Ganze doch Speicherplatz?
- 

### 6.1 Feedback von Ihrer Seite

---

Die Geschwindigkeit der Lehrveranstaltung ist

[(A)] Zu langsam [(B)] Genau richtig [(C)] Zu schnell

---

Ich ...

[(A)] ... bin ich hier, weil ich hier sein muss. [(B)] ... sehe durchaus die Anwendbarkeit des vermittelten Inhalts, Begeisterung kommt aber keine auf [(C)] ... habe Spaß am Programmieren.

---

### 6.2 Motivation

Einführungsbeispiel

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define VALUECOUNT 17
5
6 int main(void) {
7     int a [] = {1,2,3,3,4,2,3,4,5,6,7,8,9,1,2,3,4};
8
9     // Ergebnis Histogramm
10    int hist[10] = {0,0,0,0,0,0,0,0,0,0};
11    // Ergebnis Mittelwert
12    int summe = 0;
13    // Ergebnis Standardabweichung
14    float abweichung = 0;
15    for (int i=0; i<VALUECOUNT; i++){
16        hist[a[i]]++;
17        summe += a[i];
18    }
19    float mittelwert = summe / (float)VALUECOUNT;
20    for (int i=0; i<VALUECOUNT; i++){
21        abweichung += pow((a[i]-mittelwert),2.);
22    }
23    // Ausgabe
24    for (int i=0; i<10; i++){
25        printf("%d - %d\n", i, hist[i]);
26    }
27    // Ausgabe Mittelwert
28    printf("Die Summe betraegt %d, der Mittelwert %3.1f\n", summe, mittelwert);
29    // Ausgabe Standardabweichung
30    float std = sqrt(abweichung / VALUECOUNT);
31    printf("Die Standardabweichung der Grundgesamtheit betraegt %5.2f\n", std);
32    return 0;
33 }

```

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.

## 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 Rückgabetypp 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.

## Wie würden wir das vorhergehende Beispiel umstellen?



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

```

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

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](#)

## 6.2.1 Funktionsdefinition

```

1 [Spezifizierer] 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 Rückgabedatentyp 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. Logischerweise sollten keine Funktionsnamen der Laufzeitbibliothek verwenden, wie z. B. `printf()`.

- Parameterliste - Parameter sind Variablen (oder Pointer 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.
- Spezifizierer - Hier wird die Konfiguration bestimmter Speicherklassen eröffnet. Erlaubt sind Speicherklassen `extern` und `static`, wobei `static` bei Funktionen (und globalen Variablen) bewirkt, dass auf die Funktion (Variable) nur innerhalb einer Datei zugegriffen werden kann.

Die Funktionsdefinition wird für jede Funktion genau einmal benötigt.

## 6.2.2 Beispiele für Funktionsdefinitionen

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

```
1 int printf(const char * restrict format, ...){
2     /* Anweisungsblock mit Anweisungen */
3 }
4
5 //int a = printf("Hello World\n %d", 1);
```

```
1 void printDatenSatz(struct student datensatz){
2     /* Anweisungsblock mit Anweisungen */
3 }
```

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

## 6.2.3 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 ist meistens Teil einer Anweisung, z.B. einer Zuweisung oder einer Ausgabeanweisung.

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

```

22     printf("%f", shifted);
23 }
24
25 int main(void){
26     Info();
27     float input = 8.4535;
28     printf("Eingabewert %f - Ausgabewert %d\n", input, runden(input));
29     printf("Eingabewert %f - Ausgabewert %f\n", input, rundenf(input,2));
30
31     return 0;
32 }

```

## 6.2.4 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 <stdio.h>
2
3 void foo(){
4     printf("Ausgabe");
5 }
6
7 int main(void) {
8     int i = foo();
9     return 0;
10 }

```

### Implizite Convertierungen

```

1 #include <stdio.h>
2
3 float foo(){
4     return 3.123f;
5 }
6
7 int main(void) {
8     int i = foo()
9     printf("%d\n",i);
10    return 0;
11 }

```

### Parameterübergabe ohne entsprechende Spezifikation

```

1 #include <stdio.h>
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 <stdio.h>
2
3 void foo(int index, float wert){
4     printf("Index   - Wert\n");
5     printf("%5d   - %5.1f\n\n", index, wert);
6 }
7
8 int main(void) {
9     foo(4, 6.5);
10    foo(6.5, 4);
11    return 0;
12 }

```

### 6.2.5 asdfas

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void printSizeOf(int intArray[]);
5 void printLength(int intArray[]);
6
7 int main(int argc, char* argv[])
8 {
9     int array[] = { 0, 1, 2, 3, 4, 5, 6 };
10
11     printf("sizeof of array: %d\n", (int) sizeof(array));
12     printSizeOf(array);
13
14     printf("Length of array: %d\n", (int)( sizeof(array) / sizeof(array[0]) ));
15     printLength(array);
16 }
17
18 void printSizeOf(int intArray[])
19 {
20     printf("sizeof of parameter: %d\n", (int) sizeof(intArray));
21 }
22
23 void printLength(int intArray[])
24 {
25     printf("Length of parameter: %d\n", (int)( sizeof(intArray) / sizeof(intArray[0]) ));
26 }

```

### 6.2.6 Funktionsdeklaration

```

1 #include <stdio.h>
2
3 int main(void) {

```

```

4  int i = foo();           // <- Aufruf der Funktion
5  printf("i=%d\n", i);
6  return 0;
7 }
8
9  int foo(void){           // <- Definition der Funktion
10     return 3;
11 }

```

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. Hier erfolgt eine automatische (implizite) Deklaration. Der Compiler zeigt dies aber durch ein *Warning* an.

Eine explizite Deklaration zeigt folgendes Beispiel:

```

1  #include <stdio.h>
2
3  int foo(void);           // Explizite Einführung der Funktion foo()
4
5  int main(void) {
6      int i = foo();       // <- Aufruf der Funktion
7      printf("i=%d\n", i);
8      return 0;
9  }
10
11 int foo(void){           // <- Definition der Funktion foo()
12     return 3;
13 }

```

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 **extern** als Spezifizierer markiert.

```

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

```

### 6.2.7 Parameterübergabe und Rückgabewerte

Bisher wurden Funktionen betrachtet, die skalere 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, zwei Arten 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 <stdio.h>
2
3  // Definitionsteil
4  void doSomething(int a){
5      printf("%d a in der Schleife\n", ++a);
6  }
7
8  int main(void) {
9      int a = 5;
10     printf("%d a in main\n", a);
11     doSomething(a);
12     printf("%d a in main\n", a);
13     return 0;
14 }

```

#### call-by-reference

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

Mit Hilfe des Zeigers wird in C die “call-by-reference”- Parameterübergabe realisiert. In der Liste der formalen Parameter wird ein Zeiger eines passenden Typs definiert. Beim Funktionsaufruf wird als Argument statt Variable eine Adresse übergeben. Beachten Sie, dass für den Zugriff auf den Inhalt des Zeigers (einer Adresse) der Inhaltsoperator `*` benötigt wird.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void inkrementieren(int *variable){
5     (*variable)++;
6 }
7
8 int main(void) {
9     int a=0;
10    inkrementieren(&a);
11    printf("a = %d\n", a);
12    inkrementieren(&a);
13    printf("a = %d\n", a);
14    return EXIT_SUCCESS;
15 }
```

Die Adresse einer Variable wird mit dem Adressenoperator `&` ermittelt. Weiterhin kann an den Zeiger-Parameter eine Array-Variable übergeben werden.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double sinussatz(double *lookup_sin, int angle, double opositeSide){
6     return opositeSide*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    printf("Größe des Arrays %ld\n", sizeof(sin_values));
15    printf("Result = %lf \n", sinussatz(sin_values, 30, 20));
16    return EXIT_SUCCESS;
17 }
```

Der Vorteil der Verwendung der Zeiger 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 <stdio.h>
2 #include <stdlib.h>
3
4 void tauschen(char *anna, char *hanna){
5     char aux=*anna;
6     *anna=*hanna;
7     *hanna=aux;
8 }
9
10 int main(void) {
11     char anna='A',hanna='H';
12     printf("%c und %c\n", anna,hanna);
13     tauschen(&anna,&hanna);
14     printf("%c und %c\n", anna,hanna);;
15     return EXIT_SUCCESS;
16 }
```

16 }

## 6.2.8 Zeiger als Rückgabewerte

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int * doCalc(int *wert) {
6     int a = *wert + 5;
7     return &a;
8 }
9
10 int main(void) {
11     int b = 5;
12     printf("Irgendwas stimmt nicht %d", * doCalc(&b) );
13     return EXIT_SUCCESS;
14 }
```

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 <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
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     printf("Die Kreisfläche beträgt für d=%3.1lf[m] %3.1lf[m²] \n", wert, flaeche);
15     return EXIT_SUCCESS;
16 }
```

**Variante 2** Rückgabezeiger adressiert mit `static` bezeichnete Variable. Aber Achtung, diese Lösung funktioniert nicht bei rekursiven Aufrufen.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int* cumsum(int wert) {
5     static int sum = 0;
6     sum += wert;
7     return &sum;
8 }
9
10 int main(void) {
11     int wert = 2;
12     int *sum;
13     sum=cumsum(wert);
14     sum=cumsum(wert);
15     printf("Die Summe ist : %d\n", *sum);
16     sum=cumsum(wert);
```

```

17 printf("Die Summe ist : %d\n", *sum);
18 return EXIT_SUCCESS;
19 }

