Gleich wie die Parameter werden auch die *lokalen Variablen* auf dem Stack abgelegt, bei rekursiven Aufrufen ebenfalls in mehreren Inkarnationen.

Dies gilt aber nur für sogenannte *automatische* Variablen.

Mit dem Schlüsselwort **`static`** definiert man dagegen Variablen, die nicht auf dem Stack, sondern in einem statischen Speicherbereich angelegt werden.

Für *statische* Variablen gilt:

- Ihre Werte bleiben von einem Funktionsaufruf zum nächsten unverändert erhalten.
- Die Initialisierung wird nur beim ersten Mal durchgeführt.

Beispiel: Erweitern der Funktion um einen Zähler für die Anzahl verschachtelter Aufrufe (die Rekursionstiefe):

```
void hanoi(int anzahl, int von, int ueber, int nach)
{
    static int aufruf = 0;
    cout << "\n" << ++aufruf << ": ";
    if (anzahl == 1) {
      cout << von << "->" << nach;
    }
    else {
        hanoi(anzahl-1, von, nach, ueber);
        hanoi(1, von, ueber, nach);
        hanoi(anzahl-1, ueber, von, nach);
    }
}
```

Analyse der Laufzeit

Die Laufzeit eines Programms hängt im allgemeinen von der Eingabe ab, etwa von der Grösse einer Zahl, der Länge eines Arrays, etc.

Wenn n diese Eingabegrösse bezeichnet, ist also die Laufzeit eine Funktion $g(n)$, die man im allgemeinen nicht genau kennt. Man kann aber oft eine obere Schranke angeben:

Die Notation $g(n) \in O(f(n))$ heisst: Es gibt eine Konstante c so dass $g(n) < c\, f(n)$.

Für $f(n)$ wählt man meist einfache Funktionen wie $f(n) = n$, n^2, e^n, $n \log n$, etc.

Eine rekursive Problemstellung bedeutet nicht automatisch, dass die Implementation als rekursive Funktion auch effizient ist.

Beispiel: Die *Binomialkoeffizienten* lassen sich dank dem Pascal-Dreieck rekursiv definieren:

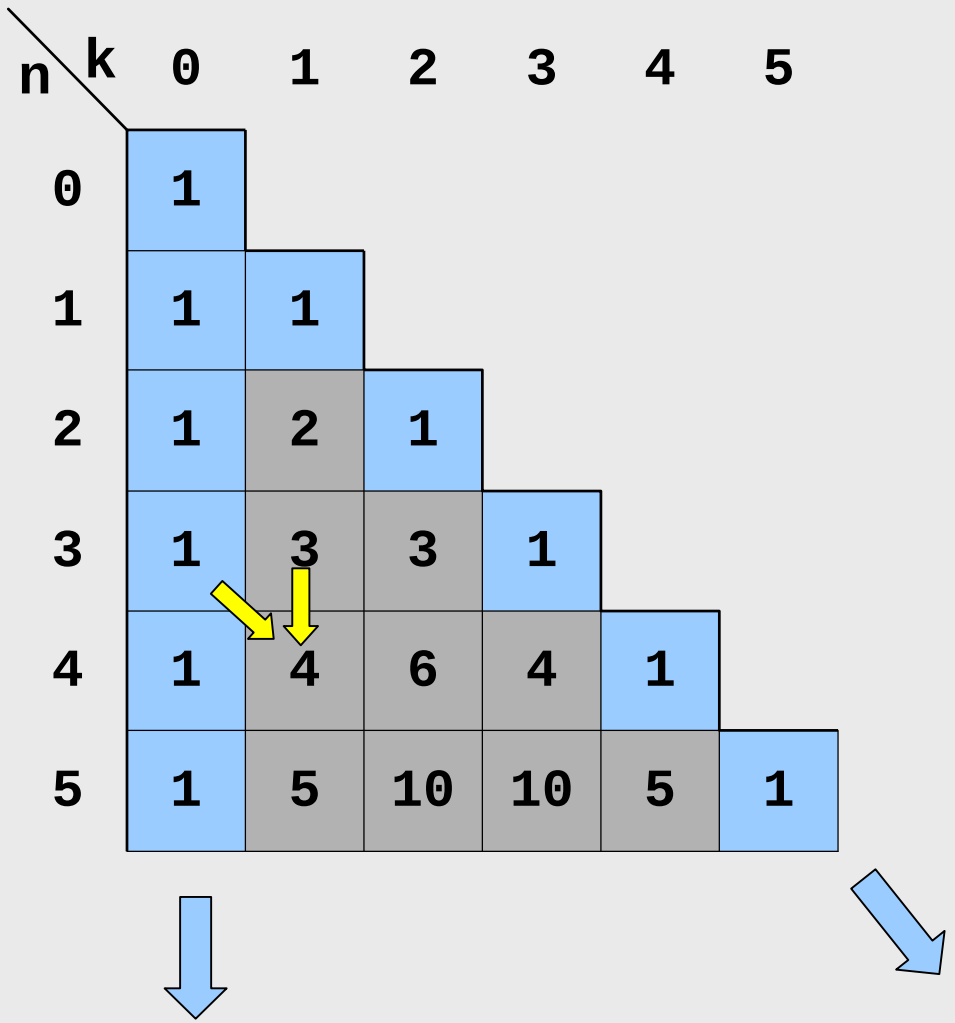
Das Pascal-Dreieck

Verankerung:

$$\binom{n}{0} = \binom{n}{n} = 1$$

Rekursion für $0 < k < n$:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



Die Umsetzung in eine C++-Funktion ist:

```
int binom(int n, int k)
{
  if (k == 0 || k == n) return 1;
  else return binom(n-1,k-1) + binom(n-1,k);
}
```

Wenn zwei rekursive Aufrufe vorkommen:

- entsteht ein binärer *Rekursionsbaum*, deswegen
- ist die Rechenzeit möglicherweise exponentiell.

