Grundlagen der Sprache C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Ein- und Ausgabe / Variablen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/01_EingabeAusgabeDatentypen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Copilot

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`

Reflexion Ihrer Fragen / Rückmeldungen

Zur Erinnerung ... Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...

Preis für die aktivste Vorlesung

Format	Informatik Studierende
Verbesserungsvorschlag	0
Fragen	1
generelle Hinweise	0

Gab es Schwierigkeiten beim Anlauf der Übungen?

Variablen

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.

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

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

QuadraticEquation.cpp

```cpp
#include <iostream>

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

  // Variante 2 - Nutzung von Variablen
  int x = 5;
  std::cout << "f(" << x << ") = " << 3 * x * x + 4 * x + 8 << "\n";
  return 0;
}
```

```
f(5) = 103
f(5) = 103
f(5) = 103
f(5) = 103
```

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 (Bezeichner)
2. Datentyp (wie `int`, `float`, etc.)
3. Wert (der gespeicherte Inhalt)
4. Adresse (Position im Speicher)
5. Gültigkeitsraum (Scope)
6. Attribute/Qualifizierer (wie `const`, `volatile`, etc.)

Die ersten fünf Kennzeichen definieren die grundlegende Struktur einer Variable, während die Attribute zusätzliche Eigenschaften und Verhaltensweisen festlegen.
Betrachten wir die verschiedenen Kennzeichen am Beispiel der mathematischen Konstante `e`:

VariableAnatomy.cpp

```cpp
#include <iostream>

int main() {
  // Attribut
  // |    Datentyp
  // |    |    Name
  // |    |    |
  // v    v    v     Wert
  const double e = 2.71828182845905;  // Die Eulersche Zahl

  std::cout << "Der Wert von e lautet " << e << "\n";
  std::cout << "Die Adresse von e lautet " << &e << "\n";

  return 0;
}
```

```
Der Wert von e lautet 2.71828
Die Adresse von e lautet 0x7ffcedcada60
Der Wert von e lautet 2.71828
Die Adresse von e lautet 0x7ffef55055f0
```

Dieses Beispiel demonstriert alle sechs Kennzeichen einer Variable:

- Jede Variable braucht einen **Namen** als Bezeichner
- Der **Datentyp** bestimmt die Interpretation der Bits im Speicher
- Der aktuelle **Wert** wird im Speicher abgelegt
- Die **Adresse** zeigt die Position im Speicher (variiert bei jedem Programmlauf)
- Der **Gültigkeitsraum** definiert, wo auf die Variable zugegriffen werden kann
- **Attribute** wie `const` legen zusätzliche Eigenschaften fest

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)

> 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.

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

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

```
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 Typenspezifikationen (`long` oder `short`) vorhanden sind kann auf die `int` Typangabe verzichtet werden.

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

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

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

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 Buchstabe oder Zeichen gespeichert wird, so garantiert der Standard, dass der gespeicherte Wert der nicht negativen Codierung im **Zeichensatz** entspricht.

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

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

<!--- TODO: Sollten wir hier kurz aufzeigen, was der Unterschied ist, oder reicht das in der Übung? -⟶

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

ASCII Zeichensatz [ASCII]

[ASCII] ASCII-Tabelle

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.

BoolExample.cpp

```cpp
#include <iostream>

int main() {
  bool a = true;
  bool b = false;
  bool c = 45;       // ungleich 0 -> true

  // Numerische Ausgabe (Standard)
  std::cout << "Numerische Darstellung:\n";
  std::cout << "a = " << a << ", b = " << b << ", c = " << c << "\n\n
  // Textuelle Ausgabe mit boolalpha
  std::cout << std::boolalpha;
  std::cout << "Textuelle Darstellung (mit boolalpha):\n";
  std::cout << "a = " << a << ", b = " << b << ", c = " << c << "\n";

  return 0;
}
```

```
Numerische Darstellung:
a = 1, b = 0, c = 1

Textuelle Darstellung (mit boolalpha):
a = true, b = false, c = true
Numerische Darstellung:
a = 1, b = 0, c = 1

Textuelle Darstellung (mit boolalpha):
a = true, b = false, c = true
```

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

`boolalpha` ermöglicht die textuelle Ausgabe von booleschen Werten als "true" oder "false".

Architekturspezifische Ausprägung (Integer Datentypen)

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

sizeof.cpp

