Prozedurale Programmierung - TU Freiberg

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

andre-dietrich JayTee42 SebastianZug galinarudollf DkPepper Lalelele Lorcc Anjuschenka

Inhaltsverzeichnis

Kapitel 1

Einführung

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

Fragen an die heutige Veranstaltung ...

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

1.1 Umfrage

Hat Sie die letztwöchige Vorstellung der Ziele der Lehrveranstaltung überzeugt?

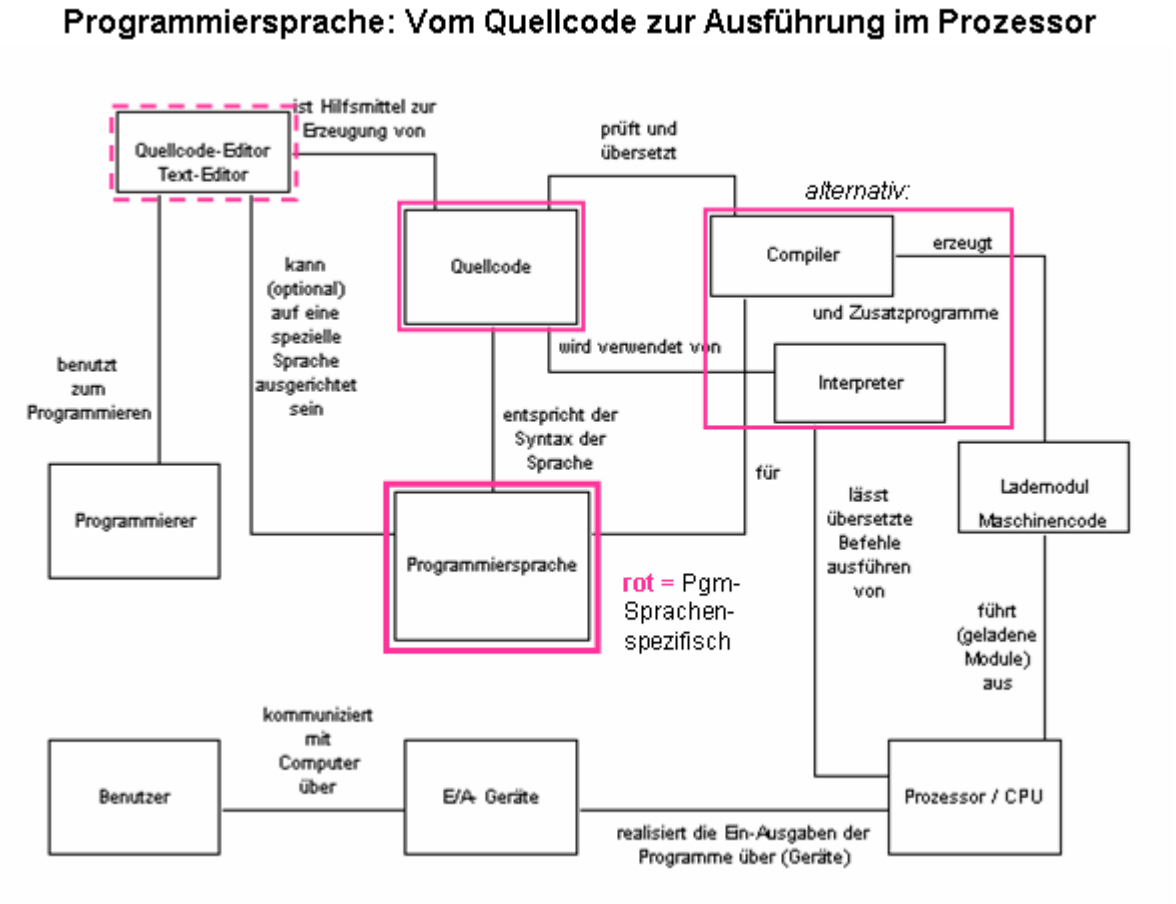
- [(ja)] Ja, ich gehe davon aus, viel nützliches zu erfahren.
- [(schau'n wir mal)] Ich bin noch nicht sicher. Fragen Sie in einigen Wochen noch mal.
- [(nein)] Nein, ich bin nur hier, weil ich muss.

1.2 Wie arbeitet ein Rechner eigentlich?

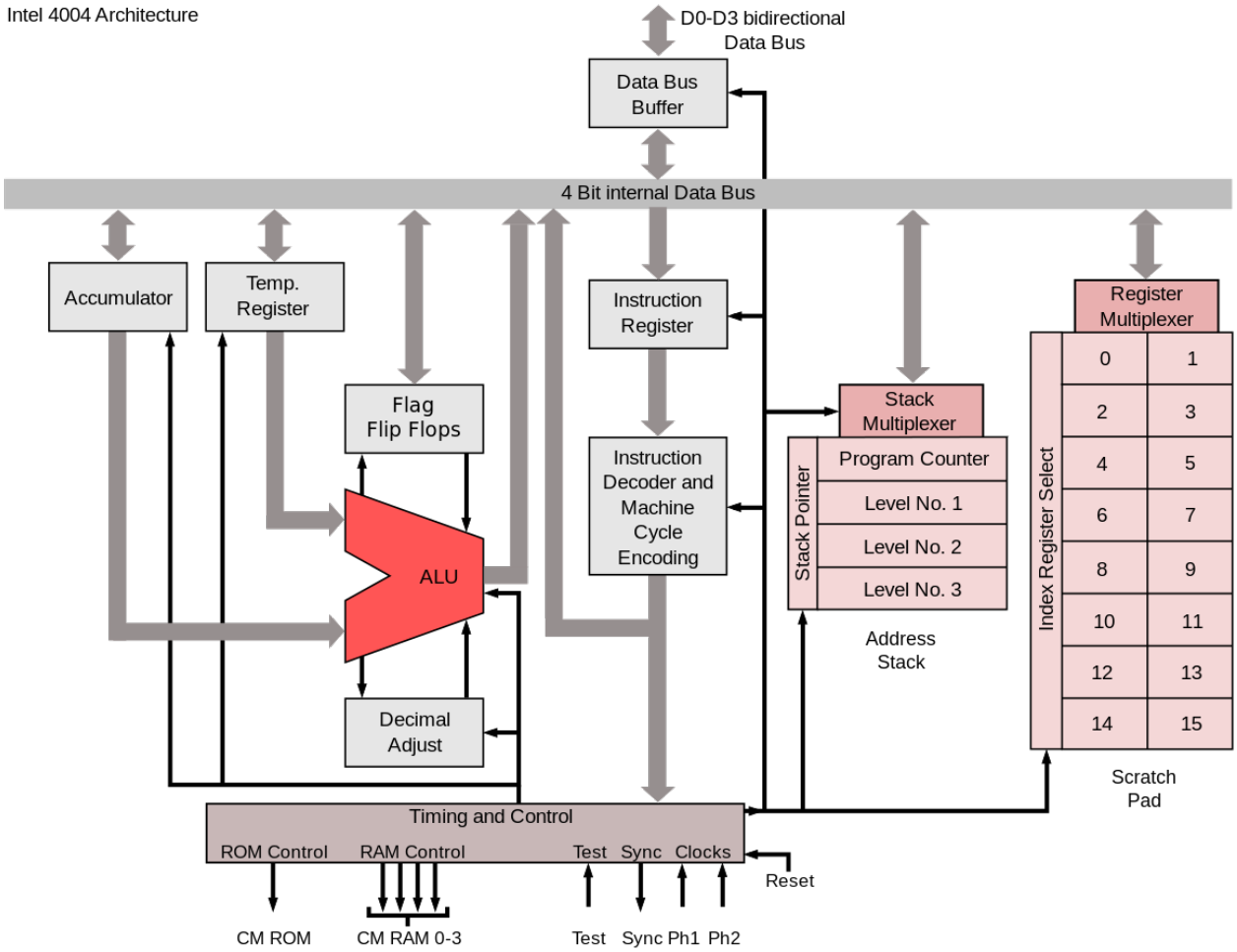
Programme sind Anweisungslisten, die vom Menschen erdacht, auf einem Rechner zur Ausführung kommen. Eine zentrale Hürde ist dabei die Kluft zwischen menschlicher Vorstellungskraft und Logik, die **implizite Annahmen und Erfahrungen** einschließt und der **"stupiden" Abarbeitung von Befehlsfolgen** in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