Im konkreten Beispiel:

- bildet sich das Ergebnis als Summe von lauter Einsen, die Zeit dafür ist also proportional zum Ergebnis und damit tatsächlich exponentiell (in *min(k,n-k)*),
- wird das gleiche Teilproblem mehrfach gelöst.

Dagegen benötigt die iterative Variante nur lineare Zeit in *min(k,n-k)* :

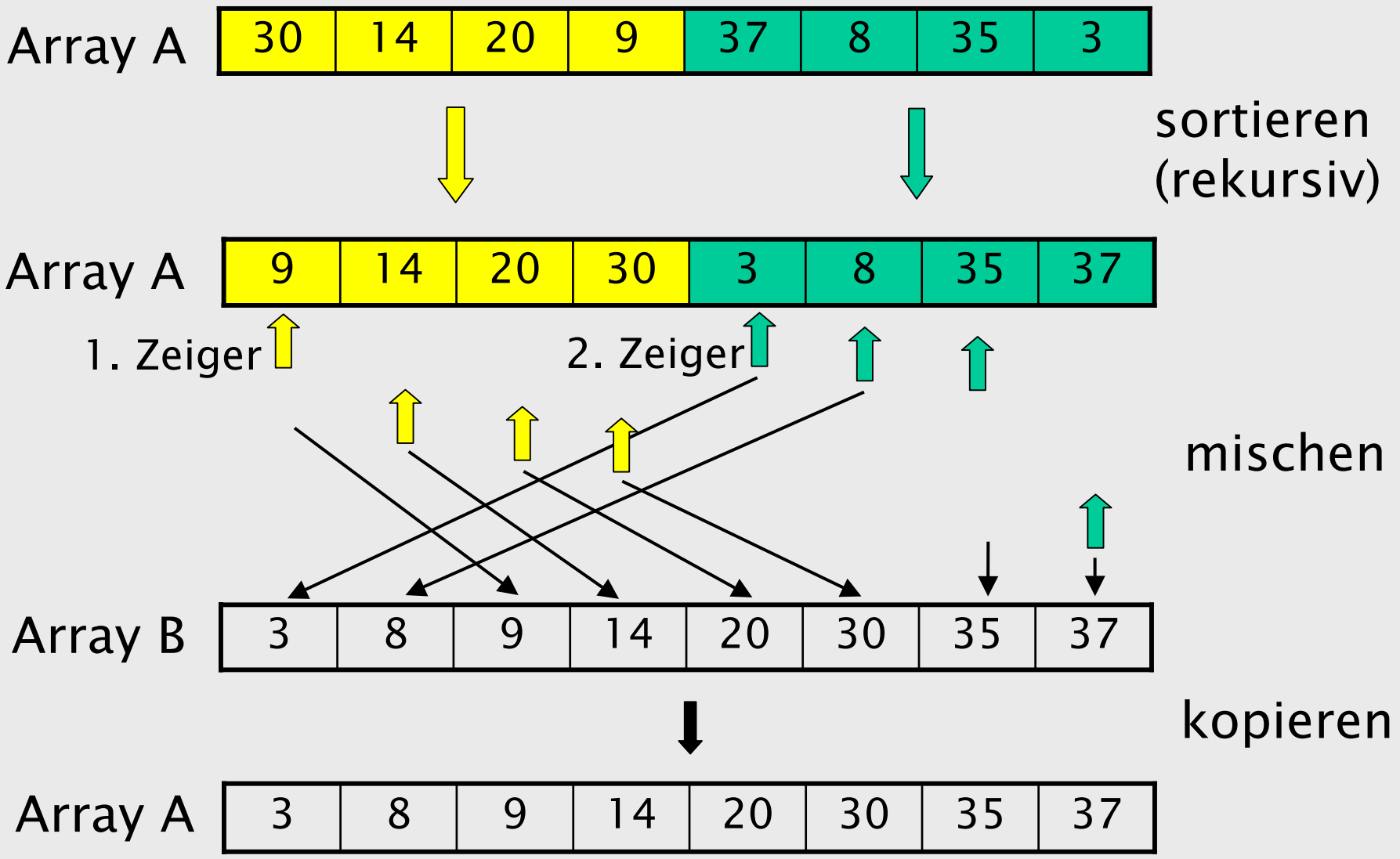
```
int binom(int n, int k)
{
  int b = 1;
  if (k > n/2) k = n-k;
  for (int i = 0; i < k; i++) b = b*(n-i)/(i+1);
  return b;
}
```

Der Merge-Sort Algorithmus

Umgekehrt gibt es viele rekursive Algorithmen, die trotz binärem Rekursionsbaum keine exponentielle Rechenzeit benötigen.

Ein klassisches Beispiel dafür ist *Merge-Sort:*
Um einen Arrays mit n Elementen aufsteigend zu sortieren, kann man so vorgehen:

1. *Sortiere* die ersten $n/2$ Elemente (rekursiv).
2. *Sortiere* die übrigen Elemente (rekursiv).
3. *Mische* die beiden erhaltenen sortierten Teil-Arrays in einen einzigen.


