# Auswertungsreihenfolge

Die Reihenfolge in der die Operanden ausgewertet werden ist im allgemeinen von links nach rechts.

Die Auswertung unterbleibt aber in gewissen Fällen wo sie für das Ergebnis nicht notwendig ist. Beispiele sind der `?:` Operator, wie auch die logischen Operatoren `&&` und `||`.

Zwar sind UND und ODER kommutativ, die C++ Operatoren `&&` und `||` aber wiederum (in einem gewissen Sinne) nicht, wie folgendes Beispiel zeigt.

Beispiel 1:

komm.cpp

```
#include <iostream>
int main()
{
  int i;  i = 0;
  bool expr1, expr2;
  expr1 = (i == 0 || 100/i < i); // ok
//   expr2 = (100/i < i || i == 0); // Fehler
  cout << expr1 << " " << expr2 << endl;
}
```

Ein ähnliches Beispiel wird in [Simon] gegeben. Das Programm verhält sich aber anders wenn IEEE Gleitkomma-Arithmetik (mit `-inf` ) zugrundeliegt.

Beispiel 2 :

komm2.cpp

```
#include <iostream>
#include <cmath>
int main()
{
  float x;  x = -3.5;  bool expr1, expr2;
  expr1 = (x <= 0 || log(x) > 5);  // ok
  expr2 = (log(x) > 5 || x <= 0);  // ok(?)
  cout << expr1 << " " << expr2 << endl;
}
```

# Logische und bitweise Operationen

Für die logischen Operatoren **&&** (UND), **||** (ODER) und **!** (NICHT) gelten viele Rechenregeln, die es erlauben eine möglichst gut lesbare Form zu finden:

| | | |
|---|---|---|
| Kommutativität | $p \vee q = q \vee p$<br>$p \wedge q = q \wedge p$ | **p \|\| q**<br>**p && q** |
| Assoziativität | $(p \vee q) \vee r = p \vee (q \vee r)$<br>$(p \wedge q) \wedge r = p \wedge (q \wedge r)$ | **p \|\| q \|\| r**<br>**p && q && r** |
| Distributivität | $(p \vee q) \wedge r = (p \wedge r) \vee (q \wedge r)$<br>$(p \wedge q) \vee r = (p \vee r) \wedge (q \vee r)$ | **(p \|\| q) && r**<br>**p && q \|\| r** |
| De Morgan-Gesetze | $\neg(p \vee q) = (\neg q) \wedge (\neg p)$<br>$\neg(p \wedge q) = (\neg q) \vee (\neg p)$ | **!p && !q**<br>**!p \|\| !q** |
| Involution | $\neg \neg p = p$ | **p** |

and.cpp

Während bei den logischen Operatoren die Operanden nach **bool** konvertiert werden (z.B. von Ganzzahl- oder Zeigertypen), so arbeiten die bitweisen Operatoren **&** (UND), **|** (ODER), **^** (XOR), **~** (NICHT), **<<** (Links-) und **>>** (Rechts-SHIFT) auf Ganzzahltypen.

Bitweise Operatoren werden oft für *flags* benutzt.

Beispiel: Rechte für Dateizugriff. Man beachte: Führende Nullen bedeuten Oktalsystem.

perm.cpp

```
const int READ    = 00400;
const int WRITE   = 00200;
const int EXECUTE = 00100;
permissions = READ|WRITE|EXECUTE; // set flags
if (permissions & WRITE) ...      // get flag
```

# Ausdrücke und Anweisungen

Jetzt wo wir Ausdrücke eingehend besprochen haben, sind wir in der Lage, die *Anweisung* etwas genauer zu definieren. Wir kennen bis jetzt zwei Formen der Anweisung.

Die erste Form ist

```
Ausdruck ;
```

Man beachte: Das Semikolon ist Teil der Anweisung, nicht ein Trennzeichen zwischen Anweisungen wie etwa in Pascal.

Diese Form der Anweisung umfasst Beispiele wie `x = x + 1;` oder `x++;` oder `cout << "x = " << x;` Es handelt sich hier syntaktisch gesehen immer um Ausdrücke (wenn auch mit “Seiteneffekt”) gefolgt vom Semikolon.

