Informatik I

Wintersemester 2003/2004

http://graphics.ethz.ch/37-847

Dr. Ronald Peikert
Institut für wissenschaftliches Rechnen
ETH - Zentrum, IFW C27.2
8092 Zürich

peikert@inf.ethz.ch
01 63 25569

Ziele der Vorlesung

Einführung ins *Programmieren*, orientiert an den Bedürfnissen in Mathematik und Naturwissenschaften

- Wichtigste Elemente der Sprache C++
- Praktische Programm-Beispiele
- Einbindung von Software-Bibliotheken

Algorithmen und *Datenstrukturen*

- Implementation in C++
- Korrektheit
- Analyse von Rechenzeit und Speicherbedarf

Warum C++?

Entwicklung der Programmiersprachen:

- Assembler – Programmierung auf der Stufe einzelner Maschineninstruktionen, symbolische Namen.
- Fortran('57), Cobol('59), Basic('64) – Frühe Programmiersprachen, Anweisungen statt Instruktionen.
- Algol ('60), Pascal('71), C ('72) – Strukturierte Programmierung, Sicherheit, Effizienz.
- Modula-2('78), Ada('79), C++('86), Java ('95) – Modulare resp. objektorienterte Programmierung, Module, Schnittstellen, Wiederverwendbarkeit, Eignung für grosse Programmsysteme.

- Lisp('58), Prolog ('71) – Spezielle Sprachen, Funktionale Programmierung, Logik-Programmierung.

C++ und Java sind recht ähnliche Sprachen. Java ist unbestritten die modernere und auch "schönere" Sprache. Es gibt aber mehrere Gründe, die für C++ als Basis dieser Vorlesung sprechen:

- *Verbreitung* und *Beständigkeit:* Die heutigen Betriebssysteme (Windows, Unix, Linux, MacOS, etc.) wurden fast ausschliesslich in C oder C++ entwickelt.
- *Kompatibilität* mit vielen Software-Bibliotheken für Numerik, Computer-Graphik, etc.
- *Effizienz,* speziell beim wissenschaftlichen Rechnen ein wichtiges Kriterium.

Die Betriebssystem-Umgebung

Betriebssysteme sind nicht Gegenstand dieser Vorlesung. C++ selber ist unabhängig von der Wahl des Betriebssystems. Mit den Programm-Bibliotheken und dem Compiler kommt aber eine gewisse Systemabhängigkeit ins Spiel.

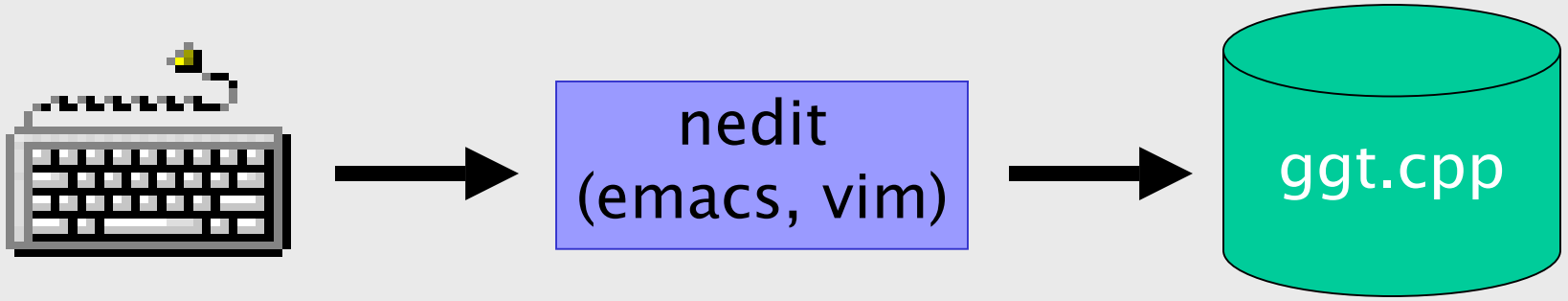
In Vorlesung und Übungen wird deshalb folgende Umgebung zugrundegelegt:

- Eine Unix-Shell als *command line interface* zum Betriebssystem.
- Der Compiler g++ vom GNU Project.

Diese "schlanke" Umgebung erlaubt, auf allen Unix-Systemen sowie den PC-Betriebssystemen Linux und Windows auf recht einheitliche Weise zu arbeiten.

Programmierung und Programme

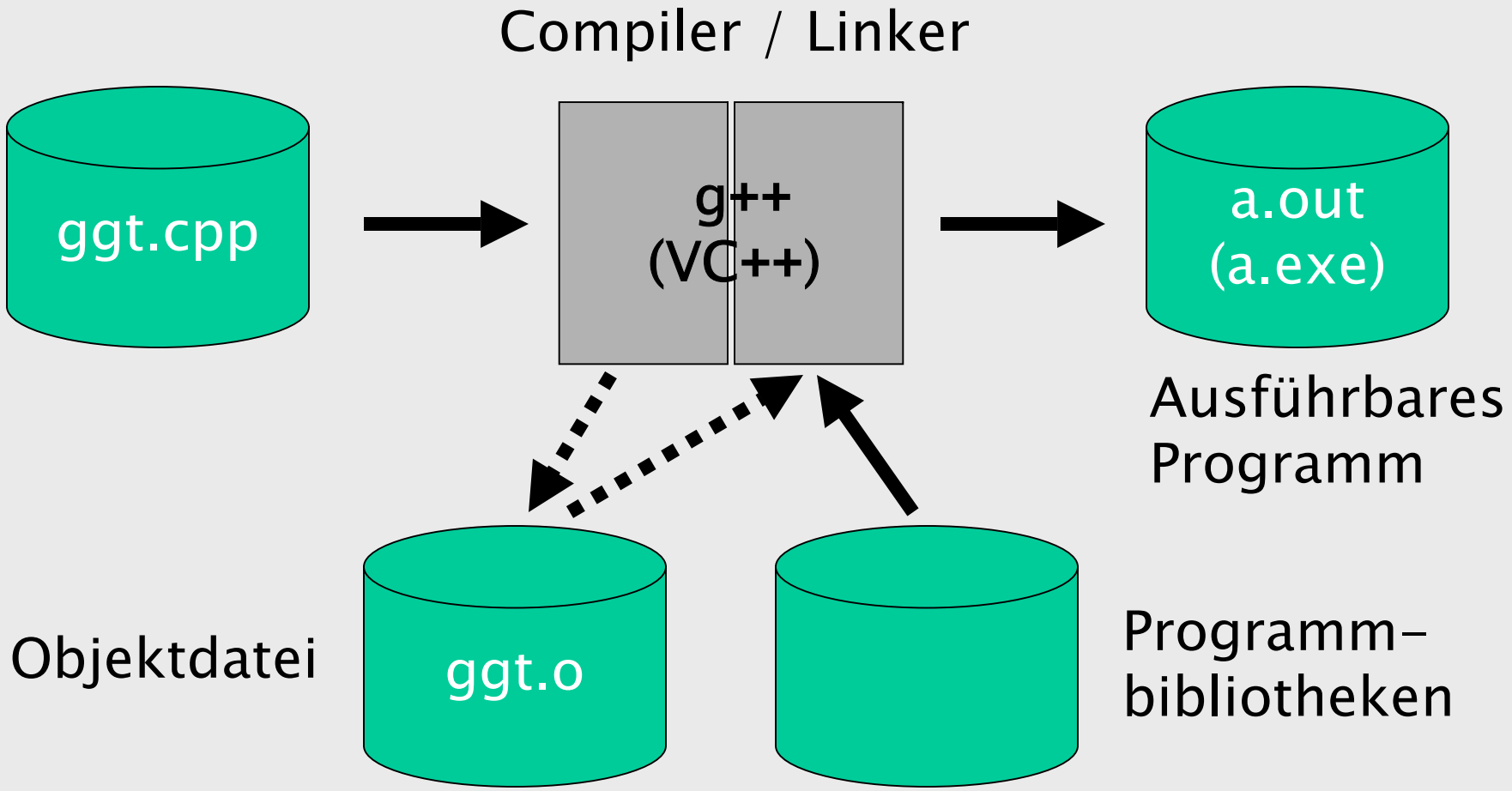
Schritt I – Erstellen eines Quellprogrammes



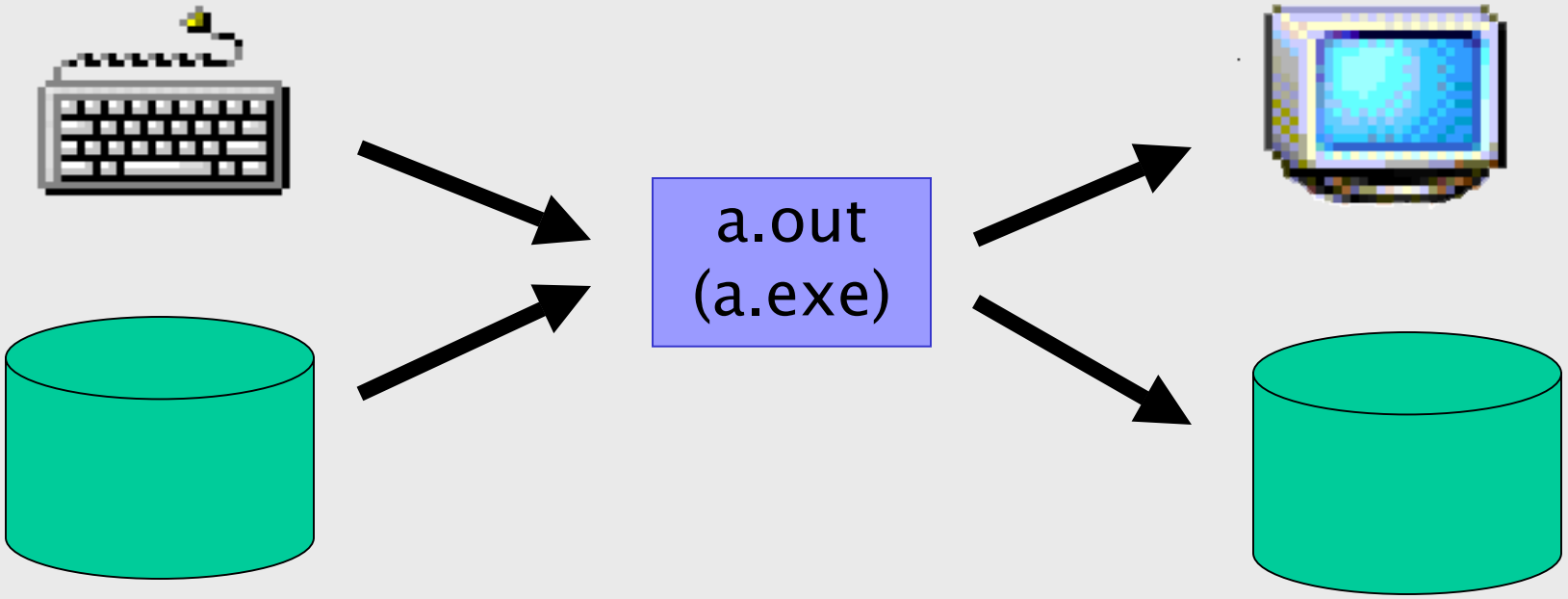
Texteditor

Quellprogramm
in der Sprache
C++

Schritt II – Übersetzen in Maschinensprache



Schritt III – Ausführung



Eingabe von Tastatur und/oder Datei

Ausgabe auf Bildschirm und/oder Datei

Erste C++ Programme

Das einfachste gültige C++ Programm lautet:

```
int main(){}
```

Es ist das “leere Programm”. Vom Compiler wird es akzeptiert und auch übersetzt.

Bei sinnvolleren Programmen befindet sich zwischen den Klammern `{}` eine Folge von *Deklarationen* und *Anweisungen*.

Eine Deklaration ist zum Beispiel

```
float x, y, summe;
```

Und eine Anweisung ist zum Beispiel

```
summe = x + y;
```

Das Programm

```
int main()
{
  float x, quadrat;
  x = 1.5;
  quadrat = x * x;
}
```

ist ebenfalls syntaktisch korrekt.

Fehlte die Deklaration `float x, quadrat;` so wäre dies ein *Syntax-Fehler*. Der Compiler würde diesen melden und das Programm nicht akzeptieren.

Fehlte die Zuweisung `x = 1.5;` so wäre dies ein *semantischer Fehler*. Manche Compiler geben eine *Warnung* aus.

Alle Programme brauchen *Dateneingabe* und *-ausgabe*. Eine Form davon ist Texteingabe und -ausgabe via Tastatur und Bildschirm.

Die Sprache C++ selber bietet keine E/A-Funktionen an, dazu müssen *Programmbibliotheken* verwendet werden. Funktionen für Text-E/A befinden sich in der *Standardbibliothek*.

Die Standardbibliothek wird automatisch dazugeladen. Die Text-E/A Funktionen erfordern aber Deklarationen, die in der Datei “iostream” stehen (in einem Verzeichnis wie /usr/include/g++, je nach System).

Statt die Datei selber ins Programm einzufügen genügt die Zeile

```
#include <iostream>
```

an der betreffenden Stelle.

Damit können wir nun ein Programm mit Ein- und Ausgabe schreiben:

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

int main()
{
      int x, y, summe;
      cout << " x = ";  cin >> x;
      cout << " y = ";  cin >> y;
      summe = x + y;
      cout << " Die Summe ist ";
      cout << summe;  cout << endl;
}
```

summe.cpp

Die ersten zwei Zeilen sind nötig, weil wir E/A-Funktionen aus der Standardbibliothek brauchen wollen.

Nach `main()` folgt syntaktisch gesehen ein sog. *Block*. Blöcke treffen wir später auch an anderen Stellen an. Sie bestehen aus einem Klammernpaar `{}` und dazwischen einer Folge von Deklarationen und Anweisungen.

Sowohl Deklarationen wie auch Anweisungen enden mit einem Semikolon `;`

Im Beispiel sind es eine Deklaration und acht Anweisungen.

Das Beispiel verwendet aus der Standardbibliothek:

- den Eingabestrom **`cin`** und den Ausgabestrom **`cout`**
- die Eingabe- und Ausgabeoperatoren **`>>`** und **`<<`**
- das Zeilenende **`endl`**

Der Operator **`cout`** passt sich dem Operanden-Typ an:

- Zeichenketten *(strings)* werden unverändert ausgegeben.
- Variablen für (ganze oder Gleitkomma-) Zahlen werden ausgewertet und im Dezimalsystem ausgegeben.
- etc.

Nicht jedes Programm braucht Text-E/A. Stattdessen kann z.B. grafische Ein-/Ausgabe verwendet werden.

zeichne.cpp

Die Zuweisung

Die einfachste Form der Anweisung ist die Zuweisung *(assignment)*. Die Zuweisung hat die Syntax

`Variable = Ausdruck ;`

d.h. sie besteht aus dem Zuweisungsoperator **`=`** mit zwei Operanden (links und rechts davon) und dem Semikolon **`;`** mit dem jede Anweisung endet.

Die beiden Operanden müssen eine Variable resp. ein Ausdruck sein, was beides noch erklärt wird.

Die Schreibweise mit den Einrahmungen ist "metasprachlich". Gemeint ist, dass man je eine gültige Variable und einen gültigen Ausdruck einsetzt.

Ein Beispiel einer Zuweisung ist

