Einführung in die Programmierung für Physiker

Die Programmiersprache C – Datentypen, Operatoren, Ausdrücke

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

WS 2017/18

Terminologie

- **Konstanten**: Z.B. `3`, `114.0`, `-5.7`, ...; nicht veränderbar.
- **Variablen**: Veränderbar; müssen vor ihrer Verwendung **definiert** werden, z.B. `int a;`, `double xyz;`, ... (dabei wird ihr **Datentyp** [z.B. `int`, `double`, ...] festgelegt).
- **Operatoren**: Z.B. `+`, `-`, `*`, `/`, ...; verknüpfen Konstanten und/oder Variablen, produzieren dabei neue Werte.
- Der Datentyp einer Konstante/Variable legt ihren Wertebereich fest und auch, mit welchen Operatoren sie kombiniert werden kann.
- **Ausdrücke**: Kombinationen von Konstanten, Variablen und Operatoren, z.B. `a + 3`, `(xyz / a) * 7.0`, `1.0 - 2.3`, ...

Variablennamen

- Variablennamen bestehen aus Buchstaben (Groß- und Kleinschreibung wird unterschieden; "`_`" zählt als Buchstabe) und Ziffern.
- Variablennamen müssen mit einem Buchstaben beginnen.
- **Reservierte Worte** (z.B. `if`, `else`, `while`, ...) sind als Variablennamen verboten.
- Beispiele für Variablennamen: `a`, `b`, `i1`, `i2`, `i3`, `z34UBr3__v`, `h_bar`, `x_square`, ...
- Insbesondere bei umfangreichen Programmcodes empfiehlt sich eine im Wesentlichen selbsterklärende Namensgebung (z.B. `h_bar`, `x_square`, ...).

Datentypen

Integer-Datentypen

- Standard Integer-Datentyp ist **`int`**:
  - Integer-Zahlen werden im Computer in Form von **Bits** ("0" oder "1") gespeichert; 1 **Byte** besteht aus 8 Bits.
  - Die Anzahl der für einen Integer-Wert verwendeten Bits ist maschinenabhängig; mindestens 16, häufig 32 Bits (4 Bytes); bei z.B. 32 Bits sind 2^{32} = 4294967296 verschiedene ganze Zahlen darstellbar, von -2^{-31} bis 2^{31}-1.
  - In den meisten Fällen ist die Anzahl der verwendeten Bits und damit der Wertebereich ohne Bedeutung; falls doch von Bedeutung, kann der Wertebereich mit Hilfe von **`limits.h`** festgestellt werden.
    - **`limits.h`**:

```
79. ...
80. /* Minimum and maximum values a `signed int' can hold.  */
81. #  define INT_MIN       (-INT_MAX - 1)
82. #  define INT_MAX       2147483647
83.
84. /* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
85. #  define UINT_MAX      4294967295U
86. ...
```

    - **Beispiel:** Abfrage des Wertebereichs von **`int`** ...

```
 1. #include<limits.h>
 2. #include<stdio.h>
 3. #include<stdlib.h>
 4.
 5. int main(void)
 6. {
 7.   printf("Wertebereich von int: %d ... %d\n", INT_MIN, INT_MAX);
 8.
 9.   if(INT_MAX < 3000000000)
10.     {
11.       printf("Fehler: Kaufen Sie sich einen besseren Computer!\n");
12.       exit(0);
13.     }
14.
15.   // Ab hier dann das eigentliche Programm ...
16. }
```

```
Wertebereich von int: -2147483648 ... 2147483647
Fehler: Kaufen Sie sich einen besseren Computer!
```

- Analog aber weniger häufig verwendet:
  - **`short`** (mindestens 16 Bits, darf nicht mehr Bits besitzen als **`int`**).
  - **`long`** (mindestens 32 Bits, darf nicht weniger Bits besitzen als **`int`**).

```
 1. #include<limits.h>
 2. #include<stdio.h>
 3.
 4. int main(void)
 5. {
 6.   printf("Wertebereich von short: %d ... %d\n", SHRT_MIN, SHRT_MAX);
 7.
 8.   // Verwendung von short analog zu int.
 9.   short a;
10.   a = 17;
11.   printf("a = %d\n", a);
```

```
12. }
```

```
Wertebereich von short: -32768 ... 32767
a = 17
```

