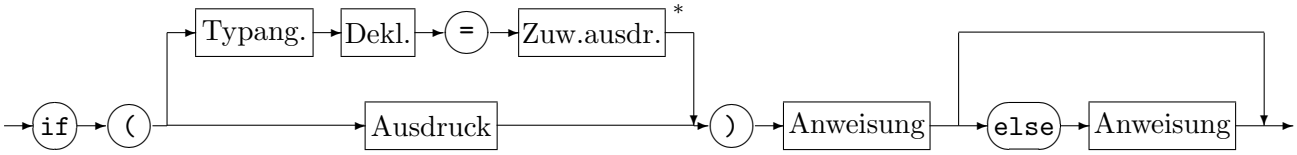
Kontrollstrukturen

Bedingte Anweisung

if-Anweisung :



Falls der Ausdruck (bzw. Zuw.ausdruck) $\neq 0$ ist, wird die erste Anweisung ausgeführt, sonst die zweite, sofern vorhanden.
Die Auswertung des Ausdrucks einschließlich der Seiteneffekte erfolgt vor der Ausführung der abhängigen Anweisung. (Der Gültigkeitsbereich einer mit dem Zuweisungsausdruck vereinbarten Variablen erstreckt sich auf die abhängigen Anweisungen.)
Bei geschachtelten if-Anweisungen wird ein else-Zweig immer dem letzten vorangehenden if, das keinen else-Zweig besitzt, zugeordnet.

Bsp.:

- Lösungen der Gleichung $ax = b$ mit $a, b \in \mathbb{R}$ ausgeben

  1.Fall $a = 0$: Wenn $b = 0$, drucke "jede reelle Zahl"; andernfalls nichts ausgeben

  2.Fall $a \neq 0$: Drucke $\frac{b}{a}$.

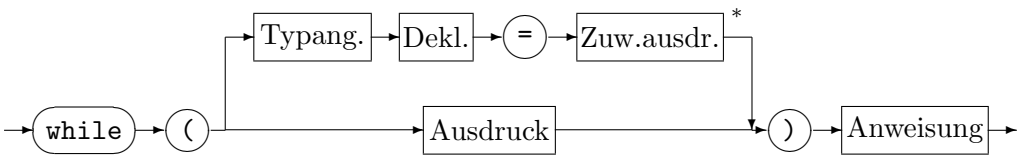
```
cout << "Loesungen von a*x=b: ";
if (a==0)
  { if (b==0) cout << "jede reelle Zahl"; } // ohne geschweifte Kammern falsch
else
  cout << b/a;
```

- Vorzeichen von a

```
if (a>0)
  {sign=1;}        // geschweifte Klammern hier nicht noetig,
else if (a<0)      // aber sinnvoll (s.o.)
  {sign=-1;}
else
  {sign=0;}
```

Wiederholungsanweisungen

while-Anweisung :



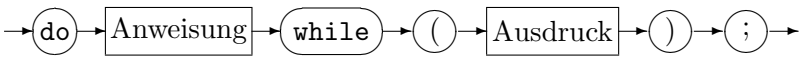
Solange der Ausdruck (bzw. Zuw.ausdruck) $\neq 0$ ist, wird die Anweisung ausgeführt.
Die Auswertung des Ausdrucks einschließlich der Seiteneffekte erfolgt vor der Ausführung der abhängigen Anweisung. (Der Gültigkeitsbereich einer mit dem Zuweisungsausdruck vereinbarten Variablen erstreckt sich auf die abhängigen Anweisungen.)

*nur C++

– $\sqrt{a+\sqrt{a+\sqrt{a+\ldots}}}$ für $a \geq 0$
Iteration: $x_0 = 0;\ x_{i+1} = \sqrt{a+x_i} \quad (i \geq 0)$
Abbruchkriterium: $|x_{i+1} - x_i| \leq \varepsilon$ oder $i > itmax$

```
   :
#include <cmath>
   :
i=0; xa=0; xn=sqrt(a);
while (abs(xn-xa)>eps && i<=itmax)
   {xa=xn; xn=sqrt(a+xa);++i}
```

do-while-Anweisung :

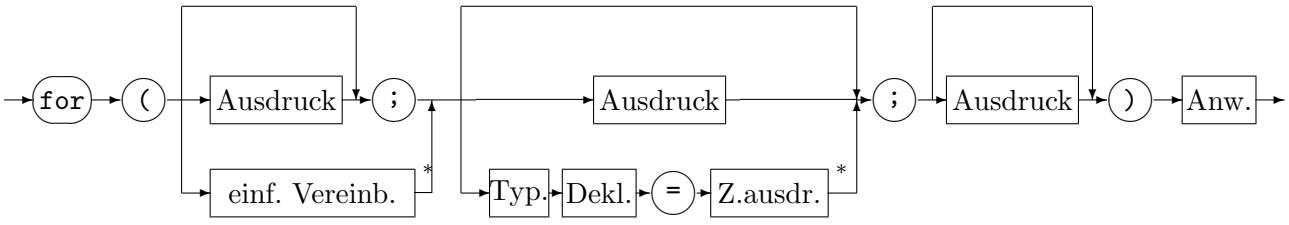


Die Anweisung wird mindestens einmal ausgeführt und solange wiederholt, wie der Ausdruck einen Wert $\neq 0$ liefert.