Der bereits bekannte *Block*

```
{ Folge von Deklarationen/Anweisungen }
```

gilt ebenfalls als Anweisung. Diese zweite Form heisst zusammengesetzte Anweisung *(compound statement)*.

Die dritte Form ist die *Leeranweisung*

```
;
```

die natürlich nur aus syntaktischen Gründen von Interesse ist.

Man beachte: Ein Block endet auf `}`, nicht auf `;`. Es wird auch kein Semikolon nach dem Block angehängt. Ein solches würde als zusätzliche (Leer-)Anweisung interpretiert. Manchmal stört dies nicht, wohl aber dann wenn der Kontext *genau eine* Anweisung verlangt.

Die Definition von Block und Anweisung ist rekursiv und erlaubt deshalb verschachtelte Blöcke!

Im Zusammenhang mit verschachtelten Blöcken ist interessant dass in jedem Block wieder Deklarationen zugelassen sind. Damit sind Deklarationen mit sehr lokalem Geltungsbereich *(scope)* möglich, was die Gefahr von Namenskonflikten reduziert.

Statt Deklaration und anschliessender Zuweisung `int i; i = 5;` kann eine *Deklaration mit Initialisierung* verwendet werden: `int i = 5;`

Wichtig: Die Initialisierung nicht mit einer Zuweisung verwechseln! Unterschiede äussern sich (später) bei abgeleiteten Typen (z.B. Feldern) und im Zusammenhang mit dem Schlüsselwort `static`.

# Kontrollstrukturen

Ausgehend von den bis jetzt bekannten Anweisungen lassen sich nun neue Anweisungen konstruieren:

Bedingte Anweisungen

- `if`
- `switch`

Schleifen-Anweisungen

- `while`
- `do`
- `for`

# Die if-Anweisung

Die **if** – Anweisung gibt es in zwei Formen:

```
if ( Ausdruck ) Anweisung
if ( Ausdruck ) Anweisung1 else Anweisung2
```

Beispiel 1:

```
spalte++;
if (spalte == anzahlSpalten) {
    zeile++;
    spalte = 0;
}
```

Beispiel 2:

```
if (x < 0) xAbs = -x;
else xAbs = x;
```

Wichtig:

- Klammern um den Ausdruck nicht vergessen!
  Fehler: `if debugModus { cout << …`

- Falls als Anweisung ein Block verwendet wird: kein Semikolon anfügen, dies wäre eine Leeranweisung.
  Beispiel: Bei Zeile 1 wäre ein abschliessendes Semikolon ein Syntax-Fehler, bei Zeile 2 nur überflüssig.

```
if (a < b) { min = a;  max = b; }
else       { min = b;  max = a; }
cout << min << " ... " << max;
```

Zusätzliche Klammern `{}` um eine Anweisung sind erlaubt, sie ergeben einen Block mit nur einer Anweisung. Manchmal ist dies sogar notwendig:

```
if (a) if (b) z = 1; else z = 2;
```

kann gleichzeitig als `if` -Anweisung innerhalb einer `if-else`-Anweisung (links) wie auch umgekehrt (rechts) konstruiert werden:

```
if (a)
   if (b) z = 1;
else z = 2;
```

```
if (a)
   if (b) z = 1;
   else z = 2;
```

Frage: Wie wird interpretiert? Antwort: So wie rechts.

Man fährt sicher wenn man immer dann Klammern setzt wenn eingerückt wird.

# Die `switch`-Anweisung

Mit der Fallunterscheidung oder **`switch`**-Anweisung wird ein Ausdruck (z.B. eine Variable) mit einer Reihe von Konstanten verglichen und abhängig davon werden Anweisungen ausgeführt.

Beispiel:

```
char auswahl;
cin >> auswahl;
switch(auswahl) {
     case 's': y = sin(x); break;
     case 'c': y = cos(x); break;
     case 't': y = tan(x); break;
     default:  y = x;      break;
}
```

Die Syntax der **switch**-Anweisung ist:

```
switch ( Ausdruck ) {
     Folge von switch-Labels
     und Anweisungen
}
```

Die **switch**-Labels haben die Form

```
case Konstante :
```

oder

```
default:
```

mit der zusätzlichen Regel, dass kein Label doppelt vorkommen darf. Die Reihenfolge der Labels ist beliebig.

