Einführung in die Programmierung

Vorlesungsteil 2
Elementare Datentypen, Variablen, Arithmetik, Typecast

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

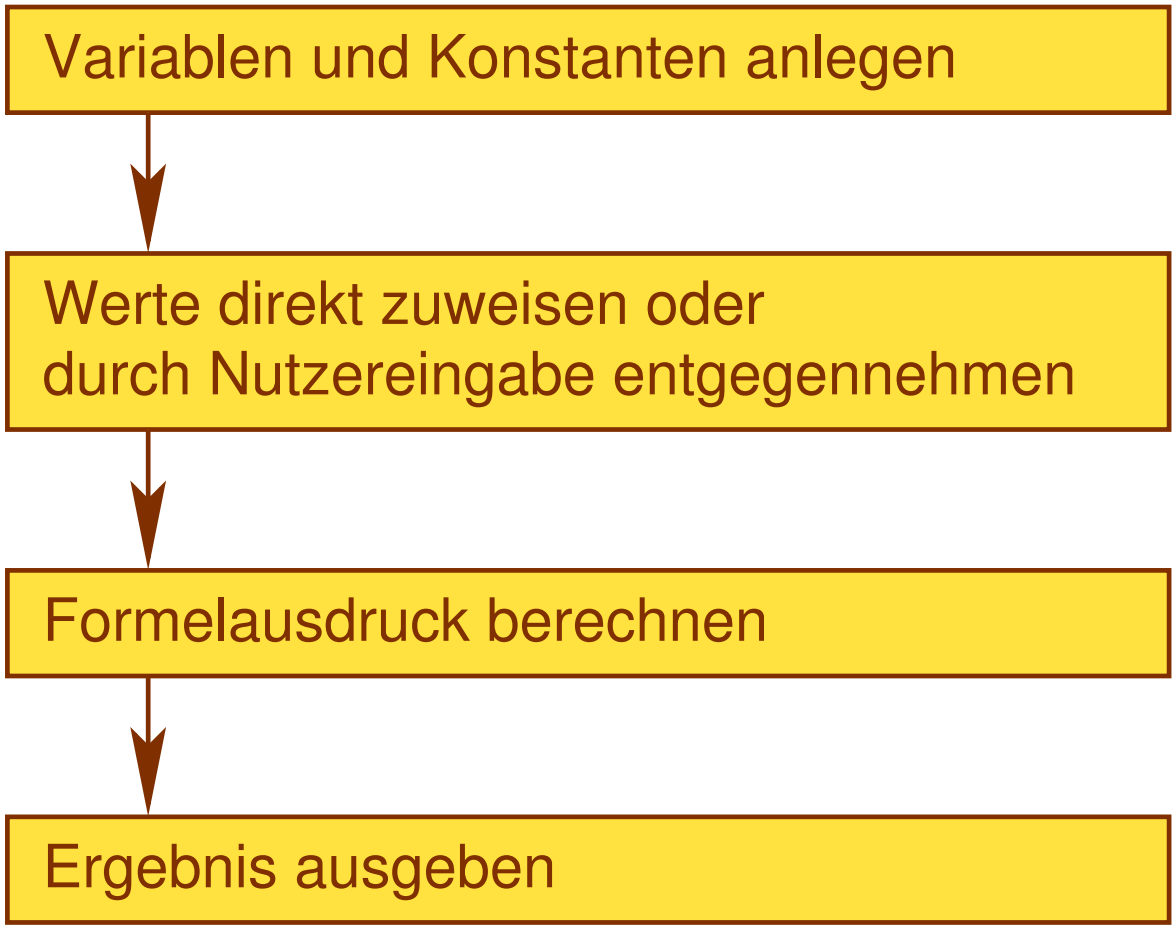
Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Heute lernen wir, wie man in C Programme zur Berechnung arithmetischer Ausdrücke schreibt.

(Bild: www.lehrfilme.eu)

Programmablauf

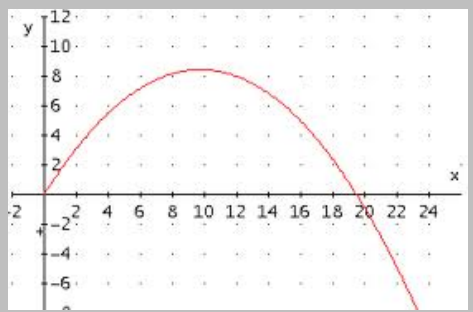


Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Beispiel: Schräger Wurf vom Boden zum Boden

Idealisiert als Wurfparabel betrachtet



$$\text{wurfweite} = \frac{v_0^2}{g} \cdot \sin(2 \cdot \phi)$$

v_0: Anfangsgeschwindigkeit
ϕ: Abwurfwinkel
g: Erdbeschleunigung

Beispiel: Schräger Wurf – C-Quelltext

schraeger-wurf-demo.c

```c
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main(void)
{
  const double G = 9.81; //Erdbeschleunigung
  double winkel = 45;    //Abwurfwinkel in Grad
  double v0;             //Anfangsgeschwindigkeit

  double phi = winkel * PI / 180; //Winkel im Bogenmasz (radiant)
  double weite;

  printf("Schraeger Wurf\n\nAnfangsgeschwindigkeit m/s: ");
  scanf("%lf", &v0);     //Nutzereingabe von v0

  weite = v0 * v0 / G * sin(2 * phi);

  printf("Wurfweite bei Abwurfwinkel %lf Grad in m: %lf\n", winkel, weite);

  return 0;
}
```

Compilieren mit `gcc schraeger-wurf-demo.c -lm`

Alle 32 Schlüsselwörter von C

C als besonders kompakte Programmiersprache

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

im ANSI-Standard C90

ergänzt durch vorbelegte Bezeichner (z.B. `main`), Bibliotheksfunktionen (z.B. `printf`) und Operatoren (z.B. `+` und `<`)

Alle 32 Schlüsselwörter von C

Heute lernen wir davon kennen . . .

`auto`	**`double`**	**`int`**	`struct`
`break`	`else`	**`long`**	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	**`float`**	**`short`**	**`unsigned`**
`continue`	`for`	**`signed`**	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

im ANSI-Standard C90

ergänzt durch vorbelegte Bezeichner (z.B. `main`), Bibliotheksfunktionen (z.B. **`printf`**) und Operatoren (z.B. **`+`** und **`<`**)

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ . . . . . Menge der natürlichen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ . . . . . Menge der natürlichen Zahlen
  $\mathbb{R}$ . . . . . Menge der reellen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll
  $<$ ..... Vergleich auf „kleiner“ nicht zwischen komplexen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll
  $<$ ..... Vergleich auf „kleiner“ nicht zwischen komplexen Zahlen
  $-$ ..... Subtraktion $3 - 5$ nicht in $\mathbb{N}$, aber auf ganzen Zahlen

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“

Der Sinn verschiedener Zahlenbereiche

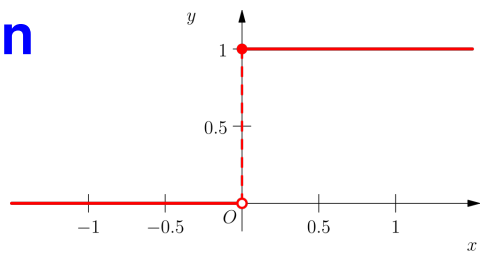
- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?
- **Beschreibungen so einfach wie möglich halten, um (Anwendungs)Fehlern entgegenzuwirken**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?
- **Beschreibungen so einfach wie möglich halten, um (Anwendungs)Fehlern entgegenzuwirken**
  - Schwellenwertfunktion



In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `ü`, `$` und viele mehr bildet einen Typ.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten können ebenfalls einen Typ bilden.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten können ebenfalls einen Typ bilden.
- Um einen Wert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten können ebenfalls einen Typ bilden.
- Um einen Wert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.
- Es gilt, sinnvolle Kompromisse zwischen *Speicherplatzbedarf*, *Wertevorrat* und *Aufwand* zur Ausführung der Operationen zu finden.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten können ebenfalls einen Typ bilden.
- Um einen Wert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.
- Es gilt, sinnvolle Kompromisse zwischen *Speicherplatzbedarf*, *Wertevorrat* und *Aufwand* zur Ausführung der Operationen zu finden.
- Aus diesen Überlegungen resultiert eine Vielzahl *elementarer Datentypen* in C.

