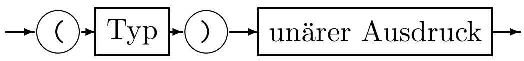
- **Explizite arithmetische Typumwandlungen bzw. implizite Typumwandlungen bei Zuweisung oder Argumentübergabe während eines Funktionsaufrufs**

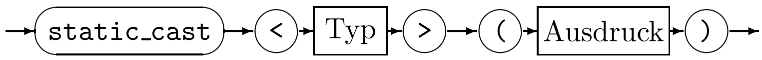
  Neben den bereits erwähnten impliziten Umwandlungen werden auch die nachfolgenden Typumwandlungen durchgeführt. Entsprechendes gilt beim Einsetzen von Argumenten in Funktionen, wenn der Argumenttyp vom Parametertyp abweicht.

ganzzahlig	$\to$	ganzzahlig mit Vorzeichen	Wert unverändert, falls im Zielbereich darstellbar; sonst implementierungsabhängig
ganzzahlig	$\to$	ganzzahlig ohne Vorzeichen	kleinster Wert modulo 2^n
ganzzahlig, reell	$\to$	reell	nächstkleinerer oder -größerer Gleitpunktwert, falls Ausgangswert innerhalb des Zielbereichs; sonst undefiniert
reell	$\to$	ganzzahlig	Abschneiden des Dezimalbruchs, falls Ergebnis im Zielbereich darstellbar; sonst undefiniert

  Explizite Typumwandlungen können mit dem bereits in C vorhandenen Cast-Operator



  oder sicherer durch



  erfolgen.

- **Mathematische Standardfunktionen** (Auszug)

  Funktionen in <cmath>

  Im folgenden stehen x und y für Argumente desselben Gleitpunktzahltyps, der auch Ergebnistyp ist, n für ein Argument vom Typ `int` und l für ein Argument vom Typ `long`.

Funktionen	*Bedeutung*
`abs(x) fabs(x)`	$\lvert x\rvert$
`sqrt(x)`	$\sqrt{x}$
`sin(x) cos(x) tan(x)`	trigonometrische Funktionen
`asin(x) acos(x) atan(x)`	trigonometrische Umkehrfunktionen
`exp(x)`	e^x
`sinh(x) cosh(x) tanh(x)`	hyperbolische Funktionen
`log(x) log10(x)`	$\ln(x)$ $\lg(x)$
`floor(x) ceil(x)`	$[x]$ $-[-x]$
`pow(x,y) pow(x,n)`	x^y x^n
`atan2(y,x)`	Argumentfunktion mit Wertebereich $[-\pi, \pi]$
`ldexp(x,n)`	$x \cdot 2^n$

  Funktionen in <cstdlib>

Funktionen	*Bedeutung*	*Bemerkung*
`abs(n)`	$\lvert n\rvert$	Ergebnistyp `int`
`abs(l) labs(l)`	$\lvert l\rvert$	Ergebnistyp `long`
`rand() srand(n)`	Pseudozufallzahl zwischen 0 und `RAND_MAX`	Ergebnistyp `int`

  In C++11 sind weitere Pseudozufallszahlengeneratoren mit unterschiedlichen Verteilungsfunktionen verfügbar (<random>).

- **Beispiele zur Fehlerbehandlung mit `errno`**

  Programm:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cerrno>
#include <cstring>

using namespace std;

int main()
{
  errno = 0;
  cout << "sqrt(-1) = " << sqrt(-1) << endl;
  cout << strerror(errno) << endl << endl;

  errno = 0;
  cout << "exp(10000) = " << exp(10000) << endl;
  cout << strerror(errno) << endl << endl;

  errno = 0;
  cout << "log(0) = " << log(0) << endl;
  cout << strerror(errno) << endl << endl;

  errno = 0;
  cout << "atan2(0,0) = " << atan2(0,0) << endl;
  cout << strerror(errno) << endl << endl;

  return(0);
}
```

  Bildschirmausgabe (g++-9.4, Ubuntu Linux 20.04):