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

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



Beispiel: Intel 4004-Architektur (1971)



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

Speicherauszug den Intel 4004:

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

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

4004 Instruction Set

BASIC INSTRUCTIONS (* = 2 Word Instructions)

Hex Code	MNEMONIC	OPR $D_3 D_2 D_1 D_0$	OPA $D_3 D_2 D_1 D_0$	DESCRIPTION OF OPERATION
00	NOP	0 0 0 0	0 0 0 0	No operation.
1 -	*JCN	0 0 0 1	$C_1 C_2 C_3 C_4$	Jump to ROM address $A_2 A_2 A_2 A_2, A_1 A_1 A_1 A_1$ (within the same ROM that contains this JCN instruction) if condition $C_1 C_2 C_3 C_4$ is true, otherwise go to the next instruction in sequence.
- -		$A_2 A_2 A_2 A_2$	$A_1 A_1 A_1 A_1$	
2 -	*FIM	0 0 1 0	R R R 0	Fetch immediate (direct) from ROM Data $D_2 D_2 D_2 D_2 D_1 D_1 D_1 D_1$ to index register pair location RRR.
- -		$D_2 D_2 D_2 D_2$	$D_1 D_1 D_1 D_1$	

▪ ▪ ▪

8 -	ADD	1 0 0 0	R R R R	Add contents of register RRRR to accumulator with carry.
9 -	SUB	1 0 0 1	R R R R	Subtract contents of register RRRR to accumulator with borrow.
A -	LD	1 0 1 0	R R R R	Load contents of register RRRR to accumulator.
B -	XCH	1 0 1 1	R R R R	Exchange contents of index register RRRR and accumulator.
C -	BBL	1 1 0 0	D D D D	Branch back (down 1 level in stack) and load data DDDD to accumulator.
D -	LDM	1 1 0 1	D D D D	Load data DDDD to accumulator.
F0	CLB	1 1 1 1	0 0 0 0	Clear both. (Accumulator and carry)
F1	CLC	1 1 1 1	0 0 0 1	Clear carry.
F2	IAC	1 1 1 1	0 0 1 0	Increment accumulator.

Quelle: Intel 4004 Assembler

1.2.1 Programmierung

Möchte man so Programme schreiben?

Vorteil:

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

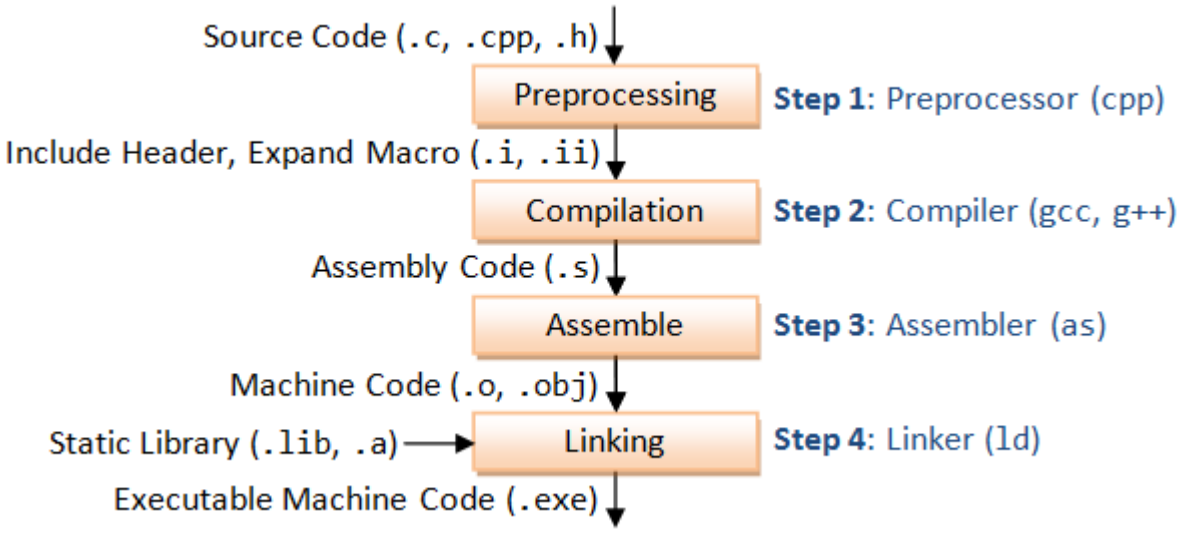
Nachteile:

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

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

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

Stufen des Compile-Vorganges:



1.2.2 Einordnung von C und C++

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

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

  deklarative Programiersprachen: Ein Programm beschreibt die allgemeinen Eigenschaften von Objekten und ihre Beziehungen untereinander. Das Programm beschreibt zunächst nur das Wissen zur Lösung des Problems (Prolog, Haskal, SQL, …).
- Wenige Schlüsselwörter als Sprachumfang

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

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

1.3 Erstes C++ Programm

1.3.1 “Hello World”

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

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

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

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

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

@AVR8js.sketch

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

1.3.2 Ein Wort zu den Formalien

```
// Karl Klammer
// Print Hello World drei mal

#include <iostream>

int main() {
  int zahl;
  for (zahl=0; zahl<3;  zahl++){
        std::cout << "Hello World! ";
  }
    return 0;
}
```

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

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

1.3.3 Gute Kommentare

1. Kommentare als Pseudocode

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

2. Kommentare zur Datei

```
// This is the mars rover control application
//
// Karl Klammer, Oct. 2018
// 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 }
```

1.3.4 Schlechte Kommentare

1. Überkommentierung von Code

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

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

Quelle: C Code Style Guidelines

2. "Merkwürdige Kommentare"

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

Sammlung von Kommentaren

1.3.5 Was tun, wenn es schief geht?

```
// Karl Klammer
// Print Hello World drei mal

#include <iostream>

int main() {
  for (zahl=0; zahl<3;  zahl++){
        std::cout << "Hello World! "
  }
    return 0;
```

Methodisches Vorgehen:

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

1.3.6 Compilerfehlermeldungen

Beispiel 1

```
#include <iostream>

int mani() {
    std::cout << "Hello World!";
    return 0;
}
```

Beispiel 2

```
#include <iostream>

int main()
    std::cout << "Hello World!";
    return 0;
}
```

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

```
#include <iostream>

imt main() {
    std::cout << "Hello World!'';
    std::cout << "Wo liegt der Fehler?";
    return 0;
}
```

1.3.7 Und wenn das Kompilieren gut geht?

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

```
#include <iostream>

int main() {
  char zahl;
  for (zahl=250; zahl<256; zahl++){
        std::cout << "Hello World!";
  }
    return 0;
}
```

