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?

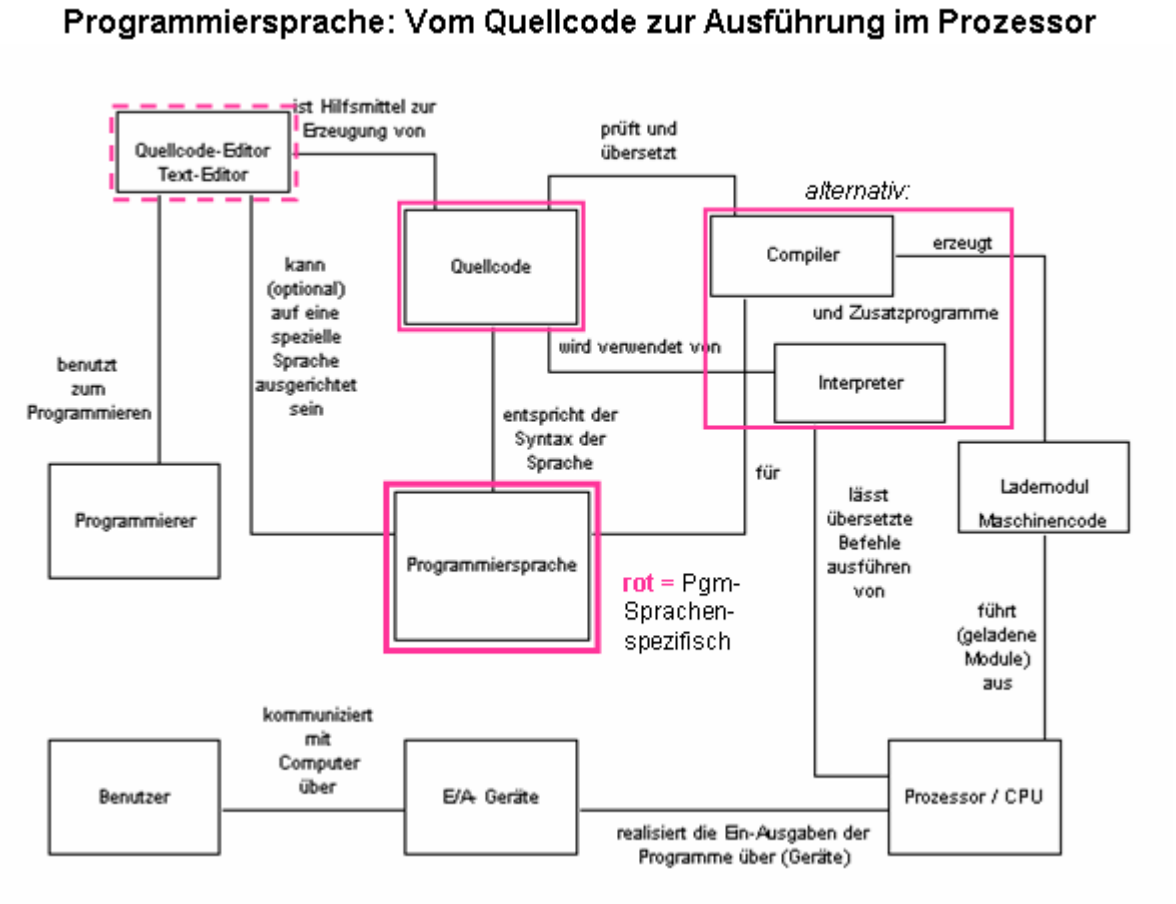
- [(ja)] Ja, ich gehe davon aus, viel nützliches zu erfahren.
- [(schau'n wir mal)] Ich bin noch nicht sicher. Fragen Sie in einigen Wochen noch mal.
- [(nein)] Nein, ich bin nur hier, weil ich muss.