Die Semantik der **`switch`**-Anweisung ist wie folgt:

- Der Ausdruck wird ausgewertet.
- Es erfolgt ein Sprung zu einer Anweisung:
  - Falls der Wert mit der Konstanten eines **`switch`**-Labels übereinstimmt: zur darauf folgenden Anweisung.
  - Sonst, falls ein Label **`default:`** existiert: zur darauf folgenden Anweisung.
  - Sonst: ans Ende der **`switch`**-Anweisung.
- Die angesprungene Anweisung wird ausgeführt und ebenso die nachfolgenden, wobei:
  - Labels übersprungen werden,
  - bei einer Anweisung **`break;`** die **`switch`**-Anweisung verlassen wird.

Ein häufiger Fehler ist das Vergessen der **break**-Anweisung.

- Frage: Wieso muss in C++ jedes Verlassen der **switch**-Anweisung explizit verlangt werden?

- Antwort: Dies erlaubt Konstruktionen wie:

```
switch (buchstabe) {
     case 'a':
     case 'A': bearbeite a/A ;
               break;
     case 'b':
     case 'B': bearbeite b/B ;
               break;
}
```

Jede **switch**-Anweisung kann durch verschachtelte **if**-Anweisungen ersetzt werden.
Wann ist also welche Form von Anweisung angebracht?

- Bei einer grösseren Anzahl Fälle ist **switch** leichter lesbar. Es ist aber auch effizienter, weil der Compiler eine Sprungtabelle *(jump table)* erzeugen kann. Mit dem ausgewerteten **switch**-Ausdruck kann dann in einem Direktzugriff *(random access)* aus der Sprungtabelle das Sprungziel ermittelt werden. Es muss also keine Folge von Vergleichen durchgeführt werden.
- Liegen allerdings die Labels verstreut über einen zu grossen Bereich (beispielsweise 1, 10, 100, 1000, ...), dann wird die **switch**-Anweisung vom Compiler wie ein verschachteltes **if** behandelt.

# Die `while`-Anweisung

Die **`while`**-Anweisung ist die vielseitigste der drei Schleifen-Anweisungen von C++. Sie hat die Syntax

**`while ( Ausdruck ) Anweisung`**

wobei für **`Anweisung`** meist ein Block eingesetzt wird.

Die Semantik der **`while`**-Anweisung ist:

- Der Ausdruck wird ausgewertet.
- Falls der Wert **`false`** oder **`0`** ist, ist die Ausführung abgeschlossen.
- Andernfalls wird die **`Anweisung`** ausgeführt und danach zur erneuten Auswertung des Ausdrucks zurück gesprungen.

Beispiel 1: Der Euklid-Algorithmus zur Berechnung des grössten gemeinsamen Teilers.

```
#include <iostream>
int main()
{
   int a, b;
   std::cout << "Bitte 2 Zahlen eingeben: ";
   std::cin >> a >> b;
   while (b > 0) {
      int c = a%b;  a = b;  b = c;
   }
   std::cout << "Der ggT ist: " << a
             << std::endl;
}
```

Beispiel 2: Quadratwurzel mit Newton-Iteration.

wurzel.cpp

```
#include <iostream>
#include <cmath>
int main()
{
   double x = 2., c;
   std::cout << "Wurzel aus ? ";
   std::cin >> c;
   while (fabs(x*x - c) >= 1e-6) {
      x = x/2. + c/(2.*x);
   }
   std::cout << "Ergebnis = " << x
             << std::endl;
}
```

Die **`while`**-Schleife hat, anders als die (typische) **`for`**-Schleife keine vorbestimmte Anzahl Durchgänge.

Es stellt sich sogar die Frage nach der *Termination* der Schleife, d.h. ob die Ausführung bei gegebener Anfangsbelegung der Variablen nach endlicher Zeit abgeschlossen ist.

Nach dem berühmten Satz über das *Halteproblem* kann diese Frage im allgemeinen nicht durch eine Analyse des Programmes (z.B. vom Compiler) beantwortet werden. Es bleibt nur die effektive Ausführung, und auch diese liefert höchstens die positive Antwort.

In unsicheren Fällen empfiehlt sich eine zweite Abbruchbedingung. Beispiel:

```
int iteration = 0;
while ( fabs(x*x-c) > 1e-6 &&
        iteration++ < maxIterationen ) {
   x = x/2. + c/(2.*x);
}
```

Jetzt würde sich aber auch eine **for**-Schleife (wird noch behandelt) anbieten, zusammen mit einer **break**-Anweisung die auch bei Schleifen möglich ist.