```

**Variante 3** Für den Rückgabezeiger wird der Speicherplatz mit `malloc` dynamisch angelegt (dazu später mehr).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double* kreisflaeche(double durchmesser) {
6     double *flaeche=(double*)malloc(sizeof(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     printf("Die Kreisfläche beträgt für d=%3.1lf[m] %3.1lf[m²] \n", wert, *flaeche);
16     return EXIT_SUCCESS;
17 }

```

### 6.2.9 main-Funktion

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

Definition der `main`-Funktion entsprechend dem C99-Standard:

```

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

1 int main(int argc, char *argv[]) {
2     /*Anweisungen*/
3 }

```

Mehr zu den Parameter `argc` und `argv` in einer der folgenden Vorlesungen.

### 6.2.10 inline-Funktionen

Der `inline`-Funktion wird das Schlüsselwort `inline` vorangestellt, z.B.:

```

1 static inline void ausgabeBruch(int z, int n) {
2     printf("%d / %d\n", z, n);
3 }

```

`inline`-Funktion wird vom Compiler direkt an der Stelle eingefügt, wo der Aufruf stattfinden soll. Gegebenenfalls ist die Ausführung der `inline`-Funktion schneller, da die mit dem Aufruf verbundenen Sicherung der Rücksprungadresse, der Sprung zur Funktion und der Rücksprung nach Ausführung entfallen. Das Schlüsselwort `inline` ist für den Compiler allerdings nur ein Hinweis und kein Befehl.

## 6.3 Lebensdauer und Sichtbarkeit von Variablen

Im Zusammenhang mit Funktionen stellt sich die Frage nach der Sichtbarkeit und der Lebensdauer einer Variablen um so mehr.

Zur Erinnerung: **globale**-Variable werden außerhalb jeder Funktionen definiert und gelten in allen Funktionen, **lokale**-Variablen gelten nur in der Funktion, in der sie definiert sind.



```

1 #include <stdio.h>
2
3 const float pi = 3.14;
4
5 float berechneUmfang(float durchmesser){
6     return durchmesser * pi;
7 }
8
9 float berechneFlaeche(float durchmesser){
10     float radius = durchmesser / 2;
11     return radius * radius * pi;
12 }
13
14 int main() {
15     char gueltig = 0;
16     float durchmesser = 23.2;
17     if (durchmesser > 0) gueltig = 1;
18     if (gueltig){
19         int linecount = 0;
20         printf("Umfang    %4.1f\n", berechneUmfang(23.2));
21         linecount++;
22         printf("Flaeche   %4.1f\n", berechneFlaeche(23.2));
23         linecount++;
24         printf("Pi kenne ich hier auch %f\n", pi);
25         linecount++;
26         printf("%d Zeilen ausgegeben\n", linecount);
27     }
28     return 0;
29 }

```

Variable	Spezifik	Bedeutung
pi	global, const	im gesamten Programm
radius	lokal	nur in <code>berechneFlaeche</code>
gueltig	lokal	nur in <code>main</code>
linecount	lokal	nur im Anweisungsblock von <code>if</code>

**\*\* Static-Variablen\*\***

static-Variablen, definiert in einer Funktion, behalten ihren Wert auch nach dem Verlassen des Funktionsblocks.

```

1 #include <stdio.h>
2
3 int zaehler(){
4     static int count = 0; //wird nur beim ersten Aufruf ausgeführt
5     return ++count;      //wird nur beim jedem Aufruf ausgeführt
6 }
7
8 int main() {
9     printf("%d \n", zaehler());
10    printf("%d \n", zaehler());
11    printf("%d \n", zaehler());
12    return 0;
13 }

```

## 6.4 Beispiel des Tages

Eine Funktion, die sich selbst aufruft, wird als rekursive Funktion bezeichnet. Den Aufruf selbst nennt man Rekursion. Als Beispiel dient die Fakultäts-Funktion  $n!$ , die sich rekursiv als  $n(n-1)!$  definieren lässt (wobei

$0! = 1$ ).

```
1 #include <stdio.h>
2
3 int fakultaet (int a){
4     if (a == 0)
5         return 1;
6     else
7         return (a * fakultaet(a-1));
8 }
9
10 int main(){
11     int eingabe;
12     printf("Ganze Zahl eingeben: ");
13     scanf("%d",&eingabe);
14     printf("Fakultaet der Zahl: %d\n",fakultaet(eingabe));
15     return 0;
16 }
```

# Kapitel 7

## Zusammengesetzte Datentypen

---

Parameter    Kursinformationen

---

**Veranstaltung:** Vorlesung Prozedurale Programmierung

**Semester:** Wintersemester 2021/22

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Verwendung von enums und structs in C-Programmen

**Link auf** [https://github.com/TUBAF-Iff-LiaScript/VL\\_ProzeduraleProgrammierung/blob/master/](https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/Reposi-06_ZusammengesetzteDatentypen.md)

**Reposi-  
tory:** 06\_ZusammengesetzteDatentypen.md

**Autoren**    @author

---

---

### Fragen an die heutige Veranstaltung ...

- Was sind *Arrays*, **structs** und **enums**?
  - Erklären Sie den Unterschied zwischen Initialisierung und Zuweisung von Variablen!
  - Wie vergleichen Sie zwei **structs**?
- 

### Wie weit waren wir gekommen?

```
1 #include <stdio.h>
2
3 void get10Values(int max, int * array){
4     for (int i=0; i<10; i++){
5         array[i] = rand() % max;
6     }
7 }
8
9 void count(int * array, int * o, int * e){
10    for (int i=0; i<10; i++){
11        switch(array[i]%2)
12        {
13            case 0 :
14                (*o)++;
15                break;
16            case 1 :
17                (*e)++;
18                break;
19        }
20    }
21 }
22
```

```

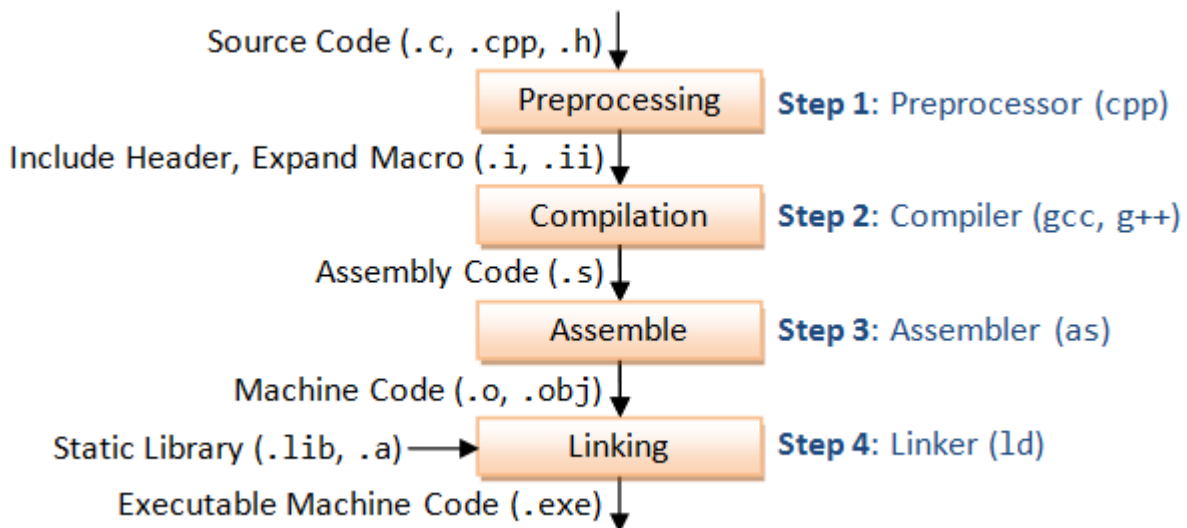
23 int main(void) {
24     int values [] = {0,0,0,0,0,0,0,0,0,0};
25     get10Values(200, values);
26     for (int i=0; i<10; i++){
27         printf("%d %4d\n", i, values[i]);
28     }
29     int o = 0, e = 0;
30     count(values, &o, &e);
31     printf("\nResults %d, %d", o, e);
32     return 0;
33 }

```

## 7.1 Exkurs: Präcompiler Direktiven

Ein Präprozessor (seltener auch Präcompiler) ist ein Computerprogramm, das Eingabedaten vorbereitet und zur weiteren Bearbeitung an ein anderes Programm weitergibt. Der Präprozessor wird häufig von Compilern oder Interpretern dazu verwendet, einen Eingabetext zu konvertieren und das Ergebnis im eigentlichen Programm weiter zu verarbeiten.

Da sich der C-Präprozessor nicht auf die Beschreibung der Sprache C stützt, sondern ausschließlich seine ihm bekannten Anweisungen erkennt und bearbeitet, kann er auch als reiner Textersetzer für andere Zwecke verwendet werden.



Der C-Präprozessor realisiert dabei die

- Zusammenfassung von Strings
- Löschung von Zeilenumbrüchen und Kommentaren (Ersetzung durch Leerzeichen)
- Whitespace-Zeichen zwischen Tokens werden gelöscht.
- Kopieren der Header- und Quelldateien in den Quelltext kopieren (`#include`)
- Einbinden von Konstanten (`#define`)
- Extrahieren von Codebereichen mit einer bedingten Kompilierung (`#ifdef`, `#elseif`, ...)

Letztgenannte 3 Abläufe werden durch den Entwickler spezifiziert. Dazu bedient er sich sogenannter Direktiven. Sie beginnen mit `#` und müssen nicht mit einem Semikolon abgeschlossen werden. Eventuell vorkommende Sonderzeichen in den Parametern müssen nicht escaped werden.