```
a = a + 1;
```

Die Zuweisung ist keine mathematische Gleichung sondern eine Operation.

Es wird zuerst der Ausdruck ausgewertet und der erhaltene Wert dann der Variablen zugewiesen.

In anderen Sprachen wie Pascal oder Maple wird der Zuweisungsoperator nicht `=` sondern `:=` geschrieben.

Variablen

Eine Variable beschreibt eine Grösse die während der Ausführung des Programmes ihren Wert ändern kann.

Die Variable wird im Computer durch eine Zelle im Arbeitsspeicher (*memory*, RAM) realisiert. Glücklicherweise erspart uns C++ aber, von „Speicherzelle Nr. 7654321“ etc. zu reden. Stattdessen werden symbolische Namen, sog. Bezeichner *(identifiers)* verwendet. Bezeichner werden später auch für weitere Dinge wie Funktionen gebraucht.

Ein gültiger Bezeichner ist eine Folge von Buchstaben, Ziffern oder dem Unterstrich `_` deren erstes Zeichen ein Buchstabe oder der Unterstrich ist.

Beispiele: **`kaese_menge, kaeseMenge, __1a__`**

Bei der Wahl eines Bezeichners muss darauf geachtet werden, dass er nicht mit einem der 63 reservierten Schlüsselwörter *(keywords)* zusammenfällt.

asm	auto	bool	break	case	catch	char
class	const	const_cast	continue	default	delete	do
double	dynamic_cast	else	enum	explicit	export	extern
false	float	for	friend	goto	if	inline
int	long	mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch	template	this
throw	true	try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile	wchar_t	while

Jede Speicherzelle von z.B. 32 Bits kann unterschiedliche Daten speichern wie:

- Ganzzahl von -2^{31} bis $2^{31}-1$
- Ganzzahl von 0 bis $2^{32}-1$
- Gleitkomma-Zahl
- 4 Zeichen (Buchstaben, Ziffern, Spezialzeichen) nach einer Codierungstabelle wie ASCII oder EBCDIC
- etc.

Die Speicherzelle selber enthält keine Angabe über den *Datentyp*. Damit deren Inhalt trotzdem korrekt interpretiert werden kann, muss der Variable ein Datentyp fest zugeordnet werden, was durch die Deklaration geschieht.

Einfache Datentypen

Die *einfachen Datentypen* dienen zur Beschreibung von

- ganzen Zahlen mit oder ohne Vorzeichen
- reellen Zahlen mit oder ohne Vorzeichen
- Buchstaben, Ziffern und anderen Textsymbolen
- Wahrheitswerten
- Aufzählungen

Auf den einfachen bauen die *abgeleiteten Datentypen* wie Feld, Verbund, Zeiger und Klasse auf, die wir später kennenlernen werden.

Ganzzahlen

short, **int** und **long** sind die Datentypen der ganzen Zahlen *(integers).*

Auf vielen Rechnern ist **short** 16, **int** 32 und **long** 32 oder 64 Bit lang.

Es gilt: **int** ist mindestens so lang wie **short** und höchstens so lang wie **long**.

Der Sinn der drei Datentypen ist die Optimierung des Speicherplatzes (**short**), der Rechenzeit (**int**), resp. des Wertebereiches (**long**).

Es gibt dazu auch die vorzeichenlosen Varianten **unsigned short**, **unsigned long** und **unsigned int** . Diese haben einen Wertebereich von 0 bis 2^N-1 (bei N Bit langen Zahlen).

Für vorzeichenbehaftete Zahlen (**`short`**, **`long`** und **`int`**) gibt es zwei Codierungen, das *Einer-* und das *Zweierkomplement:*

- Das erste Bit von links gibt das *Vorzeichen* an, 0 bedeutet positiv, 1 bedeutet negativ.
- Den *Betrag* einer negativen Zahl erhält man durch Wechseln jedes Bits. Im Falle des Zweierkomplements wird zum Ergebnis noch 1 addiert.

Es wird praktisch nur das Zweierkomplement verwendet. Seine Vorteile sind:

- Die Null hat nicht zwei verschiedene Darstellungen.
- Das Vorzeichen muss nicht speziell behandelt werden, man kann modulo 2^N rechnen!

Beispiel: Addition (–1) + 1 auf einem 8-Bit Rechner im Zweierkomplement.

	Zahl	Codierung
Erster Operand	–1	**11111111**
Zweiter Operand	1	**00000001**
Summe		**00000000**
Übertrag (ignoriert)		**1**
Ergebnis	0	**00000000**

Der Wertebereich beim Zweierkomplement ist **10...0** bis **01...1**, also -2^{N-1} bis $2^{N-1}-1$.

Die *Grenzen* der Wertebereiche werden manchmal im Programm gebraucht, z.B. zur Vermeidung von Fehlern durch Überlauf. Dann ist es besser statt der Zahlen die *symbolischen Konstanten*

```
SHRT_MIN,  SHRT_MAX,  USHRT_MAX,
INT_MIN,   INT_MAX,   UINT_MAX,
LONG_MIN,  LONG_MAX,  ULONG_MAX
```

zu verwenden. Man macht sie verfügbar durch

```
#include <climits>
```

Symbolische Konstanten machen Programme

- lesbarer, weil sie die Bedeutung einer Zahl angeben,
- universeller, weil man ihre Definition bei Bedarf ändern kann.

Gleitkommazahlen

`float`, **`double`** und **`long double`** sind die Datentypen der Gleitkomma-Zahlen *(floating point numbers)*.

Im Dezimalsystem bedeutet Gleitkommazahl:
Vorzeichen mal Mantisse mal Zehnerpotenz.

Die Mantisse hat genau eine Stelle vor dem Komma.
Diese ist ungleich Null, ausser für Null selbst.

Beispiel: $0.00000000000000000000000000000911$

- in Gleitkomma-Darstellung: $9.11 \cdot 10^{-31}$
- in C++ Schreibweise:

```
float electronMass;
electronMass = 9.11e-31;
```

Im Binärsystem bedeutet Gleitkommazahl entsprechend: Vorzeichen mal Mantisse mal Zweierpotenz.

Die Mantisse hat wiederum genau eine Stelle vor dem Komma. Diese Vorkommastelle wird aber bei der Speicherung weggespart, weil sie ja (ausser bei der Zahl Null) immer eine Eins ist.

Heutige Rechner arbeiten nach dem IEEE-Standard:

Anzahl Bits	`float`	`double`
Vorzeichen	1	1
Exponent	8	11
Mantisse	23	52

Der 128 Bit Typ `long double` ist nicht standardisiert.

Der Exponent ist auf spezielle Weise codiert *(biased)* so dass er nie lauter 0-Bits und nie lauter 1-Bits enthält. Beim Datentyp **`float`** sind z.B. Exponenten von -126 bis +127 möglich.

Die Zahl Null wird durch lauter 0-Bits dargestellt.

Nicht jedes Bitmuster entspricht einer gültigen Gleitkommazahl. Ungültige Zahlen werden als "NaN" *(not a number)* ausgegeben.

Die Datentypen **`float`** und **`double`** enthalten auch noch die Unendlich-Werte **`-inf`** und **`+inf`** (zwei Bitmuster mit lauter 1-Bits im Exponenten).

Zeichen

`char` ist der Datentyp der Textzeichen *(characters)* wie Buchstaben, Ziffern, Interpunktionszeichen, Leerzeichen, etc.

Die Zeichen werden nach einer Tabelle wie ASCII *(American Standard Code for Information Interchange)* oder EBCDIC in eine 8 Bit lange Zahl codiert.

`char` ist deshalb ein weiterer Ganzzahl-Datentyp. Es existieren sogar die Varianten **`unsigned char`** (0 bis 255) und **`signed char`** (-128 bis 127), die man verwendet, wenn wirklich Zahlen und nicht Zeichen gemeint sind.

ASCII Zeichensatz (erste 128 von 256 Zeichen):

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	NP	CR	SO	SI
1_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2_	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Nur die erste Hälfte der ASCII-Tabelle ist standardisiert (7-Bit-Zeichensatz). Die zweite Hälfte enthält unter anderem Buchstaben mit Umlauten und Akzenten.

Umlaute und Akzente sind in Bezeichnern nicht erlaubt, wohl aber bei **`char`**-Konstanten sowie innerhalb von Strings (Beispiele folgen).

Nebenbei: Für andere Alphabete bietet C++ einen Typ **`wchar_t`** *(wide character type)* . Der verwendete Code, „Unicode UTF-16“ hat 16 Bits und bis jetzt wurden darin rund 38000 Zeichen definiert. Die **`iostream`**-Operationen wie **`cout <<`** etc. funktionieren aber (noch?) nicht mit **`wchar_t`**.

Einige Beispiele:

```
char c;
c = 'a';
c = 97;       // gleichwertig in ASCII
c = 0x61;     // dito (0x heisst hexadezimal)
c = 'ä';      // erlaubt, aber nicht Standard
cout << ”Käse”;   // dito
c = 'i' + 1; // wie c = 'j', nicht für EBCDIC
c = 9;        // Tabulator in ASCII
c = '\t';     // Tabulator (besser!)
c = 0;        // wie c = '\0', Nullzeichen
c = '0';      // Ziffer 0, wie c = 48 in ASCII
```

Einige wichtige Spezialzeichen mit Ersatzdarstellung *(escape sequence)* in C++:

Beschreibung	Zeichen	ASCII	escape seq.
Zeilenende	NL	**0x0a**	**\n**
Wagenrücklauf	CR	**0x0d**	**\r**
Horizontaler Tabulator	HT	**0x09**	**\t**
Vertikaler Tabulator	VT	**0x0b**	**\v**
Alarm	BEL	**0x07**	**\a**
Backspace	BS	**0x08**	**\b**
Seitenvorschub	NP	**0x0c**	**\f**
Nullzeichen	NUL	**0x00**	**\0**
einfache Anführungszeichen	'	**0x27**	**\'**
doppelte Anführungszeichen	"	**0x22**	**\"**
Backslash	\	**0x5c**	****
Fragezeichen	?	**0x3f**	**\?**

Wahrheitswerte

`bool` ist der Datentyp der Wahrheitswerte (*Boolean,* benannt nach George Boole, 1815–1864).
Er besteht aus den zwei Werten **`false`** und **`true`**.

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

NOT	
F	T
T	F

Typkonversionen

Alle bisher behandelten Datentypen (und einige weitere, aber nicht alle) können gegenseitig zugewiesen werden. Dabei wird automatisch konvertiert:

```
double d;  d = 'a';
int pi;  pi = 3.14159;
unsigned short s;  s = -1;
```

Einige Compiler warnen in Fällen wo Werte durch die Konversion verändert werden. Um Warnungen zu vermeiden, kann man auch explizit konvertieren *(cast)* :

```
int pi;  pi = (int)3.14159;
unsigned short s; s = (unsigned short)-1;
```

Beim Zuweisen eines Wahrheitswertes an eine Zahl gilt die Konversion:

$$\texttt{false} \rightarrow 0$$
$$\texttt{true} \rightarrow 1$$

Beispiel: `int i; i = true;` weist die Zahl 1 zu.

Bei Zuweisung in die andere Richtung gilt:

$$0 \rightarrow \texttt{false}$$
$$\neq 0 \rightarrow \texttt{true}$$

Beispiel: `bool b; b = -2;` weist den Wert `true` zu.

Nicht nur bei der Zuweisung, auch bei Operationen werden Datentypen konvertiert *(propagation)*.
Für *arithmetische* Operationen gibt es 10 (!) Regeln, die man in etwa so zusammenfassen kann:

- Wenn keiner der Operanden einem der untenstehenden Datentypen angehört, werden beide Operanden in **`int`** konvertiert.

- Andernfalls werden beide Operanden in den Typ konvertiert, der in der Sequenz

  `unsigned int` → **`long`** → **`unsigned long`** → **`float`** → **`double`** → **`long double`**

  später folgt.

Aufzähltypen

Mittels **enum** können Datentypen durch explizites Aufzählen *(enumeration)* ihrer Wertebereiche erzeugt werden. Ein Beispiel ist:

```
enum wochentag { sonntag, montag, dienstag,
  mittwoch, donnerstag, freitag, samstag };

wochentag heute, morgen; // Deklaration
heute = dienstag;        // Zuweisung
morgen = heute + 1;      // leider verboten
```

enum Datentypen können, automatisch oder mittels *cast*, in Ganzzahltypen konvertiert werden, aber nicht umgekehrt.

Ausdrücke

Bei Zuweisungen haben wir auf der rechten Seite bis jetzt angetroffen:

- Konstanten wie **`1.23e-45`**, **`0xffff`**, **`'#'`**,
- Variablen wie **`i`** oder **`a1`**
- arithmetische Ausdrücke wie **`a+1`** oder **`x*x`**

Alles was auf der rechten Seite einer Zuweisung stehen darf, wird in C++ als Ausdruck *(expression)* bezeichnet.

Ein Ausdruck ist entweder

- eine Konstante
- eine Variable
- aufgebaut aus einem Operator und Operanden, oder
- eine Folge `(` `Ausdruck` `)`.

Punkt 4 bedeutet einen Ausdruck zwischen runden Klammern `()`. Wir haben es also mit einer *rekursiven* Definition zu tun.

Die Operatoren von C++ sind in der folgenden Tabelle vollständig aufgeführt.

Die Operanden sind selber wiederum Ausdrücke. (Hier ist nochmals eine Rekursion).

Operatoren und Operanden

Operatoren der Prioritätsstufen 1 bis 13:

Priorität	Assoz.	Operator	Operanden	Bedeutung
13	L	* / %	2	multiplikative Operatoren
12	L	+ -	2	additive Operatoren
11	L	>> <<	2	Shift-Operatoren
10	L	< <= > >=	2	relationale Operatoren
9	L	== !=	2	Gleichheitsoperatoren
8	L	&	2	bitweises UND
7	L	|	2	bitweises ODER
6	L	^	2	bitweises Exklusiv-ODER
5	L	&&	2	logisches UND
4	L	||	2	logisches ODER
3	L	?:	3	Konditional-Operator
2	R	= *= /= += -= >>= <<= &= ^= |=	2	Zuweisungs-Operatoren
1	L	,	2	Kommaoperator

Operatoren der Prioritätsstufen 14 bis 17:

Priorität	Assoz.	Operator	Operanden	Bedeutung
17	R	`::`	1	Globaler Geltungsbereich
	L	`::`	2	Klassen-Geltungsbereich
16	L	`->  .`	2	Elementauswahl
	L	`[]`	2	Indexoperator
	L	`()`	2	Funktionsaufruf
	L	`sizeof`	1	Grösse
	R	`dynamic_cast const_cast reinterpret_cast static_cast typeid`	2	Typkonversion
15	R	`++  --`	1	Inkrement, Dekrement
	R	`~`	1	bitweise Negation
	R	`!`	1	logische Negation
	R	`+  -`	1	Vorzeichen
	R	`*  &`	1	Inhalts-, Adressoperator
	R	`()`	2	Typkonversion (cast)
	R	`new  delete`	1	dynamische Speicherverwaltung
14	L	`->*  .*`	2	Dereferenzierung

Anzahl und Anordnung der Operanden müssen der vom jeweiligen Operator verlangten Syntax entsprechen.

Die 1-stelligen Operatoren verlangen Präfix-Notation.

Eine Ausnahme bilden die Operatoren `++` und `--` welche sowohl als Präfix- und Postfixoperatoren existieren.

Beispiele mit Ausdrücken `i++` und `++i`:

```
j = i++;  // gleichwertig mit j=i; i=i+1;
j = ++i;  // gleichwertig mit i=i+1; j=i;
i++;   // oder ++i; gleichwertig mit i=i+1;
```

Die meisten 2-stelligen Operatoren verlangen Infix-Notation.

Ausnahmen bilden beispielsweise:

- der Index-Operator (um in C++ Ausdrücke wie a_i zu bilden), z.B. `a[i]`
- der Funktionsaufruf, z.B. `f()`oder `f(j)` oder (kombiniert mit dem Komma-Operator) `f(x,0,i+1)`
- die Typkonversion, z.B. `(float)i`

Priorität

Jedem Operator ist eine Priorität *(precedence)* zugeordnet, die dafür sorgt dass beispielsweise `a + b * c` als `a + (b * c)` interpretiert wird.

Diese "Bindungsstärke" nimmt (wie in der Mathematik üblich) ab in der Reihenfolge:

- Multiplikative Operatoren `*`, `/` und `%` (Divisionsrest)
- Additive Operatoren `+` und `-`
- Vergleichsoperatoren `<`, `>`, `<=`, `>=`, `==` (Gleichheit) und `!=` (Ungleichheit)
- logisches UND `&&`
- logisches ODER `||`

Korrektes Beispiel:

```
i*i + j*j >= r2 || i==0 && j==0
```

“Beliebter” Fehler (1):

```
bool i_ist_ok;
i_ist_ok = (0 < i < 10);
```

“Beliebter” Fehler (2):

```
bool i_ist_hundert;
i_ist_hundert = (i = 100);
```

Frage: Wieso resultiert beide Male immer **true**, unabhängig von **i**?

Beispiel mit dem 3-stelligen **?:** Operator

```
cout << "100 modulo 17 = ";
int m;  cin >> m;
cout << (100%17 == m ? "richtig" : "falsch");
```

Wie später erläutert wird ist **cout** vom Typ **ostream**, einer Klasse mit **<<** als eigenem Operator. Dieser bindet stärker als **?:** was Klammern nötig macht.

Dagegen ist die Priorität der drei Operatoren **%** , **==** und **?:** absteigend (13, 9, 3) wie im Ausdruck beabsichtigt. Weitere Klammern sind also nicht nötig.

Assoziativität

Neben der Priorität besitzen die Operatoren noch eine Assoziativität. Diese regelt die Reihenfolge wenn der gleiche Operator zwei mal nacheinander auftritt wie beispielsweise in `x - y - z`. Der Minus-Operator ist linksassoziativ, deshalb wird `(x - y) - z` ausgewertet.

Dagegen ist der Zuweisungsoperator rechtsassoziativ, was beispielsweise `x = xStart = 0.5;` erlaubt.

Die Assoziativität spielt sogar bei den Operationen `+` und `*` eine Rolle, denn für die Gleitkommaarithmetik gilt (streng genommen) weder das Assoziativ- noch das Kommutativgesetz!

Beispiel (numerische Auslöschung):

assoc.cpp

```
#include <iostream>
int main()
{
  float x;  x = -10000.;
  float y;  y =  10001.;
  float z;  z =      0.0001;
  cout << (x +  y)+ z  << endl;
  cout <<  x + (y + z) << endl;
}
```

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	$p \vee q = q \vee p$ $p \wedge q = q \wedge p$	**p || q** **p && q**
Assoziativität	$(p \vee q) \vee r = p \vee (q \vee r)$ $(p \wedge q) \wedge r = p \wedge (q \wedge r)$	**p || q || r** **p && q && r**
Distributivität	$(p \vee q) \wedge r = (p \wedge r) \vee (q \wedge r)$ $(p \wedge q) \vee r = (p \vee r) \wedge (q \vee r)$	**(p || q) && r** **p && q || r**
De Morgan-Gesetze	$\neg(p \vee q) = (\neg q) \wedge (\neg p)$ $\neg(p \wedge q) = (\neg q) \vee (\neg p)$	**!p && !q** **!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:

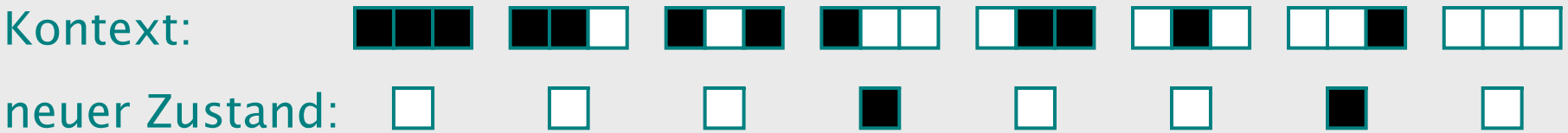
```
float x[3], y[3];
y = x;                  // Geht nicht
y = {0., 0., 10.};      // Geht auch nicht
```

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

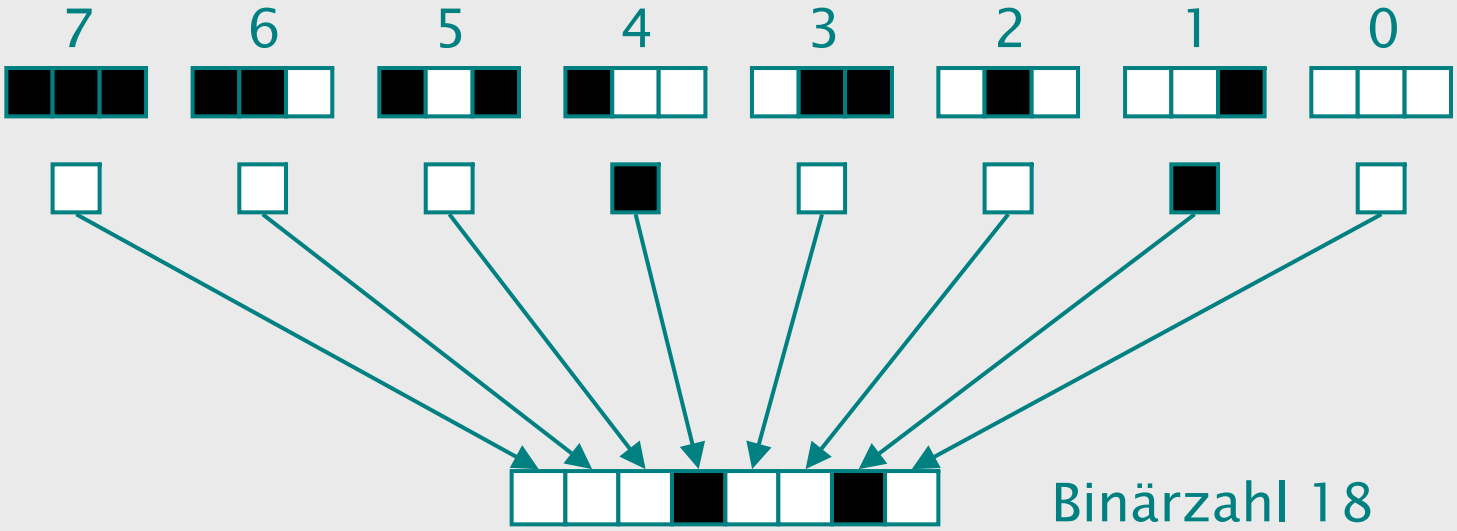
Ein eindimensionaler zellulärer Automat besteht aus:

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

- einer *Übergangstabelle*, beispielsweise:



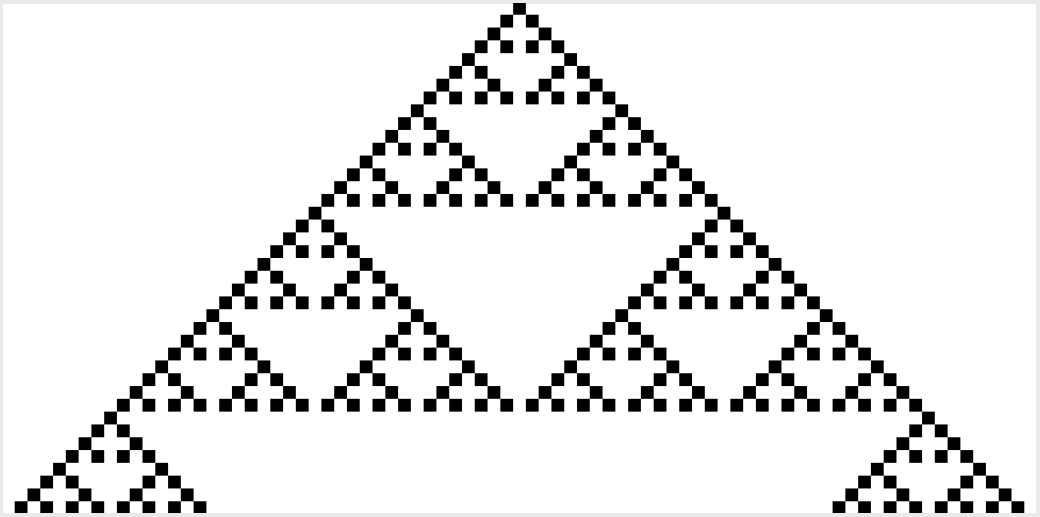
Für die Wahl der Übergangstabelle gibt es 256 Möglichkeiten. Numerierung:



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()
{
```

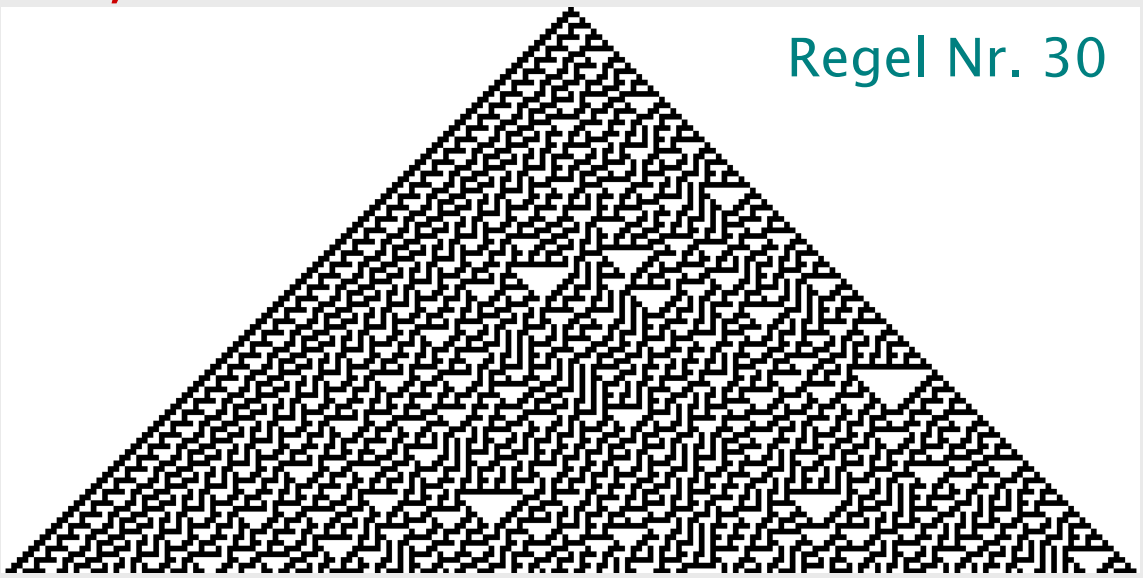
cellular.cpp

```
            // 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];
}
```

Natürlich ist die Lösung mit der Maximallänge nicht elegant:

- Es wird möglicherweise nicht benötigter Speicherplatz reserviert.
- Es besteht die Gefahr dass der Indexbereich überschritten wird. Man kann zwar (und soll auch!) auf `n > nMax` testen und damit ungültige Speicher-zugriffe vermeiden. Aber es besteht keine Möglichkeit, den Array nachträglich zu redimensionieren.

Bessere Lösungen ermöglicht die später behandelte *dynamische Speicherverwaltung*.

Variablen mit mehreren Indices sind ebenfalls möglich. In der Mathematik wären dies Matrizen oder Tensoren.

Eine Rechteckmatrix mit 2 Zeilen und 3 Spalten wird deklariert als

```
double a[2][3];
```

Eine Initialisierung ist beispielsweise

```
double a[2][3] = { {11,12,13},
                   {21,22,23} };
```

und entspricht den Zuweisungen

```
a[0][0] = 11; a[0][1] = 12; a[0][2] = 13;
a[1][0] = 21; a[1][1] = 22; a[1][2] = 23;
```

Beispiel Matrix-Multiplikation. Wenn A eine $N_i \times N_k$ Matrix und B eine $N_k \times N_j$ Matrix ist, dann ist das Produkt C=AB eine $N_i \times N_j$ Matrix mit Koeffizienten

$$c_{ij} = \sum_{k=0}^{N_k} a_{ik} b_{kj}$$

```
for (int i = 0; i < Ni; i++) {
  for (int j = 0; j < Nj; j++) {
    c[i][j] = 0;
    for (int k = 0; k < Nk; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

Das Programm zur Matrix-Multiplikation kann auf Boole'sche Matrizen angewandt werden, wobei dann **+** durch **||** und ***** durch **&&** ersetzt werden muss.

Mit einer quadratischen Boole'schen Matrix

```
bool R[N][N];
```

kann eine *Relation* ausgedrückt werden.

Beispielsweise kann R(i,j) bedeuten: "Es gibt eine direkte Zugverbindung zwischen i und j ".

Das Quadrat R·R bedeutet dann: "Es gibt eine Zugverbindung mit einmaligem Umsteigen zwischen i und j ".

Eindimensionale Arrays über dem Datentyp **char** ergeben die *Strings* der Sprache C, die aber auch in C++ verwendbar sind.

C-Strings enden mit einem Nullzeichen, darum sind folgende zwei Initialisierungen äquivalent:

```
char str[10] = "Summe";
char str[10] = {'S','u','m','m','e',0};
```

Mit dem Nullzeichen kann der String begrenzt oder "geleert" weden. Beispiele:

```
str[3] = 0; cout << str; // Druckt: Sum
str[0] = 0; cout << str; // leerer String
```

Funktionen für Länge, lexikographisches Vergleichen, Konkatenation (Zusammenfügen) etc. werden mit

```
#include <cstring>
```

deklariert.

Alternativ dazu verfügt C++ über Strings, die nicht als Arrays sondern als Objekte einer Klasse **string** realisiert sind. Beispiel:

```
string s = "Hello"; cout << s;
```

Die C++ Strings erfordern

```
#include <string>
```

Es existieren ähnliche Funktionen wie für C-Strings aber auch überladene Operatoren, wie etwa **+** für Konkatenation oder **<** für lexikographischen Vergleich.

Verbund-Datentypen

Der Array ist eine Reihung von Elementen gleichen Typs. Der Verbund *(structure*, Pascal: *record)* ist dagegen eine Zusammenfassung von Elementen (auch: Komponenten) unterschiedlicher Typen.

Wie schon bei den Aufzähltypen gesehen, geht der Variablendeklaration eine Datentypdefinition voraus.

Die Syntax der Typdefinition ist:

```
struct Verbundname  {
    Element-Deklarationen
};
```

Beispiel:

```
struct Datum { int tag, monat, jahr; };

enum Geschlecht { maennlich, weiblich };

struct Person {
  char name[80];
  Datum geburtstag;
  Geschlecht geschlecht;
};
```

Wichtig:

- Die `struct` Deklaration endet immer auf `};`
- In einer `struct` Deklaration darf keine Initialisierung verwendet werden.

Beispiele einer Variablendeklaration:

```
Person a, studierende[278];
```

Beispiele für Zugriffe auf Elemente:

```
strcpy(a.name, "Ada Byron Lovelace");
a.geburtstag.tag   =   10;
a.geburtstag.monat =   12;
a.geburtstag.jahr  = 1815;
a.geschlecht = weiblich;
```

Variablen von Verbundtypen können initialisiert werden:

```
Person b = { "Charles Babbage",
             26, 12, 1791, maennlich };
```

struct.cpp

Welche Vorteile hat das Zusammenfassen zu einem Verbund?

- Der wichtigste Vorteil wird später klar: Zwei Unterprogramme können via eines einzigen Parameters die gesamte im Verbund gespeicherte Information austauschen.
- Der Verbund kann als Ganzes an eine andere Variable zugewiesen werden. Dabei werden alle Elemente kopiert, selbst wenn es sich um weitere Verbundtypen oder sogar um Arrays handelt.

Der Fall der Arrays ist deshalb bemerkenswert, weil Arrays selber nicht zuweisbar sind.

Vereinigungs-Datentypen

Vereinigungstypen *(unions)* haben eine ähnliche Syntax wie die Verbundtypen:

```
union Name {
   Element-Deklarationen
};
```

Im Unterschied zum Verbund werden die Elemente bei der Vereinigung aber im Speicher "übereinander-gelegt". Das heisst es wird pro Variable nur soviel Speicherplatz alloziert (reserviert) wie die grösste der Elemente benötigt.

In Pascal-Terminologie wird von einem "Record mit Varianten" gesprochen.

Vereinigungstypen werden hauptsächlich als Elemente eines Verbundes verwendet. Damit lassen sich Datenstrukturen mit Varianten definieren.

Beispiel:

```
struct Ortsangabe {
  enum typ = {Ortsname, PLZ, Koordinaten};
  union wert {
    char name[16]; int plz; float xy[2];
  };
};
Ortsangabe ort[1000];
```

Beispiel für Zugriff:

```
if (ort[k].typ == Ortsname){
  cout << "in der Ortschaft "
       << ort[k].wert.name << endl;
}
```

Eine weitere Anwendung von Vereinigungstypen besteht darin, den Inhalt einer Speicherzelle auf zwei verschiedene Arten zu interpretieren.

Dies wird selten gebraucht und ist nicht zu empfehlen wegen Maschinenabhängigkeiten.

Beispiel 1: Ein Buchstaben-Code der auch als Zahl benutzt werden kann.