- Für Zeichen verwendet man in der Regel den Datentyp **`char`**:
  - 8 Bits (1 Byte), damit 256 ganze Zahlen bzw. 256 unterschiedliche Zeichen darstellbar.
  - **ASCII** (American Standard Code for Information Interchange), z.B. **`http://www.asciitable.com/`**.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   printf("%d   %c\n", 65, 65);
6.   printf("%d   %c\n", 66, 66);
7.
8.   printf("%d   %c\n", 'C', 'C');
9.
10.  char c1 = 'D';
11.  printf("%d   %c\n", c1, c1);
12. }
```

```
65   A
66   B
67   C
68   D
```

- Die Integer-Datentypen existieren auch vorzeichenlos, z.B. **`unsigned int`**, **`unsigned short`**, ...:
  - Der Wertebereich von z.B. **`unsigned int`** bei 32 Bits ist 0 ... 2^{32}-1 = 4294967295.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   unsigned int ui;
6.
7.   ui = 27;
8.   printf("ui = %u\n", ui);
9.
10.  ui = -1;
11.  printf("ui = %u\n", ui);
12. }
```

```
ui = 27
ui = 4294967295
```

Gleitkomma-Datentypen

- Gleitkomma-Datentypen sind **`float`**, **`double`** (präziser als **`float`**, am häufigsten verwendet) und **`long double`** (präziser als **`double`**):

- Gleitkomma-Zahlen werden im Computer ebenfalls in Form von Bits gespeichert:
  - Vorzeichen S (1 Bit, z.B. $0 = +$, $1 = -$).
  - Mantisse $(m_1, m_2, \ldots)$ (mehrere Bits; bei **double** häufig 52 Bits; **double**-Werte sind damit auf etwa 16 Dezimalstellen genau).
  - Exponent E (ganze Zahl, mehrere Bits; bei **double** häufig 11 Bits; damit können **double**-Werte der Größenordnung 10^{-308} ... 10^{308} dargestellt werden).
  - Umrechnung dieser Bits in eine reelle Zahl (im Prinzip) gemäß $S \times M \times 2^E$, wobei $M = 1 + \sum_{n=1}^{\ldots} m_n (1/2)^n$ (für Details siehe Informatik- oder Numerik-Bücher).
  - **Die meisten reellen Zahlen lassen sich nicht exakt durch Gleitkomma-Zahlen darstellen; es wird die nächstliegende Gleitkomma-Zahl verwendet, was zu Rundungsfehlern führt.**
- Der Wertebereich und die Genauigkeit (maschinenabhängig) kann mit Hilfe von **float.h** festgestellt werden.

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

int main(void)
{
  printf("Genauigkeit in Dezimalziffern (float) = %d\n", FLT_DIG);
  printf("Kleinster Wert x, fuer den 1.0 + x ungleich 1.0 gilt (float) = %e\n", FLT_EPSILON);
  printf("Maximaler Wert (float) = %e\n", FLT_MAX);

  printf("Genauigkeit in Dezimalziffern (double) = %d\n", DBL_DIG);
  printf("Kleinster Wert x, fuer den 1.0 + x ungleich 1.0 gilt (double) = %e\n", DBL_EPSILON);
  printf("Maximaler Wert (double) = %e\n", DBL_MAX);
}
```

```
Genauigkeit in Dezimalziffern (float) = 6
Kleinster Wert x, fuer den 1.0 + x ungleich 1.0 gilt (float) = 1.192093e-07
Maximaler Wert (float) = 3.402823e+38
Genauigkeit in Dezimalziffern (double) = 15
Kleinster Wert x, fuer den 1.0 + x ungleich 1.0 gilt (double) = 2.220446e-16
Maximaler Wert (double) = 1.797693e+308
```

Konstanten

Numerische Konstanten

- Numerische Konstanten werden wie im folgenden Beispiel geschrieben.