```
sqrt(-1) = -nan
Numerical argument out of domain

exp(10000) = inf
Numerical result out of range

log(0) = -inf
Numerical result out of range

atan2(0,0) = 0
Success
```

- **Logische Ausdrücke**

  Wertebereich des Datentyp `bool`: `true`, `false`.

  Die logischen Operatoren `&&` (und), `||` (oder) und `!` (nicht) akzeptieren Operanden vom Datentyp `bool` Datentypen und liefern ein Ergebnis vom Datentyp `bool`.

  Die Auswertung der binären Operatoren `&&` und `||` erfolgt von links nach rechts. Der zweite Operand wird nur ausgewertet, wenn das Ergebnis nach der Auswertung des ersten Operanden noch nicht feststeht.

  Bsp.: `if (x>0 && log(x)>y) { ... }` Auch die Vergleichsoperatoren `<`, `>`, `<=`, `<=`, `==` und `!=` liefern als Ergebnis den Datentyp `bool`.

  Wird ein arithmetischer Ausdruck an einer Stelle eingesetzt, an der ein Wert vom Datentyp `bool` erwartet wird, so erfolgt eine implizite Typumwandlung in `bool`. Dabei wird ein arithmetischer Wert ungleich 0 in `true` und 0 in `false` umgewandelt. Das trifft beispielsweise zu, wenn ein arithmetischer Ausdruck als Bedingung in einer `if`-Anweisung oder einer Wiederholungsanweisung verwendet wird. Dasselbe gilt für die logischen Operatoren `&&`, `||` und `!`.

  (Umgekehrt wird bei Typumwandlungen boolescher Werte in Zahlen `false` in 0 und `true` in 1 konvertiert.)

  Vorsicht:
  ```
  if (i==1) { ... }          // falls i den Wert 1 hat
  if (i=1)  { ... }          // immer
  if (1<=x && x<=2) { ... } // falls x in [1,2]
  if (1<=x<=2) { ... }      // immer
  if (!(x>0)) { ... }       // falls x<=0
  if (!x>0) { ... }          // falls x==0
  ```

  Mit dem ternären Operators `? :` läßt sich der bedingte Ausdruck

  Bedingung? *Ausdruck1* : *Ausdruck2*

  bilden. Falls die Bedingung einen Wert $\neq 0$ liefert, wird nur der Ausdruck1 ausgewertet und ist das Ergebnis des bedingten Ausdrucks. Andernfalls wird der Ausdruck2 entsprechend herangezogen.

  Bsp.:
  ```
  c = a>b ? a : b    /* c=max(a,b) */
  c = a>=0 ? a : -a  /* c=abs(a)   */
  ```

- **Bitoperatoren**

  Bitoperatoren können auf ganzzahlige Operanden angewandt werden. In der Regel sollten diese einen vorzeichenlosen Datentyp (z.B. `unsigned`) haben, weil die Wirkung einzelner Operatoren auf vorzeichenbehaftete Datentypen (z.B `int`) implementationsabhängig ist.

Operator	*Bedeutung*
`~`	Bitweise nicht
`<< >>`	Linksshift, Rechtsshift
`&`	Bitweise und
`^`	Bitweise exklusives oder (xor)
`|`	Bitweise (inklusives) oder
`<<= >>= &= |= ^=`	Bitzuweisungen

  Bsp.:
  ```
  unsigned v=14, b;
  b = (v>>2)&1u; /* Bit 2 von v (von rechts ab 0 gezaehlt) in b schreiben */
                 /* v unveraendert                                         */
  v |=   1u<<4;  /* Bit 4 in v (von rechts ab 0 gezaehlt) auf 1 setzen    */
  v &= ~(1u<<3); /* Bit 3 in v (von rechts ab 0 gezaehlt) auf 0 setzen    */
  ```

Vorrang der Operatoren

Die folgende Tabelle enthält die Operatoren von C++ nach ihrer Bindungsstärke geordnet. (Weiter oben aufgeführte Operatoren binden stärker.)

Die Tabelle gibt die Intention der Syntax wieder, in Einzelfällen kann es zu Abweichungen kommen.

Operatoren	*Assoziativität*	*Bemerkungen*
`::`	links	
`() [] -> . ++ --`	links	`++ --` postfix
`typeid static_cast const_cast`	rechts	Vorrang wie vorige Zeile
`dynamic_cast reinterpret_cast`	rechts	Vorrang wie vorige Zeile
`++ -- + - ! ~ & * sizeof new delete (`*Typ*`)`	rechts	`++ --` präfix `+ - & *` unär
`.* ->*`	links	
`* / %`	links	`*` binär
`+ -`	links	`+ -` binär
`<< >>`	links	
`< > <= >=`	links	
`== !=`	links	
`&`	links	`&` binär
`^`	links	
`|`	links	
`&&`	links	
`||`	links	
`? :`	rechts	ternär
`= += -= *= /= %= &= ^= |= <<= >>=`	rechts	
`throw`	rechts	
`,`	links	