Arithmetische Datentypen

- **Ganzzahlige Datentypen mit Vorzeichen**

  [`signed`] `short` [`int`], [`signed`] `int` oder `signed`, [`signed`] `long` [`int`]

  Zahlbereich (`short`) $\subset$ Zahlbereich (`int`) $\subset$ Zahlbereich (`long`)

  Zweierkomplementdarstellung (häufig verwendet):

ganze Zahl	*Darstellung*
$0 \leq z \leq 2^{n-1} - 1$	Dualdarstellung von z
$-2^{n-1} \leq z \leq -1$	Dualdarstellung von $z + 2^n$

- **Vorzeichenlose ganzzahlige Datentypen**

  `unsigned short` [`int`], `unsigned` [`int`], `unsigned long` [`int`]

ganze Zahl	*Darstellung*
$0 \leq z \leq 2^n - 1$	Dualdarstellung von z

  Addition, Subtraktion, Multiplikation werden modulo 2^n durchgeführt.

- **Ganzzahlliterale**

  Sind immer ≥ 0 ; Vorzeichen gelten als Operatoren.

  Dezimalschreibweise: Ziffernfolge ohne führende Null ggf. mit Suffix

Suffix	*Datentyp (gemäß Größe)*
keiner	`int` → `long`
`u,U`	`unsigned` → `unsigned long`
`l,L`	`long` → `unsigned long`
`ul,lu,Ul,lU`	`unsigned long`
`uL,Lu,UL,LU`	`unsigned long`

  Oktalschreibweise: Ziffernfolge aus `0-7` mit führender Null ggf. mit Suffix

Suffix	*Datentyp (gemäß Größe)*
keiner	`int` → `unsigned` → `long` → `unsigned long`
`u,l,ul,...`	wie bei der Dezimalschreibweise

  Hexadezimalschreibweise: Ziffernfolge aus `0-9`,`a-f`,`A-F` mit führendem `0x`,`0X` ggf. mit Suffix
  Bedeutung der Suffixe wie in der Oktalschreibweise.

- **Kenngrößen der Ganzzahlarithmetik (Auszug aus `<limits>`)**

  Zahlbereich: $[\min, \max] \cap \mathbb{Z}$

Kenngröße	*typischer Wert*[1] *für int*	*typ. Wert*[1] *für unsigned*	*Bedeutung*
`min()`	-2147483648	0	min
`max()`	2147483647	4294967295	max
`is_signed`	true	false	vorzeichenbehaftet
`is_modulo`	true	true	keine Überlaufbehandlung

  Diese Kenngrößen befindet sich für den Datentyp T jeweils in der Klasse `numeric_limits<`T`>`, daher erfolgt der Zugriff mittels `numeric_limits<`T`>::`*Kenngröße*.

  Bsp.: `numeric_limits<int>::max()` liefert den Wert der größten `int`-Zahl.

[1] z.B. g++-7.5 unter Ubuntu Linux 18.04 (amd64)

- **Gleitpunktdatentypen**

  `float, double, long double`

  Zahlbereich (`float`) ⊂ Zahlbereich (`double`) ⊂ Zahlbereich (`long double`)

  Gleitpunktdatentypen bei g++-7.5: Bitmuster $\boxed{s}\boxed{e}\boxed{f}$ (N_e: Bitzahl des Exp.felds)

Wert	*Bereich*	*Bemerkung*
$x = (-1)^s \cdot 1.f \cdot 2^{e-E}$	$0 < e < 2^{N_e} - 1$	normalisierte Gleitpunktzahl
$x = (-1)^s \cdot 0.f \cdot 2^{e-E+1}$	$e = 0,\ f \neq 0$	subnormale Gleitpunktzahl
$x = \pm 0$	$e = 0,\ f = 0$	Null mit Vorzeichen
$x = \pm\infty$	$e = 2^{N_e} - 1,\ f = 0$	unendlich mit Vorzeichen
$x =$ `NaN`	$e = 2^{N_e} - 1,\ f \neq 0$	"not a number"

Datentyp	*Entsprechung*	N_s	N_e	N_f	E
`float`	IEEE-single	1	8	23	127
`double`	IEEE-double	1	11	52	1023
`long double`[2]	IEEE-binary128	1	15	112	16383

- **Gleitpunktliterale** (Schreibweise im Programmtext)

  Immer ≥ 0 ; Vorzeichen gelten als Operatoren.
  Dezimale Festpunkt- oder Exponentialschreibweise ggf. mit Suffix.
  (Ganzzahliger Teil oder Bruchteil darf fehlen).

Suffix	*Datentyp*
`f,F`	`float`
—	`double`
`l,L`	`long double`

- **Kenngrößen der Gleitpunktarithmetik (Auszug aus `<limits>`)**

  Normalisierte Maschinenzahl ≠ 0 (NMZ):

$$x = (-1)^s \cdot B^e \cdot \sum_{i=1}^{t} z_i B^{-i} \qquad z_1 \neq 0,\ \ z_i \in \{0, \ldots, B-1\},\ \ e_{\min} \leq e \leq e_{\max}$$

Kenngröße	*Wert für float*[1]	*Wert für double*[1]	*Bedeutung*
`radix`	2	2	B
`digits`	24	53	t
`digits10`	6	15	dezimale Genauigkeit
`min_exponent`	-125	-1021	$e_{\min}$
`max_exponent`	128	1024	$e_{\max}$
`denorm_min()`	$1.4 \cdot 10^{-45}$	$4.9 \cdot 10^{-324}$	kleinste positive MZ
`min()`	$1.2 \cdot 10^{-38}$	$2.2 \cdot 10^{-308}$	kleinste positive NMZ
`max()`	$3.4 \cdot 10^{38}$	$1.8 \cdot 10^{308}$	größte NMZ
`epsilon()`	$1.2 \cdot 10^{-7}$	$2.2 \cdot 10^{-16}$	$\min\{x : x\,\text{MZ},\ x > 1\} - 1$
`infinity()`			∞
`quiet_NaN()`			`NaN`

  Bsp.: `numeric_limits<double>::infinity()` liefert den Wert ∞ als `double`-Zahl.