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

1.4 Warum dann C++?

Zwei Varianten der Umsetzung … C++ vs. Python

```cpp
#include <iostream>

int main() {
  char zahl;
  for (int zahl=0; zahl<3; zahl++){
        std::cout << "Hello World! " << zahl << "\n";
  }
    return 0;
}
```

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

1.5 Beispiele der Woche

Gültiger C++ Code

```cpp
#include <iostream>

imt main() {
  int i = 5;
  int j = 4;
  i = i + j + 2;
    std::cout << "Hello World ";
  std::cout << i << "!";
    return 0;
}
```

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

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

Algorithmisches Denken

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

```cpp
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

@AVR8js.sketch

Kapitel 2

Grundlagen der Sprache C

Parameter	Kursinformationen
Veranstaltung:	`Einführung in das wissenschaftliche Programmieren`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Ein- und Ausgabe / Variablen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/01_EingabeAusgabeDatentypen.md
Autoren	@author

Fragen an die heutige Veranstaltung ...

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

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

2.1 Variablen

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

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

```
#include<iostream>

int main() {
  // Variante 1 - ganz schlecht
  std::cout <<"f("<<x<<") = "<<3*5*5 + 4*5 + 8<<" \n";

  // Variante 2 - besser
  int x = 9;
```

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

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

Ein Programm manipuliert Daten, die in Variablen organisiert werden.

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

Kennzeichen einer Variable:

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

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

```
1 const double e = 2.71828182845905;
```

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

2.1.1 Zulässige Variablennamen

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

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

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

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

> Vergeben Sie die Variablennamen mit Sorgfalt. Für jemanden der Ihren Code liest, sind diese wichtige Informationsquellen! Link

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

Bezeichnung	denkbare Variablennamen
CamelCase (upperCamel)	`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/C++ gibt es folgende Typen für Ganzzahlen:

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

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

Gängige Zuschnitte für `char` oder `int`

Schlüsselwort	Wertebereich
`signed char`	-128 bis 127
`char`	0 bis 255 (0xFF)
`signed int`	-32768 bis 32767
`int`	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` eigentliche nicht benötigt

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

2.1.2.2 Sonderfall `char`

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

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

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

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

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

2.1.2.3 Sonderfall `bool`

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

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

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

2.1.2.4 Architekturspezifische Ausprägung (Integer Datentypen)

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

```
#include <iostream>

int main(void)
{
  int x;
  std::cout<<"x umfasst " <<sizeof(x)<<" Byte.";
  return 0;
}
```

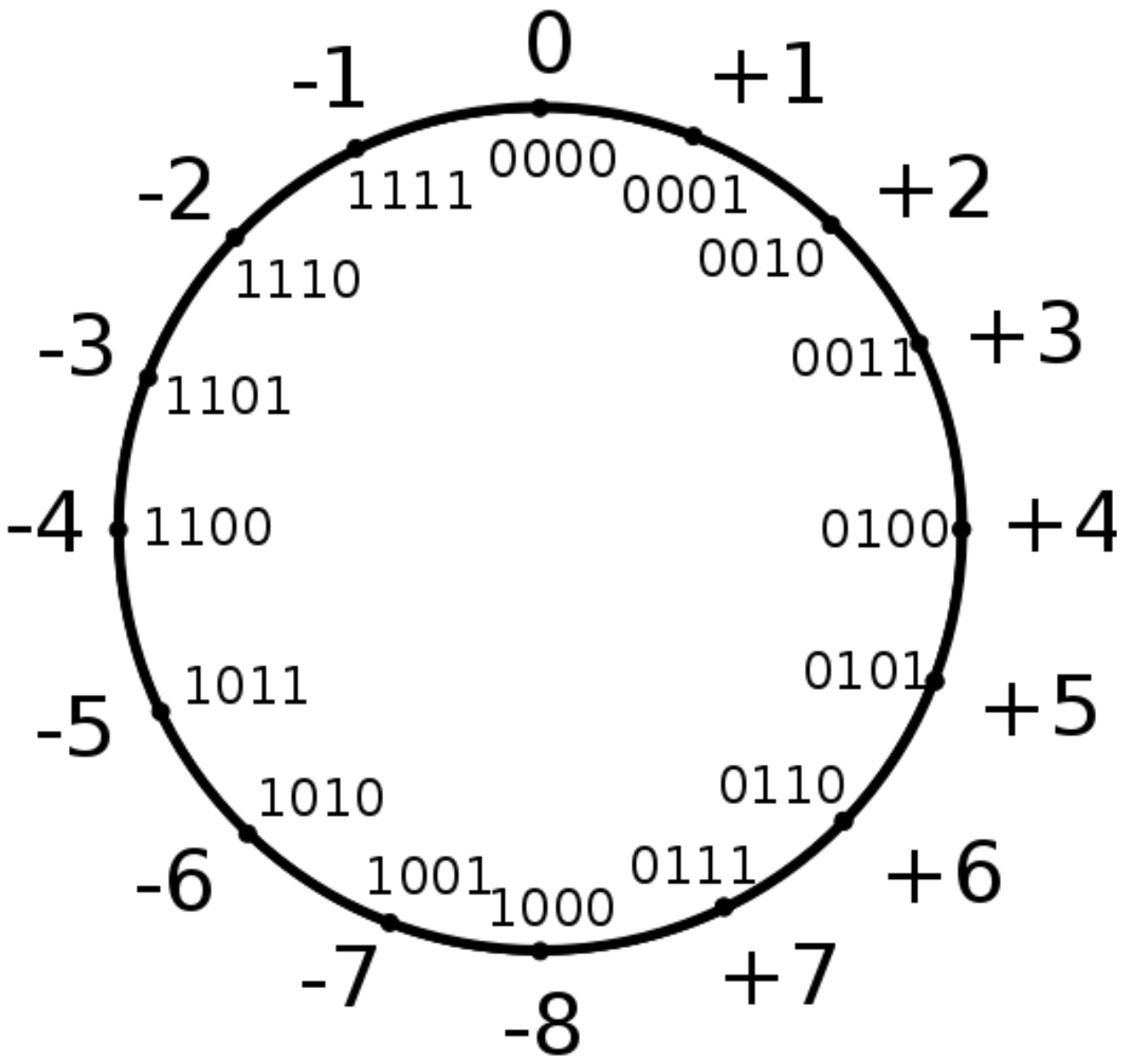
```
#include <iostream>
#include <limits.h>   /* INT_MIN und INT_MAX */

int main(void) {
   std::cout<<"int size: "<< sizeof(int)<<" Byte\n";
   std::cout<<"Wertebereich von "<< INT_MIN<<" bis "<< INT_MAX<< "\n";
   std::cout<<"char size : "<< sizeof(char) <<" Byte\n";
   std::cout<<"Wertebereich von "<< CHAR_MIN<<" bis "<<CHAR_MAX<<"\n";
   return 0;
}
```

Die implementierungspezifische Werte, wie die Grenzen des Wertebereichs der ganzzahlinen 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)

```
#include <iostream>
#include <limits.h>   /* SHRT_MIN und SHRT_MAX */

int main(){
  short a = 30000;

  std::cout<<"Berechnung von 30000+3000 mit:\n\n";

  signed short c;     // -32768 bis 32767
  std::cout<<"(signed) short c - Wertebereich von "<<SHRT_MIN<<" bis "<<SHRT_MAX<<"\n";
  c = 3000 + a;       // ÜBERLAUF!
  std::cout<<"c="<<c<<"\n";

  unsigned short d;   //      0 bis 65535
  std::cout<<"unsigned short d - Wertebereich von "<<0<<" bis "<<USHRT_MAX<<"\n";

  d = 3000 + a;
  std::cout<<"d="<<d<<"\n";
}
```

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

2.1.2.6 Fließkommazahlen

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

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

Schlüsselwort	Mindestgröße
`float`	4 Byte
`double`	8 Byte
`long double`	je nach Implementierung

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

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

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

	`float`	`double`
kleinste positive Zahl	1.1754943508e-38	2.2250738585072014E-308
Wertebereich	±3.4028234664e+38	±1.7976931348623157E+308

Achtung: Fließkommazahlen bringen einen weiteren Faktor mit - die Unsicherheit