1.2 Wie arbeitet ein Rechner eigentlich?

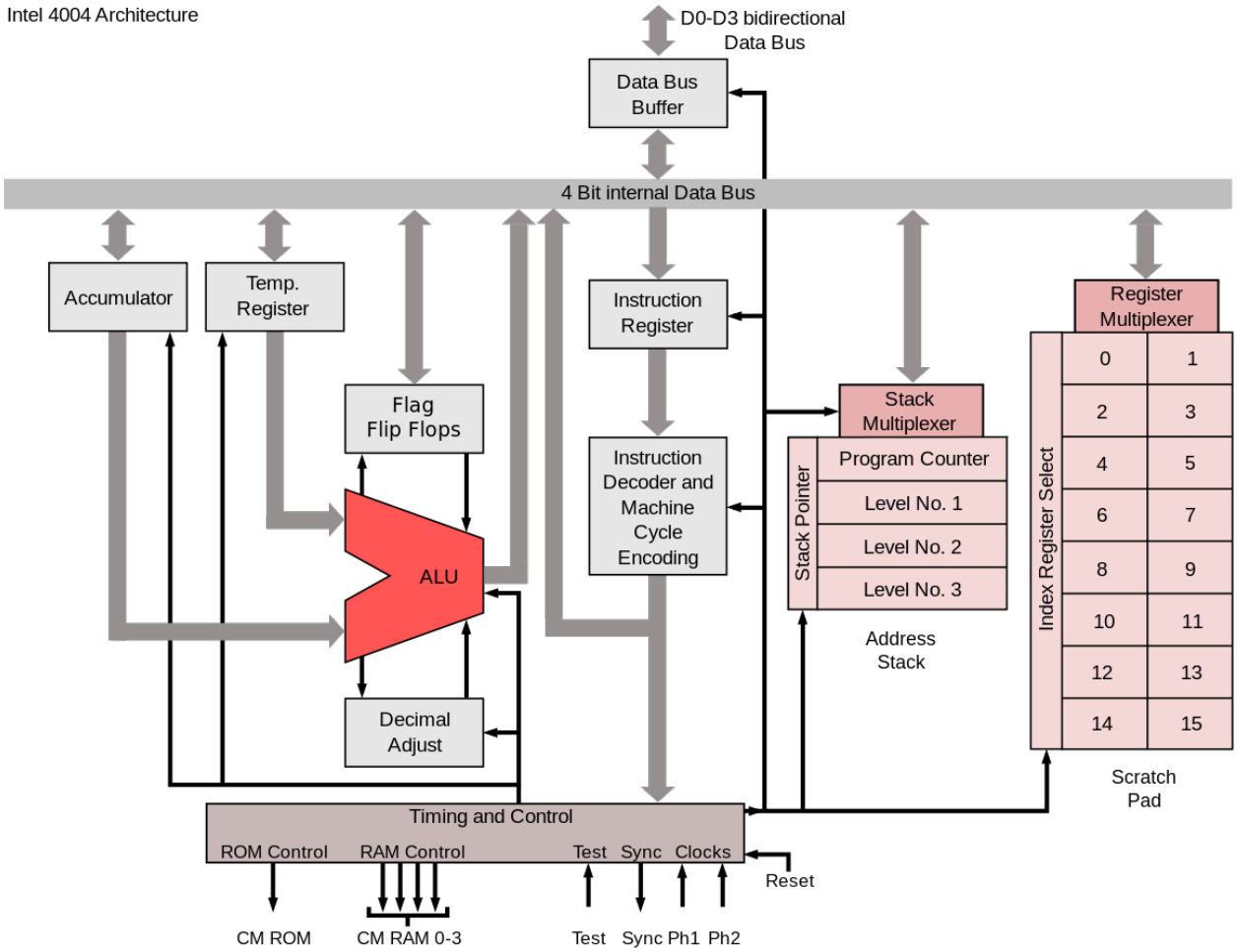
Programme sind Anweisungslisten, die vom Menschen erdacht, auf einem Rechner zur Ausführung kommen. Eine zentrale Hürde ist dabei die Kluft zwischen menschlicher Vorstellungskraft und Logik, die **implizite Annahmen und Erfahrungen** einschließt und der **"stupiden" Abarbeitung von Befehlsfolgen** in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

