technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2018/19

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

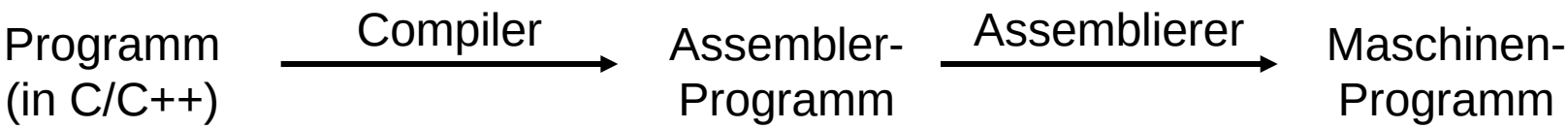
TU Dortmund

Inhalt

- Einfache Datentypen
- Zahldarstellungen im Rechner
- Bezeichner
- Datendefinition, Zuweisung, Initialisierung
- Erste Programme
- Exkurs: Grammatiken
- Zusammengesetzte Datentypen
  - Feld (array)
  - Verbund (struct)
  - Aufzählung (enum)

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

- **Zusammengesetze Datentypen**

  entstehen baukastenartig durch Zusammensetzen
  von einfachen Datentypen.

  Beispiel: ein Paar aus zwei positiven ganzen Zahlen

Wie werden Zahlen im Rechner dargestellt?

- Bit ∈ { 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} … b_2 b_1 b_0) mit b_k ∈ { 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	**`unsigned char`**
16	2	0 … 65 535	**`unsigned short int`**
32	4	0 ... 4 294 967 295	**`unsigned int`**
32	4	0 ... 4 294 967 295	**`unsigned long int`**

ACHTUNG: Wertebereiche rechnerabhängig! Hier: 32bit-Rechner.

Negative Zahlen?

- Gleicher Vorrat an verschiedenen Zahlen!

⇒ Vorrat muss anders aufgeteilt werden!

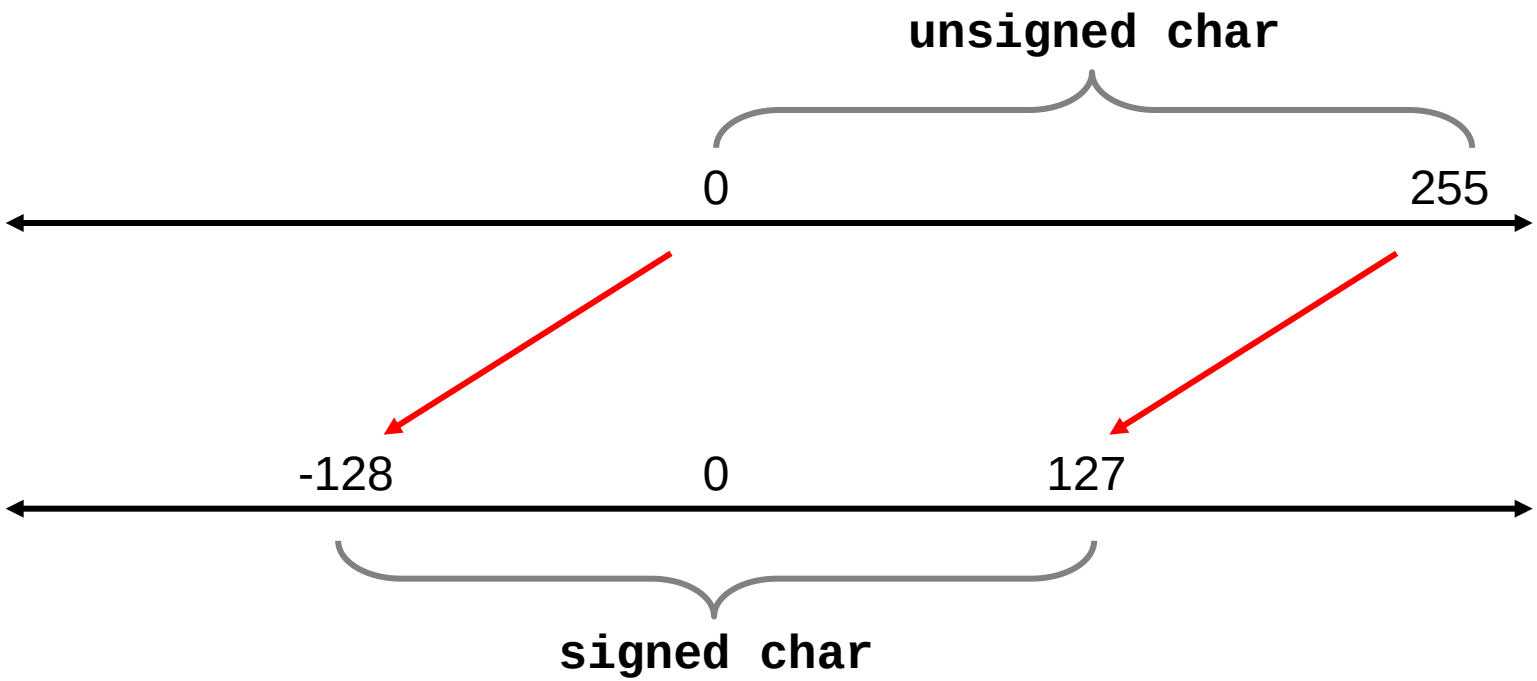
Naiver Ansatz:

- Man verwendet n-1 Bit zur vorzeichenlosen Zahldarstellung
  - ⇒ Das ergibt Zahlen im Bereich 0 … 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	`char`
16	2	-32768 … 32767	`short int`
32	4	-2147483648 ... 2147483647	`int`
32	4	-2147483648 ... 2147483647	`long int`

ACHTUNG: Wertebereiche rechnerabhängig! Hier: 32bit-Rechner.

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	
11100100	01101001	b
11100011	10011101	
11100010	11110011	
11100001	10101000	
11100000	00110001	c

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`	`typeof`
`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!

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

Zusätzliche Schlüsselwörter der Programmiersprache C++ weitere in C++11

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?

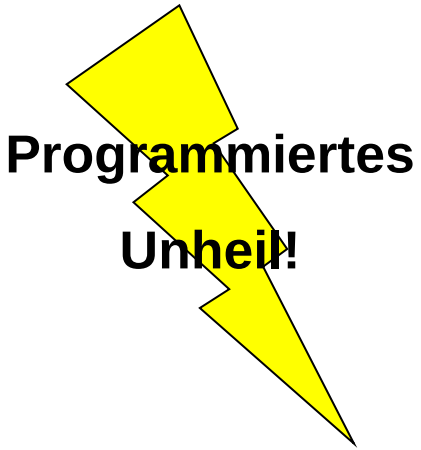
- 400 * 100 ergibt 40000 ⇒ zu groß für Datentyp **`short`** (< 32768)
- 40000 = 1001 1100 0100 0000_2
- Interpretation als Datentyp short: -32768 + 7232 = -25536
- Schließlich: -25536 / 25 = -1021

