Ausdrücke

- Ausdrücke sind programmiersprachliche Konstrukte zur Bildung von neuen Werten aus vorhandenen Werten
- Ausdrücke bestehen aus Operanden und Operatoren einschließlich Klammerungen.
- Ausdrücke bestehen mindestens aus einem Operanden.
- Jeder Ausdruck repräsentiert einen **Wert** von einem bestimmten **Typ**.
- Wert und Typ gehören immer zusammen.

Operanden (Direktwerte)

- Operanden können Direktwerte sein, ganze oder gebrochene Zahlen (Dezimalbrüche) oder einzelne Zeichen in Apostroph.
- Ganze Zahlen können dezimal, octal oder hexadezimal oder als Zeichen (dann ist der ASCII-Code der Wert) angegeben werden.

Ein kleines l , wie long hinter der Zahl, kennzeichnet einen Long-wert.

12	dezimal	int
0344	octal	int
67543l	dezimal	long
0x2aab	hexadezimal	int
12.33	dezimal	double
23.43d	dezimal	double
12.45f	dezimal	float
‘a‘	Zeichen	char/int
‘4‘	Zeichen	char/int

Konstante Ausdrücke

- Ausdrücke, die nur konstante Werte (Direktwerte) als Operanden enthalten, bezeichnet man als konstante Ausdrücke (const expressions).
- Konstante Ausdrücke werden vom Compiler währen des Compilierens ausgerechnet und das Ergebnis in den Code eingetragen.
- Für konstante Ausdrücke wird kein ausführbarer Code generiert.

Const expressions

```
int tageBis[]={
  0,
  0,
  31,
  31+28,
  31+28+31,
  31+28+31+30,
  31+28+31+30+31,
  31+28+31+30+31+30,
. . .
};
```

```
TageBis[7]   //Juli
```

Liefert den Wert von
31+28+31+30+31+30
= 181

Operanden (Variable)

- Operanden können auch Werte sein, die in Variablen gespeichert sind.
- Variablen können Werte aufnehmen, wenn sie links von einem Zuweisungszeichen = stehen (L-Value).
- Es gibt noch weitere Möglichkeiten für Operanden, die wir später kennenlernen.

Variablendefinition	Wert
`int i;`	undefiniert /zufällig
`int j=0;`	0
`float f=3.14;`	3,14
`double Pi=M_PI;`	3.14159….
`long x=1234l;`	1234

Operanden (Arrayelemente)

- Operanden können auch Werte sein, die in Arrays gespeichert sind.
- Sie werden durch einen Index selektiert.
- Der Index beginnt beim ersten Element mit dem Wert 0.
- In c wird die Gültigkeit eines Index nicht geprüft (Absturzgefahr!)!
- Der Index kann ein Direktwert sein, in einer Integervariablen stehen oder berechnet werden.

	Wert
`int arr[]={1,2,4,8};`	
`arr[0]`	1
`arr[1]`	2

`int i=2;`
`arr[i]`
`arr[i+1]`
	
	4
	8

Operanden (weitere)

- Weitere Operanden (werden teilweise später behandelt) können sein:
  - Funktionsaufrufe
  - Selektierte Strukturkomponenten
  - Pointer/dereferenzierte Pointer
  - In runden Klammern wiederum Ausdrücke

`atoi(buf)`	Funktions-aufruf
`Student.matrNr`	Selektierte Strukturkomponente
`argv[0]` `*argv[0]`	Pointer Dereferenzierter Pointer
`(i+1)`	Klammerausdruck

Operatoren

- Die Operatoren bestimmen die auszuführende Operation (z.Bsp.: +: Addition)
- Operatoren haben eine Reihe von Eigenschaften:
  - Assoziativität (links-/rechtsassoziativ)
  - Priorität(z. Bsp.: Multiplikative Operanden haben eine höhere Priorität als additive)
  - Anzahl der Operanden (unär/binär)

1	( )	[ ]	->	.			li nach re	Höchste Priorität
2	!	~	-	(…) cast				
3	++	--	&	*	sizeof		re nach li	
4	*	/	%				li nach re	
5	+	-					li nach re	
6	<<	>>					li nach re	
7	<	<=	>	>=			li nach re	
8	==	!=					li nach re	
9	&						li nach re	
10	^						li nach re	
11	|						li nach re	
12	&&						li nach re	
13	||						li nach re	
14	? :						li nach re	
15	=	*=	/=	%=	-=	&=	re nach li	
16	^=	|=	<<=	>>=			re nach li	
17	,						li nach re	Niedrigste Priorität

Zu Operatoren Zeile 1

- Runde Klammern durchbrechen die Ausführungsreihenfolge entsprechend der Operatorpriorität.

  `2*3+4` → `10` aber `2*(3+4)`→ `14`

- Eckige Klammern dienen der Indizierung von Arrays, Sie müssen Ausdrücke vom Typ int enthalten.
- Die Operatoren **`.`** und **`->`** werden später im Zusammenhang mit Strukturen betrachtet.

Zu Operatoren Zeile 2

- Die Operatoren in dieser Zeile sind unäre Operatoren, sie haben nur einen Operanden.
- Der Operator ! (logisches not) negiert Wahrheitswerte (Anmerkung dazu nächste Seite)
- Der Operator ~ (bitweises not) wird auf ganzzahlige Daten angewandt. Er negiert bitweise. Aus jedem Bit 0 wird 1 und aus jedem Bit 1 wird 0.

Anmerkungen zu Wahrheitswerten

- Wahrheitswerte werden in c durch Integerwerte repräsentiert.
- Der Wert 0 repräsentiert den Wahrheitswert false
- Jeder von 0 verschiedene Wert repäsentiert den Wahrheitswert true, Vergleichsoperatoren oder not liefern für true den Wert 1.
- Achtung!
- Für **`int i=-1;`** (logisch true) liefert **`i=!i;`** den Wert 0.
- Das gilt für jeden Ausgangswert, der ungleich 0 ist.
- Für **`int i=0;`** liefert **`i=!i;`** den Wert 1.

Zu Operatoren Zeile 2

- Der Operator – (negatives Vorzeichen) kann auf alle numerischen Daten angewandt werden.

  z.Bsp.: **`-1`** oder **`-i`** oder **`-(i+1) oder -i+1`**

- Typecast: ein Datentyp in runden Klammern vor einem Operanden wandelt temporär den Operanden in den Typ um, der in den runden Klammern angegeben ist.

  z.Bsp.: **`(long) i`** oder **`(double) i,`** wobei i eine int-Variable sei.

Zu Operatoren Zeile 3

- Die Operatoren **++** und **- -** werden Increment- und Decrementoperator genannt. Der angegebene Operand, er muss ein L-Value (beschreibbar) sein, wird um den Wert 1 erhöht oder verringert.
- Beide Operatoren gibt es als Prefix oder Postfixoperatoren, d.h. sie können vor oder nach dem Operanden stehen.

Zu Operatoren Zeile 3

- Handelt es sich um einen Postfixoperator, **`++`** oder **`--`** steht hinter dem Operanden, so wird ++ ganz am Ende des Ausdrucks ausgeführt (**`i++`**).
- Steht **`++`** oder **`--`** vor dem Operanden (++i), ist es ein Prefixoperator, die Operation wird ganz am Anfang des Ausdrucks ausgeführt .
- Leider gibt es Implementationsabhängige Unterschiede. Der Ausduck **`i=i++ + ++i;`** liefert auf verschiedenen Plattformen unterschiedliche Ergebnisse.
- Empfehlung: Diese Operatoren nicht in komplexen Ausdrücken, Macros oder Funktionsaufrufen (kommt alles später) verwenden.

Zu Operatoren Zeile 3

- Die Operatoren * und & finden im Zusammenhang mit Pointern Anwendung.
- & liefert die Adresse einer Variablen, braucht man bei scanf (nicht empfohlen).
- Der Operator sizeof liefert die Größe des nachfolgenden Operanden in Bytes. Man kann als Operand eine Variable oder auch in runden Klammern einen Datentyp angegeben.

```
int len=sizeof (long);
```

1	( )	[ ]	->	.			li nach re	Höchste Priorität
2	!	~	-	(…) cast				
3	++	--	&	*	sizeof		re nach li	
4	*	/	%				li nach re	
5	+	-					li nach re	
6	<<	>>					li nach re	
7	<	<=	>	>=			li nach re	
8	==	!=					li nach re	
9	&						li nach re	
10	^						li nach re	
11	|						li nach re	
12	&&						li nach re	
13	||						li nach re	
14	? :						li nach re	
15	=	*=	/=	%=	-=	&=	re nach li	
16	^=	|=	<<=	>>=			re nach li	
17	,						li nach re	Niedrigste Priorität

Zu Operatoren Zeile 4

- Operatoren für arithmetische Ausdrücke
- Operanden müssen typverträglich sein.
- Bei Operanden unterschiedlicher Datentypen wird (gilt als „Faustregel“) in dem Datentyp mit dem größeren Wertebereich „gerechnet“.

Zu Operatoren Zeile 4

- Zeile 4 enthält die multiplikativen arithmetischen Operatoren

*	/	%

- Auf ganze Zahlen angewandt, wird auch ganzzahlig gerechnet, bei Zahlenüberlauf gehen Stellen verloren, bei der Dision wird der ganzzahlige Rest verworfen (**`9/5` → `1`**), es wird nicht gerundet.
- Der Operator % liefert den ganzzahligen Rest einer Division (**`9%5` → `4`**) und ist nur für ganzzahlige Daten verfügbar (nicht float oder double).

Zu Operatoren Zeile 5

- Zeile 5 enthält die additiven arithmetischen Operatoren

+	-

- Auf ganze Zahlen angewandt, wird auch ganzzahlig gerechnet, bei Zahlenüberlauf gehen Stellen verloren.
- Beim Rechnen mit Gleitpunktzahlen entstehen Rechenfehler, da die Exponenten durch Verschiebung der Matisse angeglichen werden müssen. Hierbei können kleine Zahlen u.U. völlig verschwinden.

- Recherche: addition gleitpunktzahlen cancellation

Zu Operatoren Zeile 6

- Zeile 6 enthält die Bitverschiebeoperatoren

<<	>>

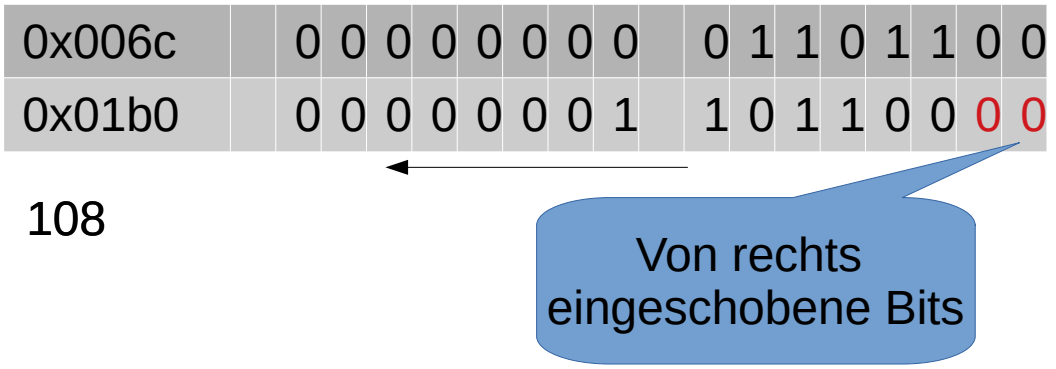
- Bei diesen Operatoren werden die Bits des linken Operanden um so viele Stellen verschoben, wie der rechte Operand angibt.

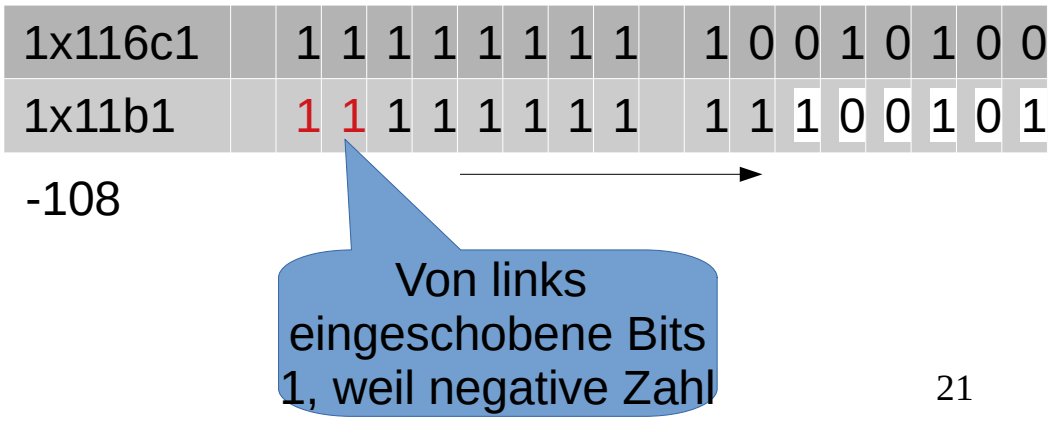
  `int x=0x6c; // … 00000000 01101100`

  `x=x<<2` verschiebt die Bits von x um zwei Stellen nach links, was einer Multiplikation mit 4 gleichkommt.

Zu Operatoren Zeile 6

- Bei einer Bitverschiebung nach links werden von rechts Nullen eingeschoben.
- Bei einer Bitverschiebung nach rechts werden Nullen von links eingeschoben, wenn die Zahl positiv ist. (Die Zahl ist unsigned oder das höchstwertige bit ist 0)
- Bei einer negativen Zahl (signed Datentyp) und die höchstewertige Bistelle ist 1, werden von links Einsen eingeschoben.





Multiplikation mit 10

```c
#include <stdio.h>
#include <stdlib.h>
char buf[128];

int main()
{
    int i, erg;
    fgets(buf,128,stdin);
    i=atoi(buf);
    erg=i*10;
    printf("i: %d, i*10: %d\n",i,erg);
    return 0;
}
```

Dieser Code wird generiert:

i → EDX

```
mov -0x8(%rbp),%edx
```

EDX → EAX (EAX=EDX=i)

```
mov %edx,%eax
```

*EAX *4 (i*4)*

```
shl $0x2,%eax
```

*EAX = EAX+EDX (i*5)*

```
add %edx,%eax
```

*EAX = EAX + EAX (i*5+2)*

```
add %eax,%eax
```

Zurückspeichern

```
mov %eax,-0x4(%rbp)
```

Multiplikation mit 10

```c
#include <stdio.h>
#include <stdlib.h>
char buf[128];

int main()
{
    int i, erg;
    fgets(buf,128,stdin);
    i=atoi(buf);
    erg=(i<<3)+(i<<1);
    printf("i: %d, i*10: %d\n",i,erg);
    return 0;
}
```

I*8 + i*2
Entspricht i*10

Zu Operatoren Zeile 7

- Vergleichsoperatoren <, >, <=, >=
- Können auf Daten aller primitiven Datentypen angewandt werden
- Werden in Verbindung mit bedingten Anweisungen oder Schleifen verwendet.
- Liefern einen Wahrheitswert (false oder true, in der Regel 0 oder 1).

Zu Operatoren Zeile 8

- Vergleichsoperation == (Test auf Gleichheit) und != (Test auf Ungleichheit)
- Man beachte unbedingt das doppelte Gleichheitszeichen ==.
- Um Verwechslungen vorzubeugen werden Vergleiche auf Gleichheit oft in folgender Form geschrieben:

  `0 == a`, anstelle **`a == 0`**.

- **`0 = a`** würde einen Compilerfehler provozieren.
- Bei Vergleichen der Art x!=0, kann !=0 auch entfallen, denn wenn x den Wert 0 enthält, entspricht dies ohnehin false.

Zu Operatoren Zeile 9, 10, 11

- Bitweise Operatoren and, or und xor
- Werden auf ganzahlige Operanden angewandt

And

x1	X2	&
1	1	1
1	0	0
0	1	0
0	0	0

Or

x1	x2	|
1	1	1
1	0	1
0	1	1
0	0	0

Xor

x1	x2	^
1	1	0
1	0	1
0	0	0
0	1	1

Tauschen ohne Zwischenspeicher

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <math.h>
 4  char vBuf[128];
 5
 6  int main()
 7  {
 8      int i=3, j=8;
 9      printf("i: %d, j: %d\n",i,j);
10      i^=j; j^=i; i^=j;
11      printf("i: %d, j: %d\n",i,j);
12    return 0;
13  }
```

Zu Operatoren Zeile 12, 13

- Logische Operatoren and, or (&& und ||)
- Werden auf Wahrheitswerte (true/false) angewandt.
- Berechnungen komplexer logischer Ausdrücke erfolgen nach dem Kurzschlussverfahren. Ist bei einer and-Vernüpfung der linke Operand false, muss der rechte Operand nicht mehr bewertet werden, da sich das Ergebnis nicht ändern kann. Bei or analog, wenn der linke Operand true ist.

And

x1	x2	&&
1	1	1
1	0	0
0	1	0
0	0	0

Or

x1	x2	||
1	1	1
1	0	1
0	1	1
0	0	0

```
int i=9, j=4;
 . . .
i && j  → 1
```
weil
true && true → true

aber:

```
int i=9, j=4;
. . .
i & j → 0
 0000 1001
&0000 0100
----------
 0000 0000
```

bitweises and

Beispiel Kurzschlussverfahren

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main()
6  {
7      int   i=8, k=0;
8
9      if (k && i++) printf("das passiert eh nicht");
10     printf("i: %d\n",i);
11
12     return 0;
13 }
```

Da in Zeile 11 k den Wert 0 (false) hat, wird i++ nicht bewertet.

```
$ ./a.out
i: 8
$
```

Zu Operatoren Zeile 14

- Der Bedingungsoperator **?**
- Der erste Operand vor dem Fragezeichen wird als Bedingung interpretiert. Liefert diese true, erhält der gesamte Ausdruck den Wert zwischen ? und :, anderenfalls den Wert hinter dem Doppelpunkt.

```
int i;
...
i = i<0 ? -i : i; // ist i negativ, wird es negiert, sonst nicht
```

Zu Operatoren Zeile 15, 16

- Zuweisungsoperatoren
- Zuweisungsoperatoren sind rechtsassoziativ, d.h. erst wird der rechte Operand bewertet, dann der linke Operand.
- Der Linke Operand muss ein L-Value, ein Operand, der beschreibbar ist, sein (z.Bsp.: eine Variable).
- Der Zuweisungsoperator schreibt den Wert des rechten Operanden auf linken Operanden. Der linke Operand verliert dabei seinen vorhergehenden Wert, er wird überschrieben.

Zu Operatoren Zeile 15, 16

```
int i=3;
...
i = i+1; // i wird mit dem um 1 erhöhten Wert von i überschrieben
```

- Nimmt man an, dass i in der letzten Zeile noch den Wert 3 hat, so wird zunächst die rechte Seite bewertet, sie ergibt 3+1 → 4, sodann wird die 4 wieder nach i geschrieben. Die Variable i enthält jetzt den Wert 4.

Zu Operatoren Zeile 15, 16

- Der Zuweisungsoperator kann mit vielen Operatoren kombiniert werden, wenn Ziel und erster Operand identisch sind.

  `i = i+1` kann auch geschrieben werden als **`i += 1`**

- Auch diese Operatoren sind rechtsassoziativ, erst wird die gesamte rechte Seite bewertet, dann erfolgt die Zuweisung einschließlich der angegebenen Operation

```
int i=3;
...
i *= i+1; // entspr. i=i*(i+1) und ergibt 12
```

Zuweisung und Initialisierung

Man unterscheidet zwischen Zuweisung und Initialisierung

Initialisierung	Zuweisung
Beispiel: int i=0;	Beispiel: i=0;
Anlegen der Variablen und Belegung mit einem Erstwert sind fest miteinander verbunden.	Eine bereits exitierende Variable bekommt einen Wert zugewiesen.
Einmalig bei der Erzeugung	Beliebig oft.

Zu Operator Zeile 17

- Der Kommaoperator ,
- Das Komma listet eine Reihe von Teilausdrücken, die un-abhängig voneinander von links nach rechts bewertet werden auf. Der gesamte Ausdruck erhält den Wert und ist vom Typ des letzten Aus-drucks in der Liste.
- Er hat die niedrigste Priorität

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  char vBuf[128];
5
6  int main()
7  {
8      int i=3, j=8;
9      printf("i: %d, j: %d\n",i,j);
10     i^=j, j^=i, i^=j;
11     printf("i: %d, j: %d\n",i,j);
12   return 0;
13 }
```

Hier ist der Komma-Operator

Schreiben Sie das Beispiel ab und probieren Sie es aus. Was passiert? Formulieren Sie Zeile 7 in drei einzelne Programmzeilen um.