```cpp
#include <iostream>

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

```
x umfasst 4 Byte.
x umfasst 4 Byte.
```

sizeof_example.c

```cpp
#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 <
    ;
  std::cout << "char size : " << sizeof(char) << " Byte\n";
  std::cout << "Wertebereich von " << CHAR_MIN << " bis " << CHAR_MAX
    "\n";
  return 0;
}
```

```
int size: 4 Byte
Wertebereich von -2147483648 bis 2147483647
char size : 1 Byte
Wertebereich von -128 bis 127
```

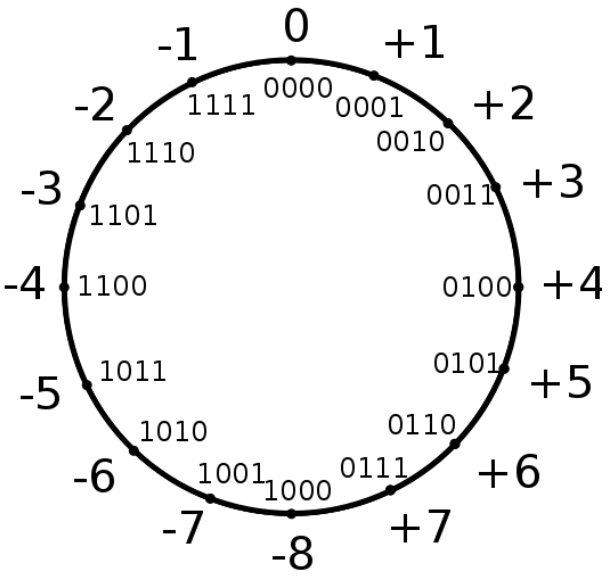
Die implementierungspezifische Werte, wie die Grenzen des Wertebereichs der ganzzahlinen Datentypen sind in `limits.h` definiert, z.B.

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

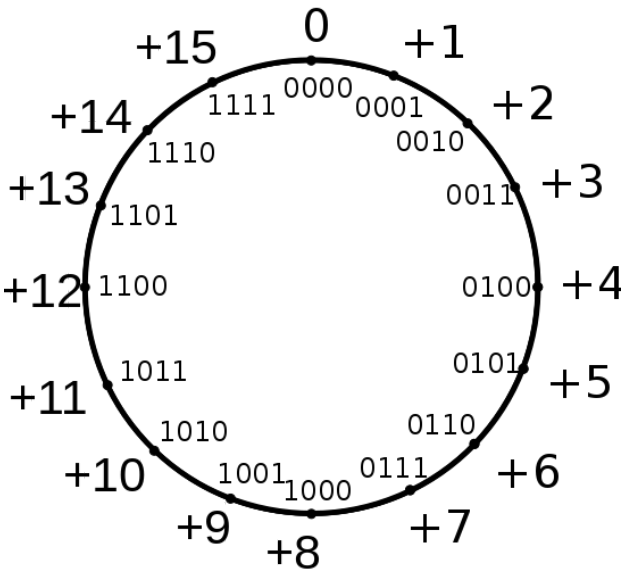
> Warum muss ich soetwas wissen?

Was passiert bei der Überschreitung des Wertebereiches

> Der Arithmetische Überlauf (Integer Overflow) tritt auf, wenn das Ergebnis einer Berechnung den gültigen Zahlenbereich eines Datentyps überschreitet.



4-Bit 2er-Komplement Darstellung



4-Bit Unsigned Darstellung

Quelle: Arithmetischer Überlauf (Autor: WissensDürster)

> 💡 *Das ist wie bei einem Kilometerzähler: Wenn man rückwärts fährt und bei 0 ist, springt er auf 99999.*

Ein berühmtes Beispiel dafür ist der "Nuclear Gandhi"-Bug aus dem Spiel Civilization:

GandhiBitwise.cpp

```cpp
#include <iostream>
#include <bitset>

int main() {
    unsigned char aggression = 1;

    std::cout << "Gandhis Startwert (1):\n";
    std::cout << "Bits:  " << std::bitset<8>(aggression) << "\n";
    std::cout << "Wert:  " << (int)aggression << "\n\n";

    // Wenn eine Zivilisation die Demokratie einführte, sank der Wert
    std::cout << "Demokratie wird eingeführt (-2 Aggression)...\n";
    aggression = aggression - 2;

    // Bei 1 - 2 passiert folgendes auf Bitebene:
    //   00000001    (1)
    // - 00000010    (2)
    // = 11111111    (255, weil unsigned! sonst -1)

    std::cout << "Nach Demokratie (1-2):\n";
    std::cout << "Bits:  " << std::bitset<8>(aggression) << "\n";
    std::cout << "Wert:  " << (int)aggression << "\n";

    return 0;
}
```

```
Gandhis Startwert (1):
Bits:  00000001
Wert:  1

Demokratie wird eingeführt (-2 Aggression)...
Nach Demokratie (1-2):
Bits:  11111111
Wert:  255
Gandhis Startwert (1):
Bits:  00000001
Wert:  1

Demokratie wird eingeführt (-2 Aggression)...
Nach Demokratie (1-2):
Bits:  11111111
Wert:  255
```

Die Erklärung auf Bit-Ebene:

- Ein `unsigned char` verwendet 8 Bits (1 Byte)
- Der Wert 1 wird als `00000001` gespeichert
- Bei der Subtraktion von 2 (`00000010`) würde theoretisch -1 entstehen
- Da es aber keine negativen Zahlen bei `unsigned` gibt, "springt" der Wert auf `11111111`
- `11111111` entspricht dem Wert 255 (höchstmöglicher 8-Bit Wert)

Resultat: Der friedlichste Anführer wurde zum nuklear bewaffneten Kriegstreiber! 😅

Solche Überläufe können in der Praxis zu schwerwiegenden Problemen führen:

- Fehlerhafte Berechnungen
- Sicherheitslücken
- Unerwartetes Programmverhalten

Fließkommazahlen

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

Seien Sie vorsichtig beim Vermischen unterschiedlicher Fließkommatypen (`float` und `double`), da dies zu unerwarteten Ergebnissen führen kann.

float_precision.cpp

```cpp
#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;
}
```

```
float Genauigkeit  :6
double Genauigkeit :15
Ungleich
float Genauigkeit  :6
double Genauigkeit :15
Ungleich
```

Achtung: Fließkommazahlen bringen einen weiteren Faktor mit - die Unsicherheit

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

Hier ein Beispiel, das die Ungenauigkeit bei der Darstellung von 0.1 zeigt:

FloatPrecision.cpp

```cpp
#include <iostream>
#include <iomanip>

int main() {
    float sum = 0.0f;

    // Addiere zehnmal 0.1
    std::cout << std::fixed << std::setprecision(20);
    std::cout << "Erwartete Werte vs. tatsächliche Werte:\n\n";

    for(int i = 1; i <= 10; i++) {
        sum += 0.1f;
    }

    // Vergleich mit 1.0 zeigt das Problem
    std::cout << "Ist:      " << sum << "\n\n";

    return 0;
}
```

```
Erwartete Werte vs. tatsächliche Werte:

Ist:      1.00000011920928955078

Erwartete Werte vs. tatsächliche Werte:

Ist:      1.00000011920928955078
```

Der Grund dafür ist, dass 0.1 im Binärsystem eine unendliche Folge ist: $0.1_{10} = 0.00011001100110011\ldots_2$

Diese Zahl kann nicht exakt in einem `float` oder `double` gespeichert werden, während Zweierpotenzen wie 0.125 (0.001_2) eine endliche Darstellung haben. Probieren Sie es im obigen Beispiel aus, indem Sie 0.125 anstelle von 0.1 verwenden.