```
#include<iostream>
#include<float.h>

int main(void) {
    std::cout<<"float Genauigkeit  :"<<FLT_DIG<<" \n";
    std::cout<<"double Genauigkeit :"<<DBL_DIG<<" \n";
    float x = 0.1;
  if (x == 0.1) {   // <- das ist ein double "0.1"
  //if (x == 0.1f) {   // <- das ist ein float "0.1"
   std::cout<<"Gleich\n";
  }else{
    std::cout<<"Ungleich\n";
  }
  return 0;
}
```

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

2.1.2.7 Datentyp `void`

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

Anwendungsbeispiele:

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

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

```
3   return 0;
4 }
```

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

2.1.3 Wertspezifikation

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

Gruppe	zulässige Zeichen
decimal-digits	`0 1 2 3 4 5 6 7 8 9`
octal-prefix	`0`
octal-digits	`0 1 2 3 4 5 6 7`
hexadecimal-prefix	`0x 0X`

hexadecimal-digits	`0 1 2 3 4 5 6 7 8 9`
	`a b c d e f`
	`A B C D E F`

unsigned-suffix	`u U`
long-suffix	`l L`
long-long-suffix	`ll LL`
fractional-constant	`.`
exponent-part	`e E`
binary-exponent-part	`p P`
sign	`+ -`
floating-suffix	`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 `int`
-234L	Ganzzahl vom Typ `signed long`
100000000000	Ganzzahl vom Typ `long`
011	Ganzzahl also oktale Zahl (Wert 9_d)
0x12	Ganzzahl (18_d)
1.23F	Fließkommazahl vom Typ `float`
0.132	Fließkommazahl vom Typ `double`
123e-2	Fließkommazahl vom Typ `double`
0xC.68p+2	hexadizimale Fließkommazahl vom Typ `double`

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

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

2.1.4 Adressen

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

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

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

2.1.5 Sichtbarkeit und Lebensdauer von Variablen

Lokale Variablen

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

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

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

Globale Variablen

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

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

```
9 }
```

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

2.1.6 Definition vs. Deklaration vs. Initialisierung

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

> Deklaration ist nur die Vergabe eines Namens und eines Typs für die Variable. Definition ist die Reservierung des Speicherplatzes. Initialisierung ist die Zuweisung eines ersten Wertes.
>
> **Merke:** Jede Definition ist gleichzeitig eine Deklaration aber nicht umgekehrt!

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

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

2.1.7 Typische Fehler

Fehlende Initialisierung

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

Redeklaration

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

Falsche Zahlenliterale

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

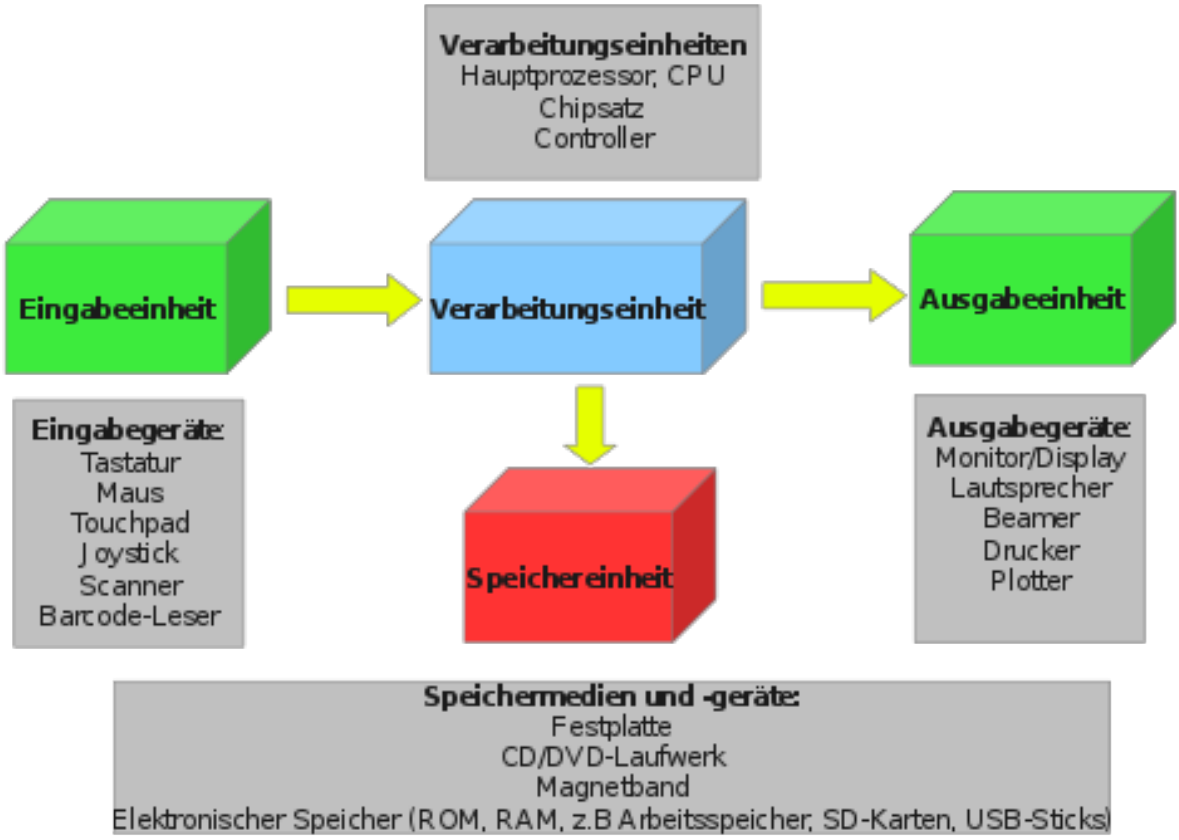
Was passiert wenn der Wert zu groß ist?

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

```
7     return 0;
8 }
```

2.2 Ein- und Ausgabe

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



Quelle: EVA-Prinzip (Autor: Deadlyhappen)

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

Als Standard werden verwendet:

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

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

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

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

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

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

2.2.1 Ausgabe

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

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

Welche Formatierungmöglichkeiten bietet der Ausgabeoperator noch?

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

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

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

Achtung: Die Manipulatoren wirken auf alle darauf folgenden Ausgaben.

2.2.1.1 Feldbreite

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

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

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

2.2.1.2 Genauigkeit

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

```
int main() {
  for (int i = 12; i > 1; i -=3) {
    std::cout << std::setprecision(i) << std::fixed << M_PI << std::endl;
  }
}
```

2.2.1.3 Escape-Sequenzen

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

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

int main(){
  cout << "123456789\r";
  cout << "ABCD\n\n";
    cout << "Vorname \t Name \t\t Alter \n";
    cout << "Andreas \t Mustermann\t 42 \n\n";
    cout << "Manchmal braucht man auch ein \"\\\"\n";
  return 0;
}
```

2.2.2 Eingabe

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

```
#include <iostream>

int main()
{
  char b;
  float a;
  int i;
  std::cout<<"Bitte Werte eingeben [char float int] : ";
  std::cin>>b>>a>>i;
  std::cout<<"char - " <<b<< " float - "<<a<<" int - "<<i;
  return 0;
}
```

2.2.3 Beispiel der Woche

Implementieren Sie einen programmierbaren Taschenrechner für quadaratische Funktionen.

```
#include<iostream>

int main() {
  // Variante 1 - ganz schlecht
  std::cout <<"f("<<x<<") = "<<3*5*5 + 4*5 + 8<<" \n";

  // Variante 2 - besser
  int x = 9;
```

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

Kapitel 3

Operatoren & Kontrollstrukturen

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

Fragen an die heutige Veranstaltung ...