```
1 #Direktive Parameter
```

### 7.1.1 `#include`

Include-Direktiven kennen Sie bereits aus unseren Beispielprogrammen. Damit binden wir Standardbibliotheken oder eigenen Source-Datei ein. Es gibt zwei Arten der `#include`-Direktive, nämlich

```

1 #include <Datei.h>
2 #include "Datei.h"

```

Die erste Anweisung sucht die Datei im Standard-Include-Verzeichnis des Compilers, die zweite Anweisung sucht die Datei zuerst im Verzeichnis, in der sich die aktuelle Sourcedatei befindet; sollte dort keine Datei mit diesem Namen vorhanden sein, sucht sie ebenfalls im Standard-Include-Verzeichnis.

Mit dem Präprozessoraufbau werden die Inhalte der Header-Files in unseren Code kopiert. Dieser wird dadurch um ein vielfaches größer, umfasst nun aber alle Funktionsdeklarationen, die genutzt werden sollen.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     printf("Präprozessorkrams \n")
6     return EXIT_SUCCESS;
7 }

1
2 gcc experiments.c -E -o experiments_pre.txt

1 ...
2 # 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
3 # 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
4 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
5 # 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
6
7 typedef unsigned char __u_char;
8 typedef unsigned short int __u_short;
9 typedef unsigned int __u_int;
10 typedef unsigned long int __u_long;
11 ...
12
13 # 4 "experiments.c"
14 int main(void) {
15     printf("Präprozessorkrams \n")
16     return
17 # 6 "experiments.c" 3 4
18     0
19 # 6 "experiments.c"
20     ;
21 }
```

Warum muss vermieden werden, dass headerfiles kreuzweise eingebunden werden?

### 7.1.2 #define

#define kann in drei verschiedenen Arten genutzt werden, um ein Symbol überhaupt zu definieren, einen konkreten Wert zuzuordnen oder aber

```

1 #define SYMBOL
2 #define KONSTANTE Wert
3 #define MAKRO(Parameter ...) Ausdruck

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MATH_PI 3.14
5 #define VOLLKREIS MATH_PI*2
6 #define HALBIEREN(wert) ((wert) / 2)
7 #define MAX_VALUE(a,b) ((a >= b) ? (a) : (b))
8
9 int main(void)
10 {
11     float r=5;
12     printf("Kreisfläche %f\n", r * r * MATH_PI);
```

```

13 printf("Kreisumfang %f\n", r * VOLLKREIS);
14 printf("Halber Wert von Pi %f\n", HALBIEREN(MATH_PI));
15 printf("Vergleich %f\n", MAX_VALUE(MATH_PI, r));
16 return EXIT_SUCCESS;
17 }

```

In allen Fällen erfolgt lediglich eine Textersetzung im Programmcode! Dies kann auch auf weitergehende Codefragmente ausgedehnt werden.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define TAUSCHE(a, b, typ) { typ temp; temp=b; b=a; a=temp; }
5
6 int main(void) {
7     int zahla=4, zahlb=7;
8     printf("zahl A: %d\nzahl B: %d\n", zahla, zahlb);
9     TAUSCHE(zahla, zahlb, int);
10    printf("zahl A: %d\nzahl B: %d\n", zahla, zahlb);
11    return EXIT_SUCCESS;
12 }

```

Im Standard-C müssen bereits einige Makros im Präprozessor vordefiniert sein. Die Namen der vordefinierten Makros beginnen und enden jeweils mit zwei Unterstrichen. Die wichtigsten vordefinierten Makros sind in der folgenden Tabelle aufgelistet.

Define	Bedeutung
__LINE__	Zeilennummer innerhalb der aktuellen Quellcodedatei
__FILE__	Name der aktuellen Quellcodedatei
__DATE__	Datum, wann das Programm compiliert wurde (als Zeichenkette)
__STDC__	Liefert eine 1, wenn sich der Compiler nach dem Standard-C richtet.
__STDC_VERSION__	Liefert die Zahl 199409L, wenn sich der Compiler nach dem C95-Standard richtet; die Zahl 199901L, wenn sich der Compiler nach dem C99-Standard richtet. Ansonsten ist dieses Makro nicht definiert.

Daneben gibt es weitere vordefinierte Makros, die das Betriebssystem zurückgeben.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define TAUSCHE(a, b, typ) { typ temp; temp=b; b=a; a=temp; }
5
6 int main(void) {
7     printf("Programm wurde compiliert am ");
8     printf("%s um %s.\n", __DATE__, __TIME__);
9
10    printf("Diese Programmzeile steht in Zeile ");
11    printf("%d in der Datei %s.\n", __LINE__, __FILE__);
12
13    #ifdef __STDC__
14        printf("Standard-C-Compiler!\n");
15    #else
16        printf("Kein Standard-C-Compiler!\n");
17    #endif
18    return EXIT_SUCCESS;
19 }

```

## 7.2 Aufzählungen

Enumerationen, kurz `enum`, dienen der Definition bestimmter Sets von Elementen, die eine Variable überhaupt annehmen kann. Wenn wir zum Beispiel die Farben einer Ampel in einem Programm handhaben wollen, sind dies lediglich “Rot”, “Gelb” und “Grün”. Im Schachspiel sind nur die Figuren “Bauer”, “Pferd”, “Springer”, “Turm”, “Dame” und “König” definiert.

Nun, dass können wir doch wunderbar mit unseren Präprozessordirektiven umsetzen!

```
1 #include <stdio.h>
2
3 #define MONDAY    1
4 #define TUESDAY   2
5 #define WEDNESDAY 3
6 #define THURSDAY  4
7 #define FRIDAY    5
8 #define SATURDAY  6
9 #define SUNDAY    7
10
11 int main(void) {
12     int day = SATURDAY;
13     printf("Wochentag: %d\n", day);
14     return 0;
15 }
```

Allerdings ist das Ganze recht unflexibel und der Evaluation des Compilers entzogen.

Die Definition eines Aufzählungsdatentyps `enum` hat die Form wie im folgenden Beispiel:

```
1 #include <stdio.h>
2
3 enum card {KARO, HERZ, PIK, KREUZ};           // Beispiel der Farben beim Skat
4 // const int kreuz=0, pik=1, karo=2, herz=3;    // Analoge Anweisung
5
6 int main(void) {
7     enum card karte = KARO;
8     printf("Wert der Karte: %d\n", karte);
9     return 0;
10 }
```

Möglicherweise sollen den Karten aber auch konkrete Werte zugeordnet werden, die bestimmte Wertigkeiten reflektieren.

```
1 #include <stdio.h>
2
3 // Beispiel der Farben beim Skat
4 enum card {KARO=9, HERZ=10, PIK=11, KREUZ=12};
5
6 int main(void) {
7     enum card karte = KARO;
8     printf("Wert der Karte: %d\n", karte);
9     return 0;
10 }
```

An dieser Stelle sind Sie aber frei, was die eigentlichen Werte angeht. Es sind zum Beispiel Konfigurationen möglich wie

```
1 enum { KARO=9, HERZ=10, PIK=11, KREUZ=12};
2 enum { KARO=9, HERZ, PIK, KREUZ};           // Gleiches Resultat
3 enum { KARO=9, HERZ, PIK=123, KREUZ};       // implizit kreuz = 124

1 #include <stdio.h>
2
3 enum textformat
```

```

4 {
5     RED = 1,
6     BOLD = 2,
7     ITALIC = 4,
8     EXCLAMATION = 8
9 };
10
11 void printColor(const char text[], enum textformat chosenFormat)
12 {
13     if (chosenFormat & RED){
14         printf("Rot - ");
15     }
16     if (chosenFormat & BOLD){
17         printf("Fett - ");
18     }
19     if (chosenFormat & ITALIC){
20         printf("Kursiv -");
21     }
22     printf("%s", text);
23     if (chosenFormat & EXCLAMATION)
24     {
25         printf(" !!!\n");
26     }
27 }
28
29 int main(void) {
30     const char text[] = "Bergakademie Freiberg";
31     enum textformat chosenFormat = RED | EXCLAMATION;
32     printColor(text, chosenFormat);
33     return 0;
34 }

```

### 7.3 Typdefinition mit typedef

Mit Hilfe des Schlüsselworts `typedef` kann für einen Datentyp, einschließlich eines `struct`-Datentyps, ein neuer Bezeichner definiert werden:

```
1 typedef Typendefinition Bezeichner;
```

Zum Beispiel kann der Datentyp `String` definiert und zur einfacheren Variablendeklaration verwendet werden:

```

1 typedef char* string;
2 string s = "Hallo";