Ganzzahlen: Häufige Fehlerquellen …

- Zahlenüberlauf: Addition

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

⇒ Resultat: `c = -32736;` ☹



- Zahlenüberlauf: Subtraktion

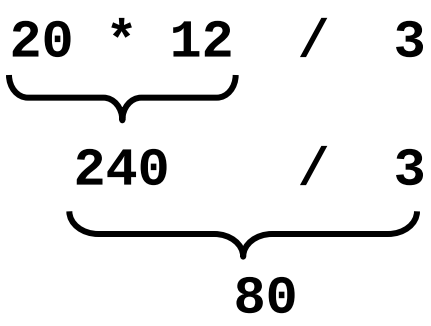
```
unsigned short m = 100, n = 101, k;
k = m - n;
```

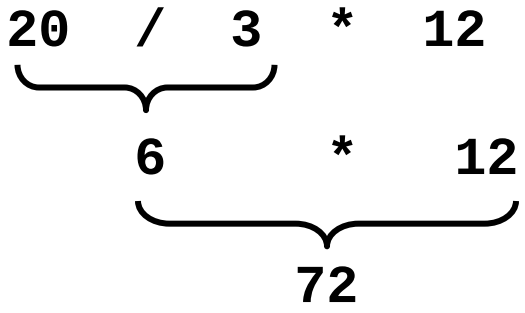
⇒ Resultat: `k = 65535;` ☹

Ganzzahlen: Häufige Fehlerquellen …

- Ganzzahldivision ist reihenfolgeabhängig!

Beispiel:





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	$\pm 3.4 * 10^{-38} \ldots \pm 3.4 * 10^{+38}$	`float`	7
64	8	$\pm 1.7 * 10^{-308} \ldots \pm 1.7 * 10^{+308}$	`double`	15

Stellen = signifikante Stellen

Reelle Zahlen

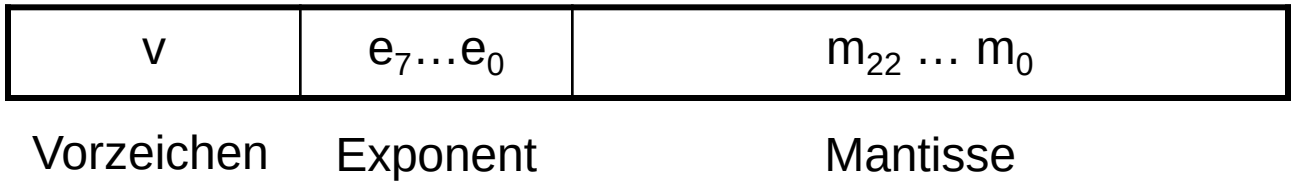
- **`Float`** vs. **`Long`**:

  beide 4 Byte, aber riesiger Unterschied im Wertebereich!

  Wie geht das denn?

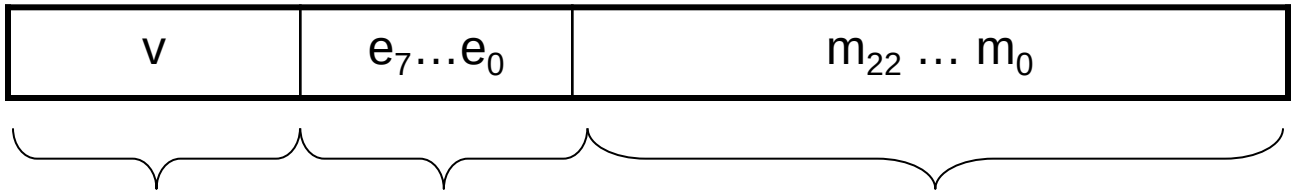
  $\Rightarrow$ Durch Verlust an Genauigkeit im niederwertigen Bereich der Zahl!

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



Reelle Zahlen

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



`unsigned char`

`0`

□

`255`

$0 \Rightarrow +1$

$1 \Rightarrow -1$

normiert: $1 \le m \le 2$,

wobei virtuelles Bit $m_{23} = 1$

`e = E + 127`

Reelle Zahlen

```
float pi1   = 3.141592;
double pi2  = 3.14159265358979;
```

7 signifikante Stellen

15 signifikante Stellen

korrekte

Weitere gültige Schreibweisen:

12345.678 Festkommazahl (*fixed format*)

1.23456e4 Fließkommazahl (*floating point*)

.345

+34.21e-91

Achtung:

Dezimaldarstellung

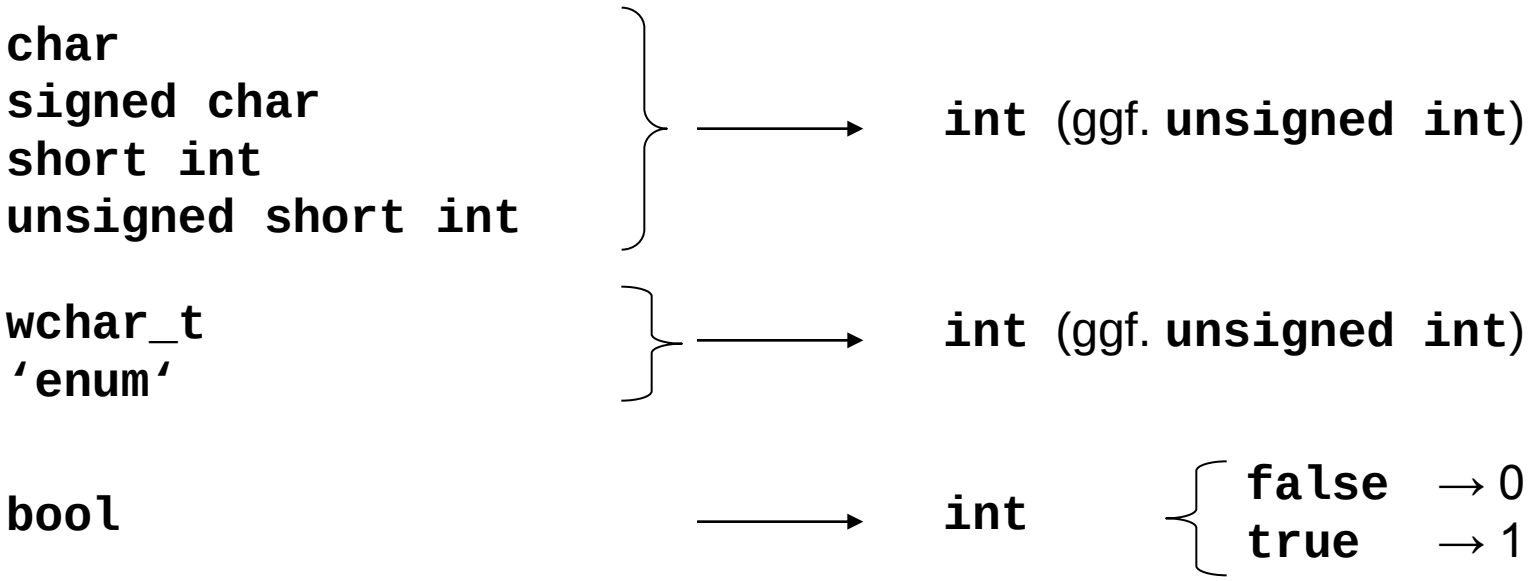
immer mit Punkt,

niemals mit Komma!