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

3.1 Operatoren

Kapitel 4

Unterscheidungsmerkmale

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

Zahl der beteiligten Operationen

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

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

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

Position

Des Weiteren wird unterschieden, welche Position der Operator einnimmt:

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

`+` und `-` können alle drei Rollen einnehmen:

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

Funktion des Operators

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

Weitere Unterscheidungsmerkmale ergeben sich zum Beispiel aus der Assoziativität der Operatoren.

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

4.0.1 Zuweisungsoperator

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

```cpp
#include <iostream>
using namespace std;

int main() {
   int zahl1 = 10;
   int zahl2 = 20;
   int ergeb = 0;
   // Zuweisung des Ausdrucks 'zahl1 + zahl2'
   ergeb = zahl1 + zahl2;

   cout<<zahl1<<" + "<<zahl2<<" = "<<ergeb<<"\n";
   return 0;
}
```

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

4.0.2 Inkrement und Dekrement

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

```cpp
#include <iostream>
using namespace std;

int main(){
  int x, result;
  x = 5;
  result = 2 * ++x;   // Gebrauch als Präfix
  cout<<"x="<<x<<" und result="<<result<<"\n";
  result = 2 * x++;   // Gebrauch als Postfix
  cout<<"x="<<x<<" und result="<<result<<"\n";
  return 0;
}
```

4.0.3 Arithmetische Operatoren

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

Achtung: Divisionsoperationen werden für Ganzzahlen und Gleitkommazahlen unterschiedlich realisiert.

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

```cpp
#include <iostream>
using namespace std;

int main(){
  int timestamp, minuten;

  timestamp = 345; //[s]
  cout<<"Zeitstempel "<<timestamp<<" [s]\n";
```

```
  minuten=timestamp/60;
  cout<<timestamp<<" [s] entsprechen "<<minuten<<" Minuten\n";
  return 0;
}
```

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

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

int main(){
  int timestamp, sekunden, minuten;

  timestamp = 345; //[s]
  cout<<"Zeitstempel "<<timestamp<<" [s]\n";
  minuten=timestamp/60;
  sekunden=timestamp%60;
  cout<<"Besser lesbar = "<<minuten<<" min. "<<sekunden<<" sek.\n";
  return 0;
}
```

4.0.4 Vergleichsoperatoren

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

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

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

int main(){
  int x = 15;
  cout<<"x = "<<x<<" \n";
  cout<<boolalpha<<"Aussage x > 5 ist "<< (x>5) << " \n";
  cout<<boolalpha<<"Aussage x == 5 ist "<< (x==-15) << " \n";
  return 0;
}
```

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

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

4.0.5 Logische Operatoren

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

Operation	Bedeutung
`&&`	UND
`||`	ODER

Operation	Bedeutung
!	NICHT

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

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

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

int main(){
  float Temperatur = -30;      // Das sind unsere Probewerte
  float Feuchte = 65;

  // Vergleichsoperationen und Logische Operationen
  bool TempErgebnis = ....     // Hier sind Sie gefragt!

  // Ausgabe
  if ... {
    cout<<"Die Messwerte kannst Du vergessen!";
  }
  return 0;
}
```

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

4.0.6 `sizeof` - Operator

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

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

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

int main(){
  double wert=0.0;
  cout<<sizeof(0)<<" "<<sizeof(double)<<" "<<sizeof(wert);
return 0;
}
```

4.1 Vorrangregeln

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

Prioritäten

In welcher Reihung erfolgt beispielsweise die Abarbeitung des folgenden Ausdruckes?

```
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:

```
c = sizeof(x) + ++a  / 3;
//    |       |  |   |
//    |       |  |   |--- Priorität 13
//    |       |  |--- Priorität 14
//    |       |--- Priorität 12
//    |--- Priorität 14

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

Assoziativität

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

```
a = 4 / 2 / 2;

// von rechts nach links (FALSCH)
// 4 / (2 / 2)   // ergibt 4

// von links nach rechts ausgewertet
// (4 / 2) / 2   // ergibt 1
```

Merke: Setzen Sie Klammern, um alle Zweifel auszuräumen

4.2 … und mal praktisch

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

```
const int button_A_pin = 10;
const int button_B_pin = 11;
const int led_pin = 13;

int buttonAState;
int buttonBState;

void setup(){
  pinMode(button_A_pin, INPUT);
  pinMode(button_B_pin, INPUT);
  pinMode(led_pin, OUTPUT);
  Serial.begin(9600);
 }

void loop() {
  Serial.println("Wait one second for A ");
  delay(1000);
  buttonAState = digitalRead(button_A_pin);
  Serial.println ("... and for B");
  delay(1000);
  buttonBState = digitalRead(button_B_pin);

  if ( buttonAState && buttonBState){
    digitalWrite(led_pin, HIGH);
    delay(3000);
  }
  else
  {
    digitalWrite(led_pin, LOW);
  }
}
```

@AVR8js.sketch

Kapitel 5

Kontrollfluss

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

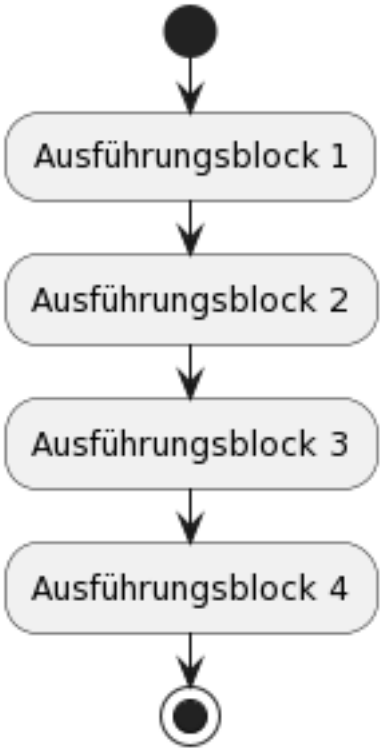


Abbildung 5.1: Modelle

Diese Einschränkung wollen wir nun mit Hilfe weiterer Anweisungen überwinden:

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

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

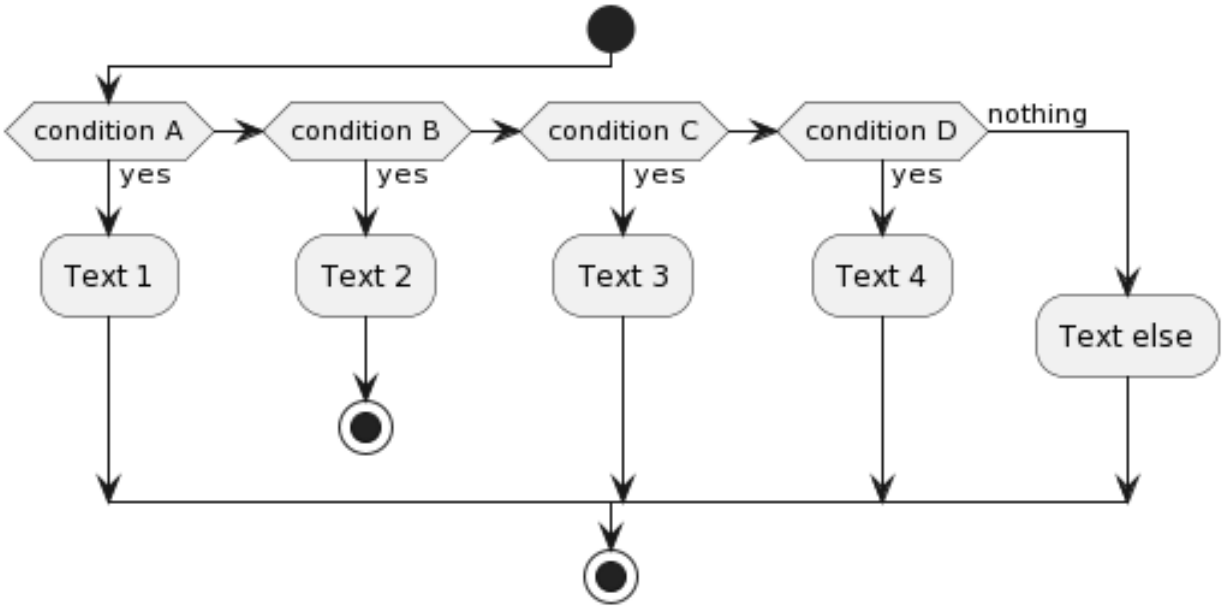
2. **Schleifen (Iteration)**: Ein Anweisungsblock wird so oft wiederholt, bis eine Abbruchbedingung erfüllt wird.

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

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

5.0.1 Verzweigungen

Verzweigungen entfalten mehrere mögliche Pfade für die Ausführung des Programms.



5.0.1.1 `if`-Anweisungen

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

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

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

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

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

Mehrere Fälle können verschachtelt abgefragt werden:

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

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

Weitere Beispiele für Bedingungen

Die Bedingungen können als logische UND arithmetische Ausdrücke formuliert werden.

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

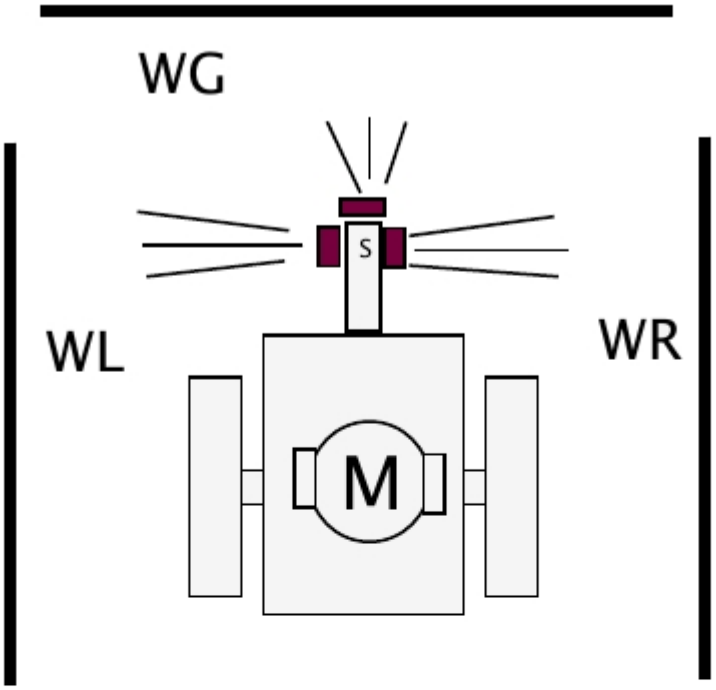
Mögliche Fehlerquellen

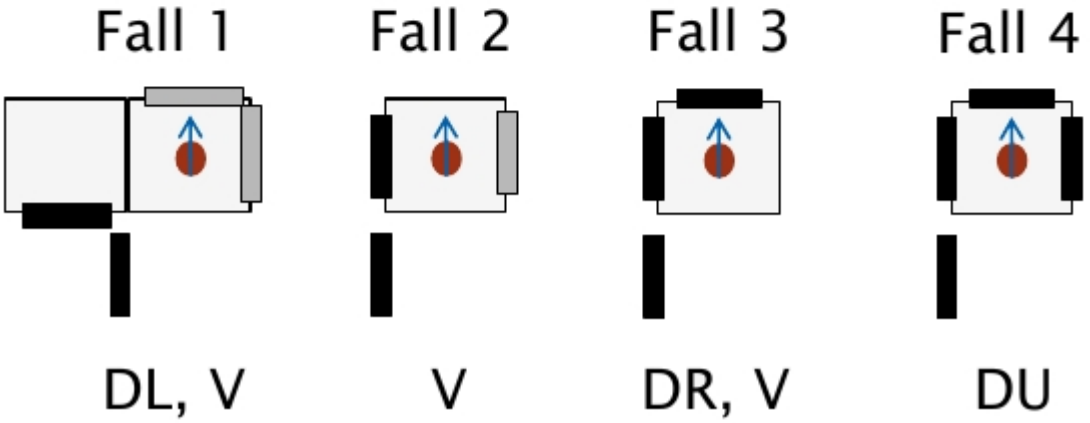
1. Zuweisungs- statt Vergleichsoperator in der Bedingung (kein Compilerfehler)
2. Bedingung ohne Klammern (Compilerfehler)
3. `;` hinter der Bedingung (kein Compilerfehler)
4. Multiple Anweisungen ohne Anweisungsblock
5. Komplexität der Statements

5.0.1.2 Beispiel

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

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

int main(){
  int WL, WG, WR;
  WL = 0; WG = 1; WR =1;
  if (!WL)                          // Fall 1
    cout<<"Drehung Links\n";
  if ((WL) && (!WG))                 // Fall 2
      cout<<"Vorwärts\n";
  if ((WL) && (WG) && (!WR))          // Fall 3
    cout<<"Drehung Rechts\n";
  if ((WL) && (WG) && (WR))           // Fall 4
    cout<<"Drehung 180 Grad\n";
    return 0;
}
```

Sehen Sie mögliche Vereinfachungen des Codes?**

5.0.1.3 Zwischenfrage

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

int main()
{
    int Punkte = 45;
    int Zusatzpunkte = 15;
    if (Punkte + Zusatzpunkte >= 50)
    {
       cout<<"Test ist bestanden!\n";
       if (Zusatzpunkte >= 15)
       {
          cout<<"Alle Zusatzpunkte geholt!\n";
       }else{
           if(Zusatzpunkte > 8) {
               cout<<"Respektable Leistung\n";
           }
       }
    }else{
       cout<<"Leider durchgefallen!\n";
    }
    return 0;
}
```

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

5.0.1.4 `switch`-Anweisungen

Too many ifs - I think I switch

Berndt Wischnewski

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.

```
switch(expression)
 {
   case const-expr: Anweisung break;
   case const-expr:
      Anweisungen
      break;
   case const-expr: Anweisungen break;
   default: Anweisungen
 }
```

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

int main() {
  int a=50, b=60;
  char op;
  cout<<"Bitte Operator definieren (+,-,*,/): ";
  cin>>op;

  switch(op) {
    case '+':
      cout<<a<<" + "<<b<<" = "<<a+b<<" \n";
      break;
    case '-':
      cout<<a<<" - "<<b<<" = "<<a-b<<" \n";
      break;
    case '*':
      cout<<a<<" * "<<b<<" = "<<a*b<<" \n";
      break;
    case '/':
      cout<<a<<" / "<<b<<" = "<<a/b<<" \n";
      break;
    default:
      cout<<op<<"? kein Rechenoperator \n";
  }
  return 0;
}
```

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.