```

**Achtung!** Hinter `string` verbirgt sich noch immer ein `char *` und nicht etwa ein “echter” String-Datentyp.

Auf ein `enum` angewandt vermeidet man damit die Wiederholung des Keywords `enum`

```

1 enum Day { monday=1, tuesday, ...};
2 void printDay(enum Day day);
3
4 // mit typedef
5 enum day_enum { monday=1, tuesday,...};
6 typedef enum day_enum Day;
7 void printDay(Day day);
8 // Alternativ in einem Rutsch
9 typedef enum { monday=1, tuesday,...} Day;

```

Und konkret am Beispiel:

```
1 #include <stdio.h>
```



```

2
3 enum textformat
4 {
5     RED = 1,
6     BOLD = 2,
7     ITALIC = 4,
8     EXCLAMATION = 8
9 };
10
11 typedef enum textformat format;
12
13 void printColor(const char text[], format chosenFormat)
14 {
15     if (chosenFormat & RED){
16         printf("Rot - ");
17     }
18     if (chosenFormat & BOLD){
19         printf("Fett - ");
20     }
21     if (chosenFormat & ITALIC){
22         printf("Kursiv -");
23     }
24     printf("%s", text);
25     if (chosenFormat & EXCLAMATION)
26     {
27         printf(" !!!\n");
28     }
29 }
30
31 int main(void) {
32     const char text[] = "Bergakademie Freiberg";
33     format chosenFormat = RED | EXCLAMATION;
34     printColor(text, chosenFormat);
35     return 0;
36 }

```

## 7.4 Strukturen

Mit `struct` werden zusammengehörige Variablen unterschiedlicher Datentypen und Bedeutung in einem Konstrukt zusammengefasst. Damit wird für den Entwickler der Zusammenhang deutlich. Die Variablen der Struktur werden als Komponenten (engl. members) bezeichnet.

Beispiele:

```

1 struct datum
2 {
3     int tag;
4     char monat[10];
5     int jahr;
6 };
7
8
9 struct Student
10 {
11     int Matrikel;
12     char name[20];
13     char vorname[25];
14 }; // <- Hier steht ein Semikolon!

```

### 7.4.1 Deklaration, Definition, Initialisierung und Zugriff

Und wie erzeuge ich Variablen dieses erweiterten Types, wie werden diese initialisiert und wie kann ich auf die einzelnen Komponenten zugreifen?

Der Beispiel zeigt, dass die Definition der Variable unmittelbar nach der **struct**-Definition oder mit einer gesonderten Anweisung mit einer vollständigen, partiellen Initialisierung bzw. ohne Initialisierung erfolgen kann.

Die nachträgliche Veränderung einzelner Komponenten ist über Zugriff mit Hilfe des Punkt-Operators möglich.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     struct datum
6     {
7         int tag;
8         char monat[10];
9         int jahr;
10 } geburtstag_1 = {18, "April", 1986};           // <- Initialisierung
11                                           // Variable geburtstag_1
12 struct datum geburtstag_2 = {13, "Januar", 2013}; // <- Initialisierung
13                                           // Variable geburtstag_2
14 geburtstag_2.tag = 13;                       // Zuweisungen
15 geburtstag_2.jahr = 1803;
16 strcpy(geburtstag_2.monat, "April");
17
18 // Unvollstaendige Initialisierung
19 struct datum geburtstag_3 = {.monat = "September"}; // <- partielle Initialisierung
20
21 printf("Person A wurde am %2d. %-10s %4d geboren.\n", geburtstag_1.tag, geburtstag_1.monat,
22        geburtstag_1.jahr);
23 printf("Person B wurde am %2d. %-10s %4d geboren.\n", geburtstag_2.tag, geburtstag_2.monat,
24        geburtstag_2.jahr);
25 printf("Person C wurde am %2d. %-10s %4d geboren.\n", geburtstag_3.tag, geburtstag_3.monat,
26        geburtstag_3.jahr);
27 return 0;
28 }
```

### 7.4.2 Vergleich von struct-Variablen

Der C-Standard kennt keine Methodik um **structs** in einem Rutsch auf Gleichheit zu prüfen. Entsprechend ist es an jedem Entwickler eine eigene Funktion dafür zu schreiben. Diese kann unterschiedliche Aspekte des **structs** adressieren.

Im nachfolgenden Beispiel werden zur Überprüfung der Gleichheit die **tag** und **monat** verglichen. Der Vergleich wird dadurch vereinfacht, dass wir für die Repräsentation des Monats ein **enum** verwenden. Damit entfällt aufwändigerer Vergleich der Strings.

```

1 #include <stdio.h>
2
3 enum MONAT {Januar, Februar, Maerz, April, Mai};
4
5 struct datum
6 {
7     int tag;
8     enum MONAT monat;
9     int jahr;
10 };
11
12 int main() {
13     struct datum person_1 = {10, Januar, 2013};
14     struct datum person_2 = {10, Maerz, 1956};
```

```

15     person_2.monat = Januar;
16
17     if ((person_1.tag == person_2.tag) && (person_1.monat == person_2.monat))
18         printf("Oha, der gleiche Geburtstag im Jahr!\n");
19     else
20         printf("Ungleiche Geburtstage!\n");
21 }

```

### 7.4.3 Structs und Funktionen

## Wie erfolgt die Übergabe von structs an Funktionen?

```

1 #include <stdio.h>
2
3 enum MONAT {Januar, Februar, Maerz, April, Mai};
4
5 struct datum
6 {
7     int tag;
8     enum MONAT monat;
9     int jahr;
10 };
11
12 int vergleich(????){
13
14 }
15
16 int main() {
17     struct datum person_1 = {10, Januar, 2013};
18     struct datum person_2 = {10, Maerz, 1956};
19     if (vergleich(person_1, person_2)){
20         printf("Person 1 ist nach Person 2 geboren");
21     }
22 }

```

#### 7.4.4 Arrays von Strukturen

Natürlich lassen sich die beiden erweiterten Datenformate auf der Basis von `struct` und Arrays miteinander kombinieren.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define ENTRIES 3
5
6 int main() {
7
8     enum {Jan, Feb, Maerz, April, Mai, Juni, Juli, Aug, Sept, Okt, Nov, Dez};
9
10    char months[12][20] = {"Januar", "Februar", "Maerz", "April",
11                           "Mai", "Juni", "Juli", "August", "September",
12                           "Oktober", "November", "Dezember"};
13
14    struct datum
15    {
16        unsigned char tag;        // wert < 31
17        unsigned char monat;      // wert < 12
18        short jahr;               // wert < 2048
19    };
20
21    {12, Mai, 1820}};

```

```

22
23 geburtstage[2].tag = 5;
24 geburtstage[2].monat = Sept;
25 geburtstage[2].jahr = 1905;
26
27 for (int i=0; i<ENTRIES; i++)
28     printf("%2d. %-10s %4d\n", geburtstage[i].tag,
29                               months[geburtstage[i].monat],
30                               geburtstage[i].jahr);
31
32 return 0;
33 }

```

## 7.5 Beispiel des Tages

Generieren Sie mit einem C-Programm Dateien im Excelformat, um diese dann weiter verarbeiten zu können. Dazu nutzen wir die Bibliothek `libxlsxwriter`, deren Dokumentation und Quellcode Sie unter

<https://libxlsxwriter.github.io/index.html>

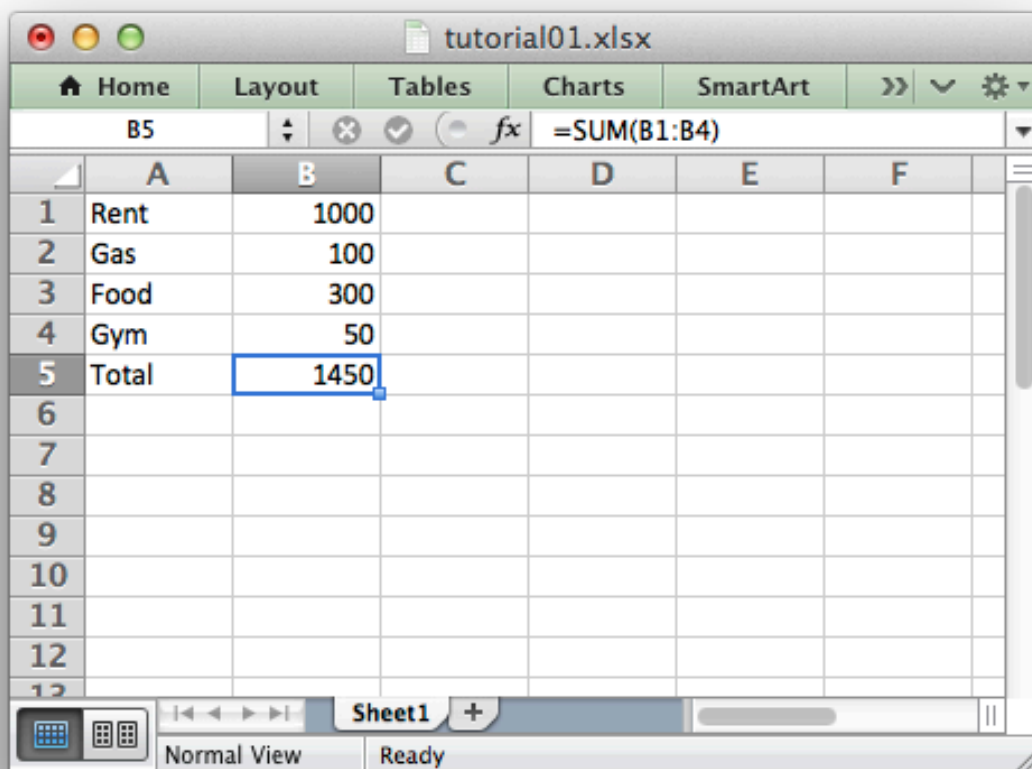
finden. Die Dokumentation für die im Beispiel verwendeten Funktionen findet sich dann für `lxw_workbook` auf der Seite