Array A
30 14 20 9 37 8 35 3
sortieren
(rekursiv)
Array A
9 14 20 30 3 8 35 37
1. Zeiger
2. Zeiger
mischen
Array B
3 8 9 14 20 30 35 37
kopieren
Array A
3 8 9 14 20 30 35 37

```
#include <iostream>
#include <cstdlib>
using namespace std;


//  IN: n: Laenge des zu kopierenden Abschnitts
//  IN: offset: Anfang des Abschnitts
//  IN: src: Quell-Array
//  OUT: dst: enthaelt kopierten Abschnitt

void copy(int n, int offset, const float src[],
          float dst[])
{
  for (int i=offset; i<offset+n; i++)dst[i]=src[i];
}
```

```
// IN: n1,n2: Laenge der 2 zu mischenden Abschnitte
// IN: offset: Anfang des ersten Abschnitts
// IN: src: Quell-Array, bereits sortiert:
//          n1 Elemente ab offset,
//          n2 Elemente ab offset+n1
// OUT: dst: Ziel-Array, (n1+n2) Elemente ab offset
//           jetzt sortiert
void merge(int n1, int n2, int offset,
           const float src[], float dst[])
{
    int i1 = offset, i2 = offset + n1;
    int j  = offset;
    while (n1 + n2 > 0) {
        if (n2 == 0 || n1 > 0 && src[i1] < src[i2])
            { n1--;  dst[j++] = src[i1++]; }
        else { n2--;  dst[j++] = src[i2++]; }
    }
}
```

```
// IN: n:    Laenge des zu sortierenden Abschnitts
// IN: offset: Anfang des Abschnitts
// IN: a:    float-Array
// IN: b:    temporaerer Speicherplatz
// OUT: a:   im spezifizierten Abschnitt jetzt
//           sortiert, ausserhalb unveraendert.

void sort(int n, int offset, float a[], float b[])
{
    if (n == 1) return;
    int m = n/2;
    sort(m, offset, a, b);
    sort(n-m, offset+m, a, b);
    merge(m, n-m, offset, a, b);
    copy(n, offset, b, a);
}
```

```
// Testprogramm fuer Merge-Sort Funktion.
// Es werden n Zufallszahlen zwischen 0 und 1
//   erzeugt und anschliessend sortiert.

int main()
{
    const int n = 23;
    float a[n], b[n];
    for (int i = 0; i < n; i++) a[i] = drand48();
    sort(n, 0, a, b);
    for (i = 0; i < n; i++) cout << a[i] << endl;
    return 0;
}
```

Für eine Laufzeit-Analyse nehmen wir an, n sei eine exakte 2er-Potenz. Wir bezeichnen

- mit $m(n)$ die Laufzeit für das Mischen und Kopieren von n Elementen
- mit $s(n)$ die Laufzeit für das Sortieren von n Elementen

Die beiden Grössen erfüllen die Gleichung

$$s(n) = 2\, s(n/2) + m(n)$$

Offensichtlich ist $m(n)$ linear in n, also $m(n) <= cn$ für eine Konstante c. Für eine Abschätzung nach oben setzen wir $m(n) = cn$ und erhalten so die Lösung

$$s(n) = c\, n\, log_2\, n$$

Verifikation:

$$\begin{aligned} s(n) &= 2\, s(n/2) + m(n) \\ &= 2\, c\, n/2\, \log_2 (n/2) + cn \\ &= c\, n \log_2 n - c\, n \log_2 2 + cn \\ &= c\, n \log_2 n \end{aligned}$$

Merge-Sort hat also eine Laufzeit $O(n \log n)$ und ist damit schneller als der nicht-rekursive Bubblesort-Algorithmus (aus den Übungen) mit Laufzeit $O(n^2)$.

Quicksort

Der Merge-Sort-Algorithmus ist zwar leicht zu verstehen und auch schnell, jedoch erfordert er einen temporären Speicherplatz von der Grösse des zu sortierenden Arrays.

Der Quicksort-Algorithmus sortiert dagegen *"in place"* .

Er basiert ebenfalls auf einer *"divide and conquer"* Strategie. Der Array wird aber nicht in der Mitte aufgeteilt, sondern die Elemente werden aufgrund ihrer Grösse in zwei Teil-Arrays aufgeteilt.

Die Grundidee des Quicksort ist:

- Wähle ein beliebiges Element des Arrays als sogenannten *Pivot*.
- Ordne den Array so um, dass
  - ein linker Teil nur Elemente <= dem Pivot
  - ein rechter Teil nur Elemente >= dem Pivot

  enthält
- Sortiere die beiden Teile rekursiv.

Für das Umordnen werden zwei Indices *i* und *j* verwendet mit der *invarianten* Eigenschaft, dass

links von *i* nur Elemente <= Pivot und
rechts von *j* nur Elemente >= Pivot

stehen .

Der Umordnungs-Algorithmus ist nun im wesentlichen:

Wiederhole in einer Schleife:

- Verschiebe die Indices gegeneinander, solange die Invariante gewahrt bleibt.
- Falls $i > j$ verlasse die Schleife.
- Vertausche die zwei Elemente mit Index *i* und *j* .
- Schiebe *i* und *j* um eine Position nach innen.

Die Quicksort-Funktion:

```
// quicksort
//     IN: a:   float-Array
//     IN: von: unterer Index, >= 0
//     IN: bis: oberer Index, <= Array-Laenge - 1
//     OUT: a:  im Bereich [von,bis] jetzt sortiert.

void quicksort(float a[], int von, int bis)
{
  if (von >= bis) return; //schon sortiert (trivial)
  float pivot = a[(von + bis)/2];
```

```
// Phase 1: Sammle Elem. <= Pivot in linkem Teil
//              und Elem.  >= Pivot in rechtem Teil

int i = von;  int j = bis; // Initialisiere i, j.
// Die Invariante ist trivialerweise erfuellt.

while (true) {
  while (a[i] < pivot) i++;
  while (a[j] > pivot) j--;
  // Die Invariante ist immer noch erfuellt.
  // Verlasse die Schleife wenn sich die Indices
  // gekreuzt haben.
  if (i > j) break;
  // Vertausche a[i] und a[j]:
  float t = a[i]; a[i] = a[j]; a[j] = t;
  i++;  j--;   // Invariante bleibt erhalten.
}
```

```
  // Phase 2: Sortiere beide entstandenen Teile
  //          rekursiv (und ebenfalls in-place).
  assert(von < i); // Dies ist Voraussetzung
  assert(j < bis); //   fuer endliche Rekursion.
  if (von < j) quicksort(a, von, j);
  if (i < bis) quicksort(a, i, bis);
}
```

Aufruf (mit `float`-Array der Länge `n`):

```
quicksort(a, 0, n-1);
```

Die Funktion `assert` (benötigt `#include <cassert>`) dient zum Testen von Invarianten und anderen Annahmen. Wenn das Argument den Wert `false` hat, bricht das Programm mit Fehlermeldung ab. Mit der Compiler-Option `-DNDEBUG` werden diese Tests abgeschaltet.