Eine typische Anwendung der **while**-Schleife ist das Lesen/ Bearbeiten von Dateien.

Beispiele: Konversion von Textdateien, mit später erklärten *Elementfunktionen* **get()**, **eof()**und **put()**.

- Von DOS nach Unix:

tounix.cpp

```
char c;
while (!cin.get(c).eof()) {
      if (c != '\r') cout.put(c);
}
```

- Von Unix nach DOS:

todos.cpp

```
char c;
while (!cin.get(c).eof()) {
      if (c == '\n') cout.put('\r');
      cout.put(c);
}
```

# Die do…while-Anweisung

Die **do…while**-Anweisung hat die Syntax

```
do Anweisung while ( Ausdruck );
```

Anweisung und Ausdruck sind gegenüber der **while**-Anweisung also vertauscht, wodurch sich natürlich auch die Semantik ändert.

Die **do…while**-Schleife ist dann praktisch, wenn im Ausdruck Variablen vorkommen, die erst bei der Ausführung der **Anweisung** einen Wert zugewiesen erhalten.

Andererseits ist die **do…while**-Schleife weniger flexibel dadurch dass sie immer *mindestens einmal* durchlaufen wird.

# Die `for`-Anweisung

Die **`for`**-Anweisung hat die Syntax

```
for ( Initialisierung ; Test ; Update )
    Anweisung
```

- Dabei sind **`Initialisierung`**, **`Test`** und **`Update`** syntaktisch gesehen drei Ausdrücke.

- Zwei davon werden zu Anweisungen gemacht und als solche ausgeführt, nämlich **`Initialisierung ;`** und **`Update ;`**. Typische Beispiele sind **`i = 0;`** resp. **`i++;`** (oder **`i--;`** oder **`i += 2;`**).

Die Semantik der `for`-Anweisung ist:

- Die Anweisung `Initialisierung ;` wird ausgeführt.
- Der Ausdruck `Test` wird ausgewertet.
- Falls der Wert `false` oder `0` ist, ist die Ausführung abgeschlossen, sonst wird fortgefahren:
- Die Anweisung `Anweisung` wird ausgeführt.
- Die Anweisung `Update ;` wird ausgeführt.
- Es wird zurückgesprungen zur erneuten Auswertung des Ausdrucks `Test` .

## Beispiel: Potenzreihen, z.B. cos(x)

cos.cpp

```
#include <iostream>
using namespace std;
int main()
{
    int i, n;
    double x, summe=0., potenz=1., fakultaet=1.;
    int vorzeichen=1;
    cout << "Wieviele Terme? ";  cin >> n;
    cout << "Argument = ";       cin >> x;
    for (i = 0; i < n; i++) {
      summe += vorzeichen * potenz / fakultaet;
      vorzeichen *= -1;  potenz *= x*x;
      fakultaet *= (2*i + 1) * (2*i + 2);
    }
    cout << "cos(" << x << ") = " << summe << endl;
}
```

Der Vollständigkeit wegen sei erwähnt, dass sowohl **Initialisierung** wie auch **Update** auch Ausdrücke von der Form **Ausdruck1** , **Ausdruck2** sein dürfen. Die entsprechenden Answeisungen werden dann jeweils von links nach rechts ausgeführt.

Beispiel 1: Schleife über die ersten *N* Zweierpotenzen

```
for (i=0, x=1.; i<N; i++, x *= 2) { … }
```

Beispiel 2: "ggT in einer Zeile", (Eingabe **a**, **b**, Ausgabe **b**)

```
for (; r = a%b; a = b, b = r);
```

Hier wird mit leerer Initialisierung, Test auf **r** ungleich Null, Komma-Ausdruck im Update und Leeranweisung die (zu?) grosse Flexibilität der **for**-Schleife ausgenützt. Folge: Das Ergebnis ist eher kryptisch.