[https://libxlsxwriter.github.io/workbook\\_8h.html](https://libxlsxwriter.github.io/workbook_8h.html)

```

1 #include "xlsxwriter.h"
2 /* Some data we want to write to the worksheet. */
3 struct expense {
4     char item[32];
5     int cost;
6 };
7 struct expense expenses[] = {
8     {"Rent", 1000},
9     {"Gas", 100},
10    {"Food", 300},
11    {"Gym", 50},
12 };
13 int main() {
14     /* Create a workbook and add a worksheet. */
15     lxw_workbook *workbook = workbook_new("tutorial01.xlsx");
16     lxw_worksheet *worksheet = workbook_add_worksheet(workbook, NULL);
17     /* Start from the first cell. Rows and columns are zero indexed. */
18     int row = 0;
19     int col = 0;
20     /* Iterate over the data and write it out element by element. */
21     for (row = 0; row < 4; row++) {
22         worksheet_write_string(worksheet, row, col, expenses[row].item, NULL);
23         worksheet_write_number(worksheet, row, col + 1, expenses[row].cost, NULL);
24     }
25     /* Write a total using a formula. */
26     worksheet_write_string(worksheet, row, col, "Total", NULL);
27     worksheet_write_formula(worksheet, row, col + 1, "=SUM(B1:B4)", NULL);
28     /* Save the workbook and free any allocated memory. */
29     return workbook_close(workbook);
30 }

```



The screenshot shows a Microsoft Excel window titled "tutorial01.xlsx". The ribbon at the top includes "Home", "Layout", "Tables", "Charts", and "SmartArt". The active cell is B5, and the formula bar displays the formula `=SUM(B1:B4)`. The spreadsheet contains the following data:

	A	B	C	D	E	F
1	Rent	1000				
2	Gas	100				
3	Food	300				
4	Gym	50				
5	Total	1450				
6						
7						
8						
9						
10						
11						
12						
13						

The status bar at the bottom indicates "Normal View" and "Ready".

Abbildung 7.1: Resultat



# Kapitel 8

## Standardalgorithmen in C

---

Parameter    Kursinformationen

---

**Veranstaltung:** Vorlesung Prozedurale Programmierung

**Semester:** Wintersemester 2021/22

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Realisierung von Standardalgorithmen in C

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

**Repository:** [LiaScript/VL\\_ProzeduraleProgrammierung/blob/master/07\\_Algorithmen.md](https://github.com/TUBAF-Iff-LiaScript/VL_ProzeduraleProgrammierung/blob/master/07_Algorithmen.md)

**Autoren**    @author

---

---

### Fragen an die heutige Veranstaltung ...

- Was ist ein Algorithmus und über welche Merkmale lässt er sich ausdrücken.
- Nennen Sie Beispiele für Algorithmen aus dem täglichen Leben.
- Wie erfolgt die Transformation des Algorithmus auf eine Programmiersprache?
- Was bedeutet der Begriff der Komplexität eines Algorithmus?
- Welchem fundamentalen Konzept der Informatik unterliegen der Quicksort Algorithmus und die binäre Suche?

---

### Wie weit waren wir gekommen?

Das folgende Beispiel wiederholt die Verwendung von `typedef` und `struct` innerhalb eines Arrays für eine Ampelsteuerung.

```
1      .-- 3s --. .-- 1s --. .-- 3s --.
2      |         | |         | |         |
3      |         v |         v |         v
4      .-.       .-.       .-.       .-.
5 Ampelzustände ( 0 )     ( 1 )     ( 2 )     ( 3 )
6      '- '      '- '      '- '      '- '
7      ^         |         |         |
8      |         |         |         |
9      .----- 1s ----- .
10
11      RED  RED/YELLOW  GREEN  YELLOW
```

```
1 typedef struct {
2     int state;
3     int next;
4     int A_red;
```

```

5   int A_yellow;
6   int A_green;
7   int timer;
8 } ampel_state_t;
9
10 ampel_state_t state_table[4] = {
11
12 // state      A_red          timer
13 // |  next  | A_yellow      |
14 // |  |    | | A_green    |
15 //-----
16 { 0, 1, 1, 0, 0, 3},
17 { 1, 2, 1, 1, 0, 1 },
18 { 2, 3, 0, 0, 1, 3},
19 { 3, 0, 0, 1, 0, 1},
20 };
21
22 const int greenPin = 13;
23 const int yellowPin = 12;
24 const int redPin = 11;
25 int state = 0;
26
27 void setup() {
28   pinMode(greenPin, OUTPUT);
29   pinMode(yellowPin, OUTPUT);
30   pinMode(redPin, OUTPUT);
31 }
32
33 void loop() {
34   if (state_table[state].A_red == 1) digitalWrite(redPin, HIGH);
35   else digitalWrite(redPin, LOW);
36   if (state_table[state].A_yellow == 1) digitalWrite(yellowPin, HIGH);
37   else digitalWrite(yellowPin, LOW);
38   if (state_table[state].A_green == 1) digitalWrite(greenPin, HIGH);
39   else digitalWrite(greenPin, LOW);
40   delay(state_table[state].timer*1000);
41   state = state_table[state].next;
42 }

```

@AVR8js.sketch

## 8.1 Rekursion

Die Rekursion ist ein Aufruf einer Funktionen aus sich selbst heraus. Da bei einem Aufruf sich die Funktion wieder selbst aufruft, benötigt die Funktion wie bei den Schleifen eine Abbruchbedingung, damit die Selbstaufrufe nicht endlos sind.

```

1 #include<stdio.h>
2
3 printLines(int x) {
4   if(x > 0) {
5     printf("\nZeile Nr. %d", x);
6     printLines(x-1);
7   }
8 }
9
10 int main() {
11   printLines(5);
12   return 0;
13 }

```



Dieses Programm ist langsamer als eine konventionelle Darstellung in einer Schleife, weil mit dem Aufruf jeder Funktion ein eigener Speicherplatz zum Anlegen von Parametern, lokalen Variablen, Rückgabewerten und Rücksprungadressen belegt wird.

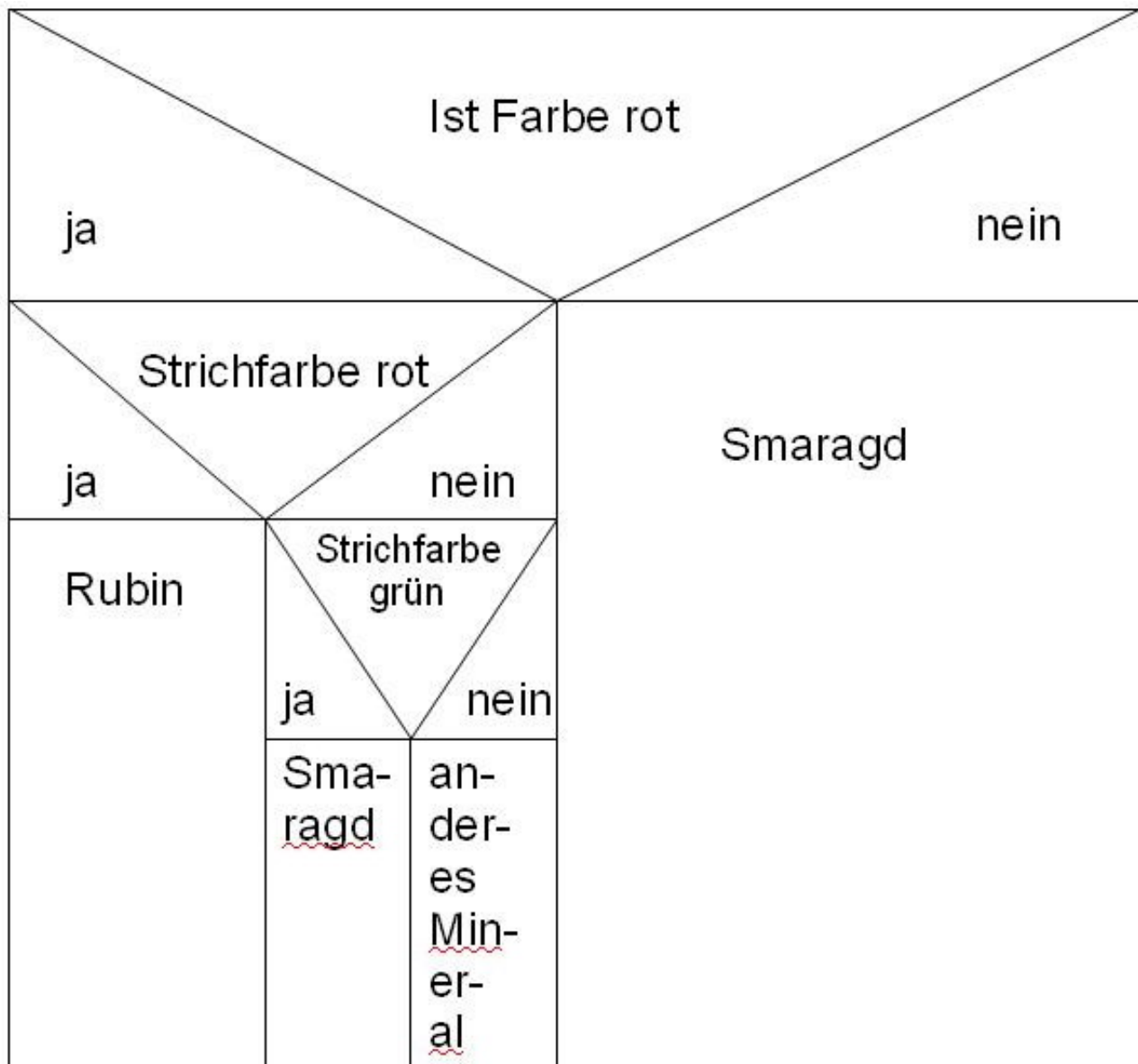
Gleichzeitig steigt aber die Lesbarkeit und Kompaktheit des Codes!

```
1 #include<stdio.h>
2
3 int fakultaet(int x) {
4     if(x > 1) {
5         return x * fakultaet(x-1);
6     }else {
7         return 1;
8     }
9 }
10
11 int main() {
12     int a = 6;
13     printf("Fakultaet von %d ist %d\n", a, fakultaet(a));
14     return 0;
15 }
```