Datentyp `void`

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

Anwendungsbeispiele:

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

```
int main(void) {
  //Anweisungen
  return 0;
}
```

```
void funktion(void) {
  //Anweisungen
}
```

Wertspezifikation

In C++ gibt es verschiedene Möglichkeiten, Zahlen zu schreiben. Hier sind die wichtigsten Schreibweisen für **dezimale** Zahlen, die Sie kennen sollten:

1. **Normale Ganzzahlen:**

```
int a = 42;         // Positive Ganzzahl
int b = -42;        // Negative Ganzzahl
int c = 011;        // !!! Oktale Schreibweise (entspricht 9 im Dezimal
  )
```

2. **Kommazahlen:**

```
double d1 = 3.14;       // Normale Schreibweise
double d2 = 0.234;      // Mit führender Null
double d3 = 123.0e-2;   // Wissenschaftliche Notation (= 1.23)
float  f1 = 1.23f;      // Als float (durch 'f' am Ende)
```

3. **Große Zahlen lesbarer machen:**

```
// Seit C++14: Zifferngruppierung mit Unterstrich
int million = 1'000'000;  // Besser lesbar als 1000000
```

Merke: Die Endung `f` bei Kommazahlen ist wichtig! Ohne sie interpretiert C++ die Zahl als `double`, was zu Genauigkeitsverlusten führen kann.

Ein praktisches Beispiel:

NumberFormats.cpp

```cpp
#include <iostream>

int main() {
    // Gängige Schreibweisen für verschiedene Anwendungen
    double preis = 19.99;               // Preisangabe
    float  temp = 36.5f;                // Temperatur
    int    anzahl = 1'000'000;          // Große Zahl
    double mikro = 0.000001;            // Sehr kleine Zahl
    double mikro2 = 1.0e-6;             // Gleicher Wert, andere
      Schreibweise

    std::cout << "Preis: " << preis << " EUR\n";
    std::cout << "Temperatur: " << temp << " °C\n";
    std::cout << "Anzahl: " << anzahl << "\n";
    std::cout << "Mikrometer: " << mikro << " m\n";
    std::cout << "Wissenschaftlich: " << mikro2 << " m\n";

    return 0;
}
```

```
Preis: 19.99 EUR
Temperatur: 36.5 °C
Anzahl: 1000000
Mikrometer: 1e-06 m
Wissenschaftlich: 1e-06 m
Preis: 19.99 EUR
Temperatur: 36.5 °C
Anzahl: 1000000
Mikrometer: 1e-06 m
Wissenschaftlich: 1e-06 m
```

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 erlangen, kann man den Operator `&` nutzen.

Pointer.cpp

```cpp
#include <iostream>

int main(void) {
  int x = 020;
  std::cout << &x << "\n";
  return 0;
}
```

```
0x7ffc94c04ad4
0x7ffd55872484
```

Hinweis: Die Ausgabe der Adresse erfolgt in der Regel im Hexadezimalformat. Mit Adressen werden wir uns im Zusammenhang mit Arrays und Zeigern intensiv beschäftigen.

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:

visibility.cpp

```cpp
#include <iostream>

int main(void) {
  int v = 1;
  int w = 5;
  {
    int v;
    v = 2;
    std::cout << v << "\n";
    std::cout << w << "\n";
  }
  std::cout << v << "\n";
  return 0;
}
```

```
2
5
1
2
5
1
```

Globale Variablen

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

visibility.cpp

```cpp
#include <iostream>

int v = 1; /*globale Variable*/

int main(void) {
  //int v = 5;
  std::cout << v << "\n";
  std::cout << ::v << "\n"; // Zugriff auf die globale Variable
  return 0;
}
```

```
1
1
1
1
```

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

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!

DeclarationVSDefinition.cpp

```
extern int a;       // Deklaration
int i;              // Definition + Deklaration
int a,b,c;
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.

Typische Fehler

Fehlende Initialisierung

MissingInitialisation.cpp

```
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  }
```

```
main.cpp: In function ‘int main()’:
main.cpp:7:29: warning: ‘y’ is used uninitialized [-Wuninitialized]
    7 |   std::cout << "y=" << y << "\n";
      |                             ^~~~

main.cpp: In function ‘int main()’:
main.cpp:7:29: warning: ‘y’ is used uninitialized [-Wuninitialized]
    7 |   std::cout << "y=" << y << "\n";
      |                             ^~~~

x=5
y=0
x=5
y=0
```

Redeklaration

Redeclaration.cpp

```cpp
#include <iostream>

int main(void) {
  int x;
  int x;
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:5:7: error: redeclaration of ‘int x’
    5 |   int x;
      |       ^
main.cpp:4:7: note: ‘int x’ previously declared here
    4 |   int x;
      |       ^
main.cpp:4:7: warning: unused variable ‘x’ [-Wunused-variable]
main.cpp: In function ‘int main()’:
main.cpp:5:7: error: redeclaration of ‘int x’
    5 |   int x;
      |       ^
main.cpp:4:7: note: ‘int x’ previously declared here
    4 |   int x;
      |       ^
main.cpp:4:7: warning: unused variable ‘x’ [-Wunused-variable]
```

Falsche Zahlenliterale

wrong_float.cpp

