

Wintersemester 2006/07

**Einführung in die Informatik für
Naturwissenschaftler und Ingenieure
(alias Einführung in die Programmierung)
(Vorlesung)**

Prof. Dr. Günter Rudolph

Fachbereich Informatik

Lehrstuhl für Algorithm Engineering





Kapitel 2: Darstellung von Information

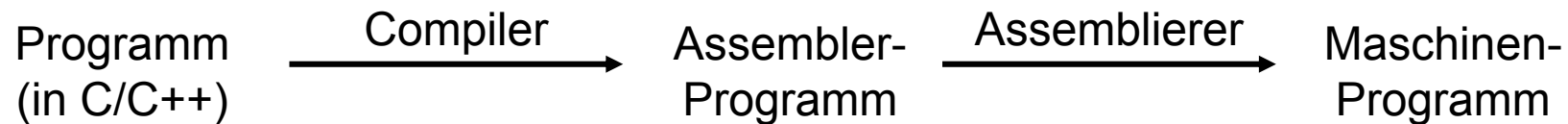
Inhalt

- Einfache Datentypen
- Zahldarstellungen im Rechner
- Bezeichner
- Datendefinition, Zuweisung, Initialisierung



Realisierung eines Programms

- Problemanalyse
- Spezifikation
- Algorithmenentwurf
- Formulierung eines Programms



- Ausführung erfolgt mit Hilfe des Laufzeitsystems



Notwendig für Programmierung:

- Ausschnitte der realen Welt müssen im Rechner abgebildet werden können!
- Dazu gehören etwa **Daten** in vielerlei Form!
- Bestimmte Formen dieser Daten haben gemeinsame, **typische** Eigenschaften!
- Solche werden zusammengefasst zu so genannten **Datentypen**.



Unterscheidung

- **Einfache Datentypen**

sind elementar bzw. nicht auf andere Typen zurückführbar.

Beispiel: positive ganze Zahlen

- **Zusammengesetzte Datentypen**

entstehen baukastenartig durch Zusammensetzen von einfachen Datentypen.

Beispiel: ein Paar aus zwei positiven ganzen Zahlen



Wie werden Zahlen im Rechner dargestellt?

- Bit $\in \{ 0, 1 \}$
- 8 Bit = 1 Byte
- Speicher im Rechner = lineare Folge von Bytes bzw. Bits
- Duales Zahlensystem:
 - n Bits: $(b_{n-1} b_{n-2} \dots b_2 b_1 b_0)$ mit $b_k \in \{ 0, 1 \}$
 - 2^n mögliche Kombinationen (= verschiedene Zahlen)
 - Umwandlung in Dezimalzahl:

$$\sum_{k=0}^{n-1} b_k 2^k$$



Einfache Datentypen

- **Ganzzahlen ohne Vorzeichen (unsigned)**

Bit	Byte	Wertevorrat	Name in C/C++
8	1	0 ... 255	<code>unsigned char</code>
16	2	0 ... 65 535	<code>unsigned short int</code>
32	4	0 ... 4 294 967 295	<code>unsigned int</code>
32	4	0 ... 4 294 967 295	<code>unsigned long int</code>

ACHTUNG: Wertebereiche rechnerabhängig! Hier: PC mit Pentium IV.



Negative Zahlen?

- Gleicher Vorrat an verschiedenen Zahlen!
- ⇒ Vorrat muss anders aufgeteilt werden!

Naiver Ansatz:

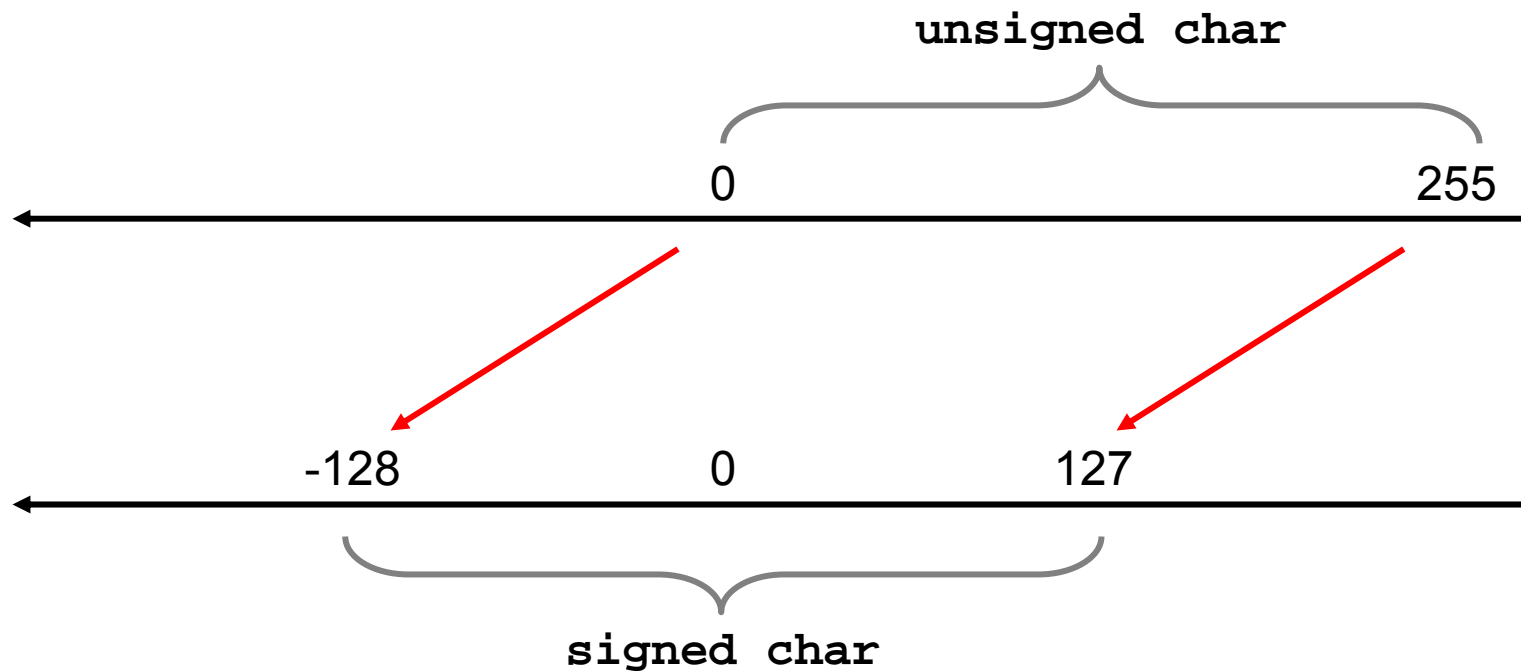
- Man verwendet $n-1$ Bit zur vorzeichenlosen Zahldarstellung
 - ⇒ Das ergibt Zahlen im Bereich $0 \dots 2^{n-1}-1$, also 0 bis 127 für $n=8$
- Bit n repräsentiert das Vorzeichen: 0 = positiv, 1 = negativ
 - ⇒ Bei $n = 8$ ergibt das Zahlen im Bereich -127 bis 127
 - ⇒ Probleme:
 - Die Null zählt doppelt: +0 und -0
 - Eine mögliche Zahldarstellung wird verschenkt!





Negative Zahlen?

- Gleicher Vorrat an verschiedenen Zahlen!
- ⇒ Vorrat muss anders aufgeteilt werden!





Bitrepräsentation von negativen Zahlen:

- Man muss nur das Stellengewicht des höchstwertigen Bits negativ machen!

Bit	7	6	5	4	3	2	1	0
unsigned	128	64	32	16	8	4	2	1
signed	-128	64	32	16	8	4	2	1

- Beispiel: $10101001_2 = -128 + 32 + 8 + 1 = -87$
- Mit Bit 0 – 6 sind Zahlen zwischen 0 und 127 darstellbar.
Falls Bit7 = 0 \Rightarrow 0 bis 127
Falls Bit7 = 1 \Rightarrow -128 bis -1



Bitrepräsentation von Ganzzahlen mit Vorzeichen: (n = 8)

7	6	5	4	3	2	1	0	unsigned	signed
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	2	2
		