```
// Fehlerhafte case Blöcke
switch(x)
{
   case x < 100: // das ist ein Fehler
     y = 1000;
  break;

  case 100.1: // das ist genauso falsch
     y = 5000;
     z = 3000;
  break;
}
```

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.

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

int main() {
  int a=5;
```

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

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

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

5.0.2 Schleifen

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

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

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

- Iterationssymbol

```
1   +-------------------------------------------------------------------+
```

```
   |                                                                    |
   |  zähle [Variable] von [Startwert] bis [Endwert], mit [Schrittweite] |
   | +------------------------------------------------------------------+
   | |                                                                  |
   | |  Anweisungsblock 1                                               |
   +-+------------------------------------------------------------------+     .
```

- Wiederholungsstruktur mit vorausgehender Bedingungsprüfung

```
   +-------------------------+
   |                         |
   |  solange Bedingung wahr  |
   | +-----------------------+
   | |                       |
   | |  Anweisungsblock 1     |
   +-+-----------------------+                                          .
```

- Wiederholungsstruktur mit nachfolgender Bedingungsprüfung

```
   +-+-----------------------+
   | |                       |
   | |  Anweisungsblock 1     |
   | +-----------------------+
   |                         |
   |   solange Bedingung wahr |
   +-------------------------+                                          .
```

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

5.0.2.1 `for`-Schleife

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

```
// generisches Format der for-Schleife
for(Initialisierung; Bedingung; Reinitialisierung) {
   // Anweisungen
}

// for-Schleife als Endlosschleife
for(;;){
   // Anweisungen
}
```

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

int main(){
    int i;
  for (i = 1; i<10; i++)
    cout<<i<<" ";

  cout<<"\nNach der Schleife hat i den Wert "<<i<<"\n";
    return 0;
}
```

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

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

int main(){
  int i;
  for (i = 1; i<10; i++);
    cout<<i<<" ";

  cout<<"Das ging jetzt aber sehr schnell ... \n"<<i;
  return 0;
}
```

5.0.2.2 `while`-Schleife

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

```
// generisches Format der while-Schleife
while (Bedingung)
  Anweisungen;

while (Bedingung){
    Anweisungen;
    Anweisungen;
}
```

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

int main(){
  char c;
  int zaehler = 0;
  cout<<"Pluszeichenzähler - zum Beenden \"_\" [Enter]\n";
  cin>>c;
  while(c != '_')
  {
    if(c == '+')
      zaehler++;
      cin>>c;
  }
  cout<<"Anzahl der Pluszeichen: "<<zaehler<<"\n";
  return 0;
}
```

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

```
// generisches Format der while-Schleife
i = 0;
while (i<10){
   // Anweisungen;
   i++;
}

for (i=0; i<10; i++){
   // Anweisungen;
}
```

5.0.2.3 `do-while`-Schleife

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

```
// generisches Format der while-Schleife
do
   Anweisung;
 while (Bedingung);
```

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

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

int main(){
  char c;
  int zaehler = 0;
  cout<<"Pluszeichenzähler - zum Beenden \"_\" [Enter]\n";
  do
  {
    cin>>c;
    if(c == '+')
      zaehler++;
  }while(c != '_');
  cout<<"Anzahl der Pluszeichen: "<<zaehler<<"\n";
  return 0;
}
```

5.0.3 Kontrolliertes Verlassen der Anweisungen

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

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

int main(){
    int i;
  for (i = 1; i<10; i++){
      if (i == 5) break;
      cout<<i<<" ";
  }
  cout<<"\nUnd vorbei ... i ist jetzt "<<i<<"\n";
    return 0;
}
```

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

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

int main(){
    int i;
  for (i = -5; i<6; i++){
      if (i == 0) continue;
      cout<<12. / i<<"\n";
  }
    return 0;
}
```

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

5.1 Beispiel des Tages

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

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

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

Kapitel 6

Grundlagen der Sprache C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik`
Semester	`Wintersemester 2022/23`
Hochschule	`Technische Universität Freiberg`
Inhalte:	`Array, Zeiger und Referenzen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/03_ArrayZeigerReferenzen.md
Autoren	@author

Fragen an die heutige Veranstaltung ...

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

6.1 Wie weit waren wir gekommen?

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

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

```
17 }
```

@AVR8js.sketch

6.2 Arrays

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

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

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

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

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

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

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

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

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

6.2.1 Deklaration, Definition, Initialisierung, Zugriff

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

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

Wie können Arrays noch initialisiert werden:

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

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

```
      cout<<a[i]<<" "<< b[i]<<"\n";
    }
  return 0;
}
```

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

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

int main(void) {
  int a[3];
  cout<<"\nNur zur Speicherplatz [Byte] "<<sizeof(a);
  cout<<"\nZahl der Elemente "<<sizeof(a)/sizeof(int)<<"\n";
  return 0;
}
```

6.2.2 Fehlerquelle Nummer 1 - out of range

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

int main(void) {
  int a[] = {-2, 5, 99};
  for (int i=0; i<=3; i++)
    cout<<a[i]<<" ";
  return 0;
  }
```

6.2.3 Anwendung eines eindimesionalen 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?

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

int main(void) {
  int a[] = {0, 1, 2, 4, 3, 5, 6, 7, 8, 9};
  int b[10];
  for (int i=0; i<10; i++)
    b[i]=i;
  for (int i=0; i<10; i++)
    if (a[i]!=b[i])
      cout<<"An Stelle "<<i<<" unterscheiden sich die Vektoren \n";
  return 0;
  }
```

Welche Verbesserungsmöglichkeiten sehen Sie bei dem Programm?

6.2.4 Mehrdimensionale Arrays

Deklaration:

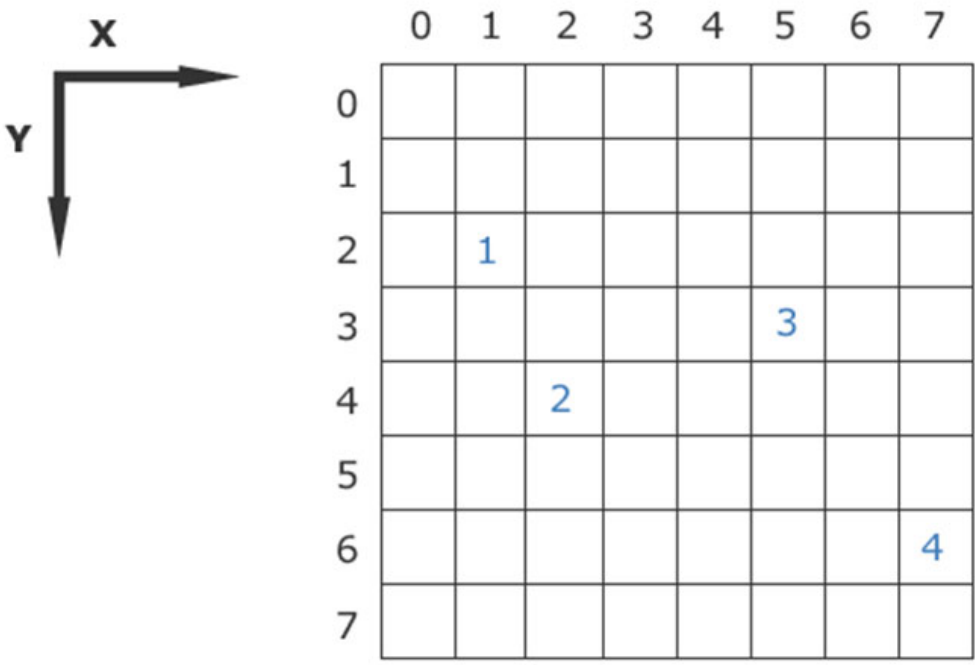
```
int Matrix[4][5];    /* Zweidimensional - 4 Zeilen x 5 Spalten */
```

Deklaration mit einer sofortigen Initialisierung aller bzw. einiger Elemente:

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