Grundbegriffe

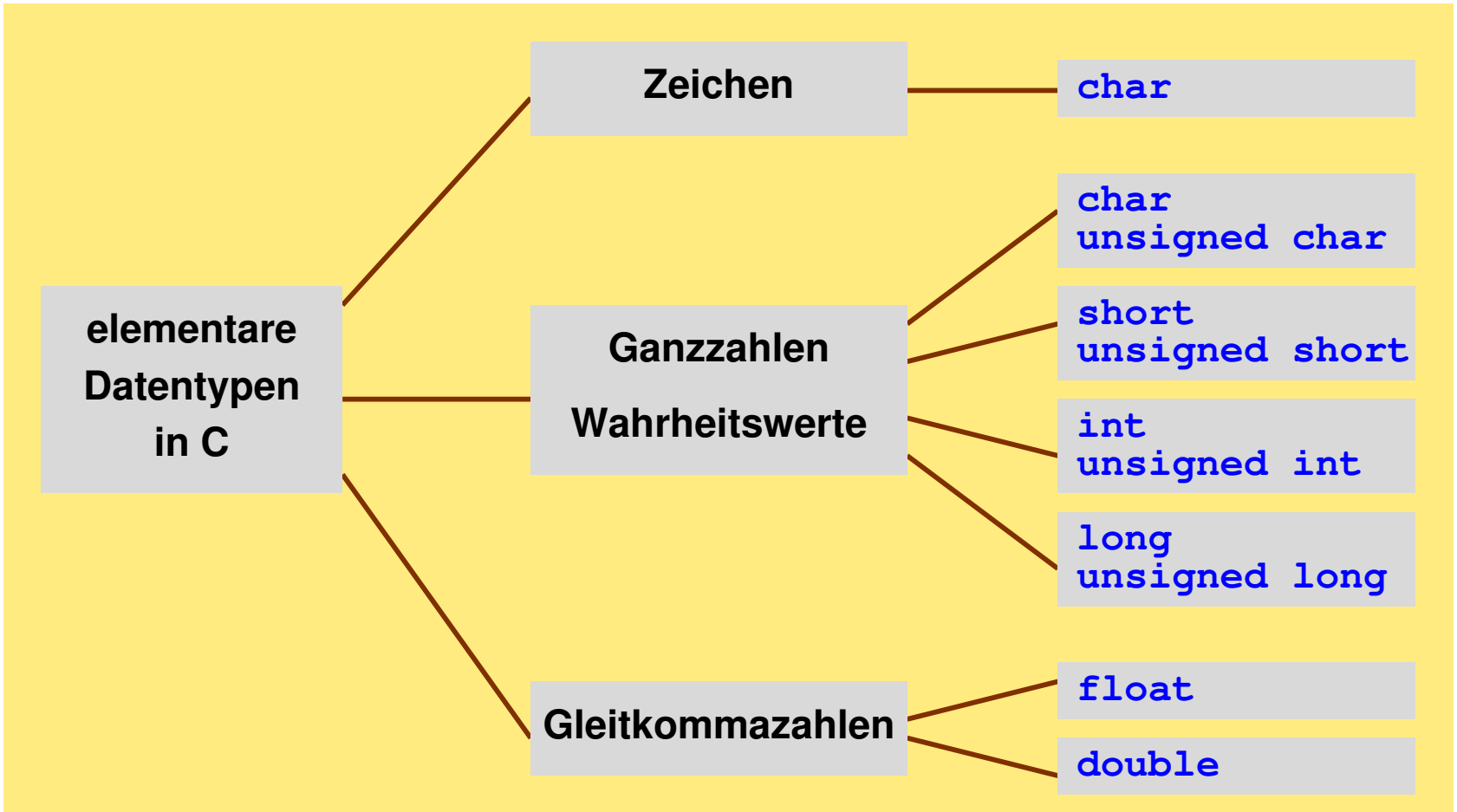
Werte sind im Allgemeinen klassische algebraische Elemente wie *Zahlen*, *Zeichen*, *Symbole* und *Zeichenketten (Strings)*.

Grundbegriffe

Werte sind im Allgemeinen klassische algebraische Elemente wie *Zahlen*, *Zeichen*, *Symbole* und *Zeichenketten (Strings)*.

Typen bezeichnen eine *Menge gleichartiger Werte*. Für jeden Typ sind bestimmte Operationen definiert. Zudem legt der Typ einheitlich fest, wie jeder seiner Werte als Bitmuster kodiert im Speicher abgelegt wird und wieviel Speicherplatz dafür nötig ist.

Übersicht der elementaren Datentypen in C



Elementare Datentypen kennenlernen

Zu elementaren Datentypen schauen wir uns im Folgenden an:

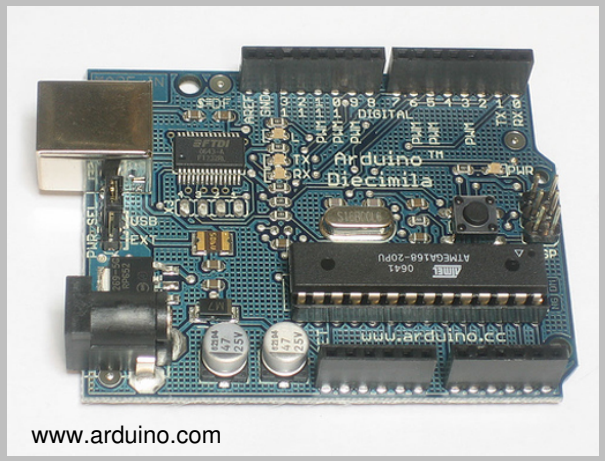
- Wie sehen *Werte* dieses Typs aus?
- Wie legt man *Variablen* für Werte dieses Typs an?
- Wieviel *Speicherplatz* belegt jeder Wert dieses Typs?
- Welchen *Wertebereich* besitzt dieser Typ?
- Welche *Operationen* sind für diesen Typ vordefiniert?
- Wie werden Werte dieses Typs als *Bitmuster* kodiert?
- Gibt es Besonderheiten und falls ja, welche?

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

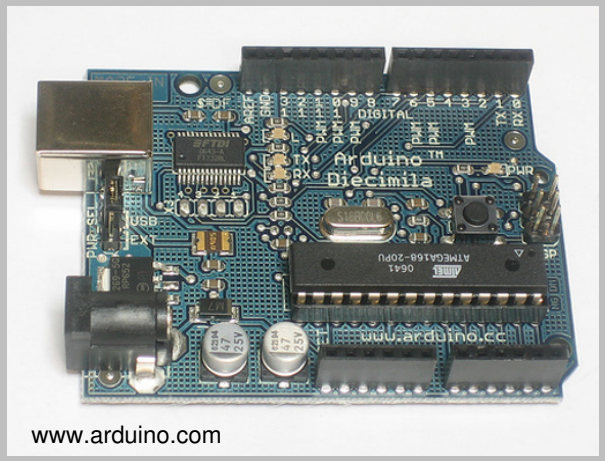


Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.



