02 - Grundlagen

Alexander Krause

IRB

Veranstaltungswebseite

Das erste Programm - Hello World

cpp Run ▶

```cpp
#include <stdio.h> // Verwende den Baukasten stdio (für printf)

int main() { // Einstiegspunkt jedes C-Programms
    printf("Hello World");

    printf("");

    printf("");
    return 0;
}
```

- `main()` ist der Einstiegspunkt jedes C-Programms
- Erstmal nur `printf` und `main()` → Rest folgt!

Aber: Wie werden Informationen im Computer abgebildet?

Hier: Der Text (*String*) `"Hello World"`

Darstellung von Informationen

- Was wollen wir speichern?
  - Name
  - Matrikelnummer
  - Temperatur
  - …
- Bestimmte Formen [illegible] Eigenschaften

Dies alles nennt man *Datentypen*

Name:	Anordnung von Buchstaben (Zeichenkette)
Matrikelnummer:	nicht-negative ganze Zahl
Temperatur:	~~nicht~~-negative reelle Zahl

Binärformat

- Computer speichert Informationen im Binärformat:

  1 oder 0 (= genannt Bit)

- Kombination von mehreren Ziffern bildet Binärzahl: 1001
- Jede Stelle der Bin

  $2^0 = 1, 2^1 = 2, 2^$

- Summe der einzelr
- Beispiel: 79 = 64 + 8 + 4 + 2 + 1 = $2^6 + 2^3 + 2^2 + 2^1 + 2^0$

8 Bits ergeben dabei 1 Byte → 2^8 = 256 mögliche Werte pro Byte

$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	1	0	0	1	1	1	1

Binärformat

- Zahlen > 255 werden genauso dargestellt, bestehen aber aus mehreren Bytes
- Beispiel: 591 = 512 + 79

512	256	128	64	32	16	8	4	2	1
1	0	0	1	0	0	1	1	1	1

- 591 = 0000 0010 x 256 + 0100 1111

Binärformat

- Byte mit dem höchsten/niedrigsten Wert heißt *Most/Least Significant Byte* (*MSB* bzw. *LSB*)
- Reihenfolge der Bytes heißt *Byte Order*: Entweder von MSB zu LSB (*Big Endian*) oder von LSB zu MSB (*Little Endian*)
- Analog für Bits: *Most/Least Significant Bit* (*MSBit* bzw. *LSBit*)

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
0 (MSBit)	1	0	0	1	1	1	1 (LSBit)

Binärformat

- 🚨 Achtung: Zahlen können gleich aussehen, aber in unterschiedlichen Zahlensystemen stehen
- Beispiel: `110` (Binär) ≠ `110` (Dezimal)
- Unterscheidung bei gemischten Zahlensystemen über **Index**: $110_B = 6_D$
- Gibt mehrere gängige Zahlensysteme:
  - Dezimal → D oder dec
  - Binär → B oder bin
  - Hexadezimal → H oder hex
  - Oktal → O oder oct

Hexadezimale Darstellung

- Dezimal: Zahlen von 0-9
- Hexadezimal: Erweiterung der dezimalen Darstellung auf 0-15
- Beispiele:
  - $32_D = 20_H = 0010\ 0000_B$
  - $255_D = FF_H = 1111\ 1111_B$
  - $1000_D = 3E8_H = 11\ 1110\ 1000_B$

Dez.	Hex.	Dez.	Hex.
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

Vorteile der hexadezimalen Darstellung

Vorteile der hexadezimalen Darstellung:

- 16 Werte → 2^4 → 4 Bit zur Darstellung → 2 x 4 Bit → 1 Byte

  → Darstellung von 1 Byte mit nur zwei Zeichen

- Deutlich leserlicher als eine Binärzahl!
- Hexadezimal ist vielfaches von Binärsystem → einfache Umwandlung
  - Besonders bei großen Zahlen sehr praktisch, weil Binärsystem unhandlich wird
  - Beispiel: FF FF_H besser als 1111 1111 1111 1111_B

Maßeinheiten für Bytes

- Bei Bytes gibt es oft Verwirrung bei den Präfixen für Einheiten: Kilo, Mega, …
- Hersteller verwenden Dezimalsystem!

  → 1000 Bytes = 1 Kilobyte
- Programmierer:innen denken hingegen im Binärsystem

  → 2^{10} = 1024 Bytes = 1 Kilobyte
- Daher Unterscheidung über zusätzliches Präfix

  → Beispiel: Anstatt Kilobyte, nun Kibibyte
- Faustformel: Erweiterung um 10 Bits ist eine Erhöhung um den Faktor ~1000 (gut für ungefähre Größenabschätzung) 10^6

Einheitenpräfixe für Bytes

Bytes (dezimal)	Bezeichnung	Bytes (binär)	Bezeichnung
10^3	Kilobyte (KB)	2^{10}	Kibibyte (KiB)
10^6	Megabyte (MB)	2^{20}	Mebibyte (MiB)
10^9	Gigabyte (GB)	2^{30}	Gibibyte (GiB)
10^{12}	Terabyte (TB)	2^{40}	Tebibyte (TiB)
10^{15}	Petabyte (PB)	2^{50}	Pebibyte (PiB)
10^{18}	Exabyte (EB)	2^{60}	Exbibyte (EiB)

Das erste Programm (Fortsetzung)

cpp Run ▶

```c
#include <stdio.h>

int main() {
    int i = 3; // Befehl 1: Erzeuge Variable mit Ganzzahl (Integer)
    i = i + 5; // Befehl 2: Addiere 5, weise die Summe i zu
    printf("%d", i); // Befehl 3: Nun gib i aus
    return 0;
}
```

- Diese Art der Programmierung heißt **imperativ**
- Jeder Befehl in C endet mit dem Semikolon (**;**)

Kommentare

- Kommentare sind wertvoll, um komplexeren Code zu erklären oder auf Tücken hinzuweisen
- Zwei Arten von Kommentaren: **Einzeilig** und **Blockkommentar**

```c
#include <stdio.h>

int main() {
    int i = 3; // Einzeilen-Kommentar, beginnt immer mit //
    // und endet mit der Zeile
    /* Blockkommentar beginnt immer mit /* und endet mit */
    /*
        Beliebiger Text dazwischen möglich
    */
    return 0;
}
```

Anlegen von Variablen

Anlegen der Variable `num`: `int num = 3;`

- `int`: Typ der Variable
- `num`: Bezeichner, führt den Namen `num` ein
- `3`: Initialisierung, initialisiert `num` mit `3`
- Was passiert hier?
  1. Übersetzer reserviert Speicher:
  2. Wertevorrat wird festgelegt (`int`)
  3. Name `num` ist von nun an eine Variable
- `num` kann ab Deklaration „abwärts” weiterverwendet werden

Elementare Datentypen

- Einfache Datentypen sind elementar → nicht auf andere Typen zurückführbar

  (*pod = plain old datatypes*)

- Beispiele:
  - (nicht-negative) ganze Zahlen (*Integers*) → `int i = 3;`
  - Zahlen mit Nachkommastellen (*Floats*). → `float f = 3.14;`
  - Zeichen (*Characters*) → `char c = 'c';`
- Achtung: verzeichenlos ↔ verzeichenbehaftet
  - Zusatz signed bzw. unsigned beim Datentyp: `unsigned int num = 3;`
  - Bei `signed` wird ein Bit geopfert für das Vorzeichen

Nicht-negative Ganzzahlen

Bits	Bytes	Werte	Name in C/C++	Min. Breite	Typ. Breite	Feste Breite
8	1	0…255	unsigned char	8	8	uint8_t
16	2	0…65			6	uint16_t
32	4	0…4			2	uint32_t
64	8	0…4	int		2	uint32_t
64	8	0…18 446 744 073 709 551 615	unsigned long long int	64	64	uint64_t

🚨 **Breite in Bits ist abhängig von der Hardwarearchitektur!** 🚨

Ganzzahlen mit Vorzeichen

Bits	Bytes	Werte	Name in C/C++	Min. Breite	Typ. Breite	Feste Breite
8	1	-128...127	char	8	8	int8_t
16	2	-327			6	int16_t
32	4	-2 14 483			2	int32_t
64	8	-2 147 483 648...2 147 483 647	long int	32	32	int32_t
64	8	-2^{63} ... $2^{63}-1$	long long int	64	64	int64_t

🚨 **Breite in Bits ist abhängig von der Hardwarearchitektur!** 🚨

Zusammengesetzte Datentypen

- Entstehen baukastenartig durch Zusammensetzen von elementaren Datentypen
- **Beispiele**:
  - Koordinate: Ein Paar aus zwei ganzen Zahlen
  - Zeichenkette: besteht aus mehreren einzelnen Characters
  - Adresse: besteht aus PLZ, Ort und Straße, Hausnummer

```
struct Koordinate {
    int x;
    int y;
};
```

Vorschriften für Bezeichner

- Es dürfen nur Buchstaben a-z sowie A-Z, Ziffern 0-9 und der Unterstrich _ vorkommen
- **Kein** Leerzeichen erlaubt
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein
- Prinzipiell keine (in der Realität erreichbare) Längenbeschränkung
- Schlüsselwörter dürfen nicht verwendet werden
- Trennung zwischen Datentypen und Variablen über *Whitespace*-Zeichen (Leerzeichen, Tabulator, *New Line*)

Beispiele für Bezeichner

Zulässig	Nicht zulässig
Winkel	nichtOk!
EinkomSteuer	7CR
einkom_steuer	x-2
wichtigeVariable	wichtige Variable
_OK	int
x3	float
__x3_und_x4_	
_99	

Einschub: Schlüsselwörter in C

Schlüsselwörter sind reservierte Wörter der jeweiligen Programmiersprache!

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	inline	int
long	register	restrict	return	short	signed
sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while	true (C23)	false (C23)

- Dies sind nur die relevantesten für C, siehe cppreference.com

Einschub: Schlüsselwörter in C++

Zusätzliche Schlüsselwörter in C++ (auszugsweise, siehe cppreference.com)

and	and_eq	asm	bitand	bitor	catch
class	compl	constexpr	const_cast	delete	dynamic_cast
explicit	for	friend	false	namespace	new
noexcept	not	not_eq	nullptr	operator	or
or_eq	private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try	typename
using	virtual	xor	xor_eq		

Konventionen für Bezeichner

- Es gibt zahlreiche Möglichkeiten für die Benennung
- Am weitesten verbreitete Konventionen
  - **Snake Case**
  - **Camel Case**
- Snake Case
  - Variablen werden klein geschrieben
  - Leerzeichen durch _ ersetzt
  - Bsp.: `int count_variable_1 = 0;`
- Konstante Variablen, deren Wert sich nicht verändert, werden nur groß geschrieben (Bsp.: `const int RESISTOR_VAL_470 = 470;`)

Konventionen für Bezeichner

- Camel Case
  - Bezeichner beginnt mit Kleinbuchstaben
  - Wörter werden aneinander gehängt
  - ein neues Wort mit Großbuchstaben signalisiert
- Bsp.: `int countVariable2 = 0;`
- Faustformeln
  - In C ist Snake Case weiter verbreitet → deswegen wird diese Notation auch hier verwendet
  - Interne oder temporäre Variablen beginnen meist mit _

Arithmetische Operationen

Operation	Operator
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Division mit Rest (Modulo)	%

Modulo-Operator

- Modulo-Operation: Division zweier Zahlen, Ergebnis entspricht dem Rest
- Beispiel: Überlauf abfangen bei einer Uhr

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int stunde = 22;
    int minute = 59;
    minute = (minute + 1) % 60;
    stunde = (stunde + 1) % 24;
    printf("%d:%d", stunde, minute);
    return 0;
}
```

Beispiel zu arithmetischen Operationen

```c
#include <stdio.h>

int main() {
    int i = 3;
    i = i + 13;
    i = i * 5;
    i = i / 8;
    i = i % 3;
    printf("%d", i);
    return 0;
}
```

A: 0

B: 3.333

C: 1

D: 3

Beispiel zu arithmetischen Operationen

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 3;
    i = i + 13; // 16
    i = i * 5; // 80
    i = i / 8; // 10
    i = i % 3; // 1 => 9/3 und Rest 1
    printf("%d", i);
    return 0;
}
```

Arithmetische Operationen *Reloaded*

Arithmetische Operation **und** Zuweisung in einem

x = x + 3;

<=>

x += 3;

Operation	Operator
Addition	+=
Subtraktion	-=
Multiplikation	*=
Division	/=
Division mit Rest (Modulo)	%=

Beispiel arithmetische Operationen mit Zuweisung

c Run

```c
#include <stdio.h>

int main() {
    int i = 3;
    i += 13;
    i *= 5;
    i /= 8;
    i %= 3;
    printf("%d", i);
    return 0;
}
```

Wissensspeicher 1

- Ihr kennt bereits das erste C-Programm 🥳
- Ihr wisst, wie
  - Ihr Variablen anlegt,
  - man Zahlen abspeichert und
  - mit Variablen rechnet.

Increment/Decrement

- **Die** häufigste Operation auf Zahlen: Addition / Substraktion von 1
- Es gibt dafür zwei eigene Operatoren
  - `++` (*Increment*)
  - `--` (*Decrement*)
- Operatoren können vor einer Variable stehen → *Pre-Increment*, `++i`
- …aber auch einer Variable folgen → *Post-Increment*, `i++`
- Analog dazu Dekrement: `--i`, `i--`
- Unterschied
  - *Pre-Increment*: Erst erhöhen, dann die Variable verwenden
  - *Post-Increment*: Erst die Variable verwenden, dann erhöhen

- Empfehlung: Benutzt Pre-Increment, wenn möglich

Beispiel zu *Pre-Post-Increment*

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 0;
    ++i;
    ++i;
    ++i;
    --i;
    printf("%d", i);
    return 0;
}
```

Konkatenation von Operationen

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 3;
    i = ((i + 13) * 5) / 8) % 3;
    printf("%d", i);
    return 0;
}
```

(Hier sind Klammern wichtig, die Operatoren haben wie in der Mathematik auch eine unterschiedliche Wichtigkeit)

Operator-Präzedenz

- Spielt eine Rolle, wenn **keine** Klammerung vorhanden ist
- Beschreibt die Bindung an Operanden
- Operatoren sind in Gruppen eingeteilt
  - Von 1-17
  - 1 ist die höchste Präzedenz
- Je höher die Präzedenz, desto stärker binden sie an die Werte / Variablen

(=„je größer die Präzedenz, desto stärker die Anziehungskraft”)

Operator-Präzedenz

- Vollständige Tabelle aller Operatoren findet sich hier unter cppreference.com
- Muss nicht auswendig gelernt werden!
- Die meisten Präzendenzen ergeben sich intuitiv,

  aber am Anfang muss man ggf. häufiger nachschauen
- Tipp: Im Zweifelsfall Klammern setzen (erhöht auch meistens die Lesbarkeit)

Beispiel Operator-Präzedenz

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 1;
    i = i++;
    printf("%d", i);
    return 0;
}
```

A: 1 **B:** 3 **C:** 2 **D:** Nicht definiert

Häufige Fehler mit *Integers*

Angenommen eine Variable hat den höchsten darstellbaren Wert

und wird dann erhöht: `uint16_t i = 65535; ++i;`

Was wird passieren?

A: `0`

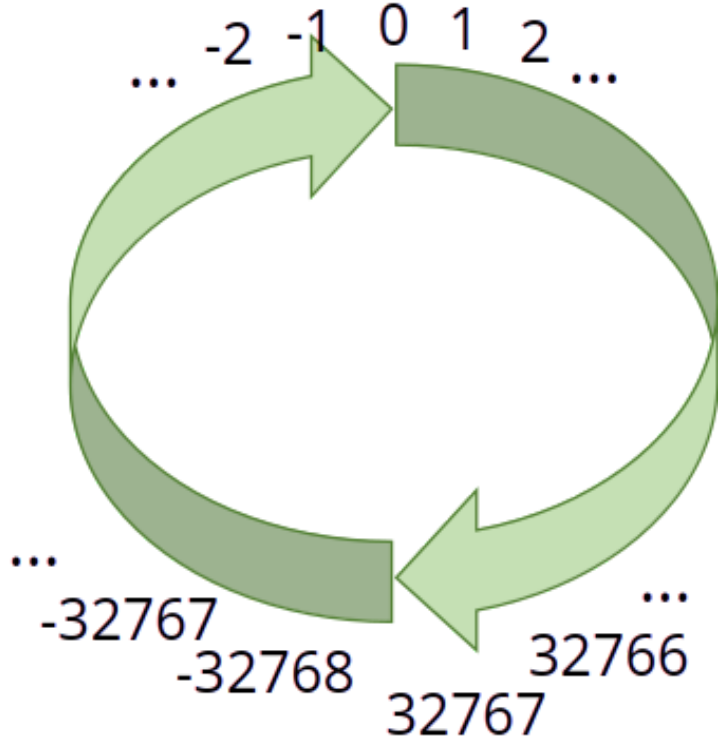
B: Nimmt 17 Bits ein (65536_D = `1 0000 0000 0000 0000`)

C: –65535

D: Wird nicht übersetzt

Integers Overflow/Underflow

- Überlauf (*Overflow*)
  - Größtmögliche Zahl und dann um 1 erhöhen
  - Kleinste Zahl tritt auf
- Unterlauf (*Underflow*)
  - Kleinmögliche Zahl und dann um 1 erniedrigen
  - Größte Zahl tritt auf
- Besonders unschön:
  - Ist nur für vorzeichlenlose definiert
  - Bei vorzeichenbehafteten Zahlen ist dies undefiniertes Verhalten (*undefined behavior*)



Undefiniertes Verhalten

- Tritt auf, wenn im Standard das Verhalten nicht definiert ist
- Kann alles bedeuten
  - Vom erwarteten Verhalten (Wrap-Around)
  - Absturz des Progra
  - Es erscheint ein Cl
  - …
- Abhilfe: Nächstgrößeren Datentyp nehmen
  - Bei der Zahl 256 nicht 1 Byte (8 Bit) sondern 2 Byte (16 Bit)
- Programmiert defensiv
  - Trefft lieber Vorkehrungen, wie Konsistenzprüfungen der Werte
  - „Macht der Wert so überhaupt Sinn?" 🤔

Verlasst euch nie auf undefiniertes Verhalten!

Beispiel häufige Fehler mit Integern

cpp Run ▶

```c
#include <stdio.h>

int main() {
    int wert = 22;
    wert = wert / 5 * 5;
    printf("%d", wert);
    return 0;
}
```

Auch Teilschritte sind betroffen!

Darstellung von reellen Zahlen

Bits	Bytes	Werte	Name in C/C++
32	4	$+/-3.4*10^{38}$	`float`
64	8	$+/-1.7*10^{308}$	`double`

- In zwei Geschmacksrichtungen
  - Einfache Genauigkeit: 32 bit → `float`
  - Zweifache Genauigkeit: 64 bit → `double`
- Repräsentation ist standardisiert nach IEEE 754 (1985)
- Sind für viele Zwecke gut, aber auch kein Allheilmittel
- Zur internen Darstellung und Tücken von Gleitkommazahlen später mehr (→ Kapitel 6)

Beispiel - Grenzen der Fließkommadarstellung

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    float f = 3.11231231231231231231231f;
    double d = 3.11231231231231231231231;
    printf("%.20f\n", f);
    printf("%.20f\n", d);
}
```

Einschub: Oktal- und Hexadezimalzahlen

- Es gibt neben den bekannten Dezimal-, Hexdezimal- und Binärsystemen noch ein weiteres
- Oktalsystem Zahlen von 0-7

8^3	8^2	8^1	8^0
0	2	0	2

$\rightarrow 2 \cdot 8^2 + 2 \cdot 8^0 = 2 \cdot 64_D + 2 \cdot 1 = 130_D$

- Häufig in Programmcode verwendet, gerade wenn Bitdarstellungen notwendig sind

Zahlennotation in C/C++

- Hexadezimalzahlen fangen in C mit dem Präfix `0x` an: `0x42` = 66_D
- Oktalzahlen beginnen mit einer `0`: `042` = 34_D
- Binärzahlen haben das Präfix `0b`: `0b101` = 5_D (nur C++; seit C++14)
- Ebenfalls nur in C++: Visuelle Trennung einer Zahl mit `'` zwecks Leserlichkeit:
  `10'500'000`

Das erste Programm (Fortsetzung)

cpp Run ▶

```c
#include <stdio.h>

int main() {
    int i = 3;
    i = i + 5;
    printf("%d", i);
    return 0;
}
```

Jetzt fehlt nur noch printf()

Darstellung Zeichen

- Durchnummerieren aller Zeichen (darstellbar und nicht-darstellbar)
- Festlegung durch ASCII (American Standard Code for Information Interchange)
  - Spendieren 7 Bit für die Darstellung (≠ 1 Byte!)
  - Bei 1 Byte werden unteren 7 Bit genutzt: 01101011
- Heute wird ein Zeichen durch 1 Byte (z. B. UTF-8) dargestellt
- In C/C++ durch ein `char` repräsentiert: `char c = 'a';`

ASCII-Zeichensatz

Spezielle Zeichen im ASCII-Zeichensatz

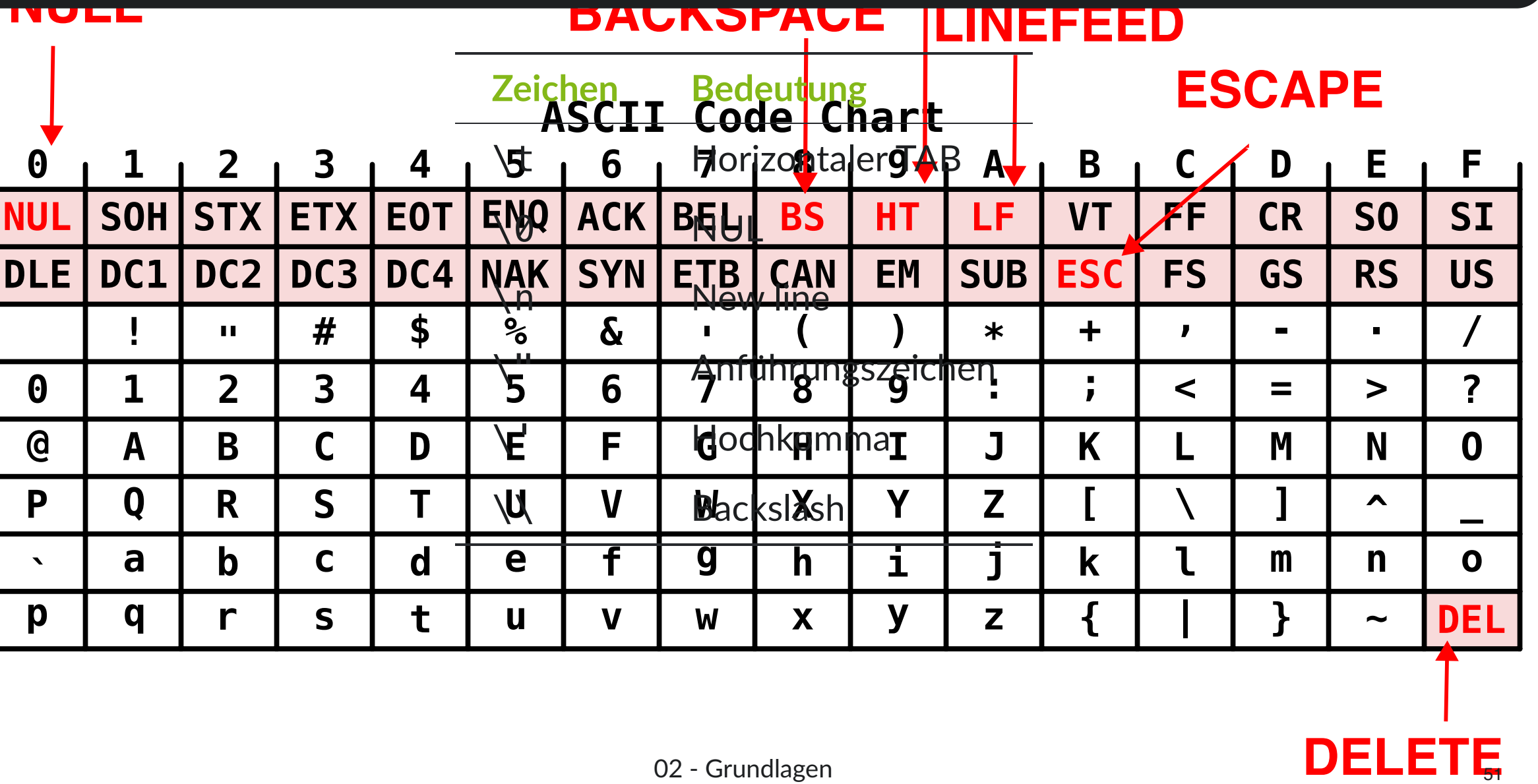
- Darstellbar sind auch Sonderzeichen: ! @ > < ? [ ] …
- Aber es gibt auch **nicht-darstellbare** Zeichen
- Die sog. Steuerzeichen dienen der Steuerung der Ausgabe
- Früher bei Terminals relevant
- Darstellung in „unserer Welt" durch vorangestelltes Backslash
  - Zeilenumbruch: `'\n'`
  - Terminierung einer Zeichenkette: `'\0'`

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ASCII-Zeichensatz

Erklärung nicht-darstellbarer ASCII-Zeichen



ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Dualismus zw. Zeichen und Zahlenwerten

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    char c1 = 'A';
    char c2 = 'a';
    printf("%d\n", c1);
    printf("%d\n", c2);
    return 0;
}
```

- Zeichen werden als Zahlen gespeichert – siehe ASCII-Tabelle
- Rechnen mittels Zeichen möglich: `int a = 'A' - 'a';`
- Vergleich von Zeichen und Zahlen: `'a' == 65`

Zeichenketten (*Strings*)

- Aneinanderreihung von Zeichen zu einem festen Ausdruck (*Literal*)
- Gekennzeichnet durch Anführungszeichen: `"`
- Beispiel: `char * text = "Hello World";`
- Genaue Bedeutung von `char *` → Kapitel 9
- 🚨 **Wichtig:** Zeichenketten sind immer `NULL` -terminiert
  - Am Ende **jeder** Zeichenkette steht das sog. `NULL` -Byte (`'\0'`)
  - Zeichenketten belgen daher **immer** 1 Byte mehr
- Beispiel: `"Hello World"`
  - Länge: 11 (sichtbare) Zeichen)
  - Belegt aber 12 Bytes: `"Hello World\0"`

Einschub: Ein-/Ausgabe

Wir waren hiermit gestartet …

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    int i = 3;
    i = i + 5;
    printf("%d", i);
    return 0;
}
```

Zurück zu printf()!

Einschub: Ein-/Ausgabe

- Fast jedes sinnvolle Programm benötigt irgendeine Form von Informationseingabe bzw. -ausgabe
- Häufige Bezeichnung: Input/Output (I/O)
- Einfachste Möglichkeit für Ausagbe in C: `printf()`
- `printf()` verwendet sogenannten *Formatstring*, um an entsprechenden Stellen den Inhalt von Variablen einzusetzen

Spezifizierer	Umwandlung
%d	Ganzzahl (Dezimal)
%x	Ganzzahl (Hexadezimal)
%s	String
%c	Character
%f	Float

Einschub: Ein-/Ausgabe

- `printf()` wandelt die gegebenen Variablen in Text um/ fügt sie in den Formatstring ein
- Vollständige Liste von Spezifizierern unter cppreference.com
- Beispiel: Float mit 2 Nachkommastellen
  `printf("%.2f", 3.14159);` → `3.14`

Spezifizierer	Umwandlung
%d	Ganzzahl (Dezimal)
%x	Ganzzahl (Hexadezimal)
%s	String
%c	Character
%f	Float

Einschub: Ein-/Ausgabe

- Eingabe analog zur Ausgabe mittels `scanf()`
- Benötigt ebenfalls Formatstring, um Text in gewünschten Datentyp umwandeln zu können

```c
#include <stdio.h>

int main() {
    float f1; // Speicher
    float f2;   // Speicherplatz anlegen
    scanf("%f %f", &f1, &f2); // Befüllung der Variablen
    printf("%f", f1 + f2);
    return 0;
}
```

Wir können nun mit unserem Programm interagieren 🥳

Mehrfach das Gleiche

Wie lege ich z. B. fünfmal einen Integer an?

```
int a_1;
int a_2;
int a_3;
int a_4;
int a_5;
```

Und für 1000 Werte?

Das will man so nicht!

Felder (*Arrays*)

- Ermöglicht das Zusammenfassen von *n* Instanzen des gleichen Typs
- Alles landet in einer Variablen
- Variablendeklaration um Größe und eckige Klammern erweitern
  - Vorher: `int num_array;` → **ein** Integer
  - Jetzt: `int num_array[5];` → ein Feld mit **fünf** Integern
- ☝️ Jetzt wird auch klar, wie Zeichenketten abgelegt werden
  - `char a = "Hello World";` → geht nicht!
  - `char a[] = "Hello World";`

Arrays

```
int num_array[5];
```

- Erstellt Variable namens `num_array` als

  aufeinanderfolgende und zusammenhängende Menge von `5` `int` -Werten

- Zugriff auf einzelne Elemente über Index
  - Zählung beginnt bei `0`
  - Indizes im Bereich `[0, n-1]`
  - Hier: `[0,4]`
- Zugriff auf erstes Element mit `[0]` → `printf("%d", num_array[0]);`
- Schreiben ist äquivalent dazu: `num_array[0] = 1;`

Arrays

```
int num_array[5];
```

- Initialisierung erfolgt …
  - separat für jeden Index: `num_arrays[0] = 1;`
  - vollständig bei Deklaration: `int num_arras[5] = {1, 2, 3, 4, 5};`
  - teilweise bei der Deklaration: `int num_arras[5] = {1, 2};`
- 🚨 Achtung: Standardwerte für restliche Werte variiert (→ später mehr dazu)
- Bis hier: Initialisiert bitte **jedes** Element eines Feldes

Beispiel zu Feldern

cpp Run ▶

```cpp
#include <stdio.h>

int main() {
    float f[3] = {1.0, 2.3};
    printf("f[0] = %.1f\n", f[0]);
    printf("f[1] = %.1f\n", f[1]);
    printf("f[2] = %.1f\n", f[2]);
    return 0;
}
```

Regeln für Arrays:

- Können von beliebigem Typ sein
- **Alle** Elemente sind vom **gleichen** Typ
- Typ kann nicht mehr geändert werden
- Länge ist ebenfalls
- Grund dafür ist ein

  Menge an benötigtem Speicher **muss** vorher bekannt sein

  (→ aufeinanderfolgend und zusammenhängend)

Denkt vorher über die Array-Größe nach!

Exkurs: Variable Length Arrays (VLAs)

- Variable Length Arrays (VLAs) sind ein (gescheiterter) Versuch, Arrays mit dynamischer Länge zu ermöglichen
- Nur in C möglich (ab C99), nicht in C++
- Notation

```c
#include <stdio.h>

int main() {
    int n = 10;
    int vla[n]; // Variable Length
    return 0;
}
```

- Hat eine Reihe von Problemen → **Vermeiden!**

Exkurs: Klammern in C/C++

- Bis hierhin wurden drei verschiedene Klammern eingeführt (`( )`, `{}`, `[]`)
- Im Englischen wird unterschieden
  - `( )` → *Parentheses* (Parenthesen)
  - `{}` → *Braces* / *Curly Braces* (Akkoladen)
  - `[]` → *Brackets* (Eckige Klammern)
- Außerdem gibt es noch *spitze* Klammern bzw. *Angled Brackets* (<>) (Winkelklammern)

Zusammenfassung

- Anhand von `Hello World`-Beispiel die Grundzüge
  - der Darstellung von Zahlen behandelt,
  - erläutert wie Zeichenketten abgelegt werden,
  - Felder vorgestellt und
  - das Zusammenfassen von mehreren Variablen zu einem Feld erklärt.