```

```
int Matrix[4][4] = { {1,},
                     {1,1},
                     {1,1,1},
                     {1,1,1,1}};

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

Initialisierung eines n-dimensionalen Arrays:




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

int main(void) {
  // Initiallisierung
  int brett[8][8] = {0};
  // Zuweisung
  brett[2][1] = 1;
  brett[4][2] = 2;
  brett[3][5] = 3;
  brett[6][7] = 4;
  // Ausgabe
  int i, j;
  // Schleife fuer Zeilen, Y-Achse
  for(i=0; i<8; i++) {
    // Schleife fuer Spalten, X-Achse
    for(j=0; j<8; j++) {
        cout<<brett[i][j]<<" ";
    }
    cout<<"\n";
  }
  return 0;
}
```

6.2.5 Anwendung eines zweidimesionalen Arrays

Elementwese Addition zweier Matrizen

```
#include <iostream>
```

```
using namespace std;

int main(void)
{
    int C[2][3];
    int i,j;
    for (i=0;i<2;i++)
        for (j=0;j<3;j++)
            C[i][j]=A[i][j]+B[i][j];
    for (i=0;i<2;i++)
    {
        for (j=0;j<3;j++)
            cout<<C[i][j]<<"\t";
        cout<<"\n";
    }
    return 0;
}
```

Weiteres Beispiel: Lösung eines Gleichungssystem mit dem Gausschen Elimnationsverfahren Link

6.3 Sonderfall Zeichenketten / Strings

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

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

int main(void) {
  cout<<"Diese Form eines Strings haben wir bereits mehrfach benutzt!\n";
  ////////////////////////////////////////////////////////////////////////////

  char a[] = "Ich bin ein char Array!";  // Der Compiler fügt das \0 automatisch ein!
  if (a[23] == '\0'){
    cout<<"char Array Abschluss in a gefunden!";
  }

  cout<<"->"<<a<<"<-\n";
  char b[] = { 'H', 'a', 'l', 'l', 'o', ' ',
               'F', 'r', 'e', 'i', 'b', 'e', 'r', 'g', '\0' };
  cout<<"->"<<b<<"<-\n";
  char c[] = "Noch eine \0Moeglichkeit";
  cout<<"->"<<c<<"<-\n";
  char d[] = { 80, 114, 111, 122, 80, 114, 111, 103, 32, 50, 48,  50, 50,  0  };
  cout<<"->"<<d<<"<-\n";
  return 0;
}
```

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

```
#include <iostream>
#include <string>
using namespace std;

int main(void) {
  string hanna = "Hanna";
  string anna = "Anna";
  string alleBeide = anna + " + " + hanna;
```

```
  cout<<"Hallo: "<<alleBeide<<std::endl;

  int res = anna.compare(hanna);

    if (res == 0)
        cout << "\nBoth the input strings are equal." << endl;
    else if(res < 0)
        cout << "String 1 is smaller as compared to String 2\n.";
        else
        cout<<"String 1 is greater as compared to String 2\n.";

  return EXIT_SUCCESS;
}
```

6.4 Grundkonzept Zeiger

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

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

```
Variablen-     Speicher-      Inhalt
name           addresse
                              +----------+
                0000          |          |
                              +----------+
                0001          |          |
                              +----------+
a   ------>     0002       +---| 00001007 | Adresse
                        z |   +----------+
                0003    e |   |          |
                        i |   +----------+
                ....    g |   |          |
                        t |   +----------+
                1005      |   |          |
                        a |   +----------+
                1006    u |   |          |
                        f |   +----------+
b   ------>     1007    <--+   | 00001101 |  Wert = 13
                              +----------+
                1008          |          |
                              +----------+
                ....          |          |                                  .
```

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

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

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

6.4.1 Definition von Zeigern

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

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

6.4.2 Initialisierung

Merke: Zeiger müssen vor der Verwendung initialisiert werden.

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

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

int main(void)
{
  int a = 0;
  int * ptr_a = &a;       /* mit Adressoperator */

  int feld[10];
  int * ptr_feld = feld;  /* mit Array */

  int * ptr_b = ptr_a;    /* mit weiterem Zeiger */

  int * ptr_Null = NULL;  /* mit NULL */

  cout<<"Pointer ptr_a    "<<ptr_a<<"\n";
  cout<<"Pointer ptr_feld "<<ptr_feld<<"\n";
  cout<<"Pointer ptr_b    "<<ptr_b<<"\n";
  cout<<"Pointer ptr_Null "<<ptr_Null<<"\n";
  return 0;
}
```

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

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

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

```
#define NULL 0
#define NULL 0L
#define NULL (void *) 0
```

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

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

int main(void)
{
```

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

6.4.3 Fehlerquellen

Fehlender Adressoperator bei der Zuweisung

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

Fehlender Dereferenzierungsoperator beim Zugriff

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

Uninitialierte Pointer zeigen “irgendwo ins nirgendwo”!

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

6.4.4 Dynamische Datenobjekte

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

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

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

int main(void)
{
  int * ptr_a;
  ptr_a=new int;
  *ptr_a = 10;
  cout<<"Pointer ptr_a                  "<<ptr_a<<"\n";
  cout<<"Wert hinter dem Pointer ptr_a  "<<*ptr_a<<"\n";
  cout<<"Aus Maus!\n";
  delete ptr_a;
  return 0;
}
```

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

int main(void)
{
  int * ptr_a;
  ptr_a=new int[3];
  ptr_a[0] = ptr_a[1] = ptr_a[2] = 42;
  cout<<"Werte hinter dem Pointer ptr_a:  ";
  for (int i=0;i<3;i++) cout<<ptr_a[i]<<" ";
  cout<<"\n";
  cout<<"Aus Maus!\n";
  delete[] ptr_a;
  return 0;
}
```

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

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

6.5 Referenz

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

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

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

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

int main(void)
{
int  a = 1;  // Variable
```

```
int &r = a;  // Referenz auf die Variable a

std::cout << "a: " << a << " r: " << r << std::endl;
std::cout << "a: " << &a << " r: " << &r << std::endl;
}
```

Die Referenzen werden verwendet:

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

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

6.6 Beispiel der Woche

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

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

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

Haben Sie eine bessere Idee?

```
#include <iostream>
using namespace std;
#define ZIELWERT 18

int main(void)
{
  int a[] = {1, 2, 5, 7, 9, 10, 12, 13, 16, 17, 18, 21, 25};
  int i_left=0;
  int i_right=12;
  cout<<"Value left "<<a[i_left]<<" right "<<a[i_right]<<"\n----------------------\n";
  do{
    cout<<"Value left "<<a[i_left]<<" right "<<a[i_right];
    if (a[i_right] + a[i_left] == ZIELWERT){
       cout<<" -> TREFFER";
    }
    cout<<"\n";
    if (a[i_right] + a[i_left] >= ZIELWERT) i_right--;
    else i_left++;
  }while (i_right != i_left);
  return 0;s
```

```
21 }
```