## 8.2 Algorithmusbegriff

Ein Algorithmus gibt eine strukturierte Vorgehensweise vor, um ein Problem zu lösen. Er implementiert Einzelschritte zur Abbildung von Eingabedaten auf Ausgabedaten. Algorithmen bilden die Grundlage der Programmierung und sind **unabhängig** von einer konkreten Programmiersprache. Algorithmen werden nicht nur maschinell durch einen Rechner ausgeführt sondern können auch von Menschen in „natürlicher“ Sprache formuliert und abgearbeitet werden.

### 1. Beispiel - Nassi-Shneiderman-Diagramm



## 2. Beispiel - Funktionsdarstellung - Berechnung der Position

$$s(t) = \int_0^t v(t)dt + s_0$$

## 3. Beispiel - Verbale Darstellung - Rezept

*“Nehmen Sie ... Schneiden Sie ... Lassen Sie alles gut abkühlen ...”*

Algorithmen umfassen Sequenzen (Kompositionen), Wiederholungen (Iterationen) und Verzweigungen (Selektionen) von Handlungsanweisungen. und besitzen die folgenden charakteristischen Eigenschaften:

- Eindeutigkeit: ein Algorithmus darf keine widersprüchliche Beschreibung haben. Diese muss eindeutig sein.
- Ausführbarkeit: jeder Einzelschritt muss ausführbar sein.
- Finitheit (= Endlichkeit): die Beschreibung des Algorithmus ist von endlicher Länge (statische Finitheit) und belegt zu jedem Zeitpunkt nur eine endliche Menge von Ressourcen (dynamische Finitheit).
- Terminierung: nach endlich vielen Schritten muss der Algorithmus enden und ein Ergebnis liefern.
- Determiniertheit: der Algorithmus muss bei gleichen Voraussetzungen stets das gleiche Ergebnis liefern.
- Determinismus: zu jedem Zeitpunkt der Ausführung besteht höchstens eine Möglichkeit der Fortsetzung. Der Folgeschritt ist also eindeutig bestimmt.

Der erste für einen Computer gedachte Algorithmus (zur Berechnung von Bernoullizahlen) wurde 1843(!) von Ada Lovelace in ihren Notizen zu Charles Babbages Analytical Engine festgehalten. Sie gilt deshalb als die erste Programmiererin. Weil Charles Babbage seine Analytical Engine nicht vollenden konnte, wurde Ada Lovelaces Algorithmus allerdings nie darauf implementiert.

„Die Grenzen der Arithmetik wurden in dem Augenblick überschritten, in dem die Idee zur Verwendung der [Programmier]Karten entstand, und die Analytical Engine hat keine Gemeinsamkeit mit schlichten Rechenmaschinen. Sie ist einmalig, und die Möglichkeiten, die sie andeutet, sind höchst interessant.“

## 8.3 Suche des Maximums

Bestimmen Sie aus drei Zahlenwerten den größten und geben Sie diesen aus  $\max(n_0, n_1, n_2)$ .

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     double n1, n2, n3;
6
7     printf("Geben Sie drei Zahlenwerte ein: \n");
8     scanf("%lf %lf %lf", &n1, &n2, &n3);
9     printf("Eingegebene Zahlen %f %f %f \n", n1, n2, n3);
10
11     if( n1>=n2 && n1>=n3 )
12         printf("%f is the largest number.", n1);
13
14     if( n2>=n1 && n2>=n3 )
15         printf("%f is the largest number.", n2);
16
17     if( n3>=n1 && n3>=n2 )
18         printf("%f is the largest number.", n3);
19
20     return EXIT_SUCCESS;
21 }
```

Welche Verbesserungsmöglichkeit sehen Sie für diesen Lösungsansatz?

Aspekt	Kritik
Userinterface	Es erfolgt keine Prüfung der Eingaben!
Design	Die Ausgabe erfolgt in 3 sehr ähnlichen Aufrufen.
Algorithmus	Es werden 6 Vergleichsoperationen und 3 logische Operationen genutzt.
Wiederverwendbarkeit	Die Funktion implementiert die Suche für genau 3 Eingaben.

Eine Lösung, die die ersten 3 genannten Kritikpunkte adressiert, könnte wie folgt entworfen werden:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     double n1, n2, n3;
6     double result = 0;
7
8     printf("Geben Sie drei Zahlenwerte ein: \n");
9     if (scanf("%lf %lf %lf", &n1, &n2, &n3) == 3){
10         printf("Eingegebene Zahlen %f %f %f \n", n1, n2, n3);
11         if( n1>=n2 && n1>=n3 ){
12             result = n1;
13         }
14         else{
15             if( n2>=n3){result = n2;}
16             else {result = n3;}
17         }
18         printf("Größter Wert ist %.1f\n", result);
19     }else{
20         printf("Ungültige Eingabe!");
21     }
```

```

21 }
22 return EXIT_SUCCESS;
23 }

```

Ein alternativer Ansatz kann mit Hilfe von Makrooperationen umgesetzt werden, die eine MAX-Methode implementieren.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX(x, y) (((x) > (y)) ? (x) : (y))
5
6 int main(void) {
7     double n1, n2, n3;
8     double result = 0;
9
10    printf("Geben Sie drei Zahlenwerte ein: \n");
11    if (scanf("%lf %lf %lf", &n1, &n2, &n3) == 3){
12        printf("Eingegebene Zahlen %f %f %f \n", n1, n2, n3);
13        result = MAX(MAX(n1, n2), n3);
14        printf("Größter Wert ist %.1f\n", result);
15    }else{
16        printf("Ungültige Eingabe!");
17    }
18    return EXIT_SUCCESS;
19 }

```

Darfs auch etwas mehr sein? Wie lösen wir die gleiche Aufgabe für größere Mengen von Zahlenwerten  $\max(n_0, \dots, n_k)$ ? Entwerfen Sie dazu folgende Funktionen:

- void generateRandomArray(int \* ptr)
- int countMaxValue(int \*ptr, int n\_samples)

die zunächst gleichverteilte Werte zwischen MAXVALUE und MINVALUE befüllt und dann die Häufigkeit des größten Wertes ermittelt.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #define MAXVALUE 100
6 #define MINVALUE 5
7 #define SAMPLES 50
8
9 void generateRandomArray(int * ptr){
10    srand(time(NULL));
11    for (int i = 0; i < SAMPLES; i++){
12        ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
13    }
14 }
15
16 int maxValue(int *ptr){
17     int max = 0;
18     for (int i = 0; i < SAMPLES; i++){
19         if (ptr[i] > max) {
20             max = ptr[i];
21         }
22     }
23     return max;
24 }
25
26 void printArray(int *ptr, int maxValue){
27     for (int i = 0; i < SAMPLES; i++){

```

```

28     if ((i>0) && (i%10==0)) {
29         printf("\n");
30     }
31     if (ptr[i]!=maxValue){
32         printf(" %3d ", ptr[i]);
33     }else{
34         printf("[%3d]", ptr[i]);
35     }
36 }
37 }
38
39 int main(void){
40     int samples[SAMPLES] = {0};
41     generateRandomArray(samples);
42     int max = maxValue(samples);
43     //int count = 0;    // Aufgabenteil 2
44     //int max = maxValue(samples, &count);
45     printArray(samples, max);
46     printf("\nIm Array wurde %d als Maximum gefunden!", max);
47     return(EXIT_SUCCESS);
48 }

```

Aufgabenteil 2: Erweitern Sie die Funktionalität von `maxValue()` um die Rückgabe der Häufigkeit des Auftretens des maximalen Wertes. Ein Kommilitone schlägt folgende Lösung vor:

```

1 int maxValue(int *ptr, int *count){
2     int max = 0;
3     for (int i = 0; i< SAMPLES; i++){
4         if (ptr[i] > max) {
5             max = ptr[i];
6             *count = 1;
7         }
8         if (ptr[i] == max) {
9             (*count)++;
10        }
11    }
12    return max;
13 }

```

Bewerten Sie diese und entwerfen Sie ggf. eine alternative Implementierung.