```
union id { char alfa[4]; int numeric; };
```

Die Reihenfolge der Bytes in Ganzzahlen ist aber maschinenabhängig *(big endian* vs. *little endian)*.

Beispiel 2 (reine Spielerei):

- Gesucht: ganzer Teil des 2er-Logarithmus.
- Lösung: eine **`float`**-Zahl hat den Zweier-Exponenten als Teil des Bitmusters. Also kann man einen Typ

```
union Hack { float f; unsigned int u; };
```

verwenden, eine Variable

```
Hack x;
```

deklarieren, dann die Zahl an **`x.f`** zuweisen und schliesslich die Bits mittels der Bit-Operatoren **`&`** und **`>>`** aus **`x.u`** extrahieren. log2.cpp

Nebenbei: Für die Extraktion von Mantisse und Exponent aus Gleitkommazahlen gäbe es die Funktion **`frexp()`** in der **`<cmath>`** Bibliothek.

Unterprogramme

Unterprogramme sind das wichtigste Strukturierungsmittel beim Programmieren. Sie erlauben

- Teilaufgaben zu definieren und diese in je einem Unterprogramm zu behandeln,
- das Programm auf verschiedene Dateien aufzuteilen, die dann separat kompilierbare *Module* bilden,
- Programmbibliotheken zu erstellen (aus einem oder mehreren Modulen),
- Unterprogramme als *„black box“* zu betrachten, die man als korrekt voraussetzt, und von denen man nur die Schnittstelle kennen muss.

Einige Programmiersprachen unterscheiden zwei Arten von Unterprogrammen:

- *Funktionen*, die einen Wert zurückgeben, und
- *Prozeduren* (Pascal) resp. *Subroutinen* (Fortran), die eine Aktion ausführen.

In C und C++ gibt es dagegen nur die Funktionen. Es ist aber zugelassen, dass eine Funktion keinen Wert zurückgibt. Eine solche Funktion hat formal den leeren Rückgabetyp **`void`**.

Standardfunktionen

C++ besitzt eine Bibliothek von mathematischen Funktionen. Mit

```
#include <cmath>
```

integriert man deren Deklarationen ins Programm.

Funktionsdeklarationen haben die Form

```
Typ Funktionsname ( Parameterliste );
```

Dabei ist `Typ` der Datentyp des Rückgabewertes und `Parameterliste` die Liste der Funktionsargumente. Eine leere Liste `()` ist zulässig, mehrere Parameter werden durch Kommata getrennt.

Die Parameterliste kann zwei Formen haben:

- Sie kann nur Datentypen enthalten wie im Beispiel (des Potenzierens):

```
float pow(float, float);
```

  In diesem Fall spricht man von einem *Prototypen* der Funktion. Der Teil `pow(float, float)` heisst die *Signatur* der Funktion.

- Sie kann Datentypen und *formale Parameter* enthalten wie im Beispiel

```
float pow(float basis, float exponent);
```

In C++ darf, anders als in C, der gleiche Funktionsname für mehrere Funktionen gebraucht werden, sofern sich deren Signaturen unterscheiden. So kennt z.B. die Standard-Mathematikbibliothek zusätzlich eine Funktion

```
float pow(float, int);
```

die auch für negative Basen funktioniert.

Die folgende Liste gibt die Prototypen der Funktionen von **<cmath>** wieder. Es existieren aber zusätzlich zu den **float**-Versionen noch entsprechende Versionen für **double** (und möglicherweise für **long double**).

```
float acos (float);           // Arcus cosinus
float asin (float);           // Arcus sinus
float atan (float);           // Arcus tangens
float atan2(float, float);    // atan(y/x) (x bel.)
float ceil (float);           // aufrunden
float cos  (float);           // Cosinus
float cosh (float);           // Cosinus hyperbolicus
float exp  (float);           // Exponentialfunktion
float fabs (float);           // Absolutbetrag
float floor(float);           // abrunden
float frexp(float, int*);     // Mantisse und Exponent
float fmod (float, float);    // Divisionsrest
float log  (float);           // Logarithmus naturalis
float log10(float);           // Zehner-Logarithmus
float pow  (float, float);    // x hoch y (x >= 0)
float pow  (float, int);      // x hoch y (x bel.)
float sin  (float);           // Sinus
float sinh (float);           // Sinus hyperbolicus
float sqrt (float);           // Quadratwurzel
float tan  (float);           // Tangens
float tanh (float);           // Tangens hyperbolicus
```

Funktionen werden benützt indem man sie vom Hauptprogramm oder einem anderen Unterprogramm aus *aufruft*.

Der Funktionsaufruf bildet einen *Ausdruck*. Damit kann er z.B. auf der rechten Seite einer Zuweisung stehen, oder aber Teil eines grösseren Ausdrucks sein.

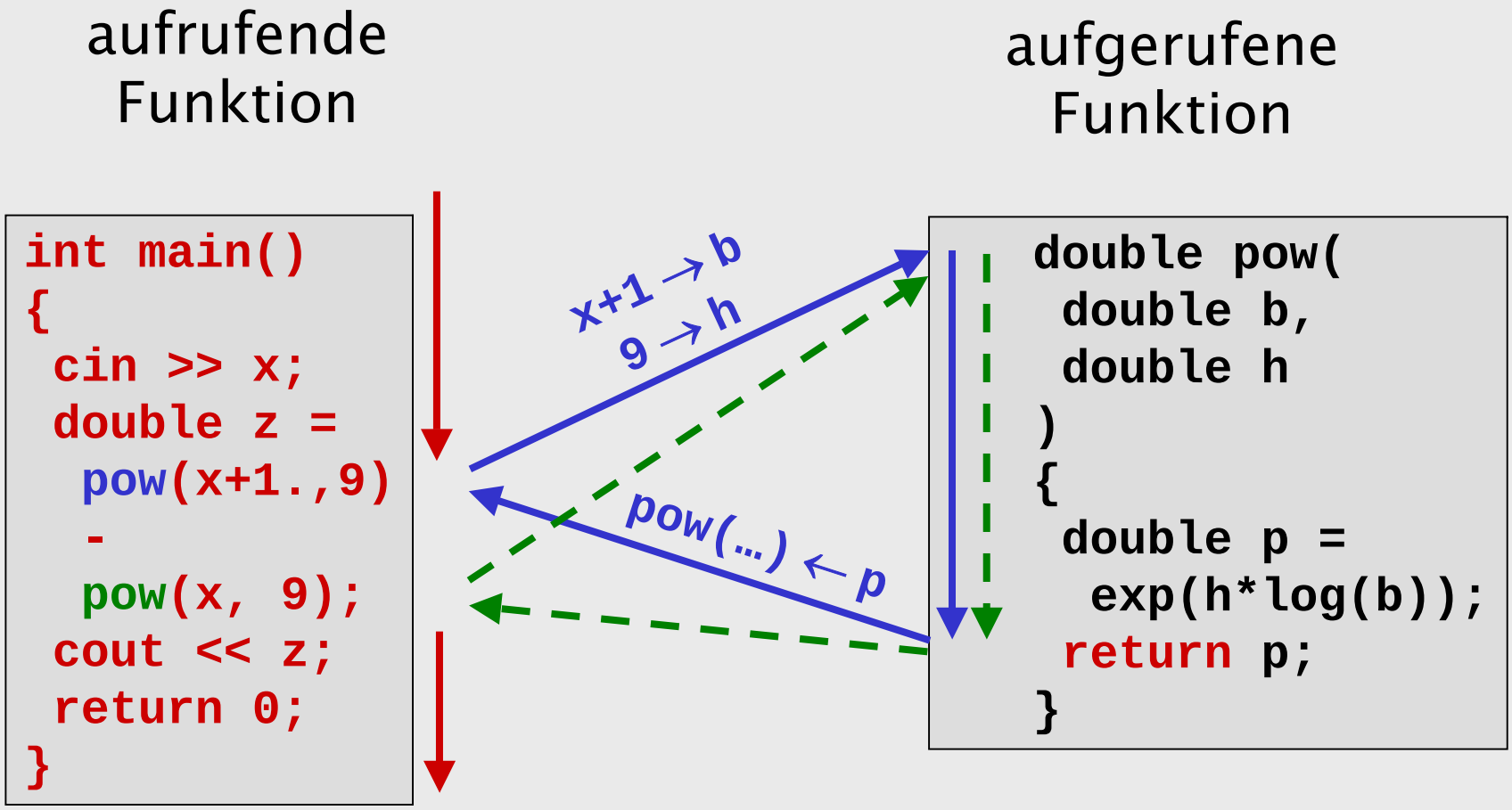
Beim Funktionsaufruf werden sog. *aktuelle Parameter* in die Parameterliste eingesetzt, das sind Ausdrücke deren Datentyp mit dem der formalen Parameter resp. der Signatur übereinstimmt.

Beispiel: `double z = pow(x+1., 9) - pow(x, 9);`

Kann es passieren, dass das Unterprogramm Variablen überschreibt?

Da nur die ausgewerteten aktuellen Parameter an das Unterprogramm weitergegeben werden, besteht diese Gefahr nicht. Das Unterprogramm müsste dazu die Adresse der Variablen kennen.

Beispiel: Der Wert der Variablen `x` kann durch den Aufruf `pow(x+1., 9)` nicht verändert werden, selbst dann nicht wenn die Funktion `pow` fehlerhaft programmiert wäre. Auch mit dem Aufruf `pow(x,9)` kann dies nicht passieren.


aufrufende Funktion
aufgerufene Funktion
int main()
{
cin >> x;
double z =
pow(x+1.,9)
-
pow(x, 9);
cout << z;
return 0;
}
x+1 → b
9 → h
pow(…) ← p
double pow(
double b,
double h
)
{
double p =
exp(h*log(b));
return p;
}

Selber definierte Funktionen

Die *Deklaration* einer Funktion ist die Angabe des Prototyps. Damit wird die Schnittstelle formal beschrieben („wieviele Argumente gibt es und von welchem Typ sind sie? Was ist der Typ des Rückgabewerts?“).

Die *Definition* ist dagegen die vollständige Beschreibung der Funktion. Die Syntax der Funktionsdefinition ist

```
Typ Funktionsname ( Parameterliste )
  Block
```

wobei in der Parameterliste diesmal nicht nur Datentypen sondern auch dazugehörige formale Parameter stehen müssen.

Es gilt zu beachten:

- Funktionen müssen in jedem Modul (d.h. in jeder Datei) in dem sie aufgerufen werden, *vor dem ersten Aufruf* deklariert werden.
- Befindet sich die Definition in einem anderen Modul, so muss der Deklaration das Schlüsselwort **`extern`** voran-gestellt werden.
- Die Definition gilt auch als Deklaration.
- Funktionen dürfen nicht mehrfach definiert werden (auch nicht in verschiedenen Modulen).

Weil eine überflüssige **`extern`**-Deklaration nicht stört, ist es zweckmässig, Funktionsdeklarationen in Header-Dateien auszulagern und diese dann in jedem Modul zu benützen.

Die Anweisung

```
return Ausdruck ;
```

bewirkt das Verlassen der Funktion und die Rückgabe von `Ausdruck` .

Die Anweisung

```
return;
```

bewirkt das Verlassen einer als `void` deklarierten Funktion.

Funktionsdefinitionen können durchaus mehrere `return`-Anweisungen enthalten. Der Block muss mit einer solchen enden (Ausnahme: bei einer Schleife der Art `while(true) { … }`).

Beispiel: Die Berechnung des grössten gemeinsamen Teilers kann jetzt als Funktion geschrieben werden:

```
int ggT(int a, int b)
{
   while (b > 0) {
     int c = a%b;  a = b;  b = c;
   }
   return a;
}
```

Wie man sieht, dürfen die formalen Parameter **a** und **b** verändert werden. Sie verhalten sich wie lokale Variablen, die beim Aufruf einen Wert zugewiesen erhalten.

Die Variablen der aufrufenden Funktion werden nicht tangiert:

```
int m = 14, n = 35;
cout << ggT(m, n) << " ist ggT von ";
cout << m << " und " << n << endl;
```

Variablenparameter

Die bis jetzt behandelten Parameter waren sogenannte *Wertparameter*. Dabei erhält das aufgerufene Unterprogramm eine *Kopie* des aktuellen Parameters.

Bei *Variablenparametern* erhält das Unterprogramm dagegen die *Speicheradresse.* Das Unterprogramm kann deshalb die Variable nicht nur lesen, sondern auch verändern.

Variablenparameter kennzeichnet man durch ein **&** (den *Referenzoperator*) vor dem formalen Parameter.

Eine typische Anwendung von Variablenparameter sind Funktionen die mehrere Werte berechnen sollen.

Beispiel: kartesische in Polarkoordinaten umrechnen:

```
void polar(double x, double y,
           double& rho, double& phi)
{
    rho = sqrt(x*x + y*y);
    phi = atan2(x, y); return;
}
```

Aufruf:

```
double x = 4., y = 3., laenge, winkel;
polar(x, y, laenge, winkel);
cout << "  Laenge:" << laenge <<
        ", Winkel:" << winkel << endl;
```

Weitere Beispiele von Funktionen mit Variablenparametern gibt es in der Bibliothek **`iostream`**.
Eine solche Funktion ist **`cin.get(char& c)`**, die das nächste Zeichen einschliesslich *whitespace* liest.
(Funktionsnamen wie **`cin.get`** beschreiben Elementfunktionen, die erst später behandelt werden).

Die Sprache C kennt keine Variablenparameter.
Darum sind z.B. die Funktionen in der Bibliothek **`cstring`** mit Zeigern programmiert (später behandelt).

Weiteres Beispiel: Eine Funktion **swap** welche die Werte zweier Variablen vertauscht.

Die Parameter sind hier sowohl Eingabe- als auch Ausgabeparameter.