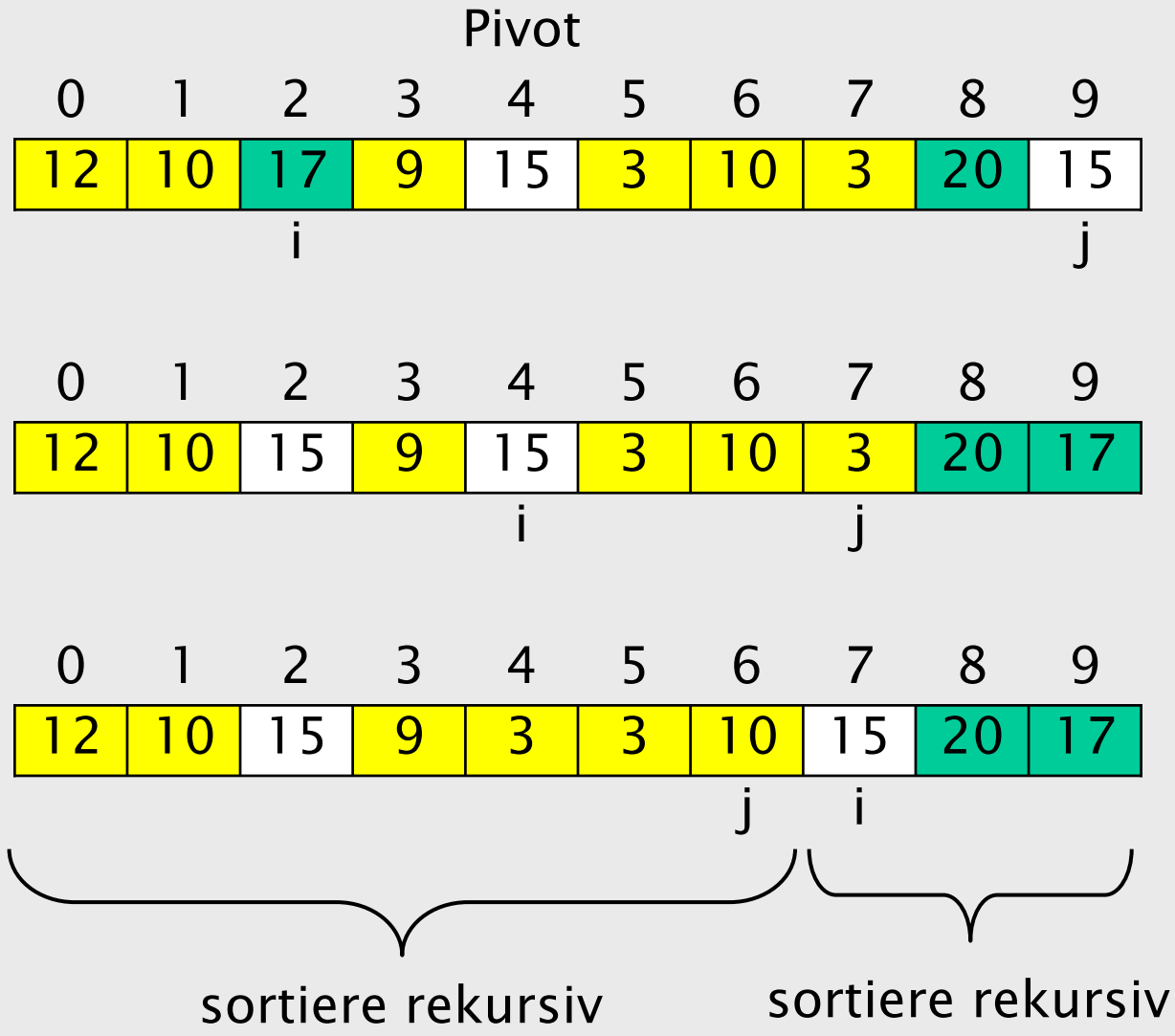
Wichtiges Detail: Beim "Nach-Innen-Verschieben" der Indices i und j wird ein Wert *gleich* dem Pivot nicht akzeptiert:

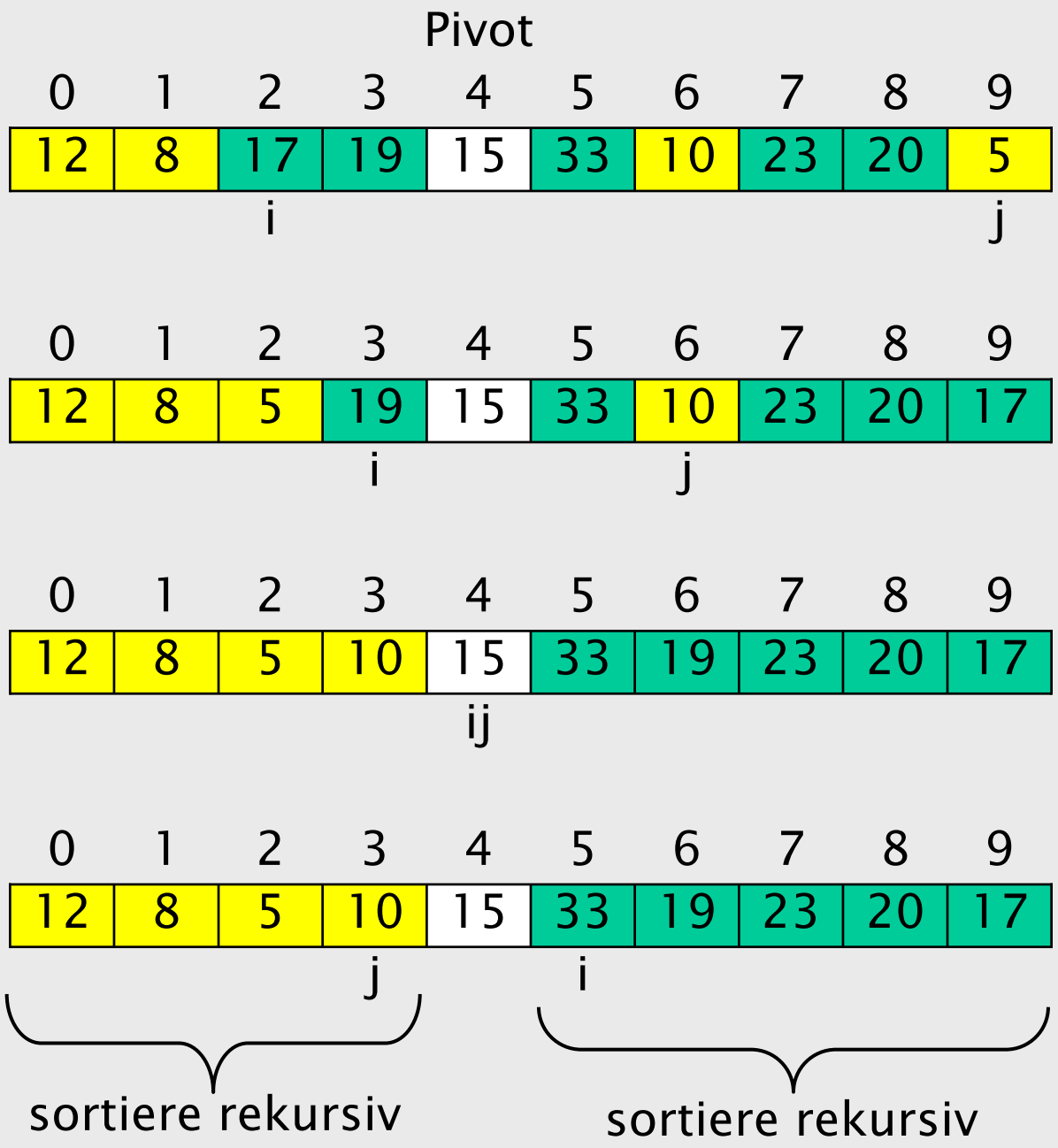
```
while (a[i] < pivot) i++;
while (a[j] > pivot) j--;
```

Grund: Man muss vermeiden, dass der Array so geteilt wird, dass ein Teil der ganze Array und der andere Teil leer ist.

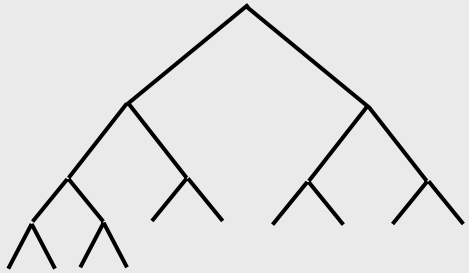
Mit obiger Lösung findet spätestens beim Pivot ein Vertauschen statt, und damit verbunden ein Verschieben *beider* Indices.

Beispiele:


Pivot
0 1 2 3 4 5 6 7 8 9
12 10 17 9 15 3 10 3 20 15
i
j
0 1 2 3 4 5 6 7 8 9
12 10 15 9 15 3 10 3 20 17
i
j
0 1 2 3 4 5 6 7 8 9
12 10 15 9 3 3 10 15 20 17
j
i
sortiere rekursiv
sortiere rekursiv


Pivot
0 1 2 3 4 5 6 7 8 9
12 8 17 19 15 33 10 23 20 5
i
j
0 1 2 3 4 5 6 7 8 9
12 8 5 19 15 33 10 23 20 17
i
j
0 1 2 3 4 5 6 7 8 9
12 8 5 10 15 33 19 23 20 17
ij
0 1 2 3 4 5 6 7 8 9
12 8 5 10 15 33 19 23 20 17
j
i
sortiere rekursiv
sortiere rekursiv

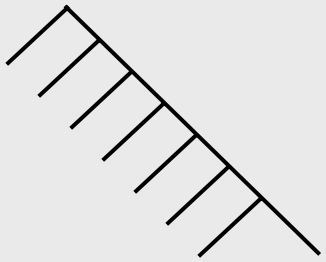
Im Gegensatz zum Quicksort hat Merge-Sort stets einen *balancierten* Rekursionsbaum: die Länge seiner Äste unterscheidet sich maximal um Eins.



Die Anzahl Knoten ist $2n - 1$ und die Teil-Arrays haben im Durchschnitt ungefähr $log_2\ n$ Elemente.

Die *Höhe* des Baumes ist $log_2\ n$ und proportional dazu ist der maximale Speicherbedarf für den Stack.

Beim Quicksort wird der Array im Extremfall jedesmal so geteilt, dass ein einziges Element abgespalten wird.



Die insgesamt n Teil-Arrays haben dann im Durchschnitt ungefähr $n/2$ Elemente, was quadratische Laufzeit bedeutet.

Noch schlimmer: Der Rekursionsbaum hat eine Höhe von n. Das bedeutet, dass mehr Speicherplatz für den Stack als für den ganzen Array gebraucht wird.

Dieser *"worst case"* existiert beim Quicksort-Algorithmus tatsächlich.

Wenn als Pivot jeweils das erste Element gewählt wird, tritt der *worst case* ausgerechnet dann ein, wenn der Array bereits sortiert ist. Diese Pivot-Wahl ist deshalb unbedingt zu vermeiden.

Wird das mittlere Element als Pivot gewählt, existiert zwar ein *worst case*, er muss aber extra konstruiert werden. In praktischen Implementationen wird die rekursive Form dennoch meist in eine iterative umgewandelt.

Ein Vorteil des Quicksort ist, dass er teilsortierte Daten (auch in verkehrter Reihenfolge) besonders effizient behandelt.