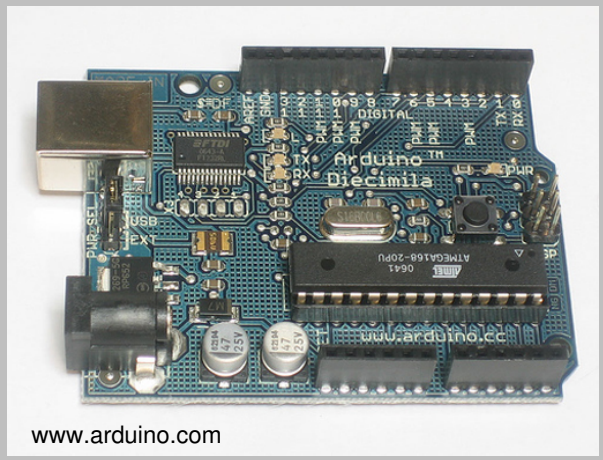
Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1

```
int v_max = messwert_maximalgeschw();
float a_max = messwert_max_insassenbeschleunigung();
char bremsstufe = messwert_bremsstufe();
```

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

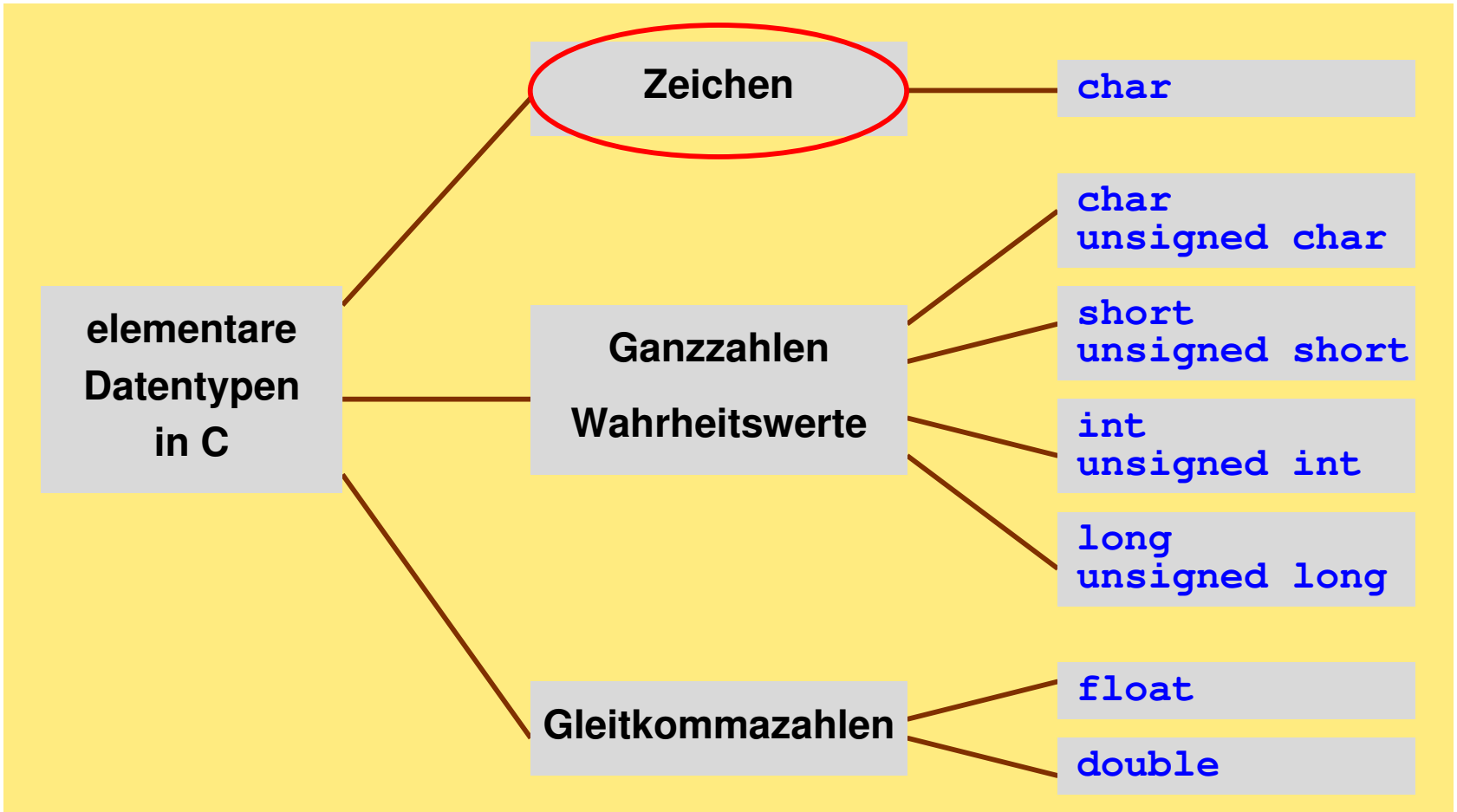


Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1			
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1	1	1	0	0	1	0			
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1			

```
int v_max = messwert_maximalgeschw();
float a_max = messwert_max_insassenbeschleunigung();
char bremsstufe = messwert_bremsstufe();
```

Übersicht der elementaren Datentypen in C



Zeichen

```
char zeichen = 'a';
```

- 1 Byte (8 Bit)
- $2^8 = 256$ Werte unterscheidbar
- Zeichenzuordnung über ASCII-Tabelle

	32	0	48	@	64	P	80	`	96	p	112
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	<	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

American Standard Code for Information Interchange (ASCII)

Zeichen

```
char zeichen = 'a';
```

- 1 Byte (8 Bit)
- $2^8 = 256$ Werte unterscheidbar
- Zeichenzuordnung über ASCII-Tabelle
- Literale immer in ' '

	32	0	48	@	64	P	80	`	96	p	112
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	<	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

American Standard Code for Information Interchange (ASCII)

Zeichen

```
char zeichen = 'a';
```

- 1 Byte (8 Bit)
- $2^8 = 256$ Werte unterscheidbar
- Zeichenzuordnung über ASCII-Tabelle
- Literale immer in `'  '`
- sinnvolle Operatoren: `==`, `!=`, `<`, `>`, `<=`, `>=`
- mit Zeichen(kodierungen) kann wie mit ganzen Zahlen gerechnet werden
- Zeichenketten später

	32	0	48	@	64	P	80	`	96	p	112
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	<	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

American Standard Code for Information Interchange (ASCII)

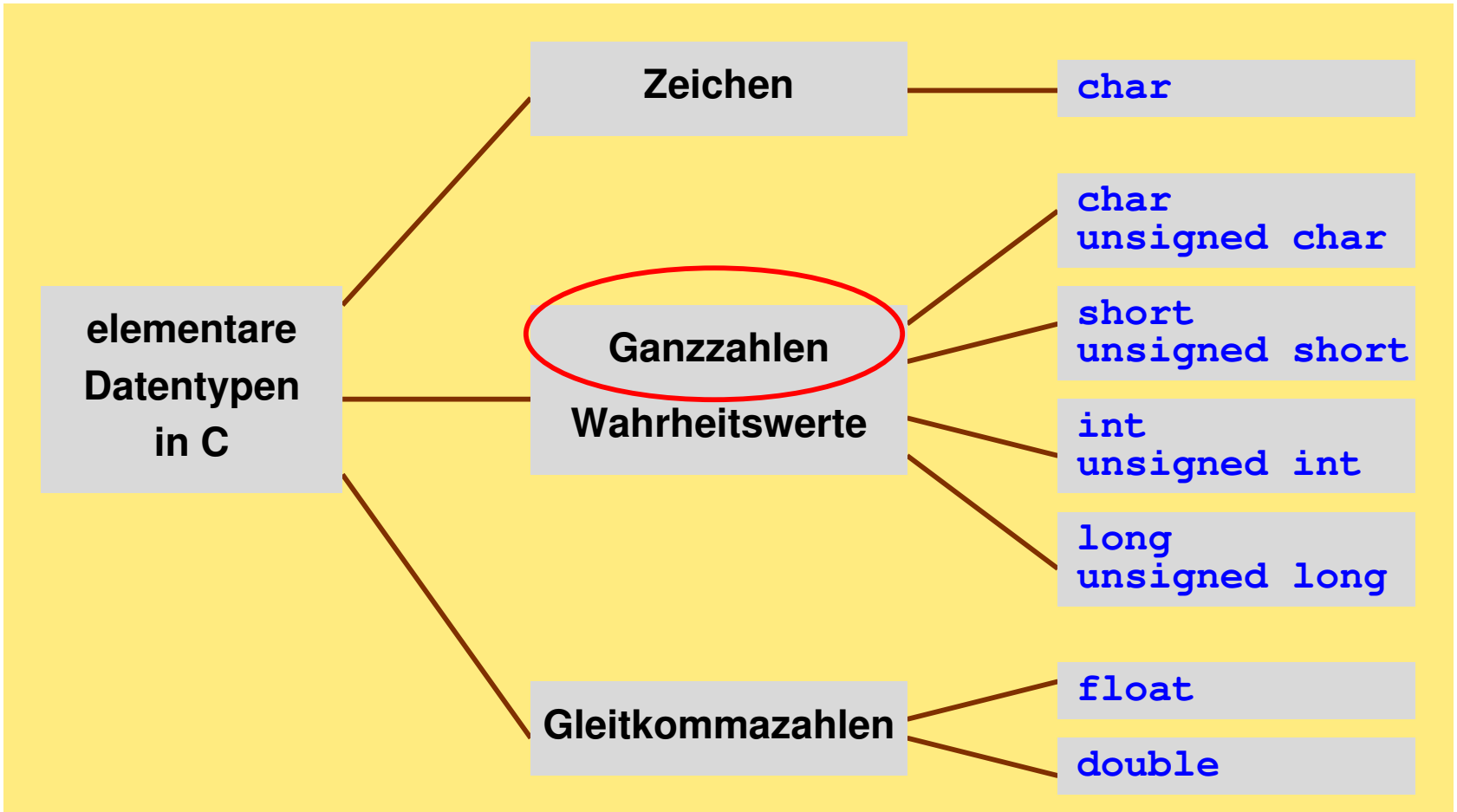
Nicht druckbare Zeichen (Escape-Sequenzen)