```c
#include<stdio.h>

int main(void)
{
  int i1 = -115;
  unsigned int ui1 = 56u;
  unsigned int ui2 = 56U;

  long l1 = -115l;
  long l2 = -115L;
  unsigned long ul1 = 56ul;
  unsigned long ul2 = 56UL;

  // **********

  char c1 = 65;  // Wert von "A" im ASCII-Zeichensatz.
  char c2 = 'A';

  // **********

  double d1 = 123.0;
  double d2 = 1.23e+2;
  double d3 = 12300.0e-2;

  float f1 = 123.0f;
  float f2 = 123.0F;

  long double ld1 = 123.0l;
  long double ld2 = 123.0L;
}
```

- Wichtig ist vor allem der Dezimalpunkt bzw. "e" bei Gleitkommakonstanten.
- **Vorsicht! Häufige Fehlerquelle ...**
  Dezimalpunkt vergessen ... z.B. **`7/2`** (ergibt **`3`**, da im Raum der ganzen Zahlen gerechnet wird) an Stelle von **`7.0/2.0`** (ergibt **`3.5`**).

```c
#include<stdio.h>

int main(void)
{
  double d1;

  d1 = 7/2;
  printf("d1 = %f\n", d1);

  d1 = 7.0/2.0;
  printf("d1 = %f\n", d1);
}
```

```
d1 = 3.000000
d1 = 3.500000
```

Symbolische Konstanten

- Mit **`#define`** *`name constant`* lassen sich aussagekräftige Namen für Konstanten definieren, sogenannte **symbolische Konstanten**.
- Bei *name* handelt es sich nicht um eine Variable; man kann *name* z.B. keinen neuen Wert

zuweisen.

- Symbolische Konstanten verbessern die Lesbarkeit von Programmcode.
- Symbolische Konstanten erlauben eine komfortable und wenig fehleranfällige Veränderung des numerischen Werts einer Konstante (auch wenn die Konstante mehrmals verwendet wird, muss der Programmcode nur an einer Stelle verändert werden).

```c
#include<stdio.h>

// Ortsfaktor in N/kg (auf der Erde).
#define ORTSFAKTOR_G 9.81

int main(void)
{
  double m = 2.0;  // Raketenmasse in kg.
  printf("Auf die Rakete wirkende Schwerkraft: %f N\n", m * ORTSFAKTOR_G);
}
```

```
Auf die Rakete wirkende Schwerkraft: 19.620000 N
```

```c
...
// Ortsfaktor in N/kg (auf dem Mond).
#define ORTSFAKTOR_G 1.57
...
```

```
Auf die Rakete wirkende Schwerkraft: 3.140000 N
```

Aufzählungskonstanten

- Sollen mehrere Integer-Konstanten, z.B. **0**, **1**, **2**, ..., mit Namen verknüpft werden, bieten sich **Aufzählungskonstanten** an.

```c
#include<stdio.h>

int main(void)
{
  // Die Aufzaehlungskonstanten MO, DI, ..., SO werden definiert; sie sind vom
  // Datentyp "enum wochentag" = int.
  enum wochentag { MO, DI, MI, DO, FR, SA, SO };

  printf("MO = %d, DI = %d, MI = %d, ...\n", MO, DI, MI);

  // **********

  enum wochentag wt = DO;

  printf("Uni-Termine heute:\n");

  if(wt == DO)
    {
      printf("08:15-09:45   PPROG-Uebung bei Alessandro.\n");
      printf("14:15-15:45   PPROG-Vorlesung.\n");
    }
  else
    {
      printf("Keine wichtigen Veranstaltungen.\n");
    }
}
```

```
MO = 0, DI = 1, MI = 2, ...
Uni-Termine heute:
08:15-09:45   PPROG-Uebung bei Alessandro.
14:15-15:45   PPROG-Vorlesung.
```

- Der "neu eingeführte Datentyp" **enum wochentag** ist äquivalent zu **int**; der folgende Programmcode ist damit äquivalent zum obigen.

```c
#include<stdio.h>

```

```c
int main(void)
{
  // Die Aufzaehlungskonstanten MO, DI, ..., SO werden definiert.
  enum { MO, DI, MI, DO, FR, SA, SO };

  printf("MO = %d, DI = %d, MI = %d, ...\n", MO, DI, MI);

  // **********

  int wt = DO;

  printf("Uni-Termine heute:\n");

  if(wt == DO)
    {
      printf("08:15-09:45   PPROG-Uebung bei Alessandro.\n");
      printf("14:15-15:45   PPROG-Vorlesung.\n");
    }
  else
    {
      printf("Keine wichtigen Veranstaltungen.\n");
    }
}
```

Definition von Variablen

- Variablen müssen vor ihrer Verwendung definiert werden; dabei wird der Name und der Datentyp der Variable festgelegt.
- Variablen können direkt bei ihrer Definition mit einem Wert **initialisiert** werden.