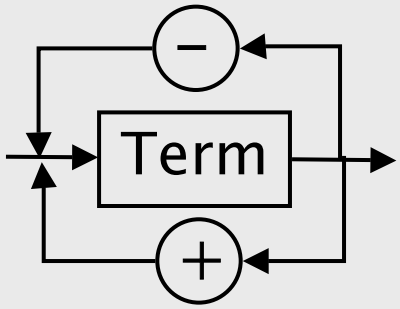
Formale Grammatiken

Eine wichtige Anwendung rekursiver Funktionen ist die Behandlung *formaler Sprachen*, darunter Programmiersprachen.

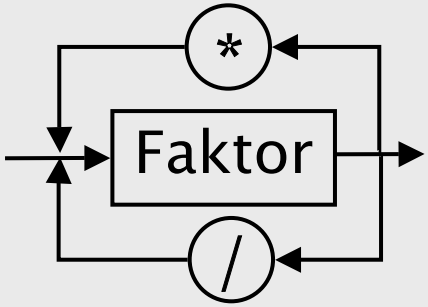
Formale Sprachen sind im allgemeinen rekursiv definiert: In C++ kann z.B. eine Anweisung einen Block enthalten, der wiederum Anweisungen enthält.

In einer einfachen Beispielsprache seien nun Ziffern, Zahlen, Ausdrücke, Terme und Faktoren durch *Regeln* definiert. Die Regeln sind durch *Syntaxdiagramme* dargestellt.

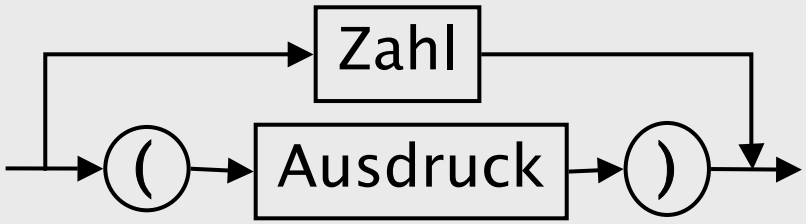
Ausdruck:



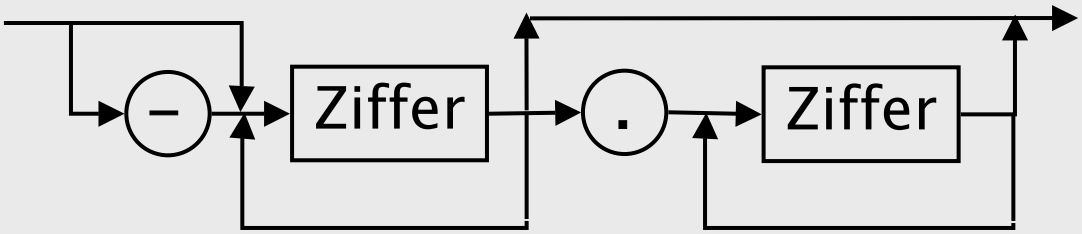
Term:



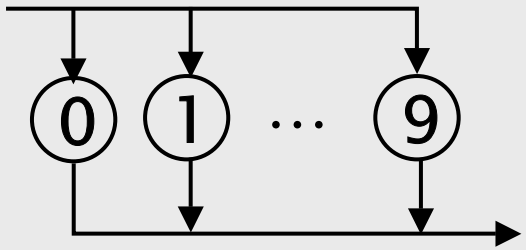
Faktor:



Zahl:



Ziffer:



Die Gesamtheit der Regeln bezeichnet man als *formale Grammatik*. Die Regeln enthalten *Terminal-* und *Nichtterminalsymbole* (Kreise resp. Rechtecke).

Anstelle von Syntaxdiagrammen werden oft auch *Metasprachen* verwendet, die dann allerdings nochmals eigene Symbole ins Spiel bringen. Beispiel:

$$\text{Term} \rightarrow \text{Faktor} \{ + | - \text{ Faktor} \}$$

Wichtig sind speziell die sog. *kontextfreien* Grammatiken, die links vom Pfeil als einziges Symbol ein Nichtterminalsymbol enthalten. Nur kontextfreie Grammatiken können durch Syntaxdiagramme beschrieben werden.

Eine *formale Sprache* ist nun die Menge der Folgen von Terminalsymbolen, die sich ausgehend von einem bestimmten NT-Symbol (z.B. `Ausdruck`) *produzieren* lassen.

Beispiele formaler Sprache sind Programmiersprachen, Formeln, Bäume etc.

Die Grammatik wird benutzt um die *Wörter* einer formalen Sprache zu

- *produzieren,*
- *akzeptieren,*
- *übersetzen,*
- oder *auszuwerten*.

In allen diesen Anwendungen ist es sinnvoll:

- für jedes NT-Symbol eine Funktion zu programmieren,
- pro Vorkommen einen Aufruf zu machen.

Zeiger

Bisher haben wir uns nur mit den *Inhalten* von Speicherzellen befasst. Für die einfachen Datentypen galt:

- Variablen entsprechen Speicherzellen.
- Zuweisung an eine Variable entspricht Schreiben der Speicherzelle.
- Benützung einer Variablen in einem Ausdruck entspricht Lesen der Speicherzelle.

Speicherzellen haben aber nicht nur einen Inhalt sondern auch eine *Adresse*.

Zeiger und Array

Adressen interessieren uns vorerst im Zusammenhang mit *Arrays*:

- Eine als Array deklarierte Variable entspricht nicht einer Speicherzelle sondern einem zusammenhängenden Speicherbereich. Durch die Deklaration wird dieser in der gewünschten Grösse alloziert.
- Jedes Array-Element entspricht wiederum dem Inhalt einer Speicherzelle.
- Die Array-Variable selber entspricht dagegen einer Adresse, genauer: der Adresse seines nullten Elementes.