Nonprinting or hard-to-print characters		
Name of character	**Written in C**	**ASCII Value**
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
form feed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11

- Escape character (\)
- Escape sequence (e.g. `\n`)

$\Longrightarrow$ dürfen in `printf`-Ausgaben verwendet werden

Übersicht der elementaren Datentypen in C



Ganzzahlige Datentypen

Typ	Größe in Byte	Wertebereich `signed`	Wertebereich `unsigned`
`char`	1	$-128 \ldots 127$	$0 \ldots 255$
`short`	2	$-32768 \ldots 32767$	$0 \ldots 65535$
`int`	4	$-2^{31} \ldots 2^{31}-1$	$0 \ldots 2^{32}-1$
`long`	8*	$-2^{63} \ldots 2^{63}-1$	$0 \ldots 2^{64}-1$

*: auf 64-Bit-Architektur, sonst 4 Byte

Größeneinheiten für Datenspeicherung

1K	2^{10}	1 024	Kilo	griech. „tausend“
1M	2^{20}	1 048 576	Mega	gr. „groß“
1G	2^{30}	1 073 741 824	Giga	gr. „riesig“
1T	2^{40}	1 099 511 627 776	Tera	gr. „ungeheuerlich“

Ganzzahlige Datentypen

```
unsigned long anzahl = 1234567890;
```

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen . . . 42
  - **Hexadezimal** (Basis 16): `0x` voranstellen . . . . . `0x1a` für 26
  - **Oktal** (Basis 8): `0` voranstellen . . . . . `022` für 18 (!!!)

Ganzzahlige Datentypen

```
unsigned long anzahl = 1234567890;
```

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen ... 42
  - **Hexadezimal** (Basis 16): `0x` voranstellen ..... `0x1a` für 26
  - **Oktal** (Basis 8): `0` voranstellen ..... `022` für 18 (!!!)
- **Operatoren**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`, `++`, `--`, `&&`, `||`, `!`
  - modulo-Operator `%` liefert den Rest bei ganzzahliger Division, z.B. `7 % 4` ist `3`

Ganzzahlige Datentypen

```
unsigned long anzahl = 1234567890;
```

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen . . . 42
  - **Hexadezimal** (Basis 16): `0x` voranstellen . . . . . `0x1a` für 26
  - **Oktal** (Basis 8): `0` voranstellen . . . . . `022` für 18 (!!!)
- **Operatoren**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`, `++`, `--`, `&&`, `||`, `!`
  - modulo-Operator `%` liefert den Rest bei ganzzahliger Division, z.B. `7 % 4` ist `3`
- **Bitmuster-Kodierung im Speicher**
  - *Zweierkomplement* (entspricht bei Ganzzahlen ≥ 0 der *einfachen Binärdarstellung*, für Ganzzahlen < 0 beginnt das Bitmuster stets mit 1)

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

Umrechnung dezimal $\longrightarrow$ binär

$$
\begin{array}{rcrl}
41 & :2 & = & 20 \quad \text{Rest } \mathbf{1} \\
20 & :2 & = & 10 \quad \text{Rest } \mathbf{0} \\
10 & :2 & = & 5 \quad \text{Rest } \mathbf{0} \\
5 & :2 & = & 2 \quad \text{Rest } \mathbf{1} \\
2 & :2 & = & 1 \quad \text{Rest } \mathbf{0} \\
1 & :2 & = & 0 \quad \text{Rest } \mathbf{1}
\end{array}
\Bigg\uparrow
$$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

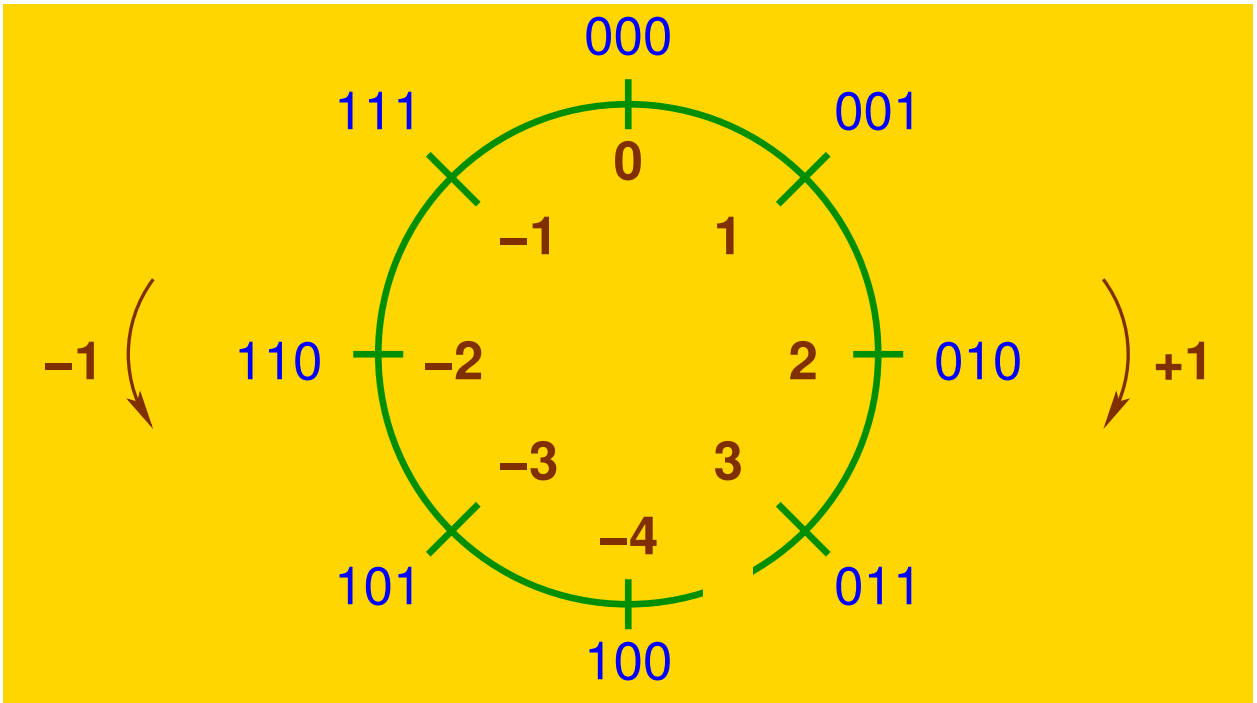
Umrechnung dezimal $\longrightarrow$ binär

$$
\begin{array}{rcrl}
41 & :2 & = & 20 \quad \text{Rest } \mathbf{1} \\
20 & :2 & = & 10 \quad \text{Rest } \mathbf{0} \\
10 & :2 & = & 5 \quad \text{Rest } \mathbf{0} \\
5 & :2 & = & 2 \quad \text{Rest } \mathbf{1} \\
2 & :2 & = & 1 \quad \text{Rest } \mathbf{0} \\
1 & :2 & = & 0 \quad \text{Rest } \mathbf{1}
\end{array}
\Bigg\uparrow
$$

Falls erforderlich, in Binärdarstellung links Nullen auffüllen
z.B. 32-Bit-Integerwert 41 ist das Bitmuster:
00000000 00000000 00000000 00101001

Ganzzahlen mit Vorzeichen im Zweierkomplement

Idee: Für beliebige Ganzzahlen soll gelten $a - b = a + (-b)$, wobei $+$ und $-$ die Addition bzw. Subtraktion auf Bitebene sind. Dadurch sind arithmetische Operationen besonders effizient ausführbar.