0	1	1	1	1	1	1	1	127	127
1	0	0	0	0	0	0	0	128	-128
1	0	0	0	0	0	0	1	129	-127
1	0	0	0	0	0	1	0	130	-126
		
1	1	1	1	1	1	1	1	255	-1



Einfache Datentypen

- **Ganzzahlen mit Vorzeichen**

Bit	Byte	Wertevorrat	Name in C/C++
8	1	-128 ... 127	<code>char</code>
16	2	-32768 ... 32767	<code>short int</code>
32	4	-2147483648 ... 2147483647	<code>int</code>
32	4	-2147483648 ... 2147483647	<code>long int</code>

ACHTUNG: Wertebereiche rechnerabhängig! Hier: PC mit Pentium IV.



Darstellung von Information

Zwischenfragen:

- Wie werden Daten im Programm angelegt bzw. abgelegt?
- Wie kann ich sie wieder finden und abrufen bzw. verändern?

⇒ Rechner muss angewiesen werden Speicherplatz für Daten zu reservieren.

⇒ Das geschieht formal im Programm durch eine **Datendefinition**:

Angabe von **Datentyp** und **Bezeichner**.

Beispiele:

```
char a;  
short b;  
unsigned long c;
```

Adresse	Daten	Name
11100110	00001001	a
11100101	10001100	b
11100100	01101001	
11100011	10011101	c
11100010	11110011	
11100001	10101000	
11100000	00110001	



Datendefinition (DD)

```
unsigned int Postleitzahl;
```

Was geschieht?

1. DD reserviert Speicher
2. DD legt Wertevorrat fest
3. DD ermöglicht eindeutige Interpretation des Bitmusters
4. DD legt zulässige Operatoren fest

Was geschieht nicht?

DD weist keinen Wert zu!

⇒ Zufällige Bitmuster im Speicher! ⇒ Häufige Fehlerquelle!



Zuweisung

- Beispiel: `Postleitzahl = 44221;`
- Vor einer Zuweisung muss eine Datendefinition stattgefunden haben!
- Was geschieht?
 - ⇒ Die Zahl wird gemäß Datentyp interpretiert & in ein Bitmuster kodiert.
 - ⇒ Das Bitmuster wird an diejenige Stelle im Speicher geschrieben, die durch den Bezeichner symbolisiert wird.

Initialisierung

- Beispiel: `unsigned int Postleitzahl = 44221;`
- Datendefinition mit anschließender Zuweisung



Bezeichner

Bauplan:

- Es dürfen nur Buchstaben **a** bis **z**, **A** bis **Z**, Ziffern **0** bis **9** und der Unterstrich **_** vorkommen.
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein.
- Prinzipiell keine Längenbeschränkung.
- **Schlüsselwörter** dürfen nicht verwendet werden.

```
Winkel
EinkomSteuer
Einkom_Steuer
einkom_Steuer
_OK
x3
_x3_und_x4_
_99
```




Schlüsselwörter

... sind reservierte Wörter der jeweiligen 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

Schlüsselwörter der Programmiersprache C



Schlüsselwörter

... sind reservierte Wörter der jeweiligen Programmiersprache!

<code>asm</code>	<code>export</code>	<code>private</code>	<code>true</code>
<code>bool</code>	<code>false</code>	<code>protected</code>	<code>try</code>
<code>const_cast</code>	<code>friend</code>	<code>public</code>	<code>typeid</code>
<code>catch</code>	<code>inline</code>	<code>static_cast</code>	<code>typename</code>
<code>class</code>	<code>mutable</code>	<code>template</code>	<code>using</code>
<code>delete</code>	<code>namespace</code>	<code>reinterpret_cast</code>	<code>virtual</code>
<code>dynamic_cast</code>	<code>new</code>	<code>this</code>	
<code>explicit</code>	<code>operator</code>	<code>throw</code>	

Zusätzliche Schlüsselwörter der Programmiersprache C++



Ganzzahlen: Binäre Operatoren

- Addition → Operator: +
- Subtraktion → Operator: -
- Multiplikation → Operator: *
- Ganzzahldivision → Operator: /
- Modulo → Operator: %

Beispiele:

A + b;

3 * x3 - 8 / Faktor;

wert % 12;



Ganzzahlen: Modulo-Operator %

- liefert den Rest der Ganzzahldivision
- aus Alltagsleben bekannt, aber selten unter diesem Namen

Beispiel: Digitaluhr

- Wertevorrat: 0:00 bis 23:59
- Stundenanzeige springt nach 23 auf 0
- Minutenanzeige springt nach 59 auf 0
- C/C++:

```
unsigned int stunde, laufendeStunde = 37;  
stunde = laufendeStunde % 24;
```



Ganzzahlen: Häufige Fehlerquellen ...