Beispiel: die wie folgt deklarierten Variablen

```
int n1 = 1234, n2 = 99999;
double a[5] = {1.,.5,.25,.125,.0625};
char c[8] = "ABC";
```

ergeben eine Speicherbelegung wie:

Adresse	Speicherinhalt								Beschreibung
0x22fd10:	1234				99999				n1, n2
0x22fd18:	1.0								a[0]
0x22fd20:	0.5								a[1]
0x22fd28:	0.25								a[2]
0x22fd30:	0.125								a[3]
0x22fd38:	0.0625								a[4]
0x22fd40:	65	66	67	0	…	…	…	…	c[0],…, c[7]

Die Anfangsadresse des Arrays **a** (hier **0x22fd18**) ist also gleichzeitig die Adresse des Elementes **a[0]**.

Die Adressen der nachfolgenden Elemente ergeben sich durch Addition der Elementgrösse (hier 8 Bytes für den Datentyp **double**).

Um mit Adressen operieren zu können, braucht man die Zeiger *(pointers)*. Es gibt zu jedem gegebenen Datentyp einen dazugehörigen Zeigertyp.

Beispiel: zu **double** existiert der Typ “Zeiger auf **double**”, der als **double*** geschrieben wird.

Die Deklaration einer Zeiger-Variablen ist also

`Typname * Variable ;`

wobei die Leerzeichen nicht obligatorisch sind:

- Die traditionelle Schreibweise ist **`double *p;`** und hat den Vorteil, dass der Komma-Operator verwendet werden kann: **`double *p, *q;`**

- Die neue Schreibweise **`double* p;`** ist intuitiver. Aber Vorsicht: **`double* p, q;`** wird als **`double *p; double q;`** interpretiert.

  Eine Alternative ist: **`typedef double* doublePtr; doublePtr p, q;`**

Wenn ein Zeiger deklariert ist, kann man ihm eine *Adresse zuweisen*. Beispiel:

```
double a[5] = {1.,.5,.25,.125,.0625};
…
double* p;
p = a;          // p zeigt auf 0-tes Element
p++;            // und jetzt auf 1-tes Element
```

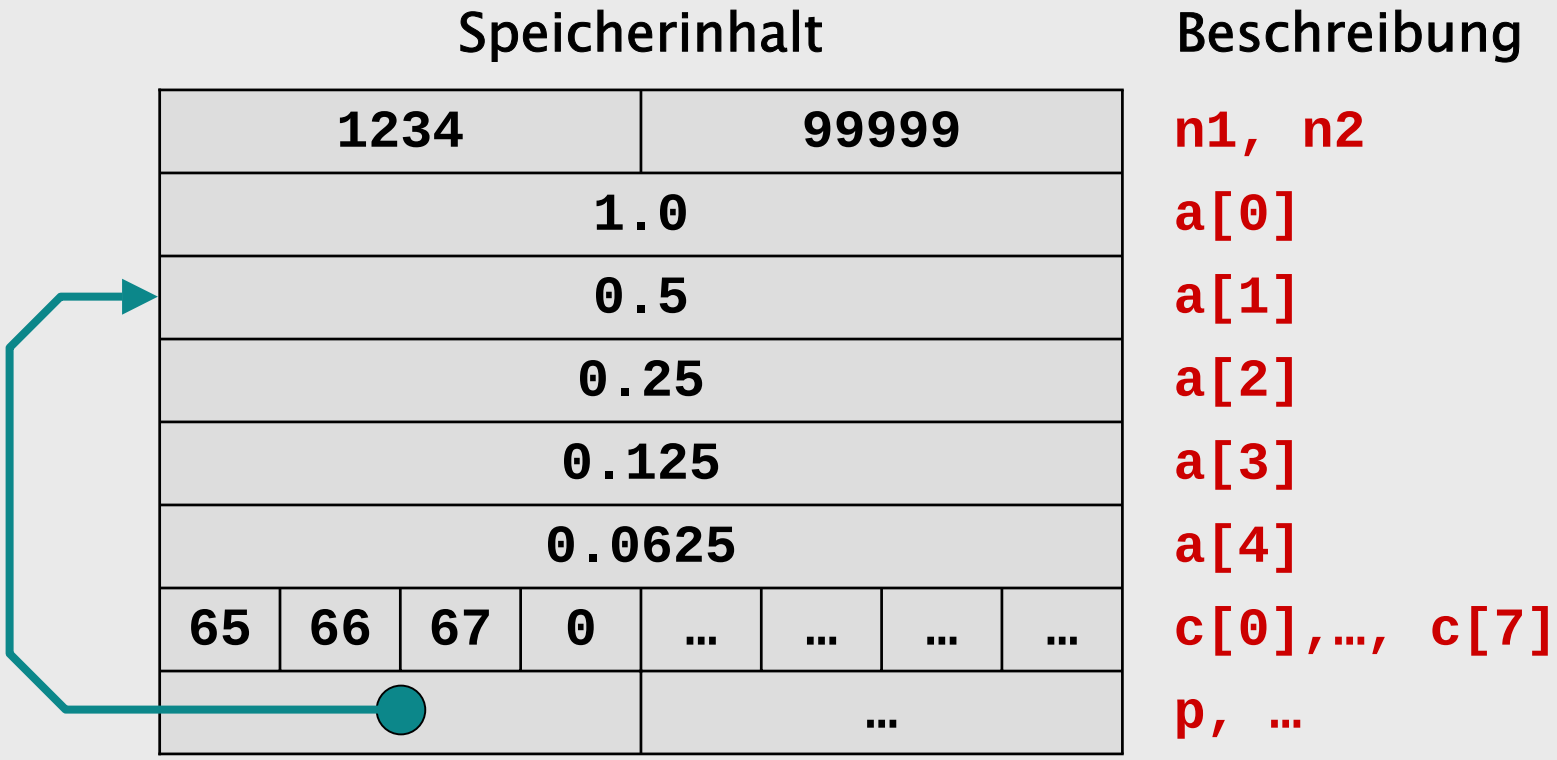
Wie das Beispiel zeigt, kann man mit Zeigern *rechnen*, was sich meist auf Addition/Subtraktion beschränkt, um sich in einem Array vor-/rückwärts zu bewegen.