Exkurs: Typumwandlung

- ***Automatisch (Promotionen)***

  → das Rechenwerk braucht gleiche Typen für Rechenoperation



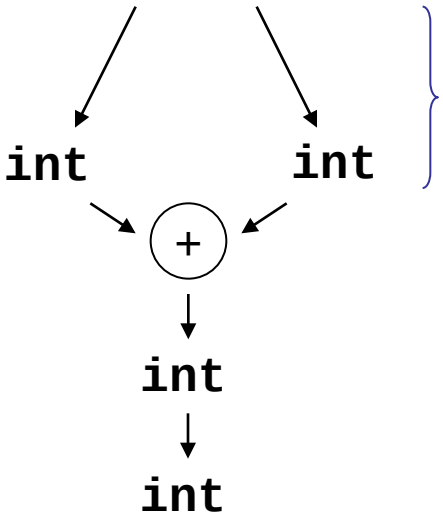
Exkurs: Typumwandlung

- ***Automatisch (Promotionen)***

  → das Rechenwerk braucht gleiche Typen für Rechenoperation

Bsp:

```
char c = 3;
short s = 1024;
int i = c + s;
```



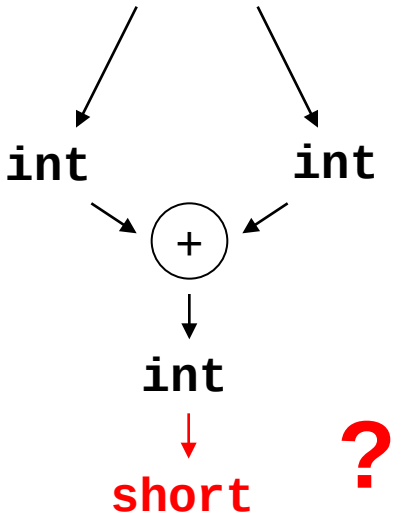
Umwandlung zu `int`

`int` -Addition

Ergebnis: `int`

Zuweisung

```
char c = 3;
short s = 1024;
short i = c + s;
```



?

Exkurs: Typumwandlung

- ***Umwandlungen***
  - Ganze Zahlen
    - Zieltyp **`unsigned`**

      → alle Bits aus der Quelle, die ins Ziel passen, werden kopiert
      → der Rest (höherwertige Bits) wird ggf. ignoriert

```
unsigned char uc = 1023; // binär 11 1111 1111
```

8 bit 10 bit ⇒ **`uc = 255`**

    - Zieltyp **`signed`**

      → Wertübernahme, wenn im Ziel darstellbar; sonst undefiniert!

```
signed char sc = 1023; // plausible Resultate 127 oder -1
```

Exkurs: Typumwandlung

- ***Umwandlungen***
  - Fließkommazahlen
    - **`float`** → **`double`**

      ⇒ passt immer ☑

    - **`double`** → **`float`**

      ⇒ Wertübernahme, wenn im Ziel darstellbar; sonst undefiniert!

    - **`float/double`** → Ganzzahl

      ⇒ Ungenauigkeiten und möglicher Datenverlust

      `int i = 2.6;` → i = 2; `char c = 2.3e8;` → c = -128;

Der Compiler **warnt** vor möglichem Datenverlust!

Warnungen des Compiler **nicht ignorieren**!

Exkurs: Typumwandlung

Trauen Sie nicht **vorbehaltslos** den Ergebnissen des Rechners!

Bsp:

$$333.75\,y^6+x^2\,(11\,x^2y^2-y^6-121\,y^4-2)+5.5\,y^8+\frac{x}{2\,y}$$

für $x = 77617\,,\quad y = 33096$

Resultat bei doppelter Genauigkeit (double): -1.18059e+021

→ exakt: ***-54767 / 66192 = -0.827396...***

Exkurs: Typumwandlung

Vorbemerkung:
Die Regeln von C++ garantieren, dass Typfehler unmöglich sind.
Theorie: Wenn Programm sauber kompiliert, dann keine Durchführung von ungültigen / unsauberen Operationen an Objekten.

→ Wertvolle Garantie! → nicht leichtfertig aufgeben!

Aber: explizite Typumwandlung (cast) untergräbt das Typsystem!

explizite Typumwandlung:

C Stil:

```
(T) Ausdruck  // wandelt Ausdruck in den Typ T um
```

mißbilligt (deprecated)

```
T(Ausdruck)   // wandelt Ausdruck in den Typ T um
```

Nicht verwenden!

Explizite Typumwandlung (C++)

Vorschau:

Hier nur zur Vollständigkeit. Wir kommen später darauf zurück!

- **`const_cast<T>(Ausdruck)`**

  → beseitigt Konstanz von Objekten

- **`dynamic_cast<T>(Ausdruck)`**

  → zum „Downcasten“ bei polymorphen Quelltypen
  → umwandeln in einen abgeleiteten Typ
  → Fehlschlag bei **`*`** ergibt Nullpointer, bei **`&`** Ausnahme **`bad_cast`**

- **`reinterpret_cast<T>(Ausdruck)`**

  → verwendet auf niedriger Ebene (Uminterpretation des Bitmusters)
  → Ziel muss mindestens soviele Bits wie Quelle haben, sonst … 💣☠☹

- **`static_cast<T>(Ausdruck)`**

  → zum Erzwingen von impliziten Typumwandlungen

Exkurs: Typumwandlung

Wenn im Code viele Casts notwendig sind,

dann stimmt meistens etwas mit dem Design des Programms nicht!

Wenn im Code ein Cast notwendig ist,

dann die Cast-Operation von C++ verwenden, weil

1. minimale automatische Typprüfung möglich (statisch / dynamisch);
2. man sich mehr Gedanken darüber macht, was man eigentlich tut;
3. für Außenstehende präziser angezeigt wird, was Sie tun.

Wenn im Code ein Cast notwendig ist,

dann die Cast-Operation in einer Funktion verbergen.