Empfehlenswert ist die **for**-Schleife vor allem in der typischen Form mit einer "Laufvariablen", welche

- initialisiert wird,
- vor jedem Durchgang gegen eine feste Grenze getestet wird,
- während dem Durchgang unverändert bleibt, und
- danach um ein festes "Delta" verändert wird.

```
for (i = iStart; i op iEnd; i += iDelta) {
    // …
    // Hier darf i benutzt aber nicht
    //   veraendert werden.
    // …
}
```

Mit **op** ist einer der Vergleichsoperatoren **< <= > >=** gemeint.

Diese eingeschränkte Form der **`for`**-Schleife hat einige praktische Vorteile gegenüber der **`while`**-Schleife:

- Die Initialisierung und die Veränderung der "Laufvariablen" geht nicht so leicht vergessen, und
- diese Teile der Schleife sind beim Lesen einfacher auffindbar.
- Da die Anzahl Durchgänge zu Beginn der Ausführung feststeht, ist auch die Termination garantiert.

Vom theoretischen Gesichtspunkt aus ist anzumerken, dass die (eingeschränkte) **`for`**-Schleife weniger mächtig ist als die **`while`**-Schleife: nicht alle berechenbaren Funktionen lassen sich damit berechnen.

Es ist auch erlaubt, die Laufvariable erst in der Schleife drin zu deklarieren:

```
for (int i = i0; i < i1; i++) { … }
```

Dies ist gleichwertig mit

```
int i;
for (i = i0; i < i1; i++) { … }
```

Das bedeutet, die Variable ist nicht nur innerhalb der Schleife bekannt.

# break und continue

Die Anweisung **break;** bedeutet: springe aus einer (**while**, **do**, **for**) Schleife oder **switch**-Anweisung heraus.

Man beachte, dass nicht nur auf den nächsthöheren Block gesprungen wird.

Früheres Beispiel, jetzt mit **for**-Schleife:

```
for (int iter = 0; iter < maxIter; iter++) {
   if (fabs(x*x-c) > 1e-6) break;
   x = x/2. + c/(2.*x);
}
```

Die Anweisung **`continue;`** bedeutet: beende den aktuellen Schleifendurchgang und springe an den Schleifenanfang zurück.

Die Sprunganweisungen **`break;`** und **`continue;`** sind nicht wirklich nötig. Das gleiche könnte mit einer **`if`**-Anweisung (und im Falle von **`break;`** mit einer zusätzlichen Abbruchbedingung) bewerkstelligt werden. Im Unterschied zur **`goto`**-Anweisung wird aber die Schleifenstruktur respektiert, so dass man nicht von "Spaghetti-Code" sprechen kann.

Ein Problem das sich manchmal bei der Behandlung von Fehlern stellt ist: Wie springt man aus mehreren verschachtelten Schleifen heraus?

Eine Lösung wäre, eine **bool**-Variable für den Fehlerzustand zu setzen und diese dann in jeder nächsthöheren Schleife abzufragen.

Eleganter geht dies mit den sog. Ausnahmen *(exceptions)*. Dieser aus den Komponenten **try**, **throw** und **catch** bestehende Mechanismus wird später behandelt. Damit kann man nicht nur aus Schleifen, sondern auch aus Unterprogrammen herausspringen.

# Abgeleitete Datentypen

Aufbauend auf den einfachen Datentypen wie **`int`**, **`float`**, **`char`**, **`bool`** oder **`enum`** können nun weitere Datentypen erzeugt werden:

- Elemente vom gleichen Typ lassen sich zu einem Feld *(array)* zusammenfassen.
- Elemente verschiedener Typen ergeben einen Verbund (**`struct`**).
- Mit dem Vereinigungstyp **`union`** kann die gleiche Speicherzelle auf zwei verschiedene Arten interpretiert werden.
- Und mit den Zeigern *(pointers)* lassen sich dynamische Datenstrukturen aufbauen.

# Arrays

Eine mathematische Variable x kann aus mehreren Komponenten bestehen. Beispielsweise besteht ein reeller 3-Vektor $\boldsymbol{x} \in \mathbb{R}^3$ aus $x_1$, $x_2$ und $x_3$.

In C++ wird ein solcher Vektor als `float x[3];` oder `double x[3];` deklariert.

Der *Indexbereich* beginnt aber immer bei 0. Die Komponenten sind daher $x_0$ bis $x_2$ oder in C++ Syntax `x[0]` bis `x[2]`.