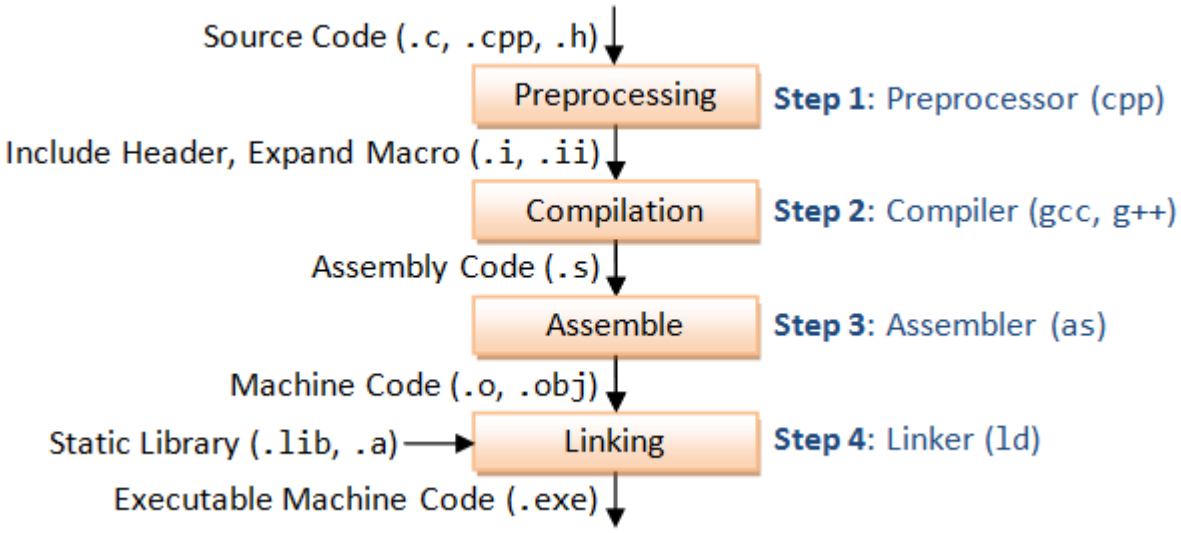
Nachteile:

- 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;
}
```

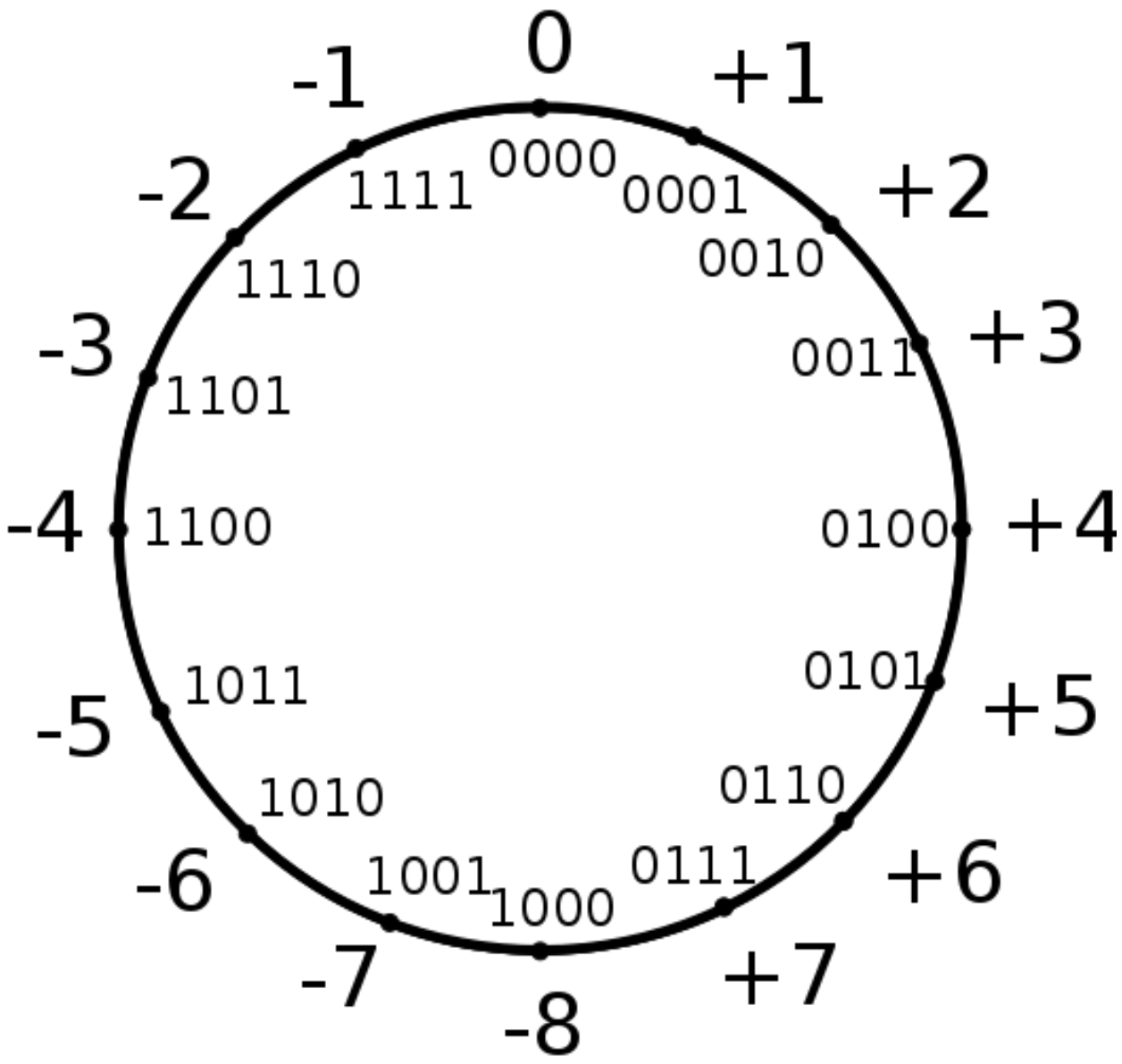
```
#include <iostream>
#include <limits.h>   /* INT_MIN und INT_MAX */