```cpp
#include <iostream>

int main(void) {
  float a = 1,5;   /* FALSCH  */
  float b = 1.5;   /* RICHTIG */
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:4:15: error: expected unqualified-id before numeric constant
    4 |   float a = 1,5;   /* FALSCH  */
      |               ^
main.cpp:4:9: warning: unused variable ‘a’ [-Wunused-variable]
    4 |   float a = 1,5;   /* FALSCH  */
      |         ^
main.cpp:5:9: warning: unused variable ‘b’ [-Wunused-variable]
    5 |   float b = 1.5;   /* RICHTIG */
      |         ^
main.cpp: In function ‘int main()’:
main.cpp:4:15: error: expected unqualified-id before numeric constant
    4 |   float a = 1,5;   /* FALSCH  */
      |               ^
main.cpp:4:9: warning: unused variable ‘a’ [-Wunused-variable]
    4 |   float a = 1,5;   /* FALSCH  */
      |         ^
main.cpp:5:9: warning: unused variable ‘b’ [-Wunused-variable]
    5 |   float b = 1.5;   /* RICHTIG */
      |         ^
```

Was passiert wenn der Wert zu groß ist?

TooLarge.cpp

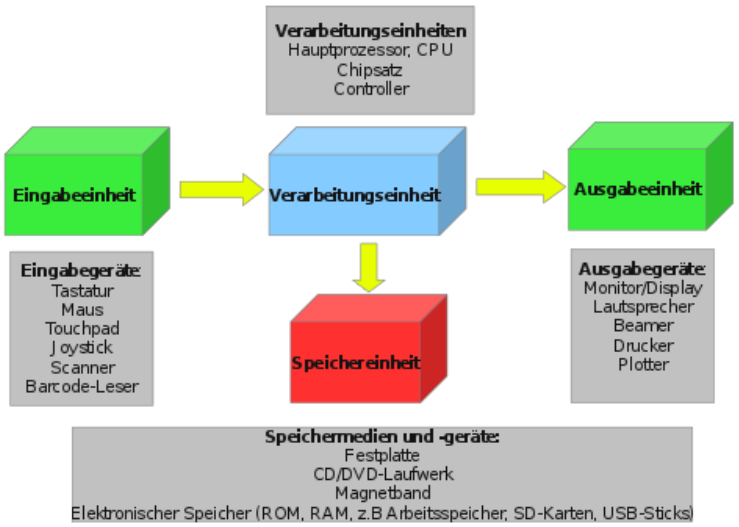
```cpp
#include <iostream>

int main(void) {
  short a;
  // 65535 ist die größte Zahl, die in 2 Byte passt
  a = 65535 + 2;
  std::cout << "Schaun wir mal ... " << a << "\n";
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:6:13: warning: overflow in conversion from ‘int’ to ‘short
int’ changes value from ‘65537’ to ‘1’ [-Woverflow]
    6 |   a = 65535 + 2;
      |       ~~~~~~^~~

main.cpp: In function ‘int main()’:
main.cpp:6:13: warning: overflow in conversion from ‘int’ to ‘short
int’ changes value from ‘65537’ to ‘1’ [-Woverflow]
    6 |   a = 65535 + 2;
      |       ~~~~~~^~~

Schaun wir mal ... 1
Schaun wir mal ... 1
```

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 einem 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.

iostream.cpp

```cpp
#include <iostream>

int main(void) {
  char hanna[] = "Hanna";
  char anna[] = "Anna";
  std::cout << "C++ stream: " << "Hallo "<< hanna << ", " << anna << "
  return 0;
}
```

```
C++ stream: Hallo Hanna, Anna
C++ stream: Hallo Hanna, Anna
```

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.

