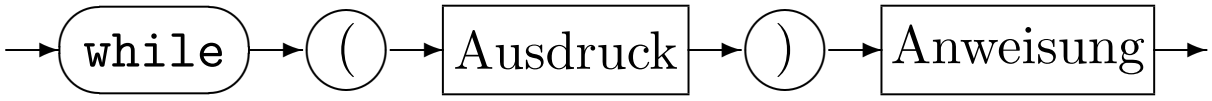
while-Anweisung *(vorläufig)*



Semantik: *Ausdruck* hat die Bedeutung einer Schleifenbedingung
Anweisung wird als "abhängige Anweisung" bezeichnet

Wirkung: Solange *Ausdruck* erfüllt ist (wahr ist),
wird *Anweisung* ausgeführt

genauer: Zuerst wird *Ausdruck* ausgewertet
- falls wahr: *Anweisung* wird ausgeführt
  dann wiederholt sich das ganze
- falls falsch: Programm wird mit der auf *Anweisung* folgenden Anweisung fortgesetzt

Häufig: *Ausdruck* $\equiv$ *Vergleich*

==	$=$	(gleich)
!=	$\neq$	(ungleich)
<	$<$	(kleiner)
<=	$\leq$	(kleiner oder gleich)

Vergleichsoperatoren

Anweisung $\equiv$ { *Anw1 Anw2 . . .* } (d.h Blockanweisung)

Bsp.: Tabelle von Quadratzahlen ausgeben

Programmfragment:

```
int i,n;              // Variablen i und n def.
cout << "n: ";       // Text "n: " ausgeben
cin  >> n;           // n einlesen
i = 1;               // i initialisieren
while (i<=n) {
  cout << i;         // i ausgeben
  cout << " ";       // Leerzeichen ausgegeben
  cout << i*i;       // i*i ausgeben
  cout << endl;      // Zeilenvorschub
  i = i+1;           // i um 1 erhoehen
}
```

Ausgabe (für n=3):

```
1 1
2 4
3 9
```

Kürzer: `cout << i << " " <<  i*i << endl;`

Bsp.: $\sum_{i=1}^{n} i \quad (n \in \mathbb{N})$ *(Inf.bl. 2, S. 4 oben)*

Programmfragment:

```
int i,n;          // i,n Ganzzahlvariable
double s;         // s Gleitpunktvariable
cout << "n: ";    // Eingabeaufforderung
cin  >> n;        // n einlesen
i = 1; s = 0;     // i und s initialisieren
while (i<=n) {
  s = s+i;        // s um i erhoehen
  // 2 Nachkommastellen, Feldbreiten 2 und 6:
  cout << fixed   << setprecision(2)
       << setw(2) << i << setw(6) << s << endl;
  i = i+1;        // i um 1 erhoehen
}
```

Ausgabe für n=3 (ohne Einlesevorgang):

```
 1   1.00
 2   3.00
 3   6.00
```

Systematischerer Zugang zum vorigen Beispiel

Indexumbenennung: $s_n := \sum_{j=1}^{n} j$

damit wir $s_i := \sum_{j=1}^{i} j$ schreiben können

Rekursion: $s_0 = 0, \quad s_i = s_{i-1} + i \ \ (i = 1, \ldots, n)$

```
i=1; s=0;
// s hat hier den Wert s_0
while (i<=n) {
  // Bhpt.: s hat hier den Wert s_{i-1}
  //   Offenbar richtig fuer i = 1
  //   Ann: Bhpt. richtig fuer ein i ∈ {1,...,n}
  s = s+i;
  // s hat hier den Wert s_{i-1} + i = s_i
  i = i+1;
  // s hat hier den Wert s_{i-1}
}
// i hat hier den Wert n+1 und s den Wert s_n
```

Komplizierteres Beispiel: $\sum_{j=0}^{n} \frac{x^j}{j!}$

$a_k := \frac{x^k}{k!} \quad (k \in \mathbb{N}_0), \qquad s_k := \sum_{j=0}^{k} a_j \quad (k \in \mathbb{N}_0)$

Rekursion: $a_0 = 1, \quad a_k = \frac{x^{k-1} \cdot x}{(k-1)! \cdot k} = a_{k-1} \cdot \frac{x}{k} \quad (k \in \mathbb{N})$

$s_0 = 1, \quad s_k = \sum_{j=0}^{k-1} a_j + a_k = s_{k-1} + a_k \quad (k \in \mathbb{N})$

```
k = 1; a = 1; s = 1;   // a: a_0,   s: s_0
while (k<=n) {
  // a: a_{k-1},  s: s_{k-1}  (richtig fuer k = 1)
  a = a*x/k;
  // a: a_{k-1} x/k = a_k
  s = s+a;
  // s: s_{k-1} + a_k = s_k
  k = k+1;
  // a: a_{k-1},  s: s_{k-1}
}
// k: n+1,     s: s_n
```

Anmerkungen zum vorigen Beispiel

- In Form von Kommentaren wurde die Richtigkeit des Programms (für $n \in \mathbb{N}$) bewiesen („Verifikation“)
- Die Verifikation unterbleibt oft, Überlegungen dieser Art sind aber dennoch erforderlich!
- Mehrere Varianten des Programmcodes sind denkbar:
  - Andere Reihenfolge der Rekursionen, z.B.
    $s_k = s_{k-1} + a_k, \quad a_{k+1} = a_k \cdot \frac{x}{k+1}$
    Änderung der Anfangswerte dann notwendig!
  - Start mit $k = 0$ erfordert andere Rekursionsbeziehungen:
    $a_{k+1} = a_k \cdot \frac{x}{k+1}, \quad s_{k+1} = s_k + a_{k+1}$ usw.
- Weshalb nicht einfach x^k und $k!$ zuerst getrennt berechnen und dann dividieren?
  - Ineffizienz: Erheblich mehr arithmetische Operationen erforderlich, wenn Zwischenergebnisse nicht gespeichert werden
  - Zahlbereichsüberlauf: Zähler und Nenner können auch bei positivem x sehr viel größer als Endergebnis werden

Anmerkungen zum letzten Programm auf Inf.bl. 2

- Programm besteht (zu Demonstrationszwecken) aus zwei Funktionen:
  `exp_reihe` und `main`
- `main` ruft `exp_reihe` mit den Argumenten `x` und `n` auf (und zwar versteckt als Argument im Ausgabeausdruck). Möglich, weil `exp_reihe` *vor* `main` definiert wurde.
- Variablen *innerhalb* des Funktionsblocks von `exp_reihe` *nicht* sichtbar (zugreifbar) in `main` und umgekehrt („lokale Variablen“).
- Parameter von `exp_reihe`: verhalten sich ähnlich wie lokale Variablen innerhalb des Fkt.blocks von `exp_reihe`, beim Funktionsaufruf werden eingesetzte Argumente auf die Parameter kopiert („Wertparameter“).
- `main` *nicht* aufrufbar von `exp_reihe`, weil *nach* `exp_reihe` vereinbart. *(Wäre auch nicht sinnvoll!)*