```
void swap(int& i, int& j)
{
    int h = i; i = j; j = h;
}
```

Aufruf:

```
int sechs = 7, sieben = 6;
swap(sechs, sieben);
cout << "sechs:"  << sechs << endl;
cout << "sieben:" << sieben << endl;
```

Arrays als Parameter

Die Regel dass bei Wertparametern Werte kopiert und bei Variablenparametern Adressen übergeben werden, gilt auch für zusammengesetzte Datentypen, z.B. den Verbund.

Der Array macht hier eine scheinbare Ausnahme: Arrays werden bei der Übergabe nicht kopiert.

Der Grund ist: In C/C++ wird ein Array intern so behandelt wie eine Speicheradresse (des nullten Elementes). Beim Unterprogrammaufruf wird deshalb nur diese Adresse kopiert, nicht aber der ganze Array.

Dass Arrays allein durch ihre Anfangsadresse repräsentiert werden, bedeutet auch dass die Arraylänge nirgends gespeichert ist.

Entsprechend ist es möglich, Funktionen zu schreiben, die beliebige Arraylängen zulassen, wie etwa:

```
float median(float folge[]);
```

Die Dimensionierungs-Klammer `[]` wird also einfach leer gelassen. Dadurch sind Aufrufe mit verschieden langen `float`-Arrays möglich.

Die Länge muss der Funktion aber mitgeteilt werden, wozu es mehrere Möglichkeiten gibt :

- fix vereinbart
- separater Parameter
- spezieller Wert als Schlussmarkierung.

Letztere Technik wird von den Funktionen in **<cstring>** verwendet, weil ja bei C-Strings eine solche Schluss-markierung existiert (das Zeichen mit ASCII-Code 0).

Beispiel: Die Funktion **strcpy** zum Kopieren von C-Strings. Sie ist deklariert als

```
char* strcpy(char* dest, const char* src);
```

was (wie wir später sehen werden) gleichwertig ist mit

```
char* strcpy(char dest[], const char src[]);
```

Beim (meist ignorierten) Rückgabewert gibt es keine Alternative zur Zeiger-Schreibweise.

Das Hauptprogramm

Das Hauptprogramm **main** ist eine selber definierbare Funktion. Es sind zwei Prototypen deklariert:

Neben dem Prototyp

```
int main();
```

existiert der Prototyp

```
int main(int argc, char **argv);
```

manchmal auch geschrieben als

```
int main(int argc, char *argv[]);
```

Die Variable **argv** ist ein Array von C-Strings. Beim "Aufruf" von **main** durch das Betriebssystem wird dem Element **argv[i]** das i-te Argument der Kommandozeile übergeben, wobei das Programm selbst als nulltes Argument gezählt wird.

Die Variable **argc** erhält die Anzahl dieser Argumente (inklusive dem nullten).

Um Zahlen von C-Strings nach **int** resp. **double** zu konvertieren, verwendet man die Funktionen:

```
#include <cstdlib>
double atof (const char str[])
int    atoi (const char str[])
```

Der durch `return` [`Ausdruck`] `;` produzierte (ganzzahlige) Rückgabewert kann vom Betriebssystem interpretiert werden, z.B. innerhalb von Scripts.
Als Konvention gilt, dass bei fehlerfreiem Abschluss ein Wert von Null zurückgegeben wird.

Mit `exit (` [`Ausdruck`] `);` kann man das Programm auch aus einem Unterprogramm heraus verlassen.

Inline-Funktionen

Der Aufruf eines Unterprogramms ist mit einem gewissen Aufwand verbunden:

- Die Registerinhalte werden auf den Stack “gerettet”.
- Die aktuellen Parameter und die Rücksprungadresse werden dem Unterprogramm via Stack übergeben.
- Vor dem Rücksprung gibt das Unterprogramm den Rückgabewert auf den Stack.
- Nach dem Rücksprung wird der Stack wieder abgebaut und die alten Registerinhalte werden wieder hergestellt.

Eine *Inline-Funktion* wird dagegen *nicht aufgerufen*.
Der Compiler ersetzt den Aufruf durch den Inhalt der Funktionsdefinition, wobei für die formalen Parameter die aktuellen Parameter(-ausdrücke) eingesetzt werden.

Inline-Funktionen machen (wenn mehrfach benutzt) das kompilierte Programm länger.

Sie bringen dafür einen Zeitgewinn

- wenn es sich um kleine Funktionen handelt, und
- wenn sie in innersten Schleifen gebraucht werden.

Beispiel: Die Inline-Funktion

```
inline double det(double mat[2][2])
{
  return mat[0][0] * mat[1][1] -
         mat[0][1] * mat[1][0];
}
```

kann z.B. so verwendet werden:

```
double produkt = det(a) * det(b);
```

und wird dann übersetzt als:

```
double produkt =
  (a[0][0]*a[1][1] - a[0][1]*a[1][0]) *
  (b[0][0]*b[1][1] - b[0][1]*b[1][0]);
```

det.cpp

Für Inline-Funktionen gilt, im Unterschied zu anderen Funktionen:

- Die Funktion muss definiert (nicht bloss deklariert) werden, bevor sie zum ersten Mal benutzt wird. Die Definition beginnt mit dem Schlüsselwort **`inline`**.

- Die Definition darf wiederholt vorkommen.

Inline-Funktionen, die in mehreren Modulen gebraucht werden, *definiert* man am besten in einer Header-Datei.

Rekursion

Funktionen dürfen andere Funktionen aufrufen, aber auch sich selbst. Man spricht dann von *Rekursion*.

Beispiel:

```
int ggT(int a, int b)
{
   if (b == 0) return a;
   else return ggT(b, a%b);
}
```

recGGT.cpp

Verlauf: `ggT(8,12)` → `ggT(12,8)` → `ggT(8,4)` → `ggT(4,0)` → `4`

Wichtig:

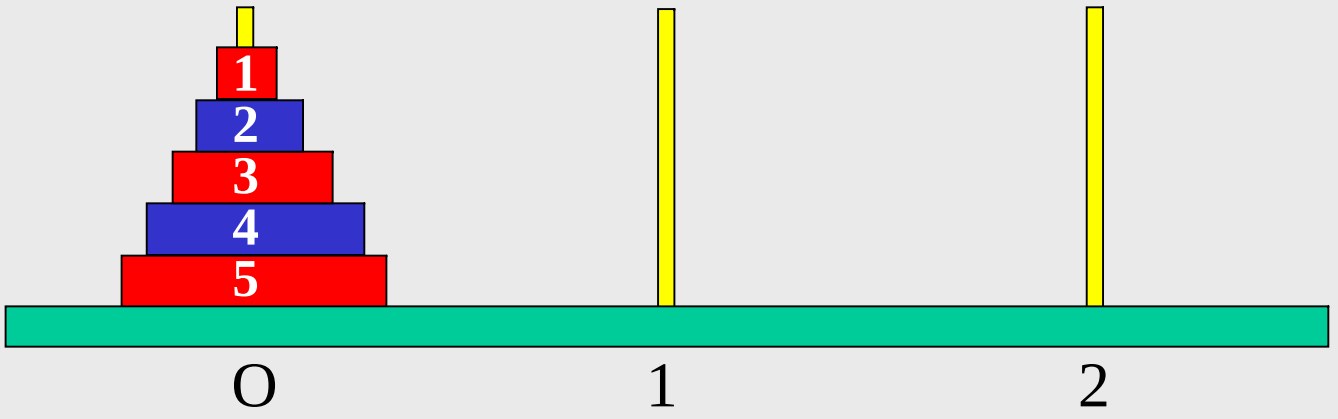
- Es braucht eine *Verankerung:* für bestimmte Parameterwerte bricht die Rekursion ab. Die Verankerung entspricht der Abbruchbedingung bei der **`while`**-Schleife.
- Die Parameter ändern sich bei jedem Aufruf und erreichen nach endlich vielen Malen eine Verankerung. Typischerweise werden die Parameter jedes Mal kleiner.

Die Rekursion kann als Alternative zur *Iteration* (**`while`**-Schleife) verwendet werden. Sie ist gleich mächtig wie diese im Sinne der Berechenbarkeit.

Die Türme von Hanoi

Die Rekursion ist besonders dann geeignet, wenn bereits die Problemstellung rekursiv ist.

Beispiel: Das Problem der Türme von Hanoi: Der Turm aus *n* Scheiben soll vom Platz 0 auf den Platz 2 gebracht werden. Dabei darf immer nur eine Scheibe aufs Mal den Platz wechseln und es darf nie eine grössere Scheibe auf eine kleinere gelegt werden.



Die Lösung kann mit Rekursion sehr einfach formuliert werden.

Man kann nämlich den Turm der Scheiben 1 bis k von Platz x auf Platz y bringen, indem man:

1. den Turm der Scheiben 1 bis $k-1$ (rekursiv) von Platz x auf Platz z bringt (Zwischenplatz, $z = 3-x-y$),
2. die Scheibe k auf Platz y setzt, und
3. den Turm der Scheiben 1 bis $k-1$ (rekursiv) von Platz z auf Platz y bringt.

Mit dem Aufruf `hanoi(n,0,1,2);` erzeugt die folgende Funktion die richtige Sequenz von Scheibenbewegungen:

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

hanoi.cpp

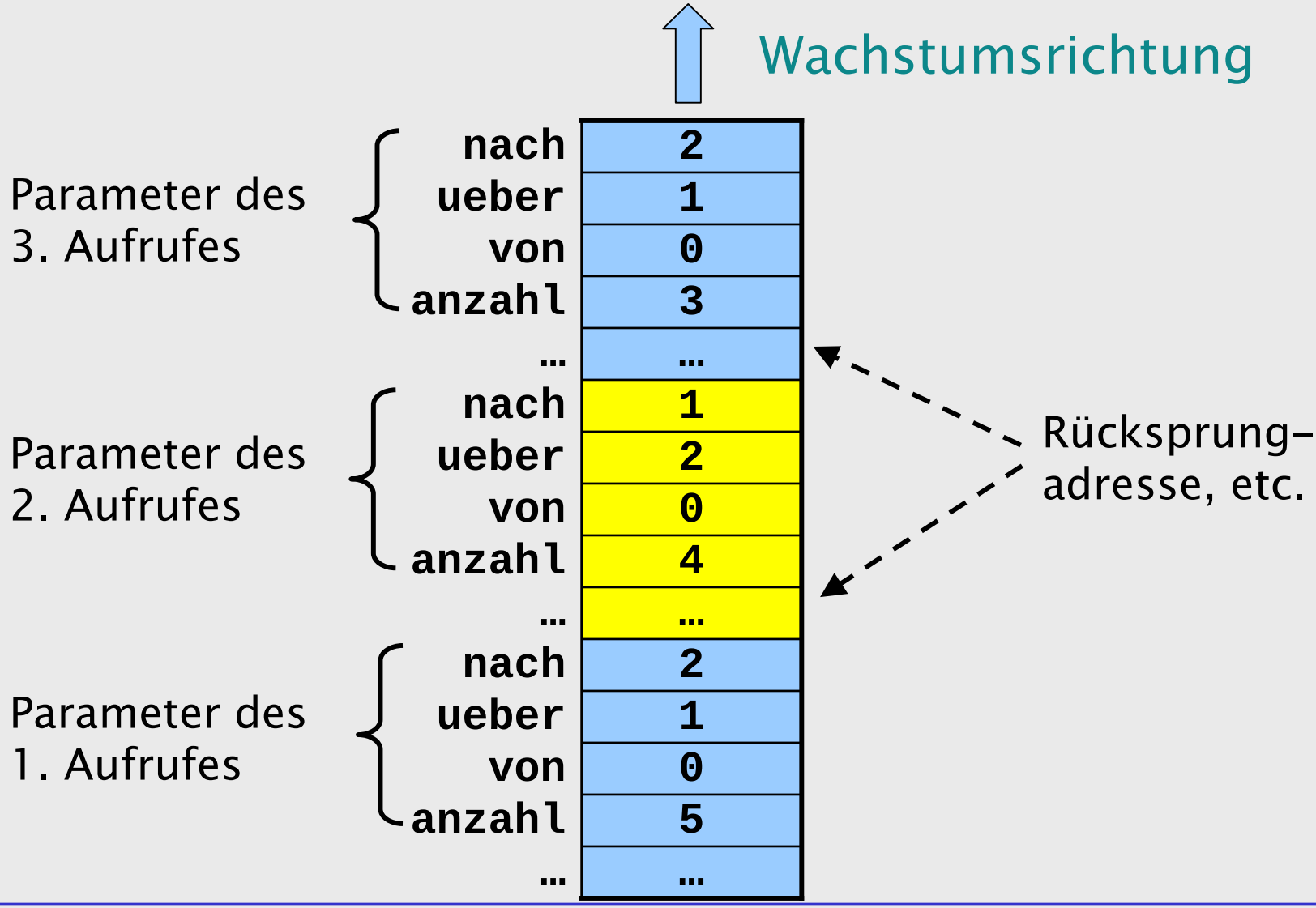
Der Stack und statische Variablen

Interessant ist, dass über den Zustand der drei Türme gar nicht Buch geführt werden muss.

Der Zustand ist aber wenigstens teilweise auf dem Stack gespeichert, denn:

Die Parameter einer Funktion werden auf dem Stack abgelegt. Im Falle von rekursiven Aufrufen gibt es mehrere *Inkarnationen* jedes Parameters.

Bild des Stacks während der Ausführung:



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:

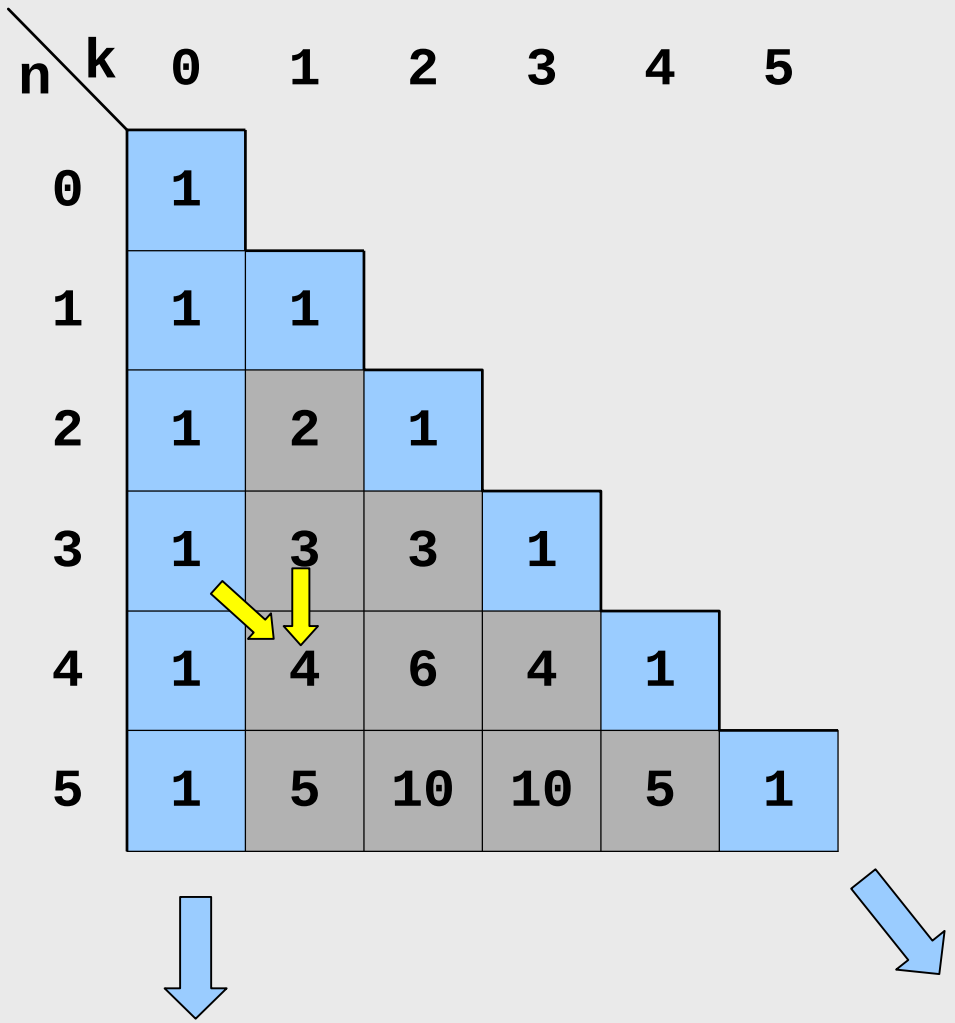
Das Pascal-Dreieck

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)* :

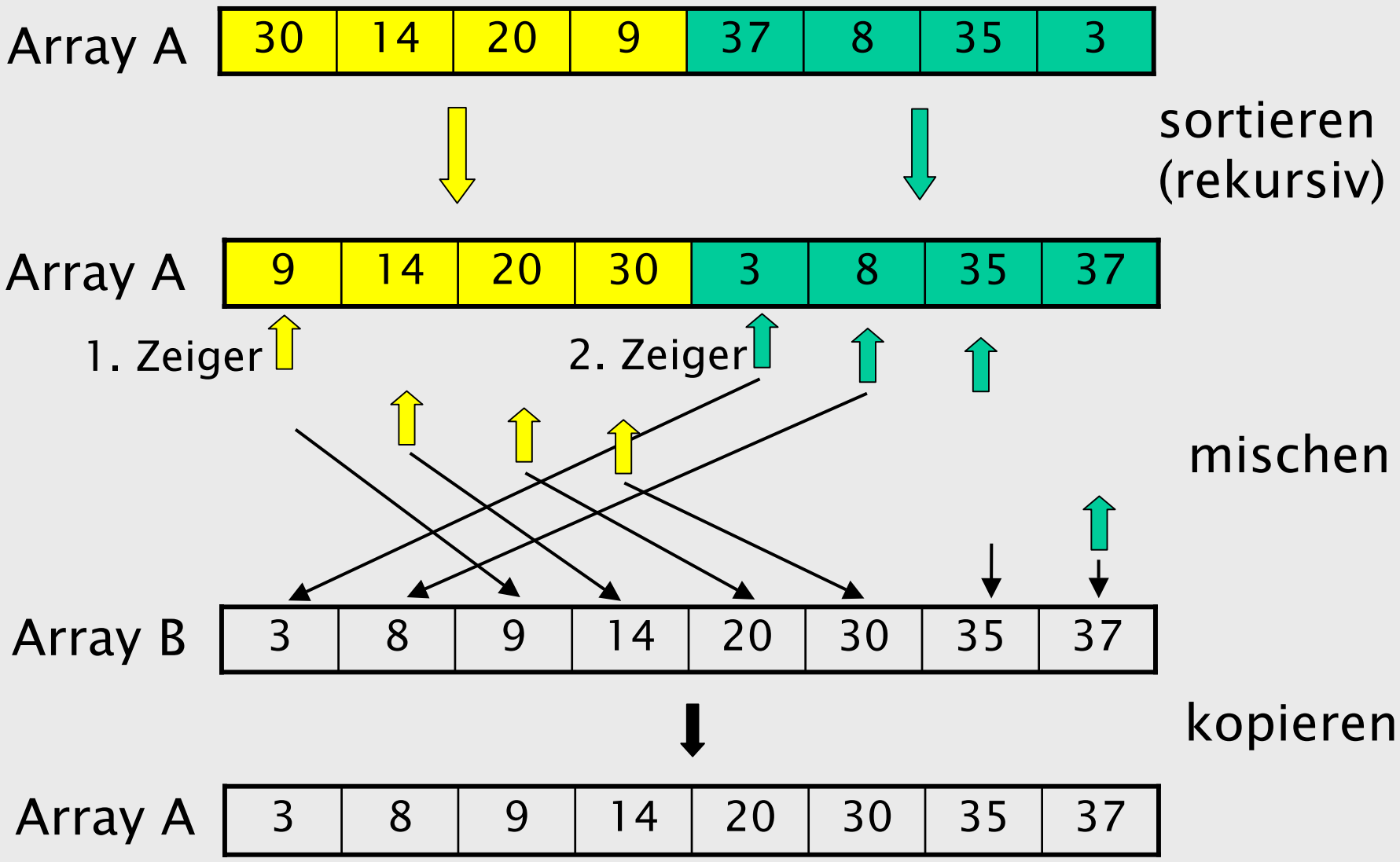
```
int binom(int n, int k)
{
  int b = 1;
  if (k > n/2) k = n-k;
  for (int i = 0; i < k; i++) b = b*(n-i)/(i+1);
  return b;
}
```

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.

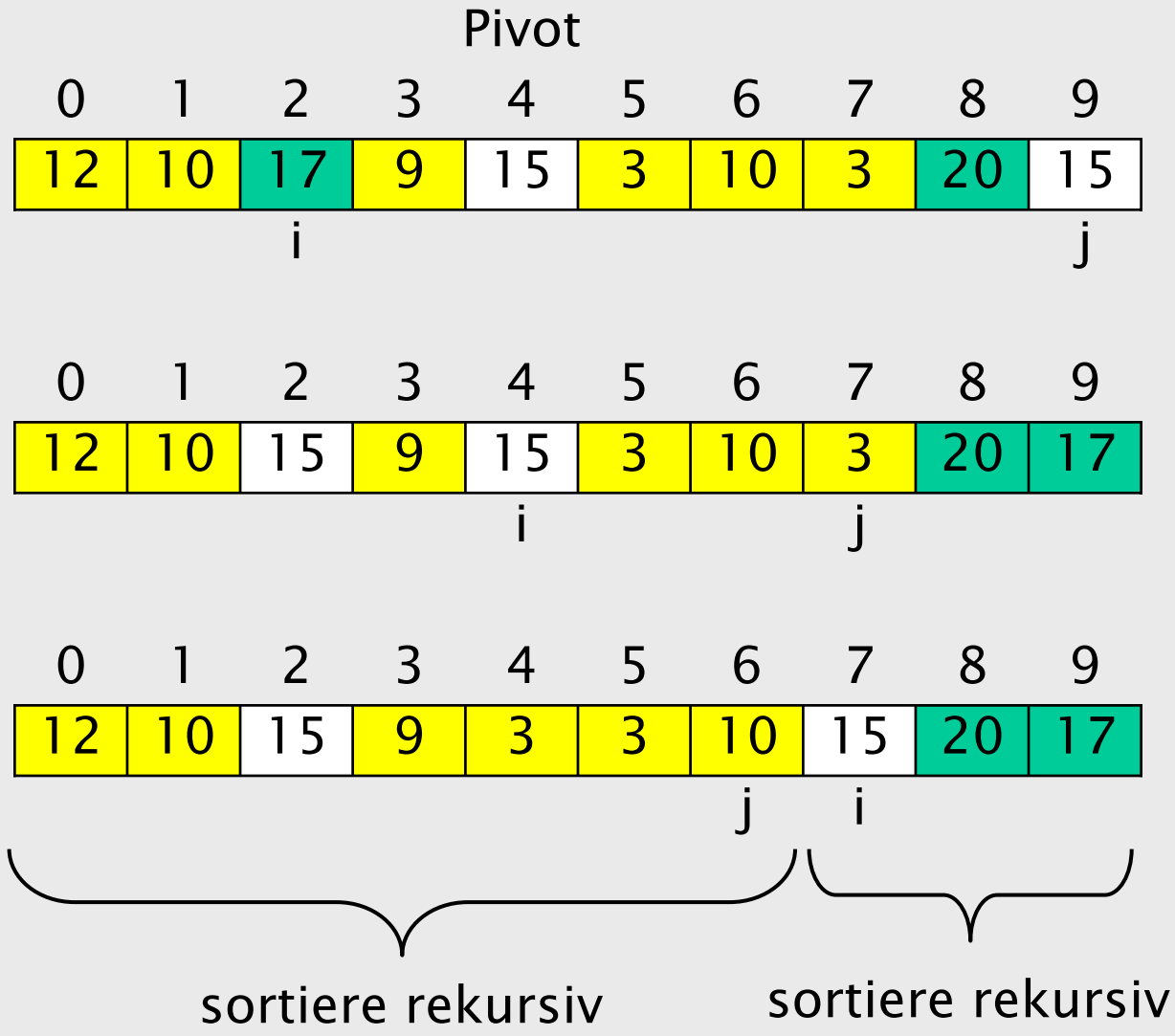
Wichtiges Detail: Beim "Nach-Innen-Verschieben" der Indices i und j wird ein Wert *gleich* dem Pivot nicht akzeptiert:

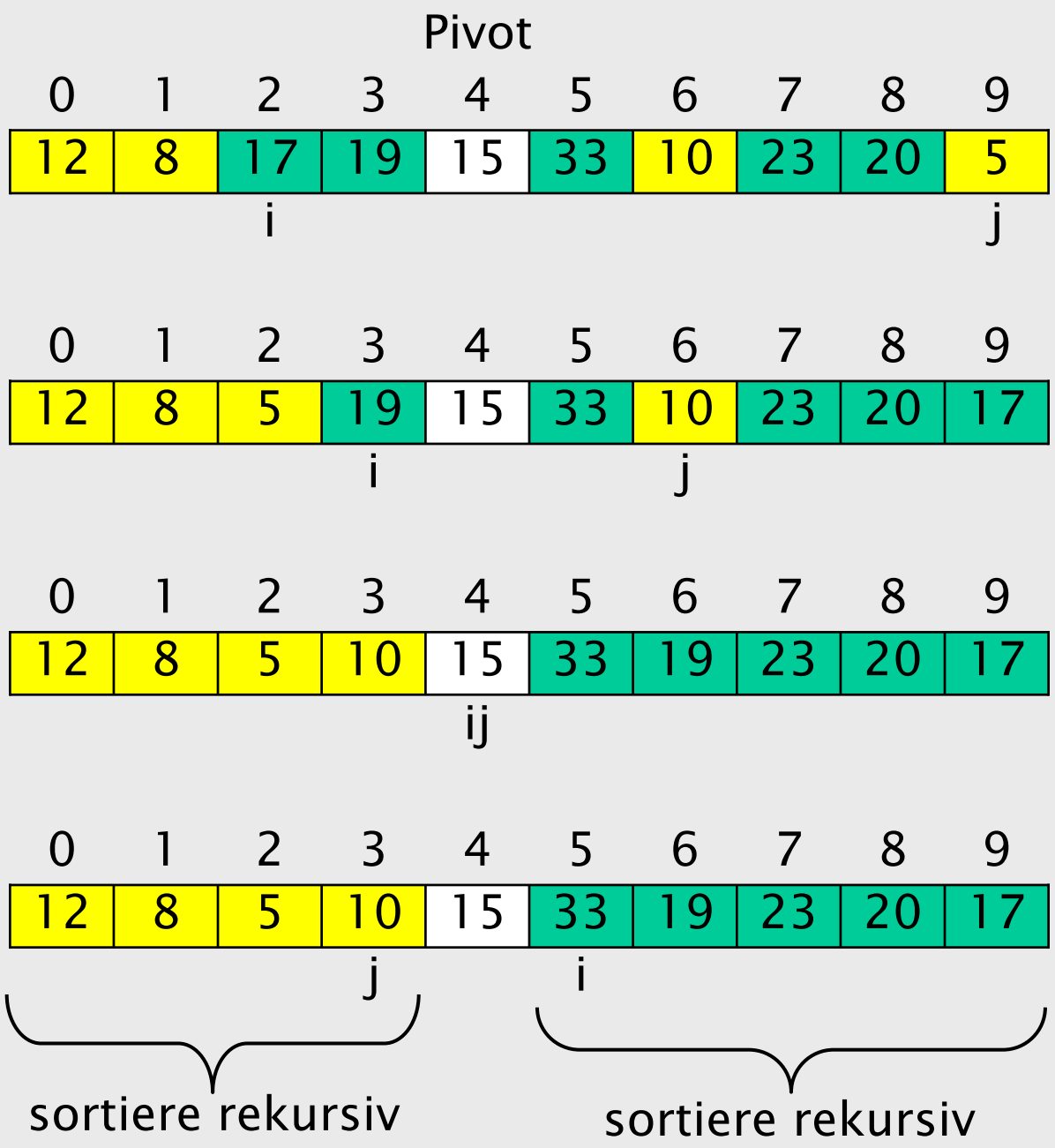
```
while (a[i] < pivot) i++;
while (a[j] > pivot) j--;
```

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

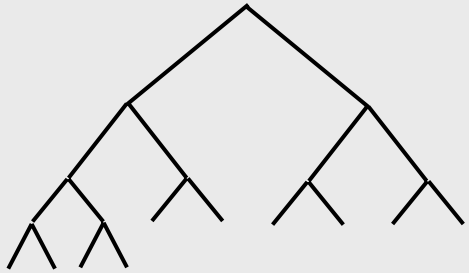
Mit obiger Lösung findet spätestens beim Pivot ein Vertauschen statt, und damit verbunden ein Verschieben *beider* Indices.

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

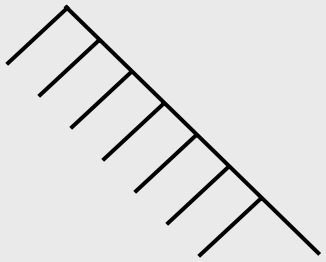
Im Gegensatz zum Quicksort hat Merge-Sort stets einen *balancierten* Rekursionsbaum: die Länge seiner Äste unterscheidet sich maximal um Eins.



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.

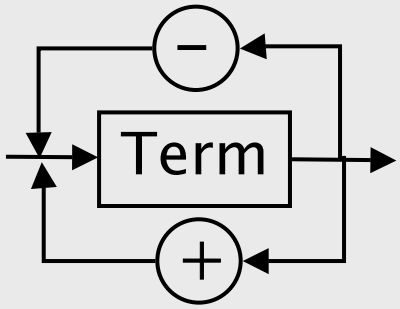
Formale Grammatiken

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

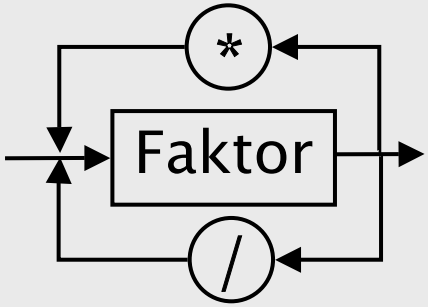
Formale Sprachen sind im allgemeinen rekursiv definiert: In C++ kann z.B. eine Anweisung einen Block enthalten, der wiederum Anweisungen enthält.

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

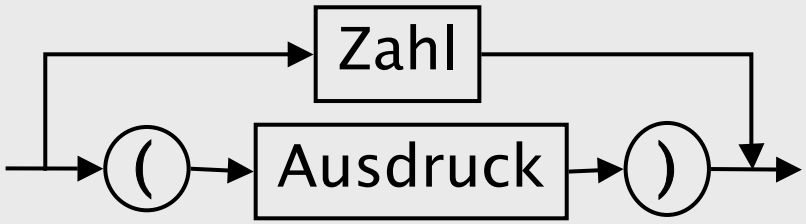
Ausdruck:



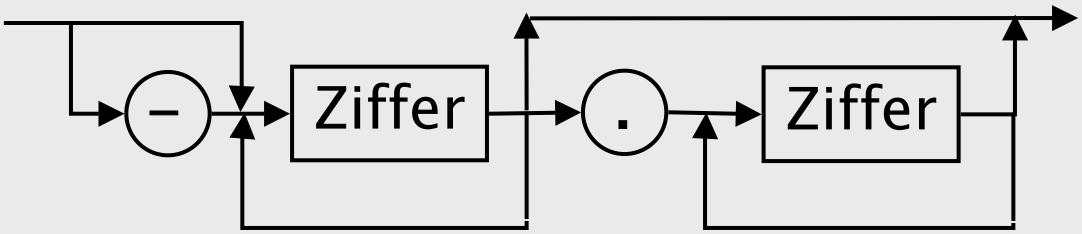
Term:



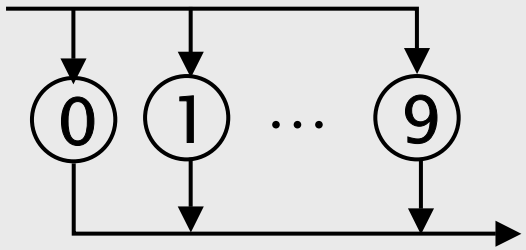
Faktor:



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
```