Bildung des n-Bit-Zweierkomplements

- in Binärdarstellung wird $-z$ durch $2^n - z$ ersetzt
- d.h. Invertieren aller Bits in der Binärdarstellung von z und Addition von 1

Beispiel

Repräsentation von -13 im 8-Bit-Zweierkomplement

	0000	1101	$z = 13_{(10)} = 1101_{(2)}$
	1111	0010	Inverses
$+$	0000	0001	Addition von 1
	1111	0011	$-z$ im Zweierkomplement

Zweierkomplement für `short`-Werte

16 Bit, d.h. Wertebereich $-2^{15} \ldots 2^{15} - 1$, also $-32768 \ldots 32767$

0111	1111	1111	1111	→	32767
0111	1111	1111	1110	→	32766
...					...
0000	0000	0000	0010	→	2
0000	0000	0000	0001	→	1
0000	0000	0000	0000	→	0
1111	1111	1111	1111	→	-1
1111	1111	1111	1110	→	-2
...					...
1000	0000	0000	0001	→	-32767
1000	0000	0000	0000	→	-32768

Übersicht der elementaren Datentypen in C

Wahrheitswerte

- Ein Wahrheitswert kann entweder den Wert *true* (wahr) oder den Wert *false* (falsch) annehmen.

Wahrheitswerte

- Ein Wahrheitswert kann entweder den Wert *true* (wahr) oder den Wert *false* (falsch) annehmen.
- Jedes Vergleichsergebnis liefert einen Wahrheitswert. Zum Beispiel: `(5 < 8)` ist true, `(7 == 4)` ist false.

Wahrheitswerte

- Ein Wahrheitswert kann entweder den Wert *true* (wahr) oder den Wert *false* (falsch) annehmen.
- Jedes Vergleichsergebnis liefert einen Wahrheitswert. Zum Beispiel: `(5 < 8)` ist true, `(7 == 4)` ist false.
- Wahrheitswerte in C durch *ganze Zahlen* repräsentiert, wobei die Zahl
  0 dem Wahrheitswert false entspricht und
  jede Zahl ungleich 0 dem Wahrheitswert true.

Wahrheitswerte

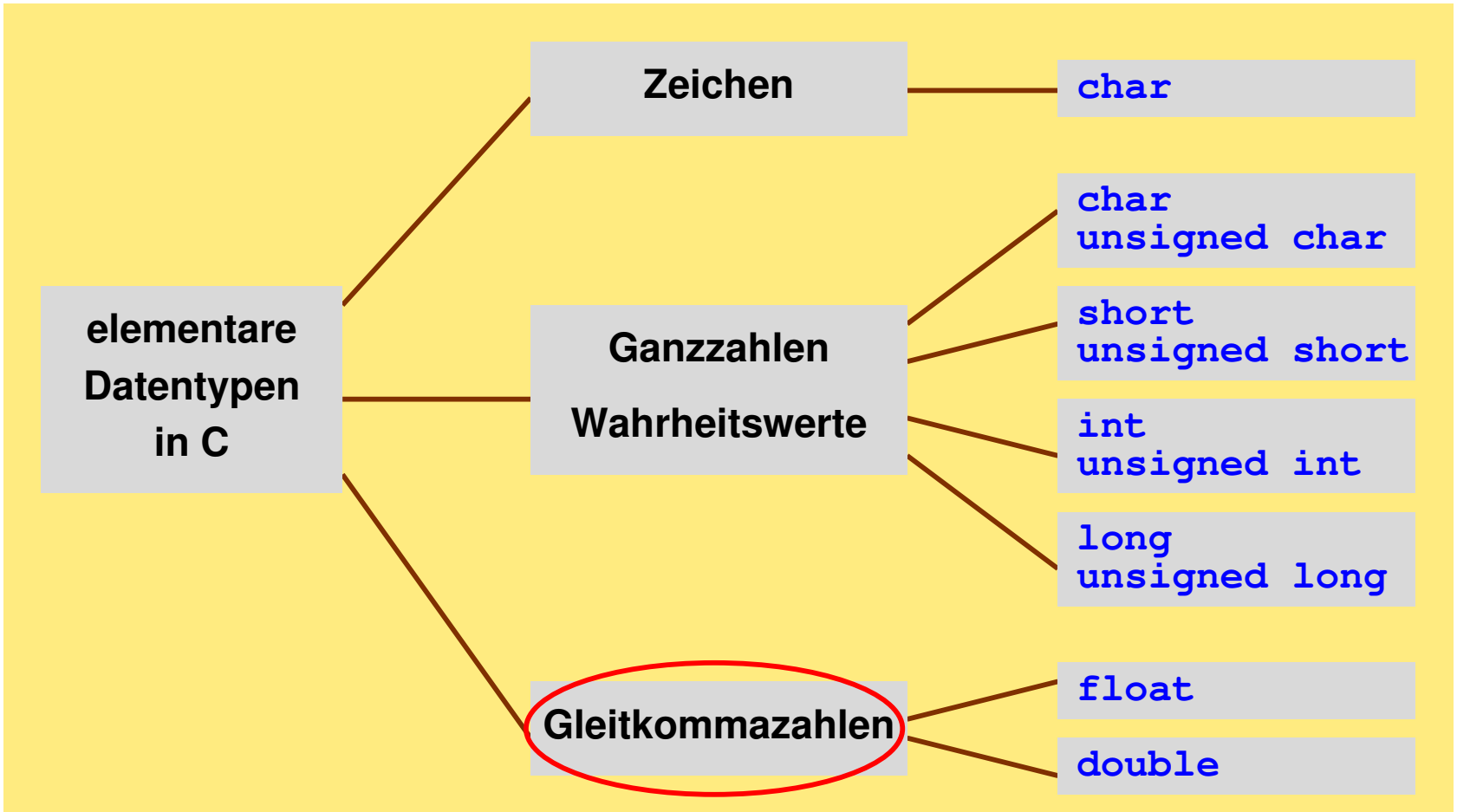
- Ein Wahrheitswert kann entweder den Wert *true* (wahr) oder den Wert *false* (falsch) annehmen.
- Jedes Vergleichsergebnis liefert einen Wahrheitswert. Zum Beispiel: `(5 < 8)` ist true, `(7 == 4)` ist false.
- Wahrheitswerte in C durch *ganze Zahlen* repräsentiert, wobei die Zahl
  0 dem Wahrheitswert false entspricht und
  jede Zahl ungleich 0 dem Wahrheitswert true.
- Für Wahrheitswerte stehen damit alle auf Ganzzahlen definierten Operationen zur Verfügung.