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 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<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 }
```

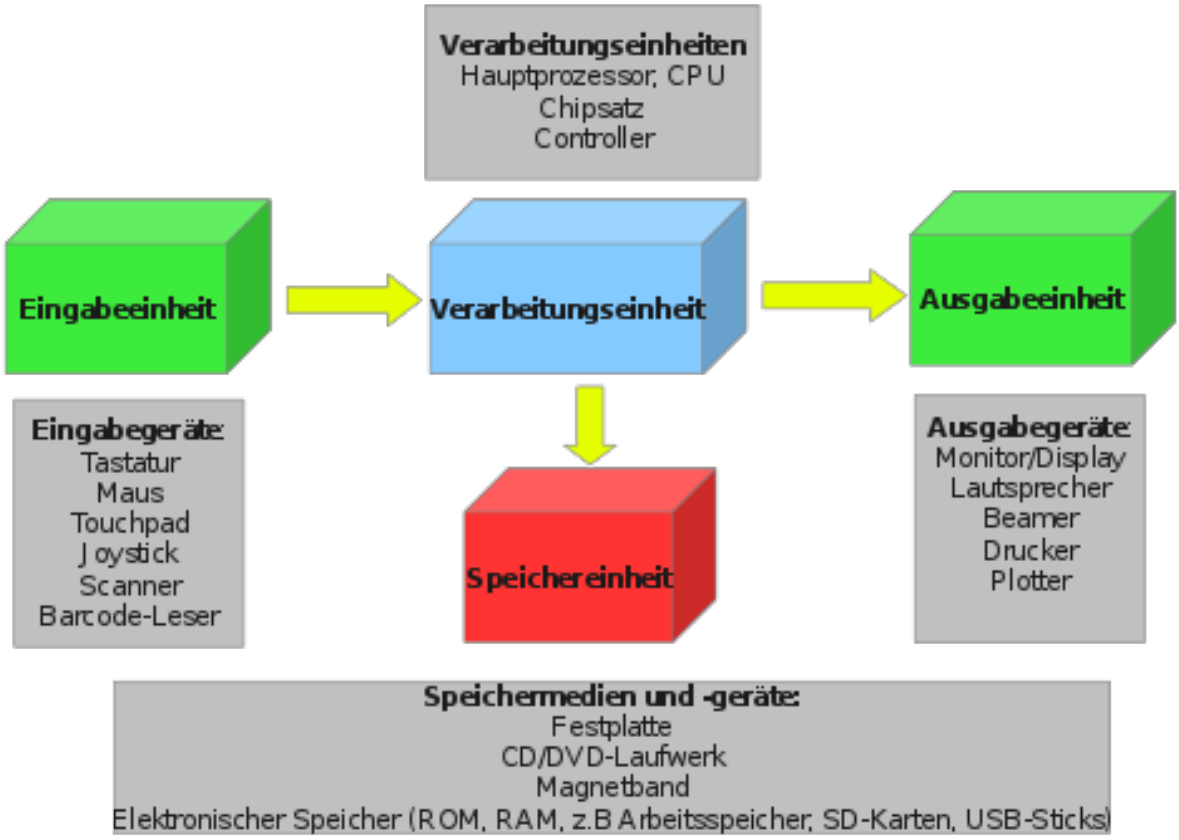
Was passiert wenn der Wert zu groß ist?

```
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 }
```