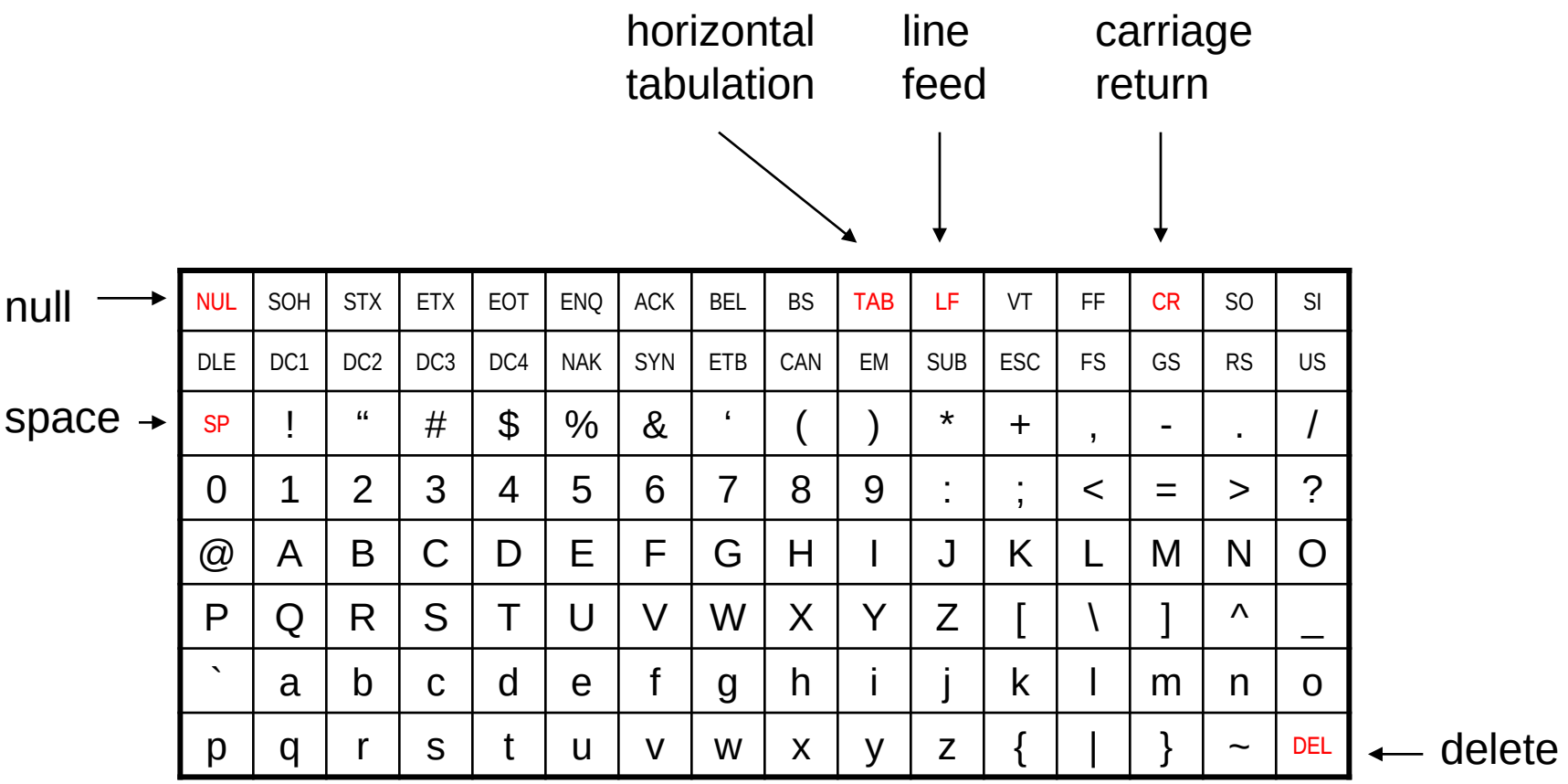
Einfache Datentypen

- **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
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

Steuer-zeichen

Einige wichtige nicht druckbare Steuerzeichen:



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

Zeichen

- Zeichen werden gemäß ihrem Code als Zahl gespeichert

⇒ deshalb kann man mit Zeichen rechnen:

```
char c = '7';
int zahl = c - '0';
```

Code von **'7'** ist 55

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): \

`\a`	alarm (BEL)	**`\"`**	Anführungsstriche
`\b`	backspace	**`\'`**	Hochkomma
`\t`	horizontal tabulator (TAB)	**`\?`**	Fragezeichen
`\n`	new line	**`\\`**	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.
```

  - **“\“The C++ Programming Language\“\n\tby B. Stroustrup“**

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

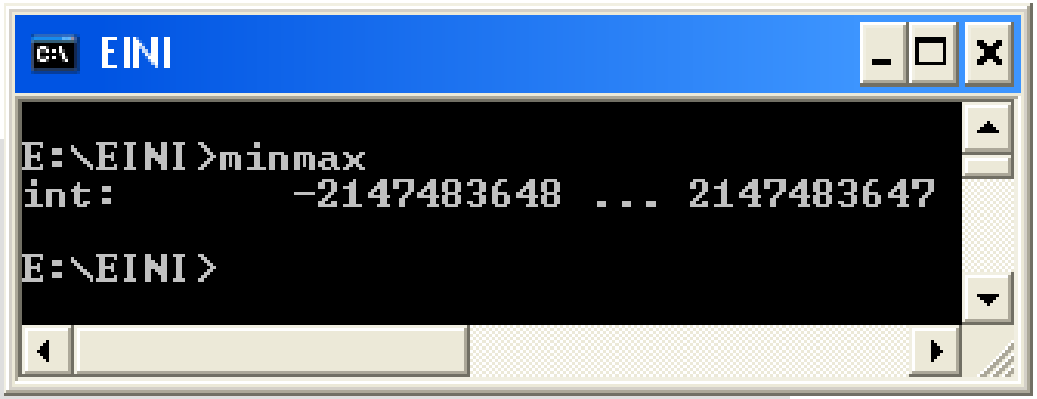
Das erste C++ Programm:

```
#include <iostream>

int main()
{
  std::cout << “Das ist eine Zeichenkette!“ << ‘\n‘;
  return 0;
}
```

- **`#include <iostream>`** bindet Ein-/Ausgabemöglichkeit aus Bibliothek ein
- **`int main()`** kennzeichnet Hauptprogramm, gibt Datentyp integer zurück
- **`std::cout`** ist der Ausgabestrom; alles rechts von << wird ausgegeben
- **`return 0`** gibt den Wert 0 an das Betriebssystem zurück (0: alles OK!)

Noch ein C++ Programm:




```
#include <iostream>
#include <climits>

int main()
{
  std::cout << "int:        "
            << INT_MIN  << " ... "
            << INT_MAX  << std::endl;
  return 0;
}
```

- **`#include <climits>`** bindet Konstanten für Wertebereiche ein
- **`INT_MIN`** und **`INT_MAX`** sind Konstanten aus Bibliothek **`climits`**
- **`std::endl`** ist eine Konstante für Beginn einer neuen Zeile

Einfache Datentypen

- **Logischer Datentyp `bool`**
  - Zum Speichern von Wahrheitswerten „wahr“ und „falsch“
  - Wertevorrat: **`true`** und **`false`**
  - Datendefinition: **`bool b;`**
  - Zuweisung: **`b = true;`**
    oder: **`int x = 9; b = x > 7;`**
  - Zum Überprüfen von **Bedingungen**
  - Operationen:

Name	*C/C++*	*Beispiel*
AND	**`&&`**	**`b && x < 7`**
OR	**`||`**	**`b || x > 8`**
NOT	**`!`**	**`!b`**

Wahrheitstafeln

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

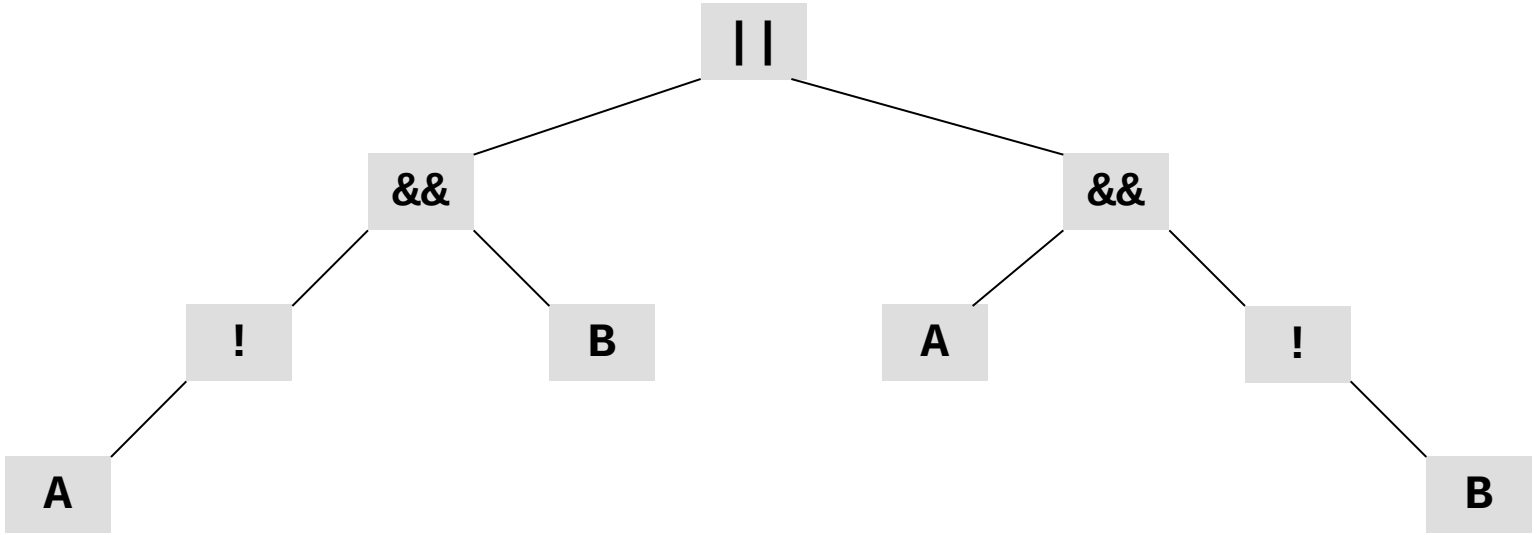
A	!A
false	true
false	true
true	false
true	false

Priorität der Operatoren

1. NOT
2. AND
3. OR

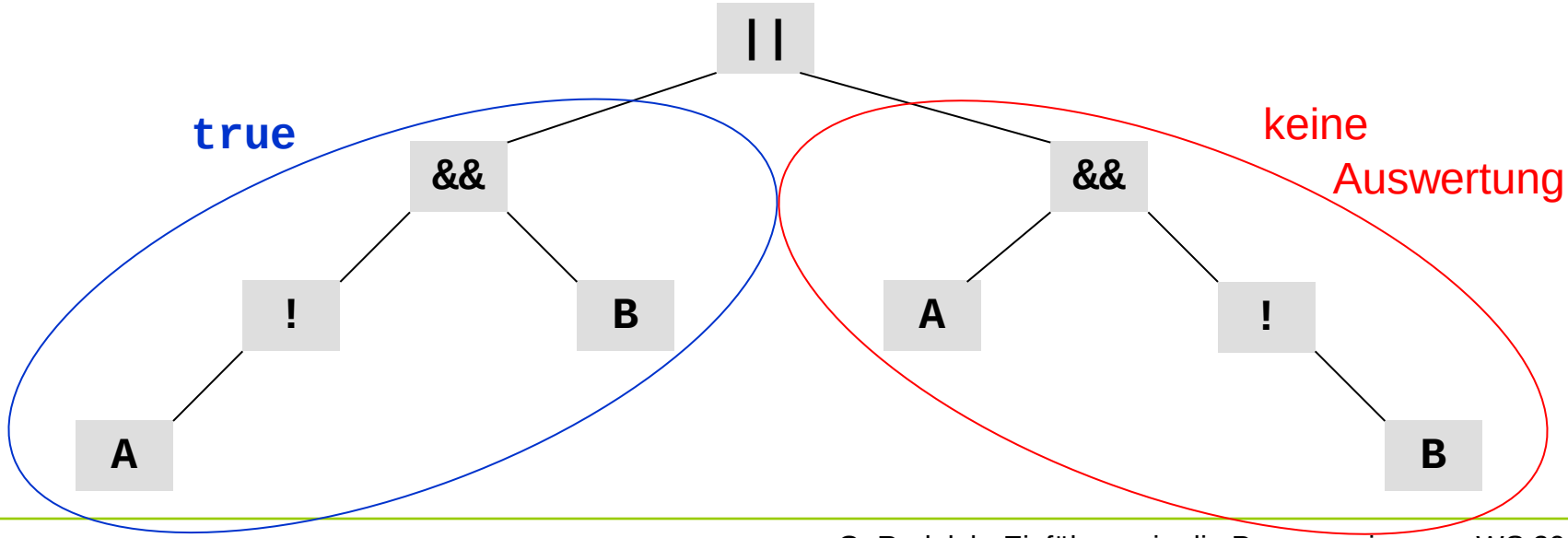
Weitere ableitbare Operationen

A NAND B	`!(A && B)`
A NOR B	`!(A || B)`
A $\Rightarrow$ B (Implikation)	`!A || B`
A XOR B (Antivalenz)	`!A && B || A && !B`



- Auswertung von links nach rechts
- Abbruch, sobald Ergebnis feststeht:
  - `A && false = false`
  - `A || true  = true`
- Beispiel:

```
bool A = false, B = true;
```



- Boolesche Ausdrücke
  - Vergleiche:

<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
==	gleich
!=	ungleich

Achtung:

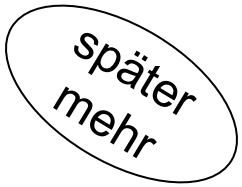
==	testet auf Gleichheit
=	wird bei einer Zuweisung verwendet

Wofür werden boolesche Ausdrücke gebraucht?

- … um Bedingungen formulieren zu können
- … um den Kontrollfluss steuern zu können
- … für Fallunterscheidungen: **`if`** *Bedingung wahr* **`then`** *mache etwas*;

```
#include <iostream>

int main()
{
  int a = 10, b = 20;
  if (a < b)  std::cout << “kleiner“;
  if (a > b)  std::cout << “groesser“;
  if (a == b) std::cout << “gleich“;
  return 0;
}
```



Im **Standard-Namensraum** wird **Standardfunktionalität** bereitgestellt:

- z.B. Ausgaben auf den Bildschirm, Eingaben von der Tastatur, …