Für alle, die mit dem Knobeln fertig sind, hier noch eine kurze Auflösung. Dadurch, dass beim Auftreten eines neuen Maximums in der ersten Verzweigung `max` dem neuen Wert zugeordnet wird, rutschen wir automatisch auch in die zweite Verzweigung, wodurch der Wert von `max` immer eins größer sein wird, als wir wollen. Eine Lösung von vielen für diese Problematik ist folgende:

```

1 int maxValue(int *ptr, int *count){
2     int max = 0;
3     for (int i = 0; i< SAMPLES; i++){
4         if (ptr[i] > max) {
5             max = ptr[i];
6             *count = 1;
7         }
8         else if (ptr[i] == max) {
9             (*count)++;
10        }
11    }
12    return max;
13 }

```

## 8.4 Sortieren

Lassen Sie uns die Idee der Max-Funktion nutzen, um das Array insgesamt zu sortieren. Dazu wird in einer Schleife (Zeile 42) der maximale Wert bestimmt, wobei dessen Eintrag aus dem bestehenden Array mit einer -1 überschrieben wird.

Welche Nachteile sehen Sie in diesem Konzept?

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5
6 #define MAXVALUE 100
7 #define MINVALUE 5
8 #define SAMPLES 1000
9
10 void generateRandomArray(int * ptr){
11     srand(time(NULL));
12     for (int i = 0; i< SAMPLES; i++){
13         ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
14     }
15 }
16
17 int maxValue(int *ptr){
18     int max = 0;
19     int max_index = 0;
20     for (int i = 0; i< SAMPLES; i++){
21         if (ptr[i] > max) {
22             max = ptr[i];
23             max_index = i;
24         }
25     }
26     ptr[max_index] = -1;
27     return max;
28 }
29
30 void printArray(int *ptr){
31     for (int i = 0; i< SAMPLES; i++){
32         if ((i>0) && (i%10==0)) {
33             printf("\n");
34         }
35         printf(" %3d ", ptr[i]);
36     }
37 }
38
39 int main(void){
40     int samples[SAMPLES] = {0};
41     generateRandomArray(samples);
42     clock_t start = clock();
43     int sorted[SAMPLES] = {0};
44     for (int i = 0; i< SAMPLES; i++){
45         sorted[i] = maxValue(samples);
46     }
47     clock_t end = clock();
48     printArray(sorted);
49     double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
50
51     printf("Der Rechner benötigt für %d Samples %f Sekunden \n", SAMPLES,cpu_time_used);
52
53     return(EXIT_SUCCESS);
54 }

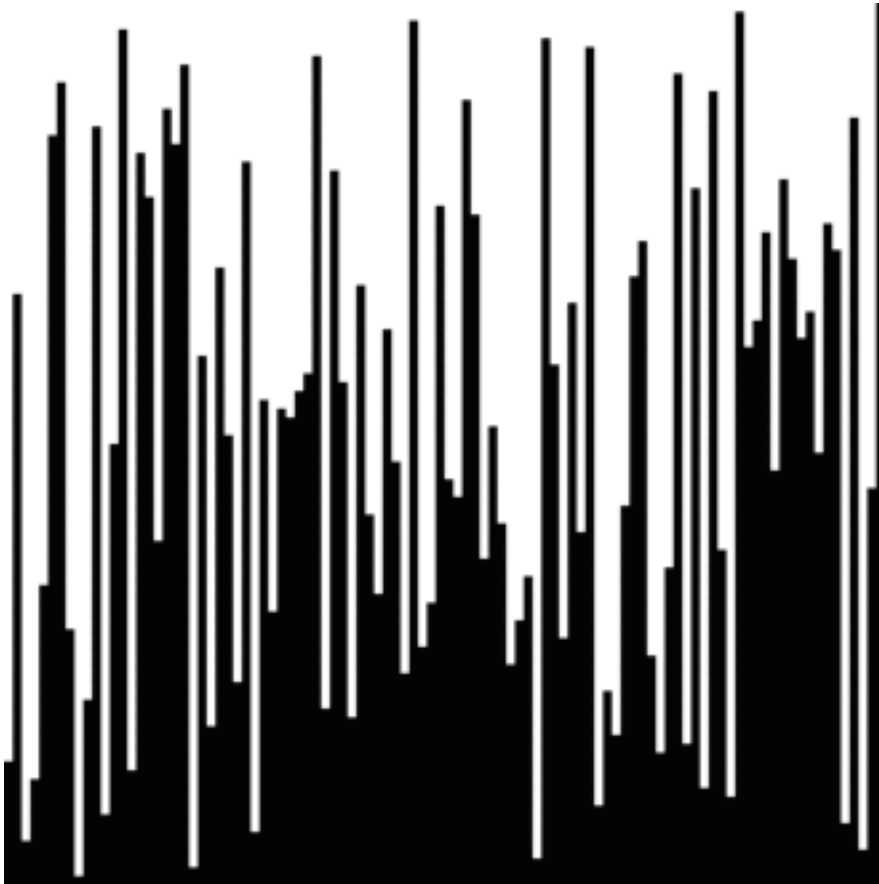
```

- Das Ursprungsarray wird beim Sortiervorgang zerstört, am Ende umfasst es ausschließlich -1-Einträge
- Die Ausführungsdauer wird durch  $\text{SAMPLES} \times \text{SAMPLES}$  Vergleichsoperationen bestimmt.

Welche Konsequenz hat dieses Verhalten?

### 8.4.1 BubbleSort

Die Informatik kennt eine Vielzahl von Sortierverfahren, die unterschiedliche Eigenschaften aufweisen. Ein sehr einfacher Ansatz ist BubbleSort, der namensgebend die größten oder kleinsten Zahlen Gasblasen gleich aufsteigen lässt.



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define MAXVALUE 100
6 #define MINVALUE 5
7 #define SAMPLES 20
8
9 void generateRandomArray(int *ptr) {
10     srand(time(NULL));
11     for (int i = 0; i < SAMPLES; i++){
12         ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
13     }
14 }
15
16 void bubble(int *array) {
17     int n_samples = SAMPLES;
18     int temp;
19     while(n_samples--){
20         for(int i = 1; i <= n_samples; i++){
21             if(array[i-1] > array[i]) {
22                 temp=array[i];
23                 array[i]=array[i-1];

```

```

24     array[i-1]=temp;
25 }
26 }
27 }
28 }
29
30 void printArray(int *ptr){
31     for (int i = 0; i< SAMPLES; i++){
32         if ((i>0) && (i%10==0)) {
33             printf("\n");
34         }
35         printf("%3d ", ptr[i]);
36     }
37     printf("\n");
38 }
39
40 int main(void) {
41     int samples[SAMPLES] = {0};
42     generateRandomArray(samples);
43     bubble(samples);
44     printArray(samples);
45     return(EXIT_SUCCESS);
46 }

```

Worin unterscheidet sich dieser Ansatz von dem vorhergehenden?

Um das erste (und größte) Element  $n$  ganz nach rechts zu bewegen, werden  $n-1$  Vertauschungen vorgenommen, für das nächstfolgende  $n-2$  usw. Für die Gesamtanzahl muss also die Summe über  $k$  von 1 bis  $n-1$  gebildet werden. Mit der Summenformel von Gauss kann gezeigt werden, dass im Falle der umgekehrt sortierten Liste werden maximal  $\frac{n \cdot (n-1)}{2}$  Vertauschungen zu führen sind.

Aufgabe: Welches Optimierungspotential sehen Sie?

In den Code sollte ein Abbruchkriterium integriert werden, wenn während eines Durchlaufes keine Änderungen vollzogen werden. Im günstigsten Fall lässt sich damit das Verfahren nach einem Durchlauf beenden.

### 8.4.2 Quicksort

Quicksort ist ein rekursiver Sortieralgorithmus, der die zu sortierende Liste in zwei Teillisten unterteilt und alle Elemente, die kleiner sind als das Pivot-Element, in die linke Teilliste, alle anderen in die rechte Teilliste einsortiert.

Die Buchstabenfolge „einbeispiel“ soll alphabetisch sortiert werden.