Man beachte, dass die Zeiger-Arithmetik "intelligent" ist: `p++` (ebenso `p = p+1` oder `p = a+1`) erhöht die Adresse nicht um Eins sondern um einmal die Grösse des jeweiligen Datentyps.

In unserem Beispiel sieht die Speicherbelegung nach der Zuweisung `p = a;` wie folgt aus (Annahme: Betriebssystem mit 32-Bit-Adressen):

Adresse	Speicherinhalt								Beschreibung
0x22fd10:	1234				99999				n1, n2
0x22fd18:	1.0								a[0]
0x22fd20:	0.5								a[1]
0x22fd28:	0.25								a[2]
0x22fd30:	0.125								a[3]
0x22fd38:	0.0625								a[4]
0x22fd40:	65	66	67	0	…	…	…	…	c[0],…, c[7]
0x22fd48:	0x22fd18				…				p, …

Nach der Anweisung **`p++;`** enthält **`p`** die Adresse **`0x22fd20`**. Damit sieht der Speicher jetzt so aus (in der geeigneteren Pfeildarstellung):



Zeigertypen sind keine Ganzzahltypen. Es existiert keine automatische Konversion von Zeigern zu Ganzzahlen und deshalb ist auch die Zuweisung von Ganzzahlen an Zeiger-Variablen verboten.

Die Ausnahme bildet die Zahl 0, die einem Zeiger zugewiesen werden darf, und ihn dadurch zu einem "leeren" Zeiger macht. Das ermöglicht Tests in bedingten Anweisungen oder Schleifen wie `if (p)` oder `while (p)`.

In C/C++ kann auch die symbolische Konstante `NULL` (mit Wert 0) verwendet werden.

Der leere Zeiger heisst in einigen anderen Sprachen *nil*.

Operatoren auf Zeigern

Wie erhält man jetzt den Inhalt der Speicherzelle mit Adresse **p**, der Zelle also worauf **p** "zeigt"?

Dazu dient der *Dereferenzierungsoperator* ***** der vor einem Zeiger (oder einem zeigerwertigen Ausdruck) stehen kann.

Im vorherigen Beispiel würde

```
cout << *p << endl;
```

die Zahl 0.5 ausdrucken (das Element **a[1]**) und

```
cout << *(a + 2) << endl;
```

die Zahl 0.25 (das Element **a[2]**).

Man könnte(!) also statt **a[i]** auch ***(a + i);** schreiben.

Manchmal kann der *-Operator bequemer sein als der []-Operator weil Indexberechnungen entfallen.

Beispiel: Die Elemente zweier zweidimensionaler Arrays abwechslungsweise in einem eindimensionalen Array sammeln:

```
const int n = 7;
int a[n][n] = {...};
int b[n][n] = {...};
int c[n*n*2];

int *aP = a, *bP = b, *cP = c;
for (int i = 0; i < n*n; i++) {
   *cP++ = *aP++;  *cP++ = *bP++;
}
```

Wichtig: Es kann nur dann dereferenziert werden, wenn die Zeiger-Variable eine gültige Adresse enthält. Dies kann auf verschiedene Arten erreicht werden:

- durch Zuweisung der Adresse einer deklarierten Variablen,
- durch Erhöhen oder Erniedrigen einer gültigen Adresse innerhalb eines allozierten Speicherbereichs,
- durch (die später erklärte) *dynamische Allokation* von Speicherplatz.

Die Umkehrung zur Deferenzierung ist der *Adress-operator* **&**. Er gibt zu Variablen oder Array-Elementen die Speicheradresse an. Das Ergebnis kann dann einer Zeiger-Variablen zugewiesen werden.

Bei Arrays gilt speziell:

- **&(a[0])** ist äquivalent zu **a**
- **&(a[i])** ist äquivalent zu **a+i**

  Zeiger*arithmetik* ist nur bei Arrays sinnvoll.

Dagegen sind Zeiger und die Operatoren ***** und **&** auf beliebige Datentypen anwendbar. Beispiel:

```
float e = 2.718281828;
float* ep = &e;
cout << "Eulers Zahl e: " << *ep << endl;
```

Dynamische Variablen

Dynamische Allokation geschieht mit dem Operator **new**.

- Die Anweisung

  Zeigervariable = new Typ ;

  erzeugt eine neue Variable vom Typ **Typ** und weist deren Adresse der Variablen **Zeigervariable** zu.

- Beispiel:

```
int* p;  p = new int;  *p = -99;
```

- Die erzeugte Variable selber ist anonym, sie kann also nur über ihre Adresse angesprochen werden.

- Der Datentyp **Typ** ist oft ein Verbund.

Dynamische Arrays kann man mit

Zeigervariable = new Typ [Dimension];

erzeugen.

Dabei kann **Dimension** eine Konstante, eine Variable oder ein Ausdruck sein, solange der Wert eine positive ganze Zahl ist. Beispiel:

```
int n;
n = (i+1) * (i+1);
int* a;
a = new int[n];
a[0] = -99;
...
a[n-1] = 123;
```

Wenn **n** eine Konstante ist, erzeugt die Deklaration

```
int* a = new int[n];
```

einen Array der auf die gleiche Art verwendet werden kann, wie wenn er als

```
int a[n];
```

deklariert worden wäre.

Aber nur die erste Form der Deklaration lässt für **n** auch Variablen zu.

Ein wichtiger Unterschied zwischen beiden Deklarationen besteht auch in der Lebensdauer der erzeugten Variablen.

Variablen, die innerhalb eines Blocks (und ohne Angabe einer Speicherklasse wie **`static`**) deklariert werden, heissen *automatische Variablen*.

Automatische Variablen "existieren" nur während der Ausführung des Blocks, d.h. der Speicherplatz steht nur während dieser Zeit zur Verfügung. Davor und danach wird er für andere Zwecke benutzt.

Dynamisch allozierter Speicher bleibt dagegen reserviert bis er explizit freigegeben wird.