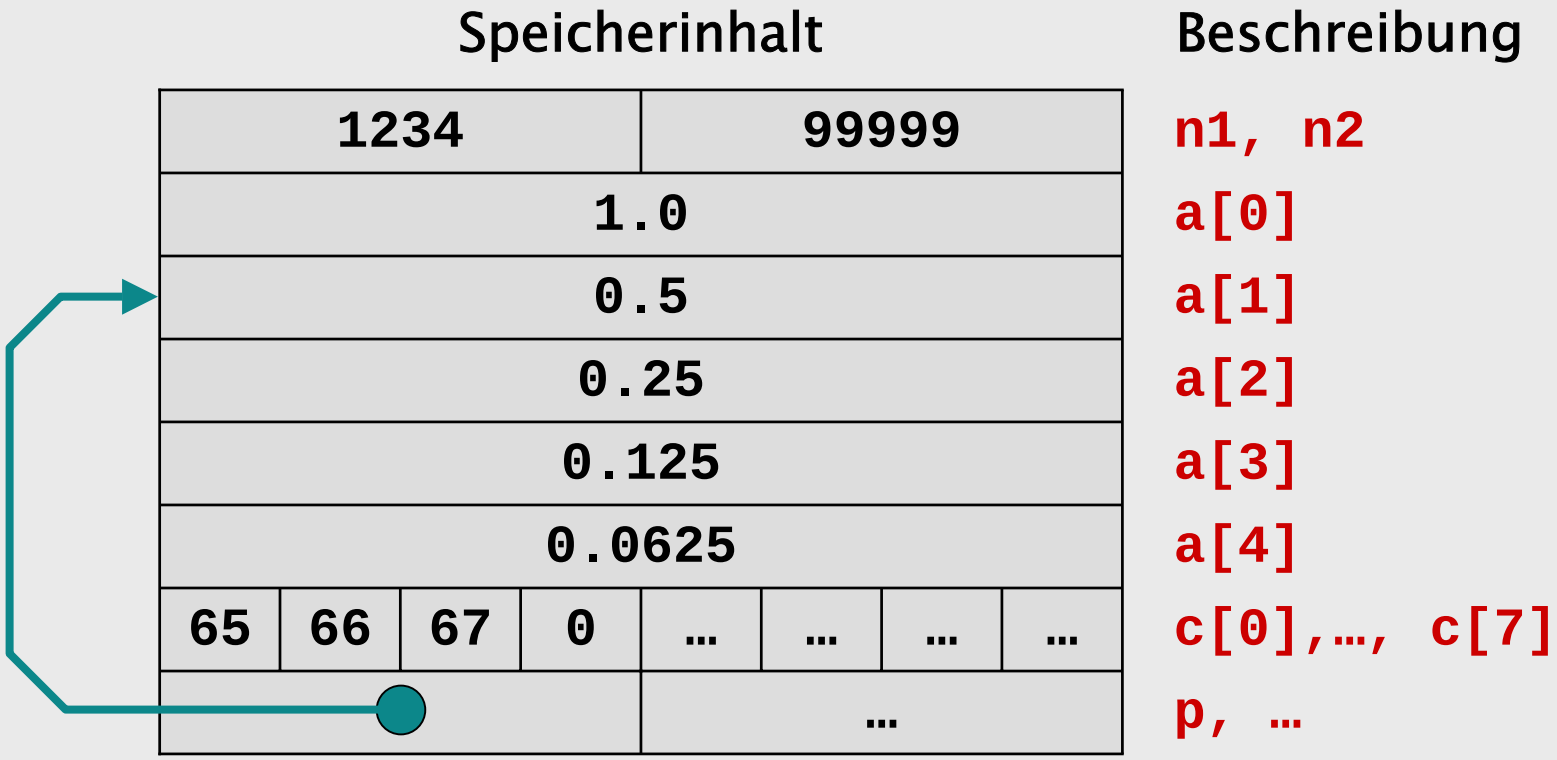
Wie das Beispiel zeigt, kann man mit Zeigern *rechnen*, was sich meist auf Addition/Subtraktion beschränkt, um sich in einem Array vor-/rückwärts zu bewegen.

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.

Das Freigeben, die Deallokation, geschieht mit dem Operator **delete**.

Entsprechend den zwei Formen von **new** (mit und ohne **[]**) gibt es diese auch bei **delete**.

Beispiele:

```
Person*  person = new Person;
Person*  group  = new Person[20];
...
delete   person;
delete[] group;
```

Die Form des **delete**-Operators muss unbedingt mit derjenigen des **new**-Operators übereinstimmen.

Das Freigeben von alloziertem Speicherplatz darf nicht vergessen werden.

Dies ist besonders wichtig, wenn innerhalb einer Schleife Speicher alloziert wird. Der verfügbare Speicherplatz würde sonst mit jedem Durchgang schrumpfen (sog. *memory leak*).

Wenn einem Zeiger ohne vorherige Deallokation ein neuer Wert zugewiesen wird, ist der alte Speicherbereich nicht mehr zugreifbar (es sei denn die Adresse wurde noch an eine zweite Zeigervariable zugewiesen).

Einige andere Sprachen wie Java oder Lisp erkennen dies und lassen solchen Speicherplatz von einem sog. *garbage collector* wieder „einsammeln“.

Wie erhält man *mehrdimensionale* dynamische Arrays?

Mit **new** kann ein solcher erzeugt werden:

```
int (*matrix)[nSpalten] =
                new int[nZeilen][nSpalten];
```

sofern alle Dimensionen **const** sind, ausser der ersten (hier **nZeilen**). Sonst müsste man so vorgehen:

```
     // Deklaration und Allokation
int** matrix = new int*[nZeilen];
for (int i = 0; i < nZeilen; i++)
     matrix[i] = new int[nSpalten];
     // Deallokation
for (int i = 0; i < nZeilen; i++)
     delete[] matrix[i];
delete[] matrix;
```

Bequemer geht es später mit speziellen C++ Klassen.

Zeiger und Verbund

Bei Verbund-Datentypen spielen die Zeiger ebenfalls eine grosse Rolle. Die Zeiger ermöglichen den Informationsaustausch ohne dass der ganze Verbund kopiert werden muss. Beispiel:

```
Person b = { "Charles Babbage", 26, … };
Person* p = &b;
```

Via den Zeiger **p** hat man nun Zugriff auf alle Komponenten des Verbundes, z.B. auf das Geburtsdatum mit

```
(*p).geburtstag
```

Für die oft gebrauchte Kombination von Dereferenzierung ***** und Auswahloperator **.** existiert der *Auswahloperator* **->** als Abkürzung:

```
p->geburtstag
```

In einem Verbund dürfen als Komponenten auch Zeiger auf den gleichen Verbund vorkommen.

Beispiel: der Verbund **`Person`** kann um die Komponenten

```
Person* vater;  Person* mutter;
```

erweitert werden. Damit könnte beispielsweise eine Datenstruktur für einen Stammbaum aufgebaut werden.

Dagegen darf der Verbund sich selber nicht “rekursiv” enthalten, eine Komponente **`Person ehepartner;`** darf also nicht innerhalb des Verbundes **`Person`** vorkommen.

Zeiger als Parameter

Bei Funktionsparametern werden oft Zeiger verwendet, vor allem in C, wo es keine Variablenparameter gibt.

Alternativ kann man Arrays auch als solche übergeben, beispielsweise:

```
double det(double mat[3][3]);
```

Die Funktion verlangt dann exakt diese Arraylänge.

Bei *mehrdimensionalen* Arrays hat Übergabe als Array den Vorteil, dass dann innerhalb der Funktionsdefinition die Schreibweise mit Klammern benützt werden darf:

`mat[i][j]` statt `*(mat + 3*i + j)`

Bei *eindimensionalen* Arrays ist die Klammerschreibweise immer erlaubt. Die Funktion

```
char* strcpy(char* dest, const char* src);
```

darf in ihrer Definition also via `dest[i]`auf das i-te Array-Element zugreifen.

In der Parameterliste ist die Schreibweise mit leeren Array-Klammern erlaubt. Man kann diese brauchen um Zeiger für Arrays visuell zu unterscheiden von Zeigern auf einfache Variablen:

```
char* strcpy(char dest[],
             const char src[]);
```

Für Zeiger gilt wie für andere Variablen: Bei der Parameter-Übergabe wird der Wert kopiert.

Bei einem Zeiger bedeutet dies:

- Man kann die Zeiger-Variable selbst nicht verändern.
- Man kann aber via den Zeiger den Speicher verändern.

Wenn der Zeiger für einen Array steht, heisst dies:

- Der Array bleibt an Ort und Stelle (die Anfangsadresse ist unveränderlich).
- Der *Inhalt* des Arrays kann verändert werden.
- Das Schlüsselwort **`const`** garantiert, dass der Inhalt des Arrays unverändert bleibt (siehe Beispiel **`strcpy`**).

Zeiger als Parameter sind nicht nur bei Arrays, sondern auch bei einfachen Variablen möglich.

Das Polarkoordinaten-Beispiel sieht mit Zeigern anstelle von Variablenparametern so aus:

```
void polar(double x, double y,
           double* rho, double* phi)
{
    *rho = sqrt(x*x + y*y);
    *phi = atan2(x, y);
}
```

Aufruf:

```
double x = 4., y = 3., laenge, winkel;
polar(x, y, &laenge, &winkel);
cout << "  Laenge:" << laenge <<
        ", Winkel:" << winkel << endl;
```

Mit Zeigern wird explizit ausgeführt, was Variablenparameter implizit leisten:

- Anwendung des Adressoperator **`&`** vor dem Aufruf (aus **`laenge`** wird **`&laenge`**)
- Übergabe als Zeiger (aus Typ **`double`** wird **`double*`**)
- Anwendung des Dereferenzierungsoperators **`*`** (aus **`rho`** wird **`*rho`**).

Zeiger als Rückgabewerte

Funktionen dürfen auch Zeiger zurückgeben. Ein Fehler wäre aber, einen Zeiger auf eine lokale Variable (z.B. Array oder Verbund) zurückzugeben. Der Speicherplatz muss mit **new** dynamisch alloziert werden (oder schon vor dem Aufruf alloziert sein).

```
double* sinusTabelle()
{
   double* s = new double[360];
    // nicht moeglich mit: double s[360];
   for (int i; i < 360; i++)
      s[i] = sin(i*M_PI/180.);
   return s;
}
…
double* sinTab = sinusTabelle();
```

Suchbäume

Beim Verwalten eines *dynamischen Datenbestandes* möchte man in beliebiger Folge Datensätze

- *suchen* (zum Lesen oder Ändern),
- *einfügen*,
- *löschen*.

Wir nehmen an dass, dieser aus n gleichartigen Datensätzen besteht. Die einzelnen Datensätze werden sinnvollerweise als Verbund realisiert.

In welcher Datenstruktur sind diese drei Operationen effizient implementierbar?

Naheliegende Lösung: den Datenbestand als *Array* organisieren.

- Einfügen: Zeitaufwand *O(1)* weil der Datensatz am Schluss angehängt werden kann. Aber: Doppeltes Einfügen wird nicht erkannt!
- Suchen, Löschen: Zeitaufwand *O(n)* weil *sequentielles Suchen* nötig.

Nächstbessere Lösung: den Datenbestand als *sortierten Array* zu organisieren. Eine Komponente des Verbundes wird als *Sortierschlüssel* gewählt. Der Schlüssel muss den Datensatz *eindeutig* identifizieren.

- Suchen: Zeitaufwand $O(\log n)$ weil *binäres Suchen* möglich. Man findet Datensätze nach (aufgerundet) $\log_2 n + 1$ Vergleichen.

- Einfügen, Löschen: Zeitaufwand $O(n)$ weil durchschnittlich $n/2$ Array-Elemente verschoben werden müssen.

Ziel: alle drei Operationen in *O(log n)* Zeit realisieren.

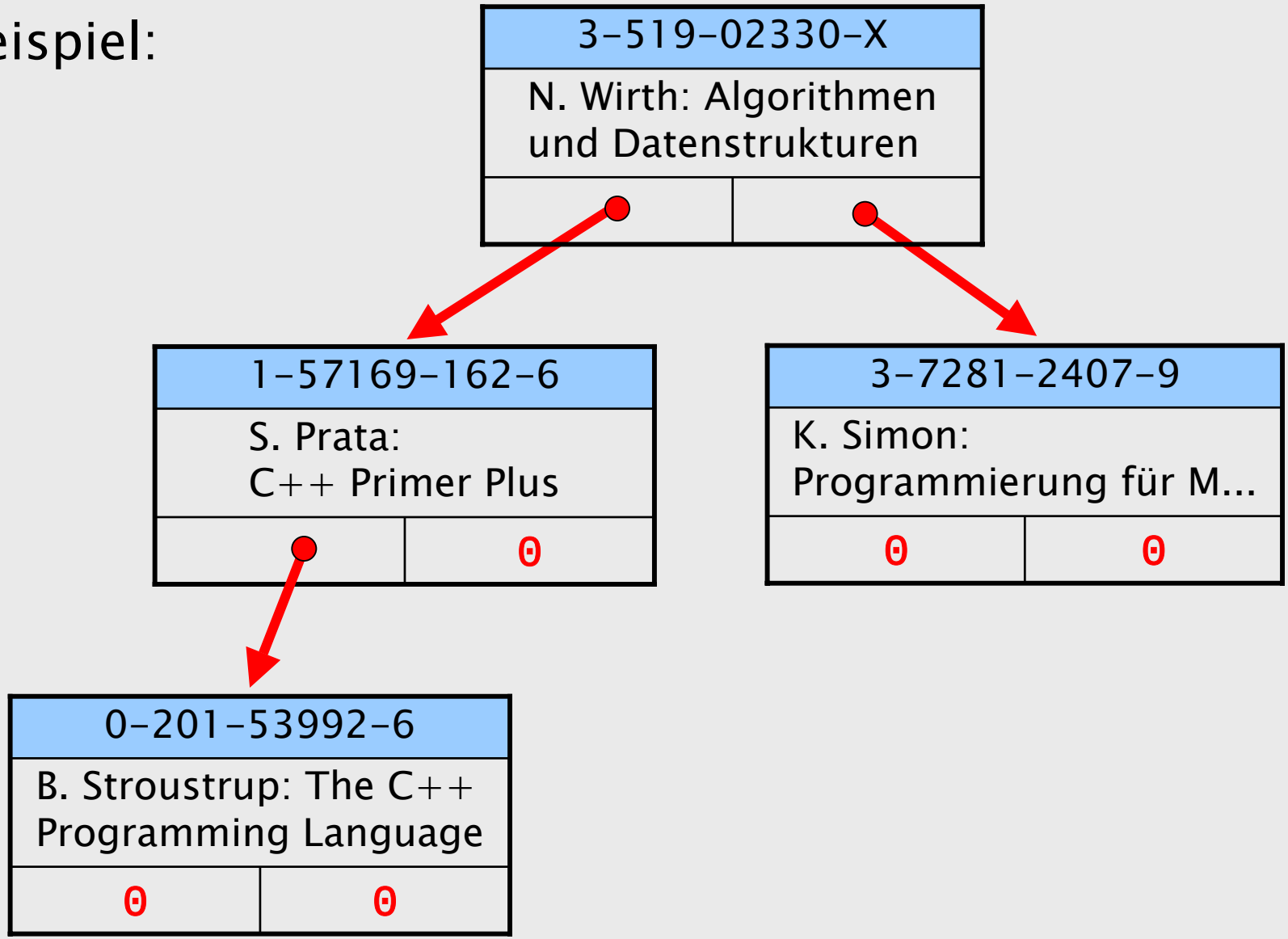
Datenstruktur: *binärer Baum*.