Als Index ist jeder Ausdruck zulässig, dessen Wert ganzzahlig ist und innerhalb des deklarierten Indexbereichs liegt, beispielsweise `x[(i+1)%3]`.

Wenn ein Array-Typ oft gebraucht wird, kann man ihm mittels **typedef** einen Namen geben.

Beispiel: statt

```
double x[3], v[3];
```

auch:

```
typedef double Vector3[3];
Vector3 x, v; // Ort, Geschwindigkeit
```

Arrays werden typischerweise mit **for**-Schleifen bearbeitet. Beispiel:

```
for (int i = 0; i < 3; i++) {
    x[i] += timeStep * v[i];
}
```

Man kann den Array bei der Deklaration *initialisieren:*

```
float x[3] = {0., 0., 10.};
```

resp. mit automatischer Dimensionierung:

```
float x[] = {0., 0., 10.};
```

Dagegen ist eine *Zuweisung* von Arrays nicht möglich:

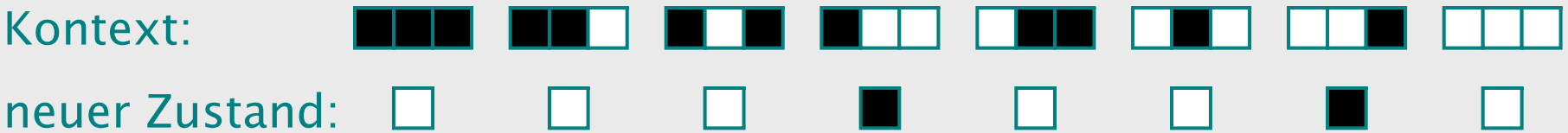
```
float x[3], y[3];
y = x;                  // Geht nicht
y = {0., 0., 10.};      // Geht auch nicht
```

Beispiel mit **`bool`** –Arrays: zellulärer Automat.
(Quelle: S. Wolfram: A New Kind of Science).

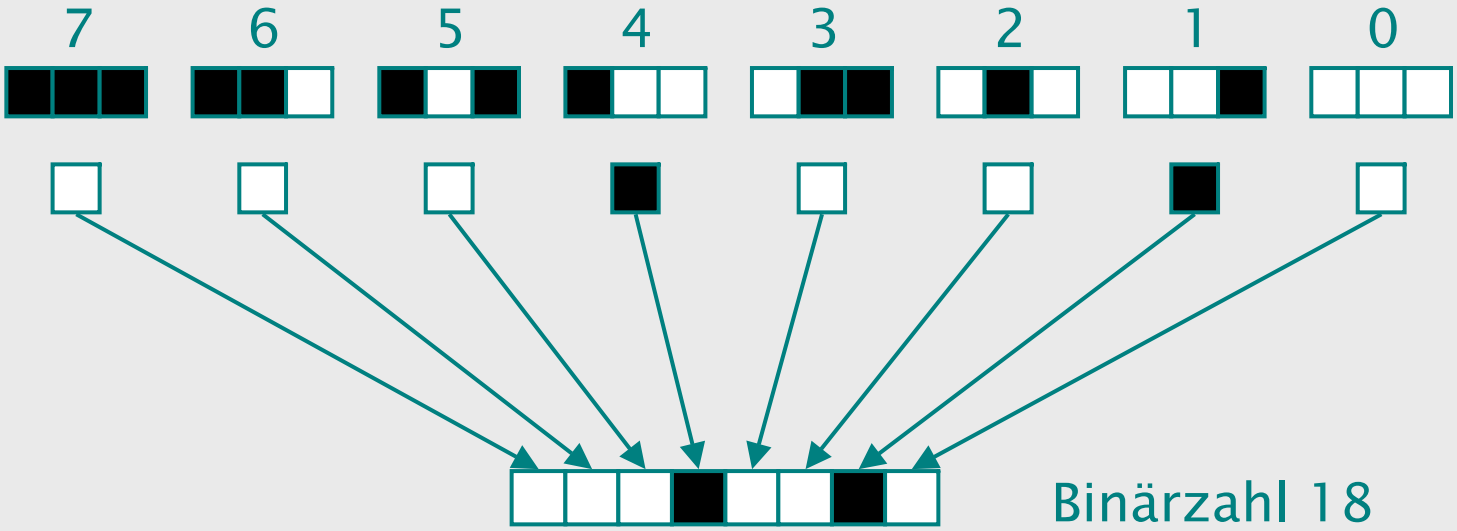
Ein eindimensionaler zellulärer Automat besteht aus:

- einem *Gitter* aus Zellen, die $N$ *Zustände* annehmen können, beispielsweise $N = 2$.
- einem *Anfangszustand*, beispielsweise:

- einer *Übergangstabelle*, beispielsweise:



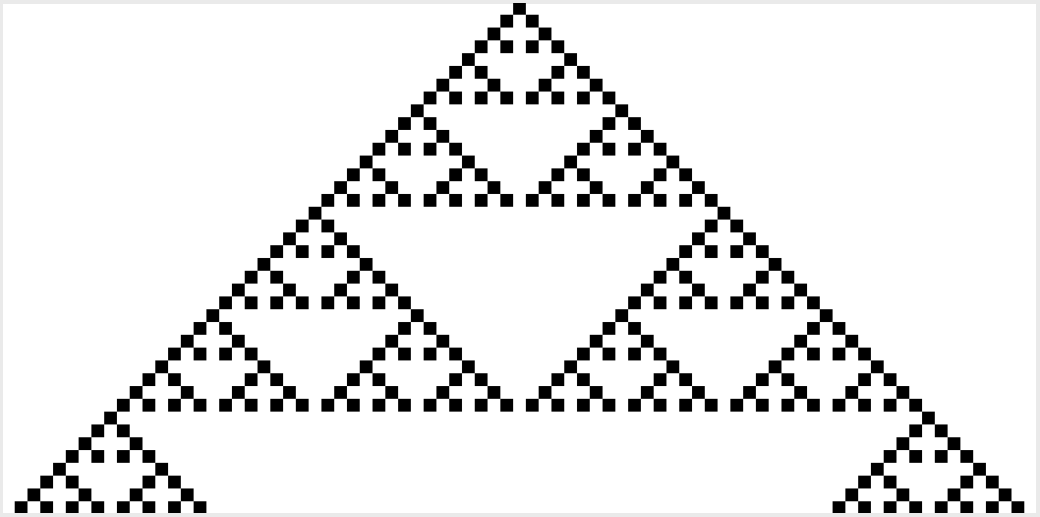
Für die Wahl der Übergangstabelle gibt es 256 Möglichkeiten. Numerierung:




Es handelt sich somit um die Übergangstabelle 18.

In Worten heisst diese „Regel Nummer 18“: eine weisse Zelle mit verschiedenfarbigen Nachbarn wird schwarz, alle anderen Zellen werden weiss.

So sehen die ersten paar Generationen aus:



Und hier ist das vollständige C++ Programm:

```
    // Eindimensionaler zellulaerer Automat
    // g++ cellular.cpp -lwindow -lgdi32

#include <ifmwindow>
#include <iostream>
int main()
{
```