```
#include <iostream>
using namespace std;
int main()
{
  int a = 10, b = 20;
  if (a < b)  std::cout << “kleiner“;
  if (a > b)  std::cout << “groesser“;
  if (a == b) std::cout << “gleich“;
  return 0;
}
```

falls Compiler einen Bezeichner nicht findet, dann Erweiterung mit **std**.

Beispiel:

Bezeichner → ???

std::Bezeichner ☺

⇒ führt zu kleineren Programmtexten

Anmerkung:

- In Programmiersprache C und vor 1993 auch in C++ existierte kein boolescher Datentyp!
- Stattdessen: Simulation mit Datentyp **`int`**
- Konvention: Wert ungleich Null bedeutet **`true`** sonst **`false`**
- Beispiele:
  - ```
    int x = 8;
    if ( x ) x = 0;
    ```
  - ```
    char c = 'y';
    if ( c ) c = '\n';
    ```
- Das ist auch jetzt noch möglich!

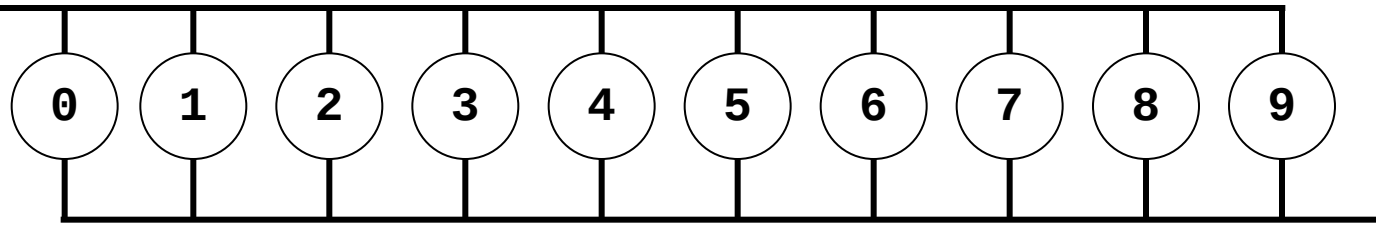
  $\Rightarrow$ Empfehlung: Besser den booleschen Datentyp verwenden!

Woher weiß man, was man in C++ schreiben darf und was nicht?

- Natürliche Sprache festgelegt durch
  - Alphabet
  - Orthografie
  - Wortbedeutungen
  - Grammatik
- Aktueller C++ Standard: ISO/IEC 14882:2002
- Es wurde u.a. eine formale Grammatik für C++ festgelegt (für alle verbindlich).

Grafische Darstellung

Ziffer :=

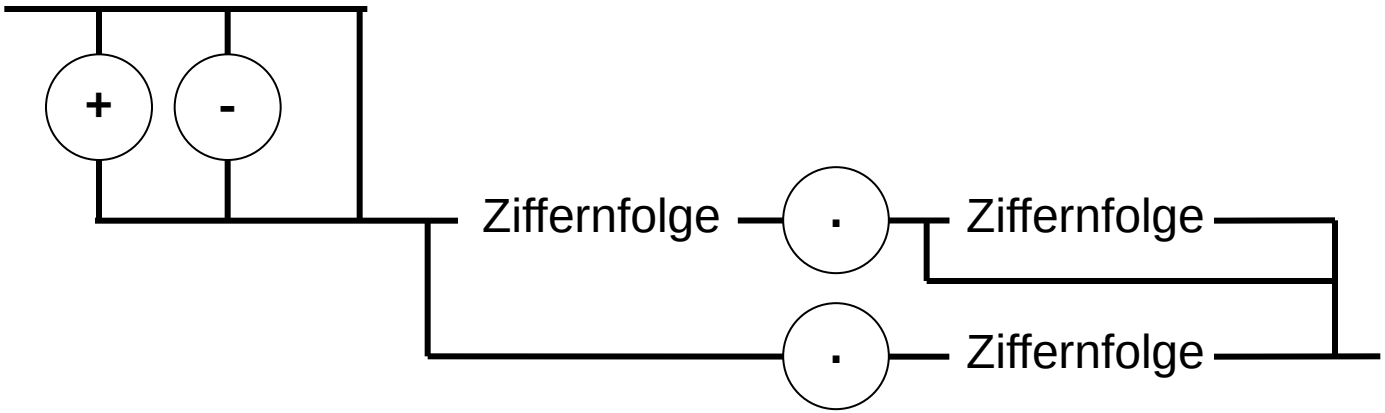


Ohne Pfeile: „von links nach rechts, von oben nach unten“

Ziffernfolge:=

Ganzzahl mit Vorzeichen :=

Festkommazahlen :=



Grafische vs. textuelle Darstellung von Grammatiken

- Grafische Darstellung anschaulich aber Platz raubend
- Textuelle Darstellung kompakter und automatisch zu verarbeiten

Ziel

- Beschreibung von syntaktisch korrekten C++ Programmen

Konkreter

- Sie sollen lernen, formale Grammatiken zu lesen und zu verstehen,
  - um sie in dieser Veranstaltung für ihre Zwecke nutzen zu können,
  - um einen fundamentalen Formalismus in der Informatik kennen zu lernen,
  - um andere Programmiersprachen leichter erlernen zu können.

Definition

Eine *kontextfreie Grammatik* G = (N, T, S, P) besteht aus

- einer endlichen Menge von Nichtterminalen N,
- einer endlichen Menge von Terminalen T,
- einem Startsymbol $S \in N$,
- einer endlichen Menge von Produktionsregeln der Form $u \rightarrow v$, wobei
  - $u \in N$ und
  - v eine endliche Sequenz von Elementen von N und T ist, sowie
- der Randbedingung $N \cap T = \varnothing$.

Beispiel

T = { +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

N = { Z, A, D }

S = { Z }

Z → +A

Z → -A

Z → A

A → D

A → AD

D → 0

D → 1

…

D → 9

= P

Kompaktere Notation:

Z → +A|-A|A

A → D|AD

D → 0|1|2|3|4|5|6|7|8|9

Beispiel

T = { +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

N = { Z, A, D }

S = { Z }

Z → +A|-A|A

A → D|AD

D → 0|1|2|3|4|5|6|7|8|9

- Nichtterminale sind Platzhalter.
- Man kann dort eine Produktionsregel anwenden.
- Der Ersetzungsprozess endet,
  wenn alle Nichtterminale durch Terminale ersetzt worden sind.

Beispiel

T = { +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

N = { Z, A, D }

S = { Z }

Z → +A|-A|A

A → D|AD

D → 0|1|2|3|4|5|6|7|8|9

Können wir mit dieser Grammatik +911 erzeugen?

Start mit Z → +A, wende Produktionsregel A → AD auf A an, ergibt Z → +AD

Wende A → AD auf A an, ergibt Z → +ADD

Wende A → D auf A an, ergibt Z → +DDD,

Wende D → 9 auf das erste D, D → 1 auf die übrigen D an, ergibt Z → +911.

Notation der Grammatik im C++ Buch von Bjarne Stroustrup

- **Nichtterminale:** Wörter in *kursiver* Schrift
- **Terminale:** Zeichen in **`nicht proportionaler`** Schrift
- Alternativen wie
  - **`D`** → **`0|1|2|3|4|5|6|7|8|9`** sind dargestellt via
  - *D:* eins von
    `0 1 2 3 4 5 6 7 8 9`
- Optionale (Nicht-)Terminale durch tiefgestelltes *opt*
  - $sign_{opt}$

Beispiel: Bezeichner

- *identifier:*
  nondigit
  identifier nondigit
  identifier digit
- *nondigit:* eins von
  universal-character-name
  _ 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
- *digit:* eins von
  0 1 2 3 4 5 6 7 8 9
- *universal-character-name:*
  \u *hex-quad*
  \U *hex-quad hex-quad*
- *hex-quad:*
  hex hex hex hex
- *hex:* eins von
  digit
  a b c d e f
  A B C D E F

Zusammengesetze Datentypen

- **Array (Feld)**
  - Einführendes Beispiel:
    Temperaturen von gestern stündlich speichern

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
8.4	8.3	8.0	7.4	7.2	7.0	7.0	7.5	8.0	8.8	9.8	11.1	13.4	13.6	13.7	13.6	12.4	12.0	10.1	9.6	9.0	8.9	8.7	8.5

  - Möglicher Ansatz:

```
float x00, x01, x02, x03, x04, x05, x06, x07,
      x08, x09, x10, x11, x12, x13, x14, x15,
      x16, x17, x18, x19, x20, x21, x22, x23;
```

  - Besser:

    Unter einem Namen zusammenfassen und
    zur Unterscheidung der Werte einen Index verwenden.

Array

Anzahl bereitzustellender Speicherplätze

- Datendefinition: `float x[24];`

Gemeinsamer Datentyp

Gemeinsamer Bezeichner

- Zugriff auf das Feldelement: `x[12];`

Achtung:

- Der Index beginnt **immer** bei 0!
- `x[12]` greift also auf das 13. Feldelement zu!
- Der maximale Index wäre hier also 23.
- Was passiert bei Verwendung von `x[24]` ? $\Rightarrow$ ABSTURZ!

Eindimensionales Array

- Ein **Array** ist eine Aneinanderreihung von **identischen** Datentypen
  - mit einer **vorgegebenen Anzahl** und
  - unter einem **gemeinsamen Bezeichner**.

- Der Zugriff auf einzelne Elemente erfolgt über einen **Index**
  - der **immer bei 0** beginnt und
  - dessen **maximaler Wert** genau **Anzahl – 1** ist.

- (Fast) alle Datentypen können verwendet werden.

Eindimensionales Array: Beispiele

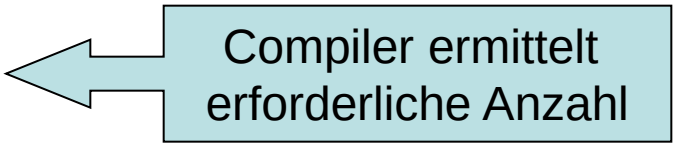
- `unsigned int Lotto[6];`
- `double Monatsmittel[12];`
- `char Vorname[20];`
- `bool Doppelgarage_belegt[2];`

- **Datendefinition**

  Datentyp Bezeichner[Anzahl];

Eindimensionales Array: Initialisierung

- `unsigned int Lotto[6] = { 27, 10, 20, 5, 14, 15 };`
- `unsigned int Lotto[]  = { 27, 10 };`



- `unsigned int Lotto[6] = { 27, 10 };`

  ist identisch zu

  `unsigned int Lotto[6] = { 27, 10, 0, 0, 0, 0 };`

- `unsigned int Lotto[6] = { 0 };`

  ist identisch zu

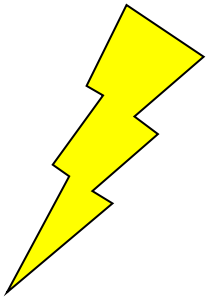
  `unsigned int Lotto[6] = { 0, 0, 0, 0, 0, 0 };`

Eindimensionales Array: Verwendung

```
float Temp[12] = { 2.3, 4.6, 8.9, 12.8 };
float x, y, z = 1.2;
Temp[4] = z;
x = Temp[0] * 0.25;
y = Temp[1] + 2.3 * Temp[2];
int i = 2, j = 3, k = 4, m = 11;
z = ( Temp[i] + Temp[j] + Temp[k] ) / 3.0;
Temp[m] = z + Temp[k - i];
```

Eindimensionales Array: Verwendung

```
float Temp[12] = { 2.3, 4.6, 8.9, 12.8 };
float TempNeu[12];
TempNeu = Temp;
```



Merken!

- Ein Array kann nicht als Ganzes einem anderen Array zugewiesen werden!
- Eine Zuweisung muss immer elementweise verfolgen!

Zwei- und mehrdimensionales Array

- **Einführendes Beispiel**
  - Pro Tag drei Temperaturmessungen: morgens, mittags, abends
  - Werte für eine Woche (7 Tage) ablegen

⇒

8.0	**20.3**	**14.2**
7.8	**18.3**	**12.2**
5.3	**12.3**	**8.8**
5.8	**13.7**	**7.5**
8.0	**19.8**	**10.2**
9.3	**21.3**	**11.1**
7.4	**17.3**	**9.9**

Tabelle

oder

Matrix

der Temperaturen

Zwei- und mehrdimensionales Array

- **Einführendes Beispiel**

`float tag0[3], tag1[3], tag2[3]` usw. bis `tag6[3];`

	0	1	2
tag0	8.0	20.3	14.2
tag1	7.8	18.3	12.2
tag2	5.3	12.3	8.8
tag3	5.8	13.7	7.5
tag4	8.0	19.8	10.2
tag5	9.3	21.3	11.1
tag6	7.4	17.3	9.9

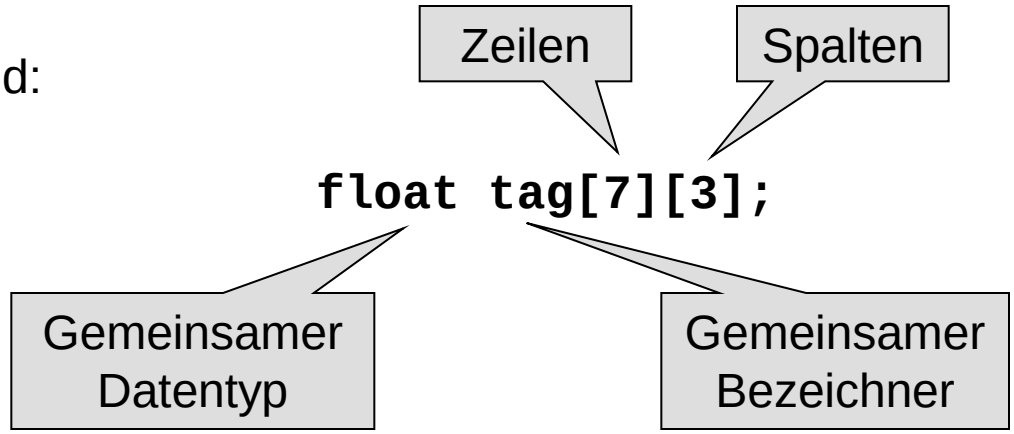
Zwei- und mehrdimensionales Array

- **Einführendes Beispiel**
  - Statt

    `float tag0[3], tag1[3], tag2[3]` usw. bis **`tag6[3];`**

    bräuchte man ein Array von Arrays vom Typ **`float`**!

  - Naheliegend:



Zwei- und mehrdimensionales Array

- **Einführendes Beispiel**

⇒ Spaltenindex

⇓ Zeilenindex

	0	1	2
0	8.0	20.3	14.2
1	7.8	18.3	12.2
2	5.3	12.3	8.8
3	5.8	13.7	7.5
4	8.0	19.8	10.2
5	9.3	21.3	11.1
6	7.4	17.3	9.9

`tag[0][2]` hat Wert `14.2`

`tag[2][0]` hat Wert `5.3`

`tag[4][2]` hat Wert `10.2`

`tag[2][4]` ist ungültig!

Zwei- und mehrdimensionales Array

- **Initialisierung**

```
float tag[7][3] = {
  { 8.0, 20.3, 14.2 },
  { 7.8, 18.3, 12.2 },
  { 5.3, 12.3,  8.8 },
  { 5.8, 13.7,  7.5 },
  { 8.0, 19.8, 10.2 },
  { 9.3, 21.3, 11.1 },
  { 7.4, 17.3,  9.9 }
};
```

oder