Wahrheitswerte

- Ein Wahrheitswert kann entweder den Wert *true* (wahr) oder den Wert *false* (falsch) annehmen.
- Jedes Vergleichsergebnis liefert einen Wahrheitswert. Zum Beispiel: `(5 < 8)` ist true, `(7 == 4)` ist false.
- Wahrheitswerte in C durch *ganze Zahlen* repräsentiert, wobei die Zahl
  0 dem Wahrheitswert false entspricht und
  jede Zahl ungleich 0 dem Wahrheitswert true.
- Für Wahrheitswerte stehen damit alle auf Ganzzahlen definierten Operationen zur Verfügung.

```
short x = 0; /* false */
short y = 1; /* true */
/* XOR: Binaeraddition ohne Uebertrag */
short z = (!x && y) || (x && !y);
```

Übersicht der elementaren Datentypen in C



Gleitkommazahlen

Typ	**Größe in Byte**	**Wertebereich (Näherungsangabe)**	**Genauigkeit (Dezimal-stellen)**
`float`	4	$-3,4 \cdot 10^{38}$ .... $3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308}$ ...$1,7 \cdot 10^{308}$	ca. 15
`long double`	10*	$-1,1 \cdot 10^{4932}$.$1,1 \cdot 10^{4932}$	ca. 20

*: prozessorabhängig auch 12 o. 16 Bytes mit noch höherer Genauigkeit mgl.

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38}$ .... $3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308}$ ...$1,7 \cdot 10^{308}$	ca. 15
`long double`	10*	$-1,1 \cdot 10^{4932}$.$1,1 \cdot 10^{4932}$	ca. 20

*: prozessorabhängig auch 12 o. 16 Bytes mit noch höherer Genauigkeit mgl.

Große Zahlen zum Vergleich

- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38}$ .... $3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308}$ ...$1,7 \cdot 10^{308}$	ca. 15
`long double`	10*	$-1,1 \cdot 10^{4932}$.$1,1 \cdot 10^{4932}$	ca. 20

*: prozessorabhängig auch 12 o. 16 Bytes mit noch höherer Genauigkeit mgl.

Große Zahlen zum Vergleich

- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen
- 10^{100} wird *Googol* genannt, Namensursprung von Google

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38}$ .... $3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308}$ ...$1,7 \cdot 10^{308}$	ca. 15
`long double`	10*	$-1,1 \cdot 10^{4932}$.$1,1 \cdot 10^{4932}$	ca. 20

*: prozessorabhängig auch 12 o. 16 Bytes mit noch höherer Genauigkeit mgl.

Große Zahlen zum Vergleich

- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen
- 10^{100} wird *Googol* genannt, Namensursprung von Google
- 10^{200} Elementarteilchen: nach heutiger Schätzung *Universum*

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* . . . . . `3.14` für 3,14

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* . . . . . `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85e-12` für $1,85 \cdot 10^{-12}$

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85e-12` für $1,85 \cdot 10^{-12}$
- **Operatoren und Eigenschaften**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`,
    viele Funktionen aus `math.h`

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85e-12` für $1,85 \cdot 10^{-12}$
- **Operatoren und Eigenschaften**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`,
    viele Funktionen aus `math.h`
  - Beim Rechnen häufig *numerische Ungenauigkeiten*
    (z.B. $0.1 + 0.1 + 0.1$ kann ergeben 0.2999999 statt 0.3)

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85e-12` für $1,85 \cdot 10^{-12}$
- **Operatoren und Eigenschaften**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, viele Funktionen aus `math.h`
  - Beim Rechnen häufig *numerische Ungenauigkeiten* (z.B. $0.1 + 0.1 + 0.1$ kann ergeben 0.2999999 statt 0.3)
  - Deshalb Gleitkommazahlen möglichst nicht auf Gleichheit (`==`) oder Ungleichheit (`!=`) vergleichen, sondern ε-Toleranzbereich definieren

Gleitkommazahlen

```
double laenge = -1234.56789;
```

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85e-12` für $1,85 \cdot 10^{-12}$
- **Operatoren und Eigenschaften**
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, viele Funktionen aus `math.h`
  - Beim Rechnen häufig *numerische Ungenauigkeiten* (z.B. $0.1 + 0.1 + 0.1$ kann ergeben 0.2999999 statt 0.3)
  - Deshalb Gleitkommazahlen möglichst nicht auf Gleichheit (`==`) oder Ungleichheit (`!=`) vergleichen, sondern ε-Toleranzbereich definieren
- **Bitmuster-Kodierung im Speicher**
  - nach Standard IEEE754 $\longrightarrow$ in der Hörsaalübung detailliert

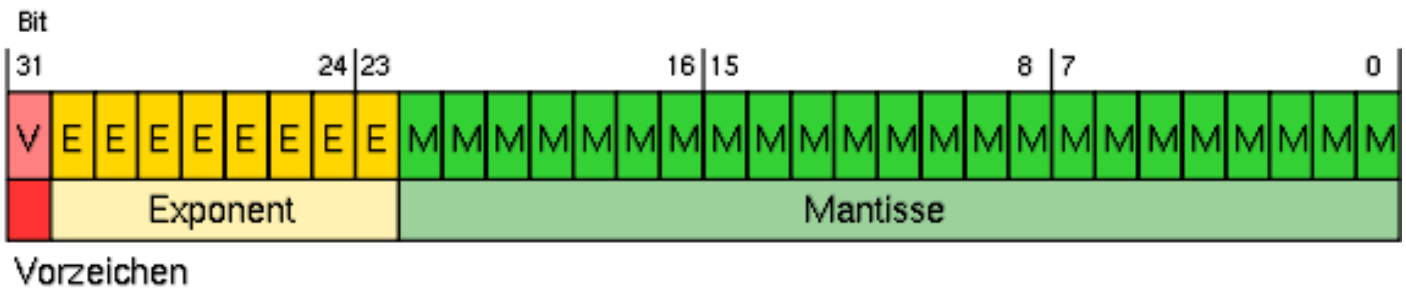
Gleitkommaformat nach IEEE754

$$\text{zahl} = (-1)^v \cdot z(m) \cdot 2^{z(e)}$$

Kodierung der drei Komponenten:

Vorzeichen v ..... 1 Bit
Exponent e ..... k Bits
Mantisse m ..... l Bits

k, l und die Berechnung von $z(m)$ und $z(e)$ sind prozessorabhängig



Beispiel `float`: 32 Bit (4 Byte)

Definitionen

Eine **Variable** ist ein Platzhalter (Behälter, *Speicherbereich*), der nacheinander verschiedene Werte (in C: gleichen Typs) annehmen kann, die über einen eindeutigen *Namen* (Bezeichner) zugänglich gemacht werden.

```
double kontostand = .0;

int punktgewinn = 2;

unsigned int anzahlStudenten = 24;

long anzahlZugriffe = 87654321;

char geschlecht = 'm';

short erledigt = 1;

float temperatur = 20.75;

kontostand = 1.5e+6;
```

Definitionen

Eine **Konstante** ist ein Name, der bei seiner Deklaration mit einem Wert verbunden wird. Diesen Wert behält die Konstante während ihrer gesamten Lebensdauer unverändert bei. In C werden Konstanten mit dem Schlüsselwort `const` eingeführt.

```
const double PI = 3.14159265;

const float G = 9.81;

const int V = 299793218;

V = 0; //Compiler meldet hier einen Fehler
```

Definitionen

Literale sind explizit im Programmtext angegebene *Konstantenwerte* oder *Konstantenausdrücke*.

```
const double G = 9.81;
double phi = 45 * 3.14159265 / 180;
```

- `G` ist eine *Konstante* vom Typ `double` mit dem Wert `9.81`.
- `phi` ist eine *Variable* vom Typ `double` mit veränderbarem Wert.
- Die Werte `9.81` und `45 * 3.14159265 / 180` sind jeweils *Literale*.
- Variablen dürfen auch ohne Wertzuweisung angelegt werden, es wird das vorhandene Bitmuster des ausgefassten Speicherbereiches als Wert interpretiert.

Variablen und Konstanten anlegen („deklarieren“)

```
int i, k, zaehler = 1, x_0, anzahl;
double phi = zaehler, dt = 0.1, alpha;
```

- Mehrere Variablen desselben Typs dürfen durch Komma getrennt hintereinander deklariert werden.
- Zur Wertzuweisung dürfen auch schon bekannte Variablen- oder Konstantennamen genutzt werden, soweit sie **typverträglich** sind.
- Der Typ einer deklarierten Variablen lässt sich (in C) während ihrer Lebensdauer nicht mehr verändern.