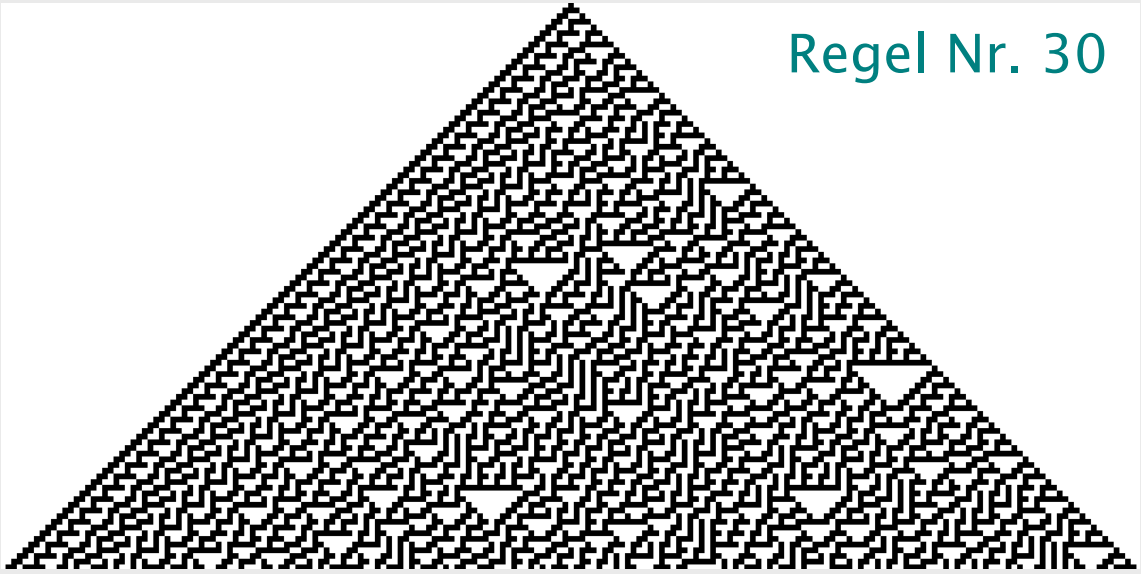
cellular.cpp

```
            // Lies Nummer der Regel ein
int rule;
std::cout << "Regel Nummer = ";
std::cin  >> rule;

            // Zerlege diese Nummer in Bits
bool transition[8];    // Neuer Wert pro Kontext
int context;
for (context = 0; context <= 7; context++) {
    transition[context] = (rule >> context) & 1;
                          // Rechts-Shift, Bit-UND
}
            // Ausgabe der Tabelle
std::cout << "*** **- *-* *-- -** -*- --* ---"
          << std::endl;
for (context = 7; context >= 0; context--) {
    std::cout << (transition[context] ? " *  "
                                      : " -  ");
}
std::cout << std::endl;
```

```
        // Lies Anzahl Generationen ein
const int maxgen = 1000;
int ngen;
std::cout << "Anzahl Generationen = ";
std::cin  >> ngen;
if (ngen > maxgen) ngen = maxgen;

        // Oeffne Fenster
int h = ngen<500 ? 500/ngen : 1;  // Pixels/Zelle
IfmWindow w((2*ngen+1)*h, ngen*h);

        // Initialisiere Gitter
bool state[2*maxgen+1] = { false };
state[ngen] = true;  // setze eine Zelle schwarz
```

```
    // Hauptschleife (ueber die Generationen)
for (int gen = 0; gen < ngen; gen++) {
        // Zeichne das Gitter
  int cell;
  for (cell = 0; cell < 2*ngen+1; cell++) {
     if (state[cell]) {
        int x = cell*h;
        int y = (ngen-gen-1)*h;
        w << FilledRectangle(x-1, y-1, x+h, y+h);
     }
  }
        // Erzeuge naechste Generation
  bool newState[2*maxgen+1];
  for (cell = 1; cell < 2*ngen; cell++) {
     context = 0;
     if (state[cell+1]) context |= 0x1;  // += 1
     if (state[cell  ]) context |= 0x2;  // += 2
     if (state[cell-1]) context |= 0x4;  // += 4
     newState[cell] = transition[context];
   }
```

```
                // Kopiere zurueck
            for (cell = 1; cell < 2*ngen; cell++) {
                state[cell] = newState[cell];
            }
    } // Ende der Hauptschleife

                // Abschluss, warte auf Programmende
    w << flush;                 // Schliesse Grafik ab
    w.wait_for_mouse_click();
    return 0;
}
```



Arrays haben konstante Längen. Eine Deklaration wie

```
double a[n];
```

ist nicht möglich, wenn **n** eine Variable ist. Was macht man nun wenn die Länge variabel sein soll?

Man möchte beispielsweise die Koeffizienten $a_i$ eines Polynoms

$$\sum_{i=0}^{n} a_i \, x^i$$

in einem Array abspeichern. Dazu muss man eine maximale Länge (beim Polynom: den "formalen Grad") festlegen, wozu man eine Konstante verwenden kann.

```
const int nMax = 100; // formaler Grad
double a[nMax+1];
int n;                // effektiver Grad
```

Anwendungs-Beispiel: Das *Horner-Schema* kann Polynome auswerten mit nur je einer Addition und Multiplikation pro Term.

$$\sum_{i=0}^{n} a_i x^i = (...(a_n x + a_{n-1})x + ... + a_1)x + a_0$$

horner.cpp

```
const int nMax = 100;
double a[nMax+1], x;
(...) // Eingabe von n, a, x

double p = a[n];
for (int i = n-1; i >= 0; i--) {
   p *= x;  p += a[i];
}
```