```
float tag[][3] = {
  { 8.0, 20.3, 14.2 },
  { 7.8, 18.3, 12.2 },
  { 5.3, 12.3,  8.8 },
  { 5.8, 13.7,  7.5 },
  { 8.0, 19.8, 10.2 },
  { 9.3, 21.3, 11.1 },
  { 7.4, 17.3,  9.9 }
};
```

Zwei- und mehrdimensionales Array

- **Datendefinition bei ansteigender Dimension**
  1. `int feld[n];`
  2. `int feld[m][n];`
  3. `int feld[k][m][n];`
  4. usw.

Zusammengesetze Datentypen

- **Zeichenkette**
  - … ist eine Aneinanderreihung von Zeichen
  - $\Rightarrow$ also ein Array/Feld von Zeichen

Datendefinition: `char wohnort[40];`

Initialisierung:

```
char wohnort[40] = {'D','o','r','t','m','u','n','d','\0'};
char wohnort[40] = "Dortmund";
char wohnort[]   = "Dortmund";
```

kennzeichnet Ende der Zeichenkette

riskant!

sicher: Compiler zählt!

- **Zeichenkette**
  - Das Ende wird durch das ASCII Steuerzeichen NUL (mit Code 0) gekennzeichnet!
  - $\Rightarrow$ Bei der Datendefinition muss also **immer ein Zeichen mehr** angefordert werden als zur Speicherung der Daten benötigt wird!

    Falsch ist: **`char wort[3] = “abc“;`**
  - Zuweisung einer Zeichenkette an eine andere nicht zulässig (weil **`array`** von **`char`**)

    Falsch ist: **`char wort[4];  wort[4] = “abc“;`**
    oder : **`wort[]  = “abc“;`**
  - Zuweisung muss immer **elementweise** erfolgen!

    Beispiel: **`char wort[4] = “abc“; wort[0] = ‘z‘;`**

Zusammengesetze Datentypen

- **Datenverbund (Struktur)**
  - Einführendes Beispiel:

    Zu speichern sei Namen und Matrikelnummer von Studierenden und ob Proseminar bestanden ist

  - Möglicher Ansatz:

    Drei verschiedene Datentypen (**`char[], unsigned int, bool`**)
    $\Rightarrow$ in Array lässt sich nur ein gemeinsamer Datentyp speichern
    $\Rightarrow$ alles als Zeichenketten, z.B. **`char stud[3][40];`**

  - Besser:

    Zusammen gehörende Daten unter einem Namen zusammenfassen aber die „natürlichen“ Datentypen verwenden!

Zusammengesetze Datentypen

- **Datenverbund (Struktur)**
  - Wir definieren uns unseren eigenen Datentyp!
  - Wir müssen die Struktur / den Bauplan definieren!
  - Wir müssen einen Namen für den Datentyp vergeben!

```
struct UnserDatenTyp
{
  char name[40];
  unsigned int matrikel;
  bool proseminar;
};
```

Name des Datentyps

Bauplan / Struktur

Zusammengesetze Datentypen

- **Datenverbund (Struktur)**
  - Zuerst das Schlüsselwort: **`struct`**
  - Dann folgt der gewählte Name (engl. *tag*).
  - In gescheiften Klammern **`{}`** steht der Bauplan. Am Ende ein Semikolon **`;`**

```
struct UnserDatenTyp
{
  char name[40];
  unsigned int matrikel;
  bool proseminar;
};
```

Name des Datentyps

Bauplan / Struktur

Datenverbund (Struktur)

- **Achtung:**
  Soeben wurde ein Datentyp definiert.
  Es wurde noch **kein Speicherplatz** reserviert!

- Datendefinition:
  ```
  UnserDatenTyp student, stud[50000];
  ```

- Initialisierung:
  ```
  UnserDatenTyp student = { “Hugo Hase“, 44221, true };
  ```

- Zugriff mit „Punktoperator“:
  ```
  unsigned int mnr = student.matrikel;
  cout << student.name << “ “ << mnr << endl;
  ```

Datenverbund (Struktur)

- Im Bauplan kann wieder jeder Datentyp vorkommen**!**
- Also auch wieder Datenverbunde (**`struct`**)!
- Beispiel:

```
struct UniStud {
  char ort[40];
  unsigned int plz;
  UnserDatentyp daten;
};

UniStud studX = {
  “Dortmund“, 44221, { “Jane Doe“, 241398, true }
};

unsigned int mnr = studX.daten.matrikel;
```

Datenverbund (Struktur)

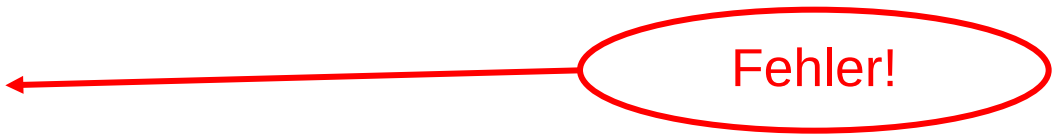
- Zuweisungen:

```
UnserDatenTyp stud[50000];
UnserDatenTyp student = { “Hugo Hase“, 44221, true };
stud[500] = student;
student = stud[501];
```

- Ganze Datensätze können strukturidentischen Variablen zugewiesen werden. Komponentenweises Zuweisen nicht nötig!

- **Achtung:**
  Anderer Name (tag) ⇒ Anderer Datentyp!
  Gilt selbst bei identischen Bauplänen!

```
struct S1 { int x; float y; };
struct S2 { int x; float y; };
S1 v1, vx;  v1 = vx;
S2 v2;      v2 = vx;
```



Zusammengesetze Datentypen

- **Aufzähltyp (enum)**
  - Umwelt beschreiben durch Begriffe statt durch Ziffern.
  - Farben: rot, blau, grün, orange, gelb, schwarz, …
  - Spielkarten: Kreuz, Pik, Herz, Karo.
  - Internet-Domains: de, uk, fr, ch, fi, ru, …

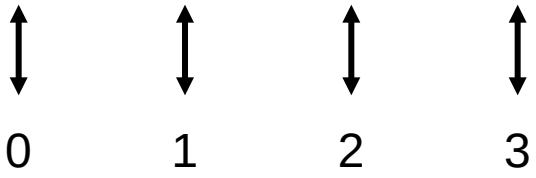
1. Schlüsselwort enum (Enumeration, Aufzählung)
2. Name der Aufzählung
3. In geschweiften Klammern die Elementnamen.

```
enum KartenTyp { kreuz, pik, herz, karo };
```

Zusammengesetze Datentypen

- **Aufzähltyp (enum)**
  - Was passiert im Rechner?
  - Interne Zuordnung von Zahlen (ein Code)

```
enum KartenTyp { kreuz, pik, herz, karo };
```



  - Zuordnung der Zahlen durch Programmierer kontrollierbar:

```
enum KartenTyp { kreuz=1, pik=2, herz=4, karo=8 };
```

  - Initialisierung: `KartenTyp Spielfarbe = kreuz;`
  - Aber: `cout << Spielfarbe << endl;`
    Ausgabe ist Zahl!