Die Knoten des Baumes sind die Datensätze, erweitert um zwei Zeiger.

```
struct TreeNode {
        int key;          // Sortierschluessel
        Data data;        // weitere Daten
        TreeNode* left;   // linker Teilbaum
        TreeNode* right; // rechter Teilbaum
};
```

Der Sortierschlüssel kann von einem beliebigen Datentyp sein, der eine *lineare Ordnung* besitzt.

Beispiel:


3-519-02330-X
N. Wirth: Algorithmen und Datenstrukturen
1-57169-162-6
S. Prata: C++ Primer Plus
0
3-7281-2407-9
K. Simon: Programmierung für M...
0
0
0-201-53992-6
B. Stroustrup: The C++ Programming Language
0
0

Jeder (gültige) Zeiger vom Typ `TreeNode*` repräsentiert einen Baum. Im Fall eines `NULL`-Zeigers (Zeigers mit Wert 0) ist dies ein leerer Baum.

Bedingungen für einen gültigen nichtleeren Baum:

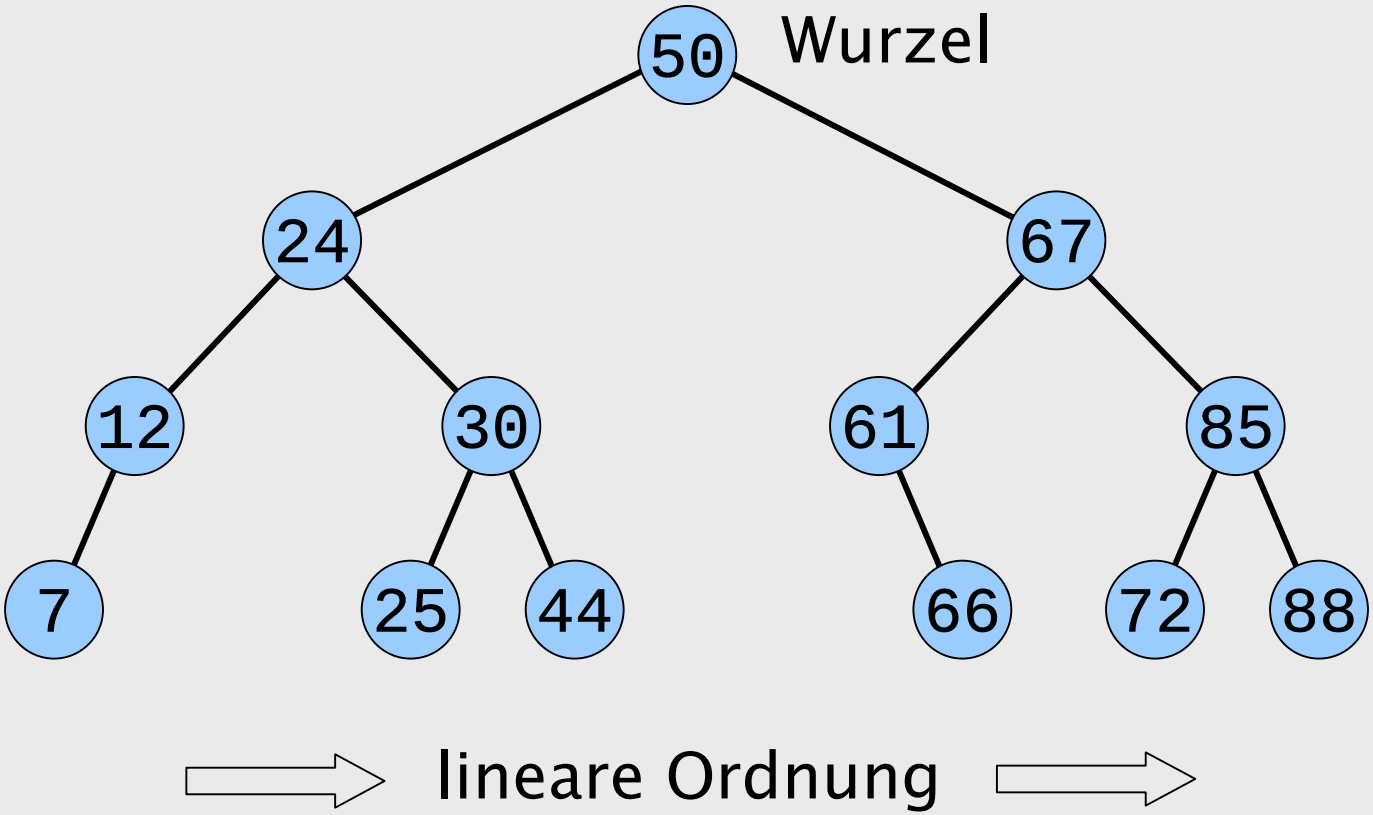
- Die Zeiger `left` und `right` sind entweder 0 oder zeigen auf einen Knoten.
- Keine zwei Zeiger zeigen auf denselben Knoten.
- Es gibt einen Wurzel-Knoten, auf den kein Zeiger zeigt.

Ein *Suchbaum* ist ein binärer Baum, bei dem die Knoten von links nach rechts sortiert sind:

Für jeden Knoten eines Suchbaums gilt:

- Der Schlüssel ist grösser als alle Schlüssel im linken Teilbaum
- Der Schlüssel ist kleiner als alle Schlüssel im rechten Teilbaum

Beispiel eines Suchbaumes:



Suchen

Das Suchen ist die einfachste der drei Operationen.

Die Funktion **search** sucht im Baum **r** einen Datensatz mit Schlüssel **z**. Rückgabewert ist ein Zeiger auf den entsprechenden Knoten, resp. **0** wenn kein solcher existiert.

```
TreeNode* search(int z, TreeNode* r)
{
  if (!r)                 return 0; // leerer Baum
  else if (z < r->key)    return search(z, r->left);
  else if (z == r->key)   return r;
  else /* z > r->key */   return search(z, r->right);
}
```

Einfügen

Die Funktion **insert** fügt einen neuen Datensatz mit Schlüssel **z** ein, sofern der Schlüssel noch nicht im Baum vorkommt. Via zurückgegebenen Zeiger kann der Datensatz anschliessend noch ergänzt werden.

```
TreeNode* insert(int z, TreeNode* &r)
{
  if (!r) {              // leerer Baum: fuege hier ein
    r = new TreeNode; r->key = z;
    r->left = 0; r->right = 0;
    return r;
  }
  else if (z < r->key)  return insert(z, r->left);
  else if (z == r->key) return 0; // exist. schon
  else /* z > r->key */ return insert(z, r->right);
}
```

Löschen

Das Löschen ist die schwierigste der drei Operationen:

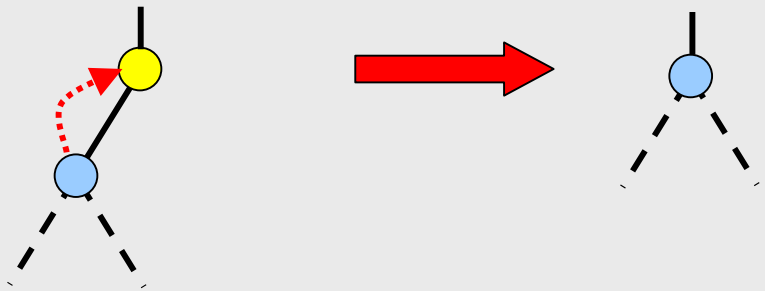
Sie beginnt mit einer Suchoperation. Ergebnis ist dann ein Baum mit dem zu löschenden Knoten in der Wurzel.

Das reduzierte Problem ist nun:

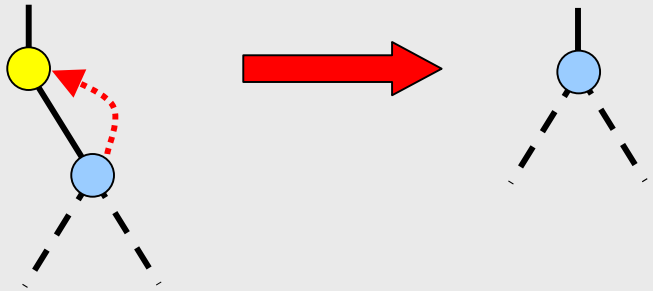
Wie entfernt man die Wurzel ohne die Suchbaum-Eigenschaft zu verletzen?

Dies ist leicht wenn (mindestens) einer der beiden Teilbäume leer ist: Man ersetzt den Baum einfach durch den zweiten Teilbaum.

rechter Teilbaum leer:



linker Teilbaum leer:

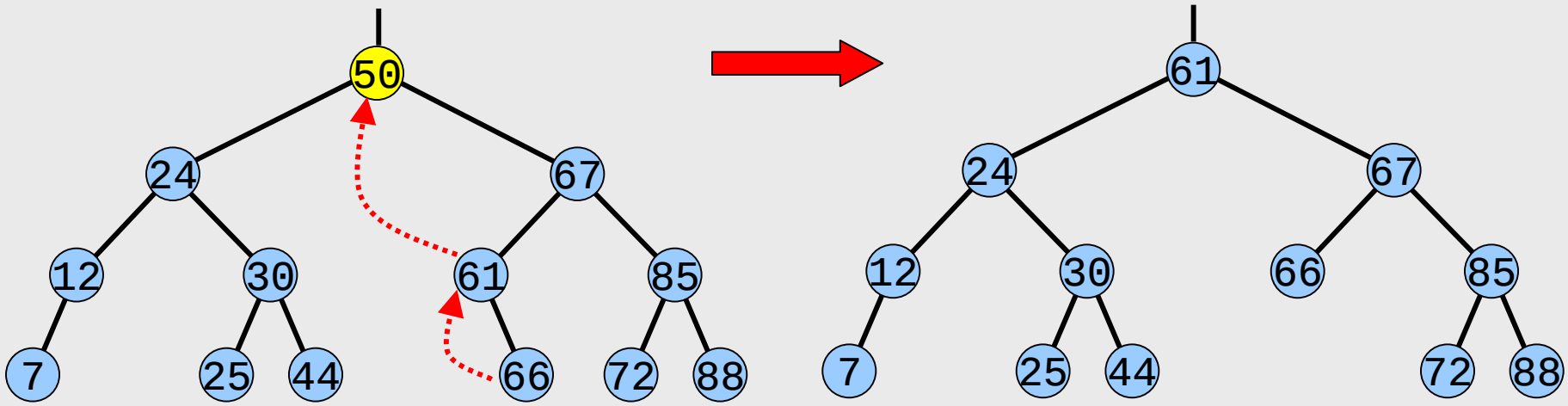


beide Teilbäume leer:

Seien nun beide Teilbäume nichtleer.

Die Idee ist nun:

- im rechten Teilbaum den linksäussersten Knoten (den Knoten mit dem kleinsten Schlüssel) entfernen, und
- diesen anstelle der Wurzel verwenden.



Die Hilfsfunktion **detachMin** entfernt im Baum **r** den Knoten mit dem kleinsten Schlüssel (ohne den Datensatz zu löschen). Rückgabewert ist ein Zeiger auf diesen "abgehängten" Datensatz.

```
TreeNode* detachMin(TreeNode* &r)
{
  if (r->left) return detachMin(r->left);
  else {
     TreeNode* p = r;
     r = r->right;
     return p;
  }
}
```

Dies erlaubt nun die Funktion **remove** zu schreiben:

```
void remove(int z, TreeNode* &r)
{
  if (!r) return;  // Fehler: nicht gefunden
  else if (z < r->key) remove(z, r->left);
  else if (z == r->key) {
    TreeNode* left  = r->left;
    TreeNode* right = r->right;
    delete r;
    if (!left) r = right;
    else if (!right) r = left;
    else {
      r = detachMin(right);
      r->right = right; r->left = left;
    }
  }
  else /* z > r->key */ remove(z, r->right);
  return;
}
```

Laufzeitanalyse

Die Laufzeit der Operationen Suchen, Einfügen und Löschen ist proportional zur *Höhe* des Suchbaumes.

Die Höhe eines Suchbaums mit n Knoten ist

- im Idealfall $log_2 (n+1)$ (aufgerundet),
- im schlimmsten Fall n

Der schlimmste Fall tritt ein, wenn ausgehend vom leeren Baum, n Datensätze mit monoton steigenden (oder fallenden) Schlüsseln eingefügt werden.

Der Idealfall liegt vor, wenn der Baum *vollständig ausgeglichen* ist:

Definition: Ein binärer Baum heisst *vollständig ausgeglichen*, wenn in jedem Knoten gilt:
Die *Anzahl Knoten* der beiden Teilbäume unterscheidet sich höchstens um Eins.

Eigenschaft der vollständig ausgeglichenen Bäume:
Die Länge aller Äste (Wurzel bis Blatt) unterscheidet sich höchstens um Eins.

Problem: Unsere Einfüge- und Löschoperationen zerstören die vollständige Ausgeglichenheit.

Das vollständige Ausgleichen nach *jeder* solchen Operation wäre zu aufwendig.

Eine praktikable Lösung ist dagegen das periodische Neuanlegen des gesamten Suchbaumes.

Besser ist es, eine schwächere Eigenschaft zu verwenden:

Definition: Ein binärer Baum heisst *ausgeglichen* (oder *AVL-Baum*), wenn in jedem Knoten gilt: Die *Höhe* der beiden Teilbäume unterscheidet sich höchstens um Eins.

Vorteil: diese Eigenschaft ist leichter wiederherzustellen.

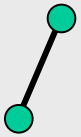
Zudem wird Einfügen und Löschen nur unwesentlich aufwendiger, denn:

Adelson-Velskii und Landis haben gezeigt, dass ausgegliche Bäume maximal 45% höher sind als die entsprechenden vollständig ausgeglichenen Bäume.

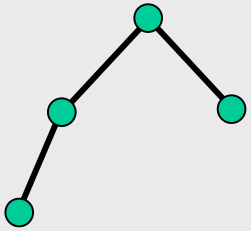
Beispiele ausgeglichener Bäume: Die Fibonacci-Bäume

$h = 1$

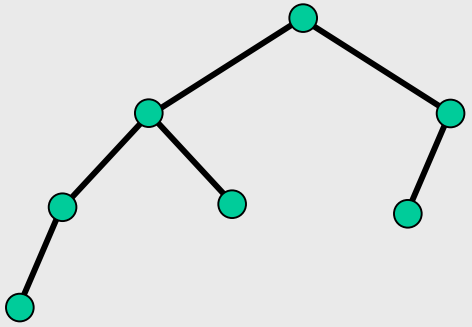
$h = 2$