Ausgangssituation nach Initialisierung von  $i$  und  $j$ , das Element rechts („l“) ist das Pivotelement:

```

1  e i n b e i s p i e l
2  ^                     ^
3  i                     j

```

Nach der ersten Suche in den inneren Schleifen hat  $i$  auf einem Element  $\geq l$  und  $j$  auf einem Element  $\leq l$  gehalten:

```

1  e i n b e i s p i e l
2  ^                     ^
3  i                     j

```

Nach dem Tauschen der Elemente bei  $i$  und  $j$ :

```

1  e i e b e i s p i n l
2  ^                     ^
3  i                     j

```

Nach der nächsten Suche und Tauschen:



```

1  e i e b e i i p s n l
2      ^      ^
3      i      j

```

Nach einer weiteren Suche sind die Indizes aneinander vorbeigelaufen:

```

1  e i e b e i i p s n l
2      ^      ^
3      j      i

```

Nach dem Tauschen von *i* und *Pivot* bezeichnet *i* die Trennstelle der Teillisten. Bei *i* steht das *Pivot-Element*, links davon sind nur Elemente  $\leq$  *Pivot* und rechts nur solche  $>$  *Pivot*:

```

1  e i e b e i i l s n p
2      ^
3      i

```

Darauf aufbauend wird der Algorithmus nun auf die beiden Teile “eiebei” und “snp” angewand.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define MAXVALUE 100
6 #define MINVALUE 5
7 #define SAMPLES 20
8
9 void generateRandomArray(int *ptr) {
10     srand(time(NULL));
11     for (int i = 0; i < SAMPLES; i++){
12         ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
13     }
14 }
15
16 void quicksort(int *ptr, int first, int last){
17     int i, j, pivot, temp;
18
19     if(first < last){
20         pivot = first;
21         i = first;
22         j = last;
23
24         while(i < j){
25             while(ptr[i] <= ptr[pivot] && i < last)
26                 i++;
27             while(ptr[j] > ptr[pivot])
28                 j--;
29             if(i < j){
30                 temp = ptr[i];
31                 ptr[i] = ptr[j];
32                 ptr[j] = temp;
33             }
34         }
35         temp = ptr[pivot];
36         ptr[pivot] = ptr[j];
37         ptr[j] = temp;
38         quicksort(ptr, first, j-1);
39         quicksort(ptr, j+1, last);
40     }
41 }
42
43 void printArray(int *ptr){
44     for (int i = 0; i < SAMPLES; i++){

```

```

45     if ((i>0) && (i%10==0)) {
46         printf("\n");
47     }
48     printf("%3d ", ptr[i]);
49 }
50 printf("\n");
51 }
52
53 int main(void) {
54     int samples[SAMPLES] = {0};
55     generateRandomArray(samples);
56     quicksort(samples, 0, SAMPLES);
57     printArray(samples);
58     return(EXIT_SUCCESS);
59 }

```

Obwohl Quicksort im schlechtesten Fall quadratische Laufzeit hat, ist er in der Praxis einer der schnellsten Sortieralgorithmen.

Die C-Standardbibliothek umfasst in der `stdlib.h` eine Implementierung von quicksort - `qsort()` an. Sie wurde in der vorangegangenen Vorlesung besprochen. Ein Anwendungsbeispiel finden Sie im nachfolgenden Abschnitt.

## 8.5 Suchen

Suchen beschreibt die Identifikation von bestimmten Mustern in Daten. Das Spektrum kann dabei von einzelne Zahlenwerten oder Buchstaben bis hin zu komplexen zusammengesetzten Datentypen reichen.

Wie würden Sie vorgehen, um in einer sortierten List einen bestimmten Eintrag zu finden?

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define MAXVALUE 100
6  #define MINVALUE 5
7  #define SAMPLES 20
8
9
10 // Generieren der Zufallszahlen
11 void generateRandomArray(int *ptr) {
12     srand(time(NULL));
13     for (int i = 0; i < SAMPLES; i++){
14         ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
15     }
16 }
17
18 // Sortieren des Arrays
19 void bubble(int *array) {
20     int n_samples = SAMPLES;
21     int temp;
22     while(n_samples--){
23         for(int i = 1; i <= n_samples; i++){
24             if(array[i-1] > array[i]) {
25                 temp=array[i];
26                 array[i]=array[i-1];
27                 array[i-1]=temp;
28             }
29         }
30     }
31 }
32
33 // Ausgabe

```

```

34 void printArray(int *ptr){
35     for (int i = 0; i< SAMPLES; i++){
36         if ((i>0) && (i%10==0)) {
37             printf("\n");
38         }
39         printf("%3d ", ptr[i]);
40     }
41     printf("\n");
42 }
43
44 // Konventionelle Suche über dem Array
45 int search (int *ptr, int pattern) {
46     for (int i = 0; i< SAMPLES; i++){
47         if (ptr[i] == pattern) return i;
48     }
49     return -1;
50 }
51
52 //int binsearch (int *ptr, int links, int rechts, int wert) {
53 //
54 //}
55
56 int main (void) {
57     int samples[SAMPLES] = {0};
58     generateRandomArray(samples);
59     bubble(samples);
60     printArray(samples);
61     int pattern = 36;
62     int index = search (samples, pattern);
63     //int index = binsearch (samples, 0, SAMPLES-1, pattern);
64     if (-1==index){
65         printf("\nPattern %d nicht gefunden!", pattern);
66     }else{
67         printf("\nIndex von %d ist %d",pattern, index);
68     }
69     return(EXIT_SUCCESS);
70 }

```

Die Suchtiefe kann mit  $\lceil \log_2(n+1) \rceil$  bestimmt werden.

## 8.6 Implementierung in der Standardbibliothek

Die Standardbibliothek umfasst eine Suchfunktion, die es erlaubt Arrays nach beliebigen Kriterien zu sortieren. Der Name `qsort()` deutet dabei an, dass der Quicksort-Algorithmus zum Einsatz kommt. Die Vergleichsoperation, die vom Anwender zu implementieren ist, akzeptiert als Übergabewerte Zeiger auf zwei Einträge im Array und setzt diese in Beziehung. Bei einem negativen Rückgabewert ist das erste Element kleiner, für einen positiven Wert größer als das zweite Übergabewert. Für den Wert 0 liegt Gleichheit vor.

```

1 void qsort(
2     void *array,          // Anfangsadresse des Arrays
3     size_t n,             // Anzahl der Elemente zum Sortieren
4     size_t size,          // Größe des Datentyps, der sortiert wird
5     int (*vergleich_func)(const void*, const void*) );

```

Eine analoge Funktion steht für die Suche in sortierten Listen bereit. `bsearch()` durchsucht diese und gibt einen Pointer zurück, der mit dem Suchkriterium übereinstimmt.

Die binäre Suche ist in der `stdlib.h` als Funktion implementiert. Die Deklaration erfasst folgende Parameter:

```

1 void *bsearch(const void *key,
2               const void *base,
3               size_t nitems,

```

```

4      size_t size,
5      int (*compar)(const void *, const void *))

```

Analog zu `qsort()` wird ein Funktionspointer `*compar()` der den Vergleich des `key` mit den Einträgen in `base` realisiert.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define MAXVALUE 100
6  #define MINVALUE 85
7  #define SAMPLES 20
8
9  // Generieren der Zufallszahlen
10 void generateRandomArray(int *ptr) {
11     srand(time(NULL));
12     for (int i = 0; i < SAMPLES; i++){
13         ptr[i] = rand() % (MAXVALUE - MINVALUE + 1) + MINVALUE;
14     }
15 }
16
17 // Ausgabe
18 void printArray(int *ptr){
19     for (int i = 0; i < SAMPLES; i++){
20         if ((i>0) && (i%10==0)) {
21             printf("\n");
22         }
23         printf("%3d ", ptr[i]);
24     }
25     printf("\n");
26 }
27
28 int cmpfunc (const void * a, const void * b) {
29     return ( *(int*)a - *(int*)b );
30 }
31
32 int main (void) {
33     int samples[SAMPLES] = {0};
34     generateRandomArray(samples);
35     qsort(samples, SAMPLES, sizeof(int), cmpfunc);
36     printArray(samples);
37     int pattern = 90;
38     int *item;
39     item = (int*) bsearch (&pattern, samples, SAMPLES, sizeof(int), cmpfunc);
40     if( item != NULL ) {
41         printf("%d an der Speicherstelle %p gefunden!\n", pattern, item);
42         printf("Das Array beginnt im Speicher bei %p\n", samples);
43         printf("Der Index von %d beträgt folglich %d\n", pattern, item - samples);
44     }else{
45         printf("%d nicht gefunden!", pattern);
46     }
47     return(EXIT_SUCCESS);
48 }

```

`bsearch` evaluiert die Existenz eines Eintrages, dabei ist es wichtig, den Rückgabewert auf den richtigen Typ des Eintrages zu casten.

## 8.7 Beispiel des Tages

Schätzen Sie die Größe der Kreiszahl  $\pi$  mittels Monte-Carlo-Simulation ab. Nutzen Sie dafür den Ansatz, dass bei der Projektion von  $n$  gleichverteilten Paaren  $(x, y)$   $\pi$  über den Anteil zwischen denjenigen Paaren innerhalb eines Quadranten unter dem Einheitskreis und denjenigen außerhalb bestimmt werden kann.

Die Fläche des Quadrates ist 4, der Flächenanteil des Kreises beträgt  $1^2 \cdot \pi$ . Somit gilt

$$\pi \approx \frac{\text{count}_{in}}{\text{count}_{all}} \cdot 4$$

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5
6 double bestimmePI(unsigned int samples){
7     double x, y, z;
8     int count = 0;
9     srand(time(NULL));
10    count=0;
11    for (int i=0; i<samples; i++) {
12        x = rand() / (double) RAND_MAX;
13        y = rand() / (double) RAND_MAX;
14        z = x*x+y*y;
15        if (x*x+y*y<1) count++;
16    }
17    return (double)count/samples*4;
18 }
19
20 int main(void) {
21     for (int number=5; number<15000; number=number+50){
22         printf("%3d, %f, %f\n", number,
23                M_PI,
24                bestimmePI(number));
25     }
26     return EXIT_SUCCESS;
27 }

```

Das korrekte Ergebnis lautet 3.1415926535...