```
...
  int a;  // Definition einer int-Variable.
  int b, c, d;  // Definition mehrerer int-Variablen.
  int e = 27;  // Definition und Initialisierung einer int-Variable.
...
```

- **Vorsicht! Häufige Fehlerquelle ...**
  Initialisierung einer Variable vergessen (im folgenden Beispiel **`m_neutron`**) ... liefert falsche, "zufällige" Ergebnisse.

```
#include<stdio.h>

int main(void)
{
  double m_proton, m_neutron, m_electron;

  m_proton = 938.272;  // Masse des Protons in MeV/c^2.
  m_electron = 0.510999;  // Masse des Elektrons in MeV/c^2.

  double m_Helium_constituents = 2.0*m_proton + 2.0*m_neutron + 2.0*m_electron;

  printf("Konstituentenmasse des Helium-Atoms: %f MeV/c^2.\n", m_Helium_constituents);
}
```

```
Konstituentenmasse des Helium-Atoms: 1877.565998 MeV/c^2.
```

- Mit **`const`** können unveränderbare Variablen (also Konstanten) definiert werden (ähnlich, aber nicht identisch zu **`#define`**).

```
#include<stdio.h>

int main(void)
{
  const double pi = 3.1415;
  printf("pi = %f\n", pi);
}
```

```
pi = 3.141500
```

```
#include<stdio.h>

int main(void)
{
  const double pi = 3.1415;
  printf("pi = %f\n", pi);

  pi = 3.141592;
}
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o pi_const pi_const.c
pi_const.c: In Funktion "int main()":
pi_const.c:8:8: Fehler: Zuweisung der schreibgeschützten Variable "pi"
```

Operatoren

Unäre arithmetische Operatoren

Operator/Ausdruck	Beschreibung
+a	plus **a**
-a	minus **a**

Binäre arithmetische Operatoren

Operator/Ausdruck	Beschreibung
a + b	**a** plus **b**
a - b	**a** minus **b**
a * b	**a** mal **b**
a / b	**a** geteilt durch **b**
a % b	**a** modulo **b** (Anwendung nur auf positive Integer-Ausdrücke)

```c
#include<stdio.h>

int main(void)
{
  printf("23 / 5 = %d, Rest %d\n", 23 / 5, 23 % 5);
}
```

```
23 / 5 = 4, Rest 3
```

Vergleichsoperatoren

Operator/Ausdruck	Beschreibung
a == b	Ist **a** gleich **b**?
a != b	Ist **a** ungleich **b**?
a < b	Ist **a** kleiner **b**?
a > b	Ist **a** größer **b**?
a <= b	Ist **a** kleiner oder gleich **b**?
a >= b	Ist **a** größer oder gleich **b**?

- Vergleiche liefern "1" bei Antwort "ja" ("true"), "0" bei Antwort "nein" ("false"), sogenannte **logische Ausdrücke**.

```c
#include<stdio.h>

int main(void)
{
  printf("7 > 2   -->   %d\n", 7 > 2);
  printf("3 == 4   -->   %d\n", 3 == 4);
}
```

```
7 > 2   -->   1
3 == 4   -->   0
```

Logische Operatoren

- Anwendung nur auf logische Ausdrücke.

Operator/Ausdruck	Beschreibung

a && b	logisches "und"; "1", wenn **a** ungleich "0" (also "true") und **b** ungleich "0" (also "true"); "0" sonst; **b** wird nur ausgewertet, falls **a** ungleich "0" (also "true") ist
a || b	logisches "oder"; "1", wenn **a** ungleich "0" (also "true") oder **b** ungleich "0" (also "true"); "0" sonst; **b** wird nur ausgewertet, falls **a** gleich "0" (also "false") ist
!a	logische "Negation"; "1", wenn **a** gleich "0" (also "false"); "0" sonst

```c
#include<stdio.h>

int main(void)
{
  printf("(7 > 2) && (3 == 4)   -->   %d\n", (7 > 2) && (3 == 4));
  printf("(7 > 2) || (3 == 4)   -->   %d\n", (7 > 2) || (3 == 4));
  printf("!0 --> %d   ,   !1 --> %d\n", !0, !1);
}
```

```
(7 > 2) && (3 == 4)   -->   0
(7 > 2) || (3 == 4)   -->   1
!0 --> 1   ,   !1 --> 0
```

- Zur Erinnerung: Logische Ausdrücke (Vergleiche und Kombinationen von logischen Operatoren und Vergleichen) werden in Kontrollstrukturen verwendet, z.B.
  - **if(***expr***){...}else{...}**,
  - **while(***expr***){...}**,
  - **for(...; *expr*; ...){...}**.

```c
...
  if((x > 3) && (x < 7))
  {
    ...
  }
  else
  {
    ...
  }
...
```

```c
...
  while( !((x == 2) || (y != z)) )
  {
    ...
  }
...
```

```c
...
  for(i = 0; i < 10; i = i+1)
  {
    ...
  }
...
```

Operatoren zur Bit-Manipulation

- Anwendung nur auf Integer-Ausdrücke.

Operator/Ausdruck	Beschreibung
a & b	"und"-Verknüpfung von Bits
a | b	"oder"-Verknüpfung von Bits

`a ^ b`	"exclusive oder"-Verknüpfung von Bits
`~a`	Bit-Komplement
`a << b`	Bit-Verschiebung nach links
`a >> b`	Bit-Verschiebung nach rechts

```c
#include<stdio.h>

int main(void)
{
  int a =  6;  // Entspricht der Bitfolge ...000110.
  int b = 20;  // Entspricht der Bitfolge ...010100.

  printf("a & b = %d\n", a & b);
  printf("a | b = %d\n", a | b);

  printf("a >> 1 = %d\n", a >> 1);
  printf("a >> 2 = %d\n", a >> 2);
  printf("a >> 3 = %d\n", a >> 3);
}
```

```
a & b = 4
a | b = 22
a >> 1 = 3
a >> 2 = 1
a >> 3 = 0
```

Zuweisungsoperatoren

Operator/Ausdruck	Beschreibung
`a = b`	der Variable **a** wird der Wert von **b** zugewiesen; der Wert des Ausdrucks ist der zugewiesene Wert
`a` *op*`= b`	*op* ∈ {+,-, *, /, %, &, |, ^, <<, >>}; äquivalent zu `a = (a)` *op* `(b)`

```c
#include<stdio.h>

int main(void)
{
  double x;

  printf("(x = 5.0) = %f\n", x = 5.0);

  x += 2.5;
  printf("x = %f\n", x);

  printf("(x /= 2.5) = %f\n", x /= 2.5);
  printf("x = %f\n", x);
}
```

```
(x = 5.0) = 5.000000
x = 7.500000
(x /= 2.5) = 3.000000
x = 3.000000
```

Inkrement- und Dekrement-Operatoren

- Anwendung nur auf einzelne Variablen.

Operator/Ausdruck	Beschreibung
`a++`	der Wert dieses Ausdrucks ist der ursprüngliche Wert von **a**; **a** wird um 1 erhöht
`++a`	**a** wird um 1 erhöht; der Wert dieses Ausdrucks ist der neue Wert von **a** (äquivalent zu `a = a+1`)
`a--`	der Wert dieses Ausdrucks ist der ursprüngliche Wert von **a**; **a** wird um 1 erniedrigt

`--a`	**a** wird um 1 erniedrigt; der Wert dieses Ausdrucks ist der neue Wert von **a** (äquivalent zu **a = a-1**)

```c
#include<stdio.h>

int main(void)
{
  int a;

  a = 7;
  printf("a++ = %d\n", a++);
  printf("a = %d\n", a);

  a = 7;
  printf("++a = %d\n", ++a);
  printf("a = %d\n", a);

  a = 7;
  printf("(a = a+1) = %d\n", a = a+1);
  printf("a = %d\n", a);
}
```

```
a++ = 7
a = 8
++a = 8
a = 8
(a = a+1) = 8
a = 8
```

Bedingter Ausdruck

Operator/Ausdruck	Beschreibung
`a ? b : c`	falls der logische Ausdruck **a** "true" ist (also ein Integer-Wert ungleich 0), wird **b** berechnet und dessen Wert ist der Wert des bedingten Ausdrucks, sonst wird **c** berechnet und dessen Wert ist der Wert des bedingten Ausdrucks; sind **b** und **c** von verschiedenem Datentyp, finden die unten diskutierten Regeln zur Umwandlung von Datentypen Anwendung

```c
#include<stdio.h>

int main(void)
{
  double x, y;

  // (x > y) ? x : y   entspricht   max(x,y).

  x = 4.0;
  y = 5.5;
  printf("(x > y) ? x : y   =   %f\n", (x > y) ? x : y);

  x = 3.2;
  y = 1.4;
  printf("(x > y) ? x : y   =   %f\n", (x > y) ? x : y);
}
```

```
(x > y) ? x : y   =   5.500000
(x > y) ? x : y   =   3.200000
```

Umwandlung von Datentypen

- Sind die Operanden eines Operators wie z.B. **+** oder **=** von unterschiedlichem Datentyp, findet eine automatische **Typumwandlung** statt.
- In der Regel findet eine Umwandlung in den Datentyp statt, der den größeren Wertebereich besitzt; z.B. bei Addition eines **int**-Werts und eines **double**-Werts ist das Ergebnis vom Datentyp **double**.
- Bei Zuweisungsoperatoren wie z.B. **=** oder **+=** wird der Wert auf der rechten Seite in den Datentyp der Variable auf der linken Seite umgewandelt.
- Wird ein Gleitkomma-Wert in einen Integer-Wert umgewandelt (z.B. bei einer Zuweisung), werden die Ziffern nach dem Dezimalpunkt ignoriert.

```c
#include<stdio.h>

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

  i = 3;
  x = 7.5;
  printf("i + x = %f\n", i + x);  // Umwandlung von int in double: 3 --> 3.0.

  i = 4.25;
  printf("i = %d\n", i);  // Umwandlung von double in int: 4.25 --> 4.
  i = 4.75;
  printf("i = %d\n", i);  // Umwandlung von double in int: 4.75 --> 4.
  i = -4.25;
  printf("i = %d\n", i);  // Umwandlung von double in int: -4.25 --> 4.
  i = -4.75;
  printf("i = %d\n", i);  // Umwandlung von double in int: -4.75 --> 4.
}
```

```
i + x = 10.500000
i = 4
i = 4
i = -4
i = -4
```

- **Liegt der umzuwandelnde Wert nicht im Wertebereich des Datentyps, in den umgewandelt wird, ist das Ergebnis undefiniert, d.h. sinnlos.**

```c
#include<stdio.h>

int main(void)
{
  char c;  // Wertebereich von -128 bis +127.

  c = 355;
  printf("c = %d\n", c);

  c = 355.0;
  printf("c = %d\n", c);

  double x = 355.0;
  c = x;
  printf("c = %d\n", c);
}
```

```
c = 99
c = 127
c = 99
```

- Eine Typumwandlung kann auch durch einen **Cast-Operator** **(***type***)** erzwungen werden: **(***type***)***expr* wandelt den Wert von *expr* in den Datentyp *type* um.

```c
#include<stdio.h>

int main(void)
{
  printf("(double)(7/2) = %f\n", (double)(7/2));
  printf("((double)7)/((double)2) = %f\n", ((double)7)/((double)2));
}
```

```
(double)(7/2) = 3.000000
((double)7)/((double)2) = 3.500000
```

- **Beispiel:** Ausgabe einer Trajektorie, z.B. als Input-Datei für **gnuplot** ...

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

int main(void)
{
  const double g = 9.81;  // Ortsfaktor in m/s^2.

  // Die Trajektorie wird im Zeitfenster t_min ... t_max ausgegeben (num_samples Abtastwerte).
  const double t_min = 3.0;  // In s.
  const double t_max = 6.0;  // In s.
  const int num_samples = 7;

  for(int i = 0; i < num_samples; i++)
    {
      double t = t_min   +   (((double)i) / ((double)(num_samples - 1))) * (t_max - t_min);
      double z = -0.5 * g * pow(t, 2.0);  // Senkrechter freier Fall.
      printf("%.3f %+8.3f\n", t, z);
    }
}
```

```
3.000  -44.145
3.500  -60.086
4.000  -78.480
4.500  -99.326
5.000 -122.625
5.500 -148.376
6.000 -176.580
```

- **Vorsicht! Häufige Fehlerquelle ...**
  Cast-Operator bei der Division von Integer-Werten vergessen ... z.B. **1/6** ergibt 0, da im Raum der ganzen Zahlen gerechnet wird.

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

int main(void)
{
  const double g = 9.81;  // Ortsfaktor in m/s^2.

  // Die Trajektorie wird im Zeitfenster t_min ... t_max ausgegeben (num_samples Abtastwerte).
  const double t_min = 3.0;  // In s.
  const double t_max = 6.0;  // In s.
  const int num_samples = 7;

  for(int i = 0; i < num_samples; i++)
    {
      double t = t_min   +   (i / (num_samples - 1)) * (t_max - t_min);
      double z = -0.5 * g *pow(t, 2.0);  // Senkrechter freier Fall.
      printf("%.3f %+8.3f\n", t, z);
    }
}
```

```
3.000  -44.145
3.000  -44.145
3.000  -44.145
3.000  -44.145
3.000  -44.145
3.000  -44.145
6.000 -176.580
```

- Eine vollständige Darstellung der Regeln der Typumwandlung findet sich z.B. in **Programmieren in C**, 2. Auflage (B. W. Kernighan, D. M. Ritchie, Hanser), Anhang A.6.
- **Programmcode sollte so geschrieben werden, dass er einfach lesbar ist die Ergebnisse nicht von den Details und Feinheiten dieser Regeln abhängen (z.B. durch Verwendung von Cast-Operatoren).**

Vorrang und Assoziativität von Operatoren

Operator	Assoziativität
`()`, `[]`, `.`, `->`	links
`+` (unär), `-` (unär), `!`, `~`, `++`, `--`, `(type)`, `&` (unär), `*` (unär), `sizeof()`	rechts
`*` (binär), `/`, `%`	links
`+` (binär), `-` (binär)	links
`<<`, `>>`	links
`<`, `>`, `<=`, `>=`	links
`==`, `!=`	links
`&` (binär)	links
`^`	links
`|`	links
`&&`	links
`||`	links
`?:`	rechts
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`	rechts
`,`	links

- **Vorrang:**
  - Operatoren in der gleichen Tabellenzeile besitzen den gleichen Vorrang.
  - Die Zeilen sind gemäß dem Vorrang der enthaltenen Operatoren geordnet (erste Zeile höchster Vorrang, letzte Zeile niedrigster Vorrang).
  - Die Auswertung von Ausdrücken findet entsprechend dem Vorrang der enthaltenen Operatoren statt.
  - Beispiel: `a + b * c` ist äquivalent zu `a + (b * c)`.
- **Assoziativität:**
  - Linksassoziative binäre Operatoren mit gleichem Vorrang `×` und `∗`:
    `a × b ∗ c`
    ist äquivalent zu
    `(a × b) ∗ c`.
  - Rechtsassoziative binäre Operatoren mit gleichem Vorrang `×` und `∗`:
    `a × b ∗ c`
    ist äquivalent zu
    `a × (b ∗ c)`.
  - Rechtsassoziative unäre Operatoren mit gleichem Vorrang `×` und `∗`:
    `×∗a`
    ist äquivalent zu
    `×(∗a)`.
  - Beispiel: `a - b - c` ist äquivalent zu `(a - b) - c`.
- **Die Reihenfolge, in der die Operanden eines Operators ausgewertet werden, ist nicht festgelegt, d.h. compiler-abhängig (Ausnahmen bilden `&&`, `||` und `?:`).**

```c
#include<stdio.h>

int main(void)
```

```
4. {
5.   int i;
6.
7.   i = 3;
8.   printf("%d   (hier steht 7 oder 8, je nach verwendetem Compiler)\n", (i=i+1) + i);
9.
10.  i = 3;
11.  printf("%d   (hier steht 6 oder 7, je nach verwendetem Compiler)\n", (i++) + i);
12. }
```

```
8   (hier steht 7 oder 8, je nach verwendetem Compiler)
6   (hier steht 6 oder 7, je nach verwendetem Compiler)
```

- **Programmcode sollte so geschrieben werden, dass er einfach lesbar ist und die Ergebnisse nicht von den Details und Feinheiten dieser Regeln abhängen (z.B. durch die Verwendung von Klammern).**