```cpp
std::cout << 55 << "55" << 55.5 << true << "\n";
```

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
`std::fixed`	Dezimaldarstellung (ohne wissenschaftliches Format) erzwingen

manipulatoren1.c

```
#include <iostream>
#include <iomanip>

int main() {
  std::cout << std::setbase(16) << std::fixed << 55 << "\n";
  std::cout << std::setbase(10) << std::fixed << 55 << "\n";
  return 0;
}
```

```
37
55
37
55
```

Achtung: Die Manipulatoren wirken auf alle daruf folgenden Ausgaben.

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.

manipulatoren2.c

```
#include <iostream>
#include <iomanip>
int main() {
  std::cout << std::right << std::setw(5) << 55 << "\n";
  std::cout << std::right << std::setfill('0') << std::setw(5) << 55 <
    ;
  std::cout << std::left << std::fixed << std::setw(15) << 0.000000023
    << "\n";
  return 0;
}
```

```
   55
00055
0.0000000000000
   55
00055
0.0000000000000
```

Genauigkeit

genauigkeit.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <math.h>

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

```
3.141592653590
3.141592654
3.141593
3.142
```

Escape-Sequenzen

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

esc_sequences.c

```c
#include <iostream>

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

```
123456789ABCD

Vorname          Name            Alter
Andreas          Mustermann      42

Manchmal braucht man auch ein "\"
123456789ABCD

Vorname          Name            Alter
Andreas          Mustermann      42

Manchmal braucht man auch ein "\"
```

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.

istream.cpp

```cpp
#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;
}
```

```
Bitte Werte eingeben [char float int] : Bitte Werte eingeben [char
float int] :
```

Beispiel der Woche

Implementieren Sie einen programmierbaren Taschenrechner für quadaratische Funktionen.

QuadraticEquation.cpp

```cpp
#include <iostream>

int main() {
  // Variante 3 - verglichen mit dem Anfang der Vorlesung
  int x;
  std::cin >> x;
  std::cout << "f(" << x << ") = " << 3 * x * x + 4 * x + 8 << "\n";
  return 0;
}
```

Quiz

Variablennamen

Welche dieser Variablennamen sind grundsätzlich zulässig?

- [ ] geschwindigkeit
- [ ] hasjdLASJdssa
- [ ] speed
- [ ] speed of robot
- [ ] sp33d
- [ ] 99Speed
- [ ] speed_of_triangle
- [ ] _speed
- [ ] speed.forwards
- [ ] int
- [ ] speedOfRobot

Datentypen

Ordnen Sie die Datentypen die korrekten Zahlentypen zu.

Ganzzahl	Fließkommazahl	
◯	◯	int
◯	◯	float
◯	◯	double
◯	◯	bool
◯	◯	char

Boolean

Welche Werte können `bool`-Variablen zugewiesen werden?

- ◯ 0 und 1
- ◯ 0 bis 1

Fließkommazahlen

Welche dieser Zahlen kann präzise im Speicher abgebildet werden?

- ◯ 0.3
- ◯ 0.125
- ◯ 0.111
- ◯ 0.783
- ◯ 0.420

Adressen

Mit welchem Symbol kann auf die Speicheradresse einer Variable zugegriffen werden?

Globale und lokale Variablen

Wählen Sie aus, welche Variablen global und welche lokal sind.

```
#include <iostream>

int w = 5;

int main(void)
{
  int v = 1;
  {
    int v;
    v = 2;
    std::cout << v << "\n";
    std::cout << w << "\n";
  }
  std::cout << v << "\n";
  return 0;
}
```

Global	Lokal	
◯	◯	v
◯	◯	w

Definition, Deklaration und Initialisierung

Wählen Sie aus in welchen Fällen eine Deklaration, Definition oder Initialisierung vorliegt.

Deklaration	Definition	Initialisierung	
☐	☐	☐	int i;
☐	☐	☐	int i = 99;
☐	☐	☐	double d;

Escape-Sequenzen

Wie lautet die Escape-Sequenz für BACKSPACE?

Wie lautet die Escape-Sequenz für NEWLINE?

Wie lautet die Escape-Sequenz für HORIZONTAL TAB?

Wie lautet die Escape-Sequenz für SINGLE QUOTATION MARK?

Wie lautet die Escape-Sequenz für DOUBLE QUOTATION MARK?

Wie lautet die Escape-Sequenz für CARRIAGE RETURN?