Namen für Variablen und Konstanten

Bezeichner (Namen)

- beginnen mit einem Buchstaben oder Unterstrich _
- dürfen enthalten: engl. Buchstaben, Ziffern und Unterstriche _
- dürfen beliebig lang sein, aber nur erste 32 Zeichen ausgewertet
- Groß- und Kleinbuchstaben werden unterschieden.
- C-Schlüsselwörter (wie **`return`** oder **`double`**) nicht zugelassen
- Reservierte Wörter (wie **`main`**) sollten nicht verwendet werden

Namen für Variablen und Konstanten

Bezeichner (Namen)

- beginnen mit einem Buchstaben oder Unterstrich _
- dürfen enthalten: engl. Buchstaben, Ziffern und Unterstriche _
- dürfen beliebig lang sein, aber nur erste 32 Zeichen ausgewertet
- Groß- und Kleinbuchstaben werden unterschieden.
- C-Schlüsselwörter (wie **`return`** oder **`double`**) nicht zugelassen
- Reservierte Wörter (wie **`main`**) sollten nicht verwendet werden

Konventionen

- Aussagekräftige Namen verwenden
- Variablennamen primär mit Kleinbuchstaben schreiben
- In zusammengesetzten Wörtern beginnen neue Wortteile mit Großbuchstaben (**`anzahlStudenten`**)
- Konstanten mit Großbuchstaben schreiben

Grundoperationen

Operator	Beispiel	Wirkung
`+`	`a + b`	Addiert **a** und **b**
`-`	`a - b`	Subtrahiert **b** von **a**
`*`	`a * b`	Multipliziert **a** und **b**
`/`	`a / b`	Dividiert **a** durch **b**
`%`	`a % b`	Liefert den Rest bei der ganzzahligen Division **a / b**

- Operation `%` (Divisionsrest) nur auf Ganzzahltypen definiert
- Division `/` auf Ganzzahltypen schneidet Nachkommastellen ab
- Division durch 0 führt zum Programmabsturz
- Punktrechnung vor Strichrechnung
- Gleichrangige Operatoren von links nach rechts abgearbeitet
- Runde Klammern wie in der Mathematik zulässig

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen
- Bei Ganzzahltypen leider keine Fehlermeldung, es entstehen dann *falsche Ergebnisse*

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen
- Bei Ganzzahltypen leider keine Fehlermeldung, es entstehen dann *falsche Ergebnisse*

Berechnungsidee, so dass keine zu großen Zwischenergebnisse entstehen:

$$\frac{n \cdot (n-1)}{2} \cdot \frac{n-2}{3} \cdot \frac{n-3}{4} \cdot \frac{n-4}{5} \cdot \frac{n-5}{6}$$

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

```c
#include <stdio.h>

int main(void)
{
   // Anzahl verschiedener Lotto-Tipps bei "6 aus 49" bzw. "6 aus n"

   long n = 49; //Italien: 90, Ungarn: 45, Litauen: 30

   long komb = n * (n-1) / 2 * (n-2) / 3 * (n-3) / 4 * (n-4) / 5 * (n-5) / 6;

   printf("Es gibt %ld Kombinationen 6 aus %ld\n", komb, n);
   return 0;
}
```

Der Zuweisungsoperator = und seine Funktionsweise

xz.c

```c
#include <stdio.h>

int main(void) {
  int x = 5;
  int z = 3;

  x = x + 2;
  z = x * z;

  printf("x: %d  z: %d\n", x, z);
  return 0;
}
```

- Variablen können während Programmabarbeitung ihren Wert verändern.
- Rechts vom Zuweisungsoperator = haben die Variablen ihren alten Wert, daraus wird der neue Wert berechnet und dann der Variablen links vom = zugewiesen.

Abkürzende Schreibweisen

xz2.c

```c
#include <stdio.h>

int main(void) {
  int x = 5;
  int z = 3;

  x += 2;
  z *= x;

  printf("x: %d  z: %d\n", x, z);
  return 0;
}
```

- `<Variable> = <Variable> <ArithOp> <Operand>;` kann gleichwertig kürzer geschrieben werden durch `<Variable> <ArithOp>= <Operand>;`
- `<ArithOp>` ist beliebige arithmetische (oder Bit-)Operation
- `<Operand>` beliebige typverträgliche Variable oder Literal

Inkrementieren ++ und Dekrementieren – –

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen

Inkrementieren ++ und Dekrementieren – –

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)

Inkrementieren ++ und Dekrementieren −−

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`

Inkrementieren ++ und Dekrementieren − −

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)

Inkrementieren ++ und Dekrementieren – –

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)
- `a = b * (x++) - c;` entspricht der Abarbeitung

```
a = b * x - c; x++;
```

Inkrementieren ++ und Dekrementieren – –

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)
- `a = b * (x++) - c;` entspricht der Abarbeitung
  `a = b * x - c; x++;`
- `a = b * (++x) - c;` entspricht der Abarbeitung
  `x++; a = b * x - c;`

Die Standard-Mathematikbibliothek `math.h`

Teil 1

`double sin(double x);`	sin(x)
`double cos(double x);`	cos(x)
`double tan(double x);`	tan(x)
`double asin(double x);`	$\sin^{-1}$(x) in [-pi/2,pi/2] (für x in[-1,1])
`double acos(double x);`	$\cos^{-1}$(x) in [0,pi] (für x in[-1,1])
`double atan(double x);`	$\tan^{-1}$(x) in [-pi/2,pi/2]
`double atan2(double x, double y);`	$\tan^{-1}$(x/y) in [-pi,pi]
`double sinh(double x);`	sinh(x)
`double cosh(double x);`	cosh(x)
`double tanh(double x);`	tanh(x)

- durch Präprozessorbefehl `#include <math.h>` am Quelltextanfang einbinden
- Argumente bei den Winkelfunktionen `sin`, `cos`, `tan` stets im Bogenmaß (radiant)
- Funktionswerte stets im Typ `double` zurückgegeben

Die Standard-Mathematikbibliothek `math.h`

Teil 2

`double exp(double x);`	e^x
`double log(double x);`	ln(x) (für x>0)
`double log10(double x);`	$\log_{10}(x)$ (für x>0)
`double pow(double x, double y);`	x^y (für (x != 0 oder y > 0) und (x >=0 oder y ganzzahlig))
`double sqrt(double x);`	Quadratwurzel von x (für x >= 0)
`double ceil(double x);`	kleinste ganze Zahl >= x
`double floor(double x);`	größte ganze Zahl <= x
`double fabs(double x);`	|x|
`double ldexp(double x, int n);`	$x \cdot 2^n$
`double frexp(double x, int *exp);`	liefert Mantisse in [1/2,1) und speichert Exponent nach `*exp`
`double modf(double x, double *ip);`	liefert y in **Z** und in `*ip` den Rest in [0,1), so daß y+Rest=x
`double fmod(double x, double y);`	Gleitpunktrest von x/y in [0,y)

- **`ceil`** rundet auf, **`floor`** rundet ab
- **`fmod`** liefert den Nachkommaanteil bei Division **`x / y`**
- **`frexp`** und **`modf`** unterstützen schnelles Gleitkommarechnen

(Pseudo)Zufallszahlen erzeugen mithilfe `stdlib.h`

wuerfel.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    // Wuerfel
    srand(time(NULL));
    printf("1. Wurf: %d\n", rand()%6+1);
    printf("2. Wurf: %d\n", rand()%6+1);
    printf("3. Wurf: %d\n", rand()%6+1);
    return 0;
}
```

- `srand(time(NULL));` initialisiert Generator mit Startwert aus aktueller Systemzeit
- Bibliotheksfunktion `rand` liefert Pseudozufallszahl zwischen `0` und `RAND_MAX` (vordefinierte Konstante) als `int` zurück
- Skalieren, so dass Ganzzahl zwischen 1 und 6 ausgegeben wird

Bildschirmausgabe mit `printf`