– *Größter gemeinsamer Teiler von* $a \in \mathbb{N}_0$ *und* $b \in \mathbb{N}$

```
int ggt(int a, int b)
{ // a>=0 b>0 vorausgesetzt (keine Fehlerbehandlung)
  int r;
  do {
    r=a%b; a=b; b=r;
  } while (r!=0);
  return a;
}
```

for-Anweisung :



Die Anweisung ist (bis auf die Wirkung von `break` und `continue`) und den Gültigkeitsbereich der einfachen Vereinbarung äquivalent zu

```
Ausdruck1;  bzw.  einfache Vereinbarung
while(Ausdruck2)  bzw.  while(Typang. Deklarator = Zuw.ausdr)
    { Anweisung Ausdruck3; }
```

Die Ausdrücke in der for-Anweisung können ganz oder teilweise fehlen; fehlt der mittlere Ausdruck so gilt er als wahr.
Der Gültigkeitsbereich der einfachen Vereinbarung reicht bis zum Ende der for-Anweisung.

– $a^n \quad (n \in \mathbb{N})$

```
pot=1; for (int i=1; i<=n; ++i) pot*=a; // weniger gebraeuchlich
pot=1; for (int i=0; i<n; ++i)  pot*=a; // ueblich
```

– Skalarprodukt $\sum_{i=0}^{n-1} a_i b_i \quad (n \in \mathbb{N})$

```
vector<double> a(n),b(n); double s=0;
for (int i=0; i<n; ++i) s+=a[i]*b[i];
```

– Binomialkoeffizient $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdots k} \quad (k, n \in \mathbb{N})$

```
int b,i,k,n;
for (i=1,b=1; i<=k; --n,++i) b=b*n/i;
```

Der linksassoziative Kommaoperator fasst zwei Ausdrücke syntaktisch zu einem einzigen Ausdruck zusammen. Datentyp und Wert des Ausdrucks ist der des rechten Operanden. Die durch Kommas getrennten Funktionsargumente oder Deklarationen sind keine Kommaausdrücke.

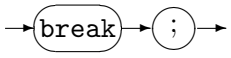
– Vertafelung einer Funktion in $[a, b]$

```
double a,b,h,eps=0.001;  // eps wg. Rundungsfehlern
for (double x=a; x<=b+eps*h; x+=h)
  cout << x << " " << exp(x) << endl;
```

Sprunganweisungen

goto-Anweisung :

Marken brauchen nicht vereinbart zu werden, ihr Gültigkeitsbereich ist die Funktion innerhalb der sie sich befinden. Sie befinden sich in einem eigenen Namensraum. Sprünge mit `goto` sind nur innerhalb einer Funktion möglich.



(nur innerhalb einer Wiederholungs– oder Auswahlanweisung)

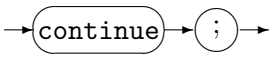
`break` beendet die kleinste umschließende Wiederholungsanweisung oder Auswahlanweisung.

– $x \in \{a_0, \ldots, a_{n-1}\}$?

```
vector<double> a(n);
        :
for (int i=0; i<n; ++i)
  if (a[i]==x) {
    cout << "x=a[" << i << "]" << endl;
    break;
  }
```

– $x \in \{a_{ij} : \ i, j = 0, \ldots, n-1\}$?

```
vector<vector<double>> a(n);
for (int i=0; i<n; ++i) a[i].resize(n);
        :
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    if (a[i][j]==x) {
      cout << "x=a[" << i <<"][" << j << "]" << endl;
      goto fertig;
    }
fertig:  // usw.
```

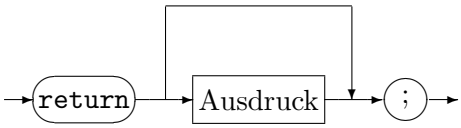


(nur innerhalb einer Wiederholungsanweisung)

`continue` bewirkt die unmittelbare Fortsetzung der kleinsten umschließenden Wiederholungsanweisung.
Sie entspricht einem Sprung zu einer Marke am Ende der abhängigen Anweisung:

```
while (...) {        do {             for(...) {
    :                   :                :
  marke: ;            marke: ;         marke: ;
}                    } while(...);    }
```

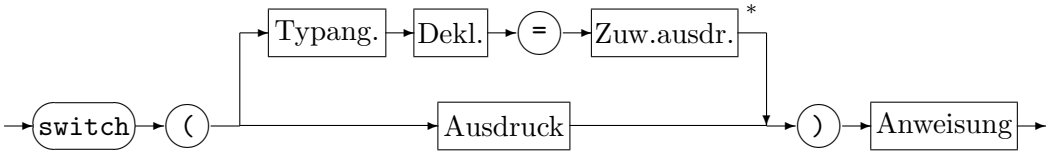
return-Anweisung :



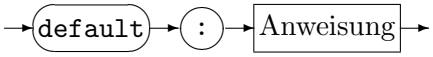
Beendigt die Funktion und weist ihr den Wert des Ausdrucks zu. Bei Funktionen vom Ergebnistyp `void` darf *kein* Ausdruck angegeben werden, in allen anderen Fällen ist er erforderlich.

Auswahlanweisung

switch-Anweisung :



Nur innerhalb der Auswahlanweisung sind die beiden folgenden Anweisungen erlaubt:



`case ...` und `default` wirken als Marken zu denen entsprechend dem Wert des ganzzahligen Ausdrucks ein Sprung erfolgt. Falls der Wert *keiner* Fallkonstanten entspricht, erfolgt der Sprung zur default-Marke. break-Anweisungen innerhalb der abhängigen Anweisung bewirken die Beendigung der switch-Anweisung.

- Mehrfache case-Marken

```
int i;
  :
switch(i) {
   case 1: case 2: case 3: cout << "Medaille" << endl; break;
   default: cout << "keine Medaille" << endl; break;
}
```