- Zahlenüberlauf

```
short m = 400, n = 100, p = 25, k;  
k = m * n / p;
```

⇒ Resultat: **k = -1021;** ☹️

Warum?

- $400 * 100$ ergibt 40000 ⇒ zu groß für Datentyp **short** (< 32768)
- $40000 = 1001\ 1100\ 0100\ 0000_2$
- Interpretation als Datentyp short: $-32768 + 7232 = -25536$
- Schließlich: $-25536 / 25 = -1021$



Ganzzahlen: Häufige Fehlerquellen ...

- Zahlenüberlauf: Addition

```
short a = 32600, b = 200,  
c = a + b;
```

⇒ Resultat: `c = -32736;` ☹️

- Zahlenüberlauf: Subtraktion

```
unsigned short m = 100, n = 101, k;  
k = m - n;
```

⇒ Resultat: `k = 65535;` ☹️

Programmiertes
Unheil!





Ganzzahlen: Häufige Fehlerquellen ...

- Ganzzahldivision ist reihenfolgeabhängig!

Beispiel:

$$\begin{array}{r} 20 * 12 / 3 \\ \hline 240 / 3 \\ \hline 80 \end{array}$$

$$\begin{array}{r} 20 / 3 * 12 \\ \hline 6 * 12 \\ \hline 72 \end{array}$$



Merken!

- Wird Zahlenbereich bei Ganzzahlen über- oder unterschritten (auch bei Zwischenergebnissen), dann entstehen unvorhersehbare, falsche Ergebnisse **ohne Fehlermeldung!**
- Es liegt im **Verantwortungsbereich des Programmierers**, die geeigneten Datentypen auszuwählen (Problemanalyse!).
- Die Verwendung von „größeren“ Datentypen verschiebt das Problem nur auf größere Wertebereiche: es wird i.A. dadurch **nicht gelöst!** Es müssen ggf. Vorkehrungen getroffen werden: z. B. Konsistenzprüfungen.



Reelle Zahlen

- In C/C++ gibt es zwei Datentypen für reelle Zahlen:

Bit	Byte	Wertebereich	Name in C/C++	Stellen
32	4	$\pm 3.4 * 10^{-38} \dots \pm 3.4 * 10^{+38}$	<code>float</code>	7
64	8	$\pm 1.7 * 10^{-308} \dots \pm 1.7 * 10^{+308}$	<code>double</code>	15

Stellen = signifikante Stellen



Reelle Zahlen

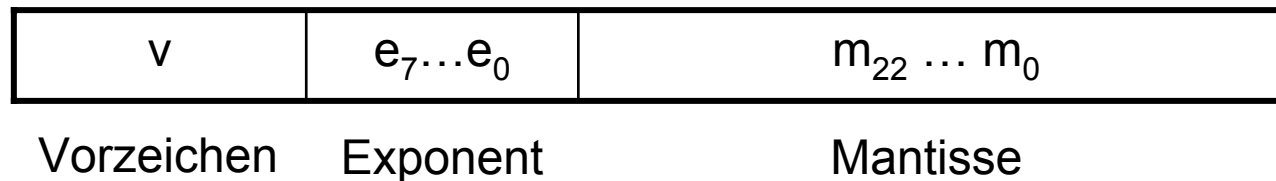
- **Float** vs. **Long**:

beide 4 Byte, aber riesiger Unterschied im Wertebereich!

Wie geht das denn?

⇒ Durch Verlust an Genauigkeit im niederwertigen Bereich der Zahl!