```
int a = 5;
double x = 9.81;
printf("a ist %d und x ist %lf\n", a, x);
```

- `printf` steht für *formatierte* Ausgabe, in `stdio.h` definiert
- Argumente sind der *Formatstring* begrenzt durch `"  "`, danach folgen durch Kommas getrennt die auszugebenden Variablen
- Im Formatstring *Platzhalter* für jede Variable, eingeleitet durch `%`. Reihenfolge der Platzhalter von links nach rechts entspricht Reihenfolge der Variablen
- Bezeichnungen der Platzhalter fest vorgegeben nach Variablentyp und Formatierungsoption
- Steuerzeichen (Escape-Sequenzen) wie `\n` zulässig

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** . . . . . vorzeichenbehaftete Ganzzahl dezimal („digits“)

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%lf` ..... Alternative für Format double („long float“)

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%lf` ..... Alternative für Format double („long float“)
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: **`-1.23456789e+3`**

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%lf` ..... Alternative für Format double („long float“)
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: **`-1.23456789e+3`**
`%s` ..... Zeichenkette („string“)

Ein- und Ausgabe von Variablenwerten

In den Formatstrings von **`printf`** und **`scanf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` oder **`%i`** ..... vorzeichenbehaftete Ganzzahl dezimal („digits“)
`%u` ..... vorzeichenlose Ganzzahl dezimal („unsigned“)
`%c` ..... einzelnes Zeichen („character“)
`%f` ..... Gleitkommazahl in Einzelstellenschreibweise („float“)
Beispiel: **`-1234.56789`**
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%lf` ..... Alternative für Format double („long float“)
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: **`-1.23456789e+3`**
`%s` ..... Zeichenkette („string“)

seltener genutzt:

`%x` ..... Ganzzahl hexadezimal („heX“)
`%p` ..... Speicheradresse einer Variablen („pointer“)
`%%` ..... Ausgabe des Prozentzeichens

Umrechnen zwischen dezimal und hexadezimal allein mit `printf`

```c
#include <stdio.h>

int main(void) {
  int a = 45054;

  printf("Hex: %x\n", a);
  return 0;
}
```

```c
#include <stdio.h>

int main(void) {
  int a = 0xaffe;

  printf("Dez: %d\n", a);
  return 0;
}
```

$$45054 = 10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 \cdot 16^0 = \text{affe}_{(16)}$$

Tastatureingaben mittels `scanf`

```
int a;

printf("Wert eingeben: ");
scanf("%d", &a);
```

- `scanf` in `stdio.h` verfügbar
- Formatstring darf nur Platzhalter enthalten
- Übergeben wird die *Speicheradresse*, ab der der einzugebende Variablenwert abgelegt wird
- `scanf` nutzt diese Adresse und legt dort den Wert ab
- Der Adressoperator `&` liefert zu einer Variablen ihre Anfangsadresse im Speicher, z.B. `&a`
- `scanf` fängt keinen Pufferüberlauf ab

Einzelzeicheneingabe mit `getchar`

```c
#include <stdio.h>

int main(void) {
  char c;

  printf("Druecken Sie eine Taste\n");
  c = getchar();
  printf("Gedrueckt: %c\n", c);
  return 0;
}
```

- `getchar` in `stdio.h` verfügbar
- Eingelesenes Zeichen wird als `char` zurückgegeben
- Rückgabe muss nicht zwingend ausgewertet werden, bsp. bewirkt die Programmzeile `getchar();` das Warten auf einen Tastendruck des Nutzers. Sinnvoll u.a. bei umfangreichen Ausgaben zum „Weiterschalten“

Typecast (I) – Welche Ausgabe erwarten Sie hier?

```c
#include <stdio.h>

int main(void) {
  int i = 5;
  double x = i;

  printf("%lf\n", x);
  return 0;
}
```

Typecast (I) – Welche Ausgabe erwarten Sie hier?

```c
#include <stdio.h>

int main(void) {
  int i = 5;
  double x = i;

  printf("%lf\n", x);
  return 0;
}
```

- Mitunter kommt es vor, dass Variablen unterschiedlichen Typs im gleichen Ausdruck verarbeitet werden (sollen).
- Daraus resultiert Notwendigkeit, Variablen- und/oder Konstantenwerte möglichst *werterhaltend* von einem Typ in einen anderen umzuwandeln.
- Ist der Wertebereich des Zieltyps gleich oder größer (z.B. `int` nach `double`), geht keine Information verloren.
- Solche Konvertierungen werden automatisch und problemlos ausgeführt (*impliziter Typecast*)

Typecast (II)

```c
#include <stdio.h>

int main(void) {
  double x = 9.81;
  int i = x;
  printf("%d\n", i);
  return 0;
}
```

- Anders ist die Situation, wenn bei einer Typkonvertierung Information verloren gehen kann.
- Sprachspezifikation von C definiert Verhalten in solchen Fällen.
- Vielfach wird eine automatische Konvertierung unter Informationsverlust vorgenommen, z.B. von Gleitkommatyp in Ganzzahltyp werden Nachkommastellen einfach abgeschnitten (wie oben: Ausgabe **9**)

Typecast (III)

```c
#include <stdio.h>

int main(void) {
  int i = 5;
  float f  = &i; //Adresse von i

  printf("%f\n", f);
  return 0;
}
```

- Alle elementaren Datentypen sind zueinander typverträglich.
- Darüber hinaus kann es aber *Typinkompatibilitäten* geben.
- Beispiel oben: Die Speicheradresse `&i` der `int`-Variablen `i` lässt sich nicht sinnvoll als `float`-Wert darstellen.
- Bei Typinkompatibilitäten Fehlermeldung(en) beim Compilieren

Expliziter Typecast

```c
#include <stdio.h>

int main(void) {
  float f = 7 / 9;
  float g = (float) 7 / (float) 9;
  float h = (float) 7 / 9;

  printf("%f  %f  %f\n", f, g, h);
  return 0;
}
```

- Als Programmierer kann man Typecasts im Quelltext erzwingen, indem man den Zieltyp in runde Klammern vor die Variable oder das Literal schreibt.
- Dieser *explizite Typecast* hat eine höhere Priorität (bindet stärker) als alle arithmetischen Operatoren.
- Ausgaben oben: `f`: 0.000000, `g`: 0.777778, `h`: 0.777778
- Bei mehrstelligen arithmetischen Operatoren Ausführung im wertebereichsgrößten Typ der Operanden

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` ...“

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Speicherschonende Programmierung

- Wenn ein Schalter z.B. nur 8 unterscheidbare Stufen kennt, braucht man dafür wahrlich keine 64 Bit im Speicher

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Speicherschonende Programmierung

- Wenn ein Schalter z.B. nur 8 unterscheidbare Stufen kennt, braucht man dafür wahrlich keine 64 Bit im Speicher

Möglichst zeiteffiziente Operationsausführung

- Gleitkommaoperationen sind zeitaufwendiger als Ganzzahloperationen

Prioritäten von Operatoren in C (Auswahl)

hoch	`()`	Funktionsaufruf und Klammerung

	`+ -`
	`++ -- &`
	`(Typ)`
		Vorzeichen
		Inkrement, Dekrement, Adresse
		Typecast

	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= >` `>=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung `=`) von links nach rechts abgearbeitet

Prioritäten von Operatoren in C (Auswahl)

hoch	`()`	Funktionsaufruf und Klammerung

	`+ -`
	`++ -- &`
	`(Typ)`
		Vorzeichen
		Inkrement, Dekrement, Adresse
		Typecast

	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= >` `>=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung =) von links nach rechts abgearbeitet
- `int i = (3 <= 5) * (int) (9.81 - sqrt(9));`
  ergibt .....

Prioritäten von Operatoren in C (Auswahl)

hoch	`()`	Funktionsaufruf und Klammerung

	`+ -`
	`++ -- &`
	`(Typ)`
		Vorzeichen
		Inkrement, Dekrement, Adresse
		Typecast

	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= >` `>=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung `=`) von links nach rechts abgearbeitet
- `int i = (3 <= 5) * (int) (9.81 - sqrt(9));`
  ergibt ..... `6`