[2] Oft nicht als IEEE-binary128 implementiert, dadurch reduzierte Mantissenlänge, z.B. 64 Bit statt 112 Bit
[3] gemäß IEEE-754

- **Arithmetische Ausdrücke**

  Seiteneffekte

  Ausdrücke in C++ besitzen in der Regel einen Wert. Bei der Auswertung eines Ausdrucks können zusätzlich Seiteneffekte eintreten. Im folgenden sei `v` eine ganzzahlige oder Gleitpunktvariable, v_0 ihr Wert *vor* Ausführung des `++`- bzw. `--`-Operators und `a` sei ein Ausdruck mit Wert a_0.

Ausdruck	*Wert*	*Seiteneffekt*
`v++`	v_0	`v` um 1 erhöhen
`v--`	v_0	`v` um 1 erniedrigen
`++v`	v_0+1	`v` um 1 erhöhen
`--v`	v_0-1	`v` um 1 erniedrigen
`v=a`	a_0	Zuweisung von a_0 an `v`

  Bsp.:

```
int i=5,j=5,k;
k=i++;  /* i=6, k=5 */
k=++j;  /* j=6, k=6 */
```

  Die Seiteneffekte sind eingetreten, wenn die Ausdrucksanweisung beendet ist. Jede Variable darf in einer Ausdrucksanweisung nur einmal modifiziert werden, andernfalls ist das Ergebnis undefiniert (z.B. `j=i++*i--`).
  Syntaktisch unzulässig ist die mehrfache Anwendung des Postfixinkrement/dekrement-Operators auf eine Variable (z.B. `i++++`) und die Zuweisung an postfixinkrementierte/dekrementierte Variablen (z.B. `i++=j`).

  Assoziativität

  Klammerung, Operatorenvorrang und Assoziativität bestimmen die Bildung von Teilausdrücken, jedoch wird die Auswertungsreihenfolge dadurch *nicht* festgelegt. Der Übersetzer darf Kommutativ- und Assoziativgesetze nur dann anwenden, wenn die Maschinenoperationen sie auch erfüllen.

  Bsp.:

```
#include <limits>
using namespace std;
      :
double eps=numeric_limits<double>::epsilon();
1.0 + eps/2 - 1.0 ;    /* Ergebnis: 0              */
1.0 - 1.0 + eps/2 ;    /* Ergebnis: eps/2          */
```

  Vorzeichen

  Die Anwendung des einwertigen Plus/Minusoperators kann bereits implizite Typumwandlungen bewirken (s.u.). Klammerung ist nicht unbedingt nötig; `+a` ist gleichbedeutend mit `(0+a)`.

  Bsp.:

```
int i=5,j=2,k;
k=i*-j;     /* gleichbedeutend mit k=i*(-j); */
k=--i;      /*  i=4,  (j=2), k=4             */
k=- -j;     /* (i=4),  j=2 , k=2             */
k=-i++;     /*  i=5,  (j=2), k=-4            */
k=-+i+-++j; /*  i=5,   j=3 , k=-8            */
k=i+++++j;  /*        ?                      */
```

Division und Rest

Bei ganzzahligen (typgleichen) Operanden gilt

`i/j` $= \left\lfloor \frac{\texttt{i}}{\texttt{j}} \right\rfloor$ falls `i`≥ 0, `j`> 0), sonst implementierungsabhängig

`i%j` = `i-(i/j)*j`

Bsp.:

```
int i; double x;
i=9/4;   /* i=2    */
i=9%4;   /* i=1    */
x=9.0/4; /* x=2.25 */
x=9/4;   /* x=2.0  */
```

Zuweisungen

Zuweisungen sind rechtsassoziativ, Mehrfachzuweisungen sind möglich.

Bsp.:

```
double x,y,z,a=5;
x=y=z=a; /* gleichbedeutend mit x=(y=(z=a))                    */
x+=y;    /* gleichbedeutend mit x=x+y (analog fuer -,*,/,%)  */
x*=y+5;  /* gleichbedeutend mit x=x*(y+5), nicht mit x=x*y+5 */
```

- **Implizite arithmetische Typumwandlungen (ohne Zuweisung)**

  Binäre Operatoren

  Ziel: Umwandlung eines oder beider Operanden in denselben Datentyp, der auch Ergebnistyp ist.

  1. Fall: Mindestens ein Gleitpunktoperand

  Ergebnistyp ist der längste beteiligte Gleitpunktoperand; der andere Operand wird in diesen umgewandelt.

  Bsp.:

  ```
  double x;
  x=1/2*0.5;   /* x=0.0  */
  x=0.5*1/2;   /* x=0.25 */
  ```

  2. Fall: Nur ganzzahlige Operanden

  Zunächst Umwandlung kürzerer Datentypen als `int` in den Datentyp `int`, sofern `int` alle Werte des kürzeren Datentyps darstellen kann. Andernfalls Umwandlung in `unsigned int`. („Integererweiterung“)
  Ergebnistyp ist dann der “größte“ vorkommende Datentyp unter Zugrundelegung der Ordnung `int < unsigned < long < unsigned long` (bzw. `unsigned long`, falls `long` nicht alle Werte von `unsigned` darstellen kann). Der andere Operand wird in den Ergebnistyp umgewandelt.

  Unäre Operatoren (Vorzeichen)

  Integererweiterung des Operanden.