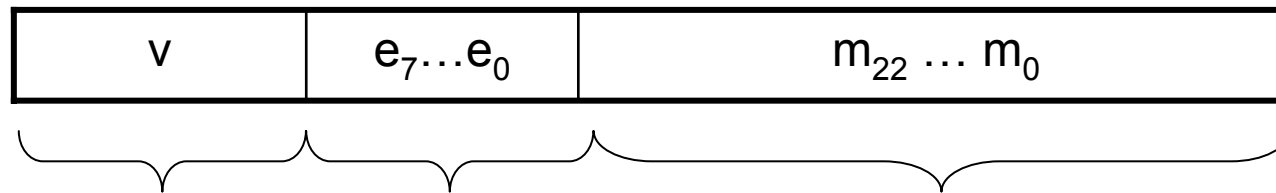
- Repräsentation ist standardisiert: IEEE-Standard P754 (1985)
- Beispiel: `float` (32 bit)





Reelle Zahlen

- Repräsentation ist standardisiert: IEEE-Standard P754 (1985)
- Beispiel: `float` (32 bit)



**signed
char**

0 ⇒ +1

-128

⋮

1 ⇒ -1

+127

normiert: $1 \leq m \leq 2$,
wobei virtuelles Bit $m_{23} = 1$



Reelle Zahlen

`float pi1 = 3.141592;` ← 7 signifikante Stellen
`double pi2 = 3.14159265358979;` ← 15 signifikante Stellen
korrekte

Weitere gültige Schreibweisen:

12345.678 Festkommazahl (*fixed format*)

1.23456e5 Fließkommazahl (*floating point*)

.345

+34.21e-91

Achtung:

Dezimaldarstellung

immer mit Punkt,

niemals mit Komma!



Einfache Datentypen

- Zeichen

- Ein Zeichen wird in einem Byte gespeichert (**char**)
- Zuordnung: Zeichen ↔ Zahl (Code)
- ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), 7-Bit-Code

0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI	} Steuer- zeichen
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
32	SP	!	“	#	\$	%	&	'	()	*	+	,	-	.	/	
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL	



Darstellung von Information

Einige wichtige nicht druckbare Steuerzeichen:

horizontal tabulation line feed carriage return

null →

space →

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

← delete



Zeichen

- Zeichen werden gemäß ihrem Code als Zahl gespeichert

⇒ deshalb kann man mit Zeichen rechnen:

```
char c = '7';
```

Code von '7' ist 55

```
int zahl = c - '0';
```

Code von '0' ist 48

Resultat:

zahl = 7

- ... und man kann Zeichen vergleichen:

```
'a' < 'b'
```

ist wahr, weil 97 < 98

- Erst bei der Ausgabe wird Datentyp `char` wieder als Zeichen interpretiert.



Zeichen

- Datendefinition: `char Zeichen;`
- Zuweisung: `Zeichen = 'x';`
- Darstellbare Zeichen:
 - Buchstaben: `'a'` bis `'z'` und `'A'` bis `'Z'`
 - Ziffern: `'0'` bis `'9'`
 - Satzzeichen: z.B. `'!'` oder `':'`
 - Sonderzeichen: z.B. `'@'` oder `'>'` oder `'}'` oder Leerzeichen

- Steuerzeichen
mit Fluchtsymbol
(Umschalter): `\`

<code>\a</code>	alarm (BEL)	<code>\"</code>	Anführungsstriche
<code>\b</code>	backspace	<code>\'</code>	Hochkomma
<code>\t</code>	horizontal tabulator (TAB)	<code>\?</code>	Fragezeichen
<code>\n</code>	new line	<code>\\</code>	backslash



Zeichenketten (Strings)

*Datendefinition etc.
kommt später!*

- Aneinanderreihung von Zeichen
- Gekennzeichnet durch doppelte Hochkommata: `"`
- Beispiele:

- `"Dies ist eine Zeichenkette!"`

```
Dies ist eine Zeichenkette!
```

- `"Das ist jetzt\nneu."`

```
Das ist jetzt
neu.
```

- `"\nThe C++ Programming Language"\n\tby B. Stroustrup"`

```
"The C++ Programming Language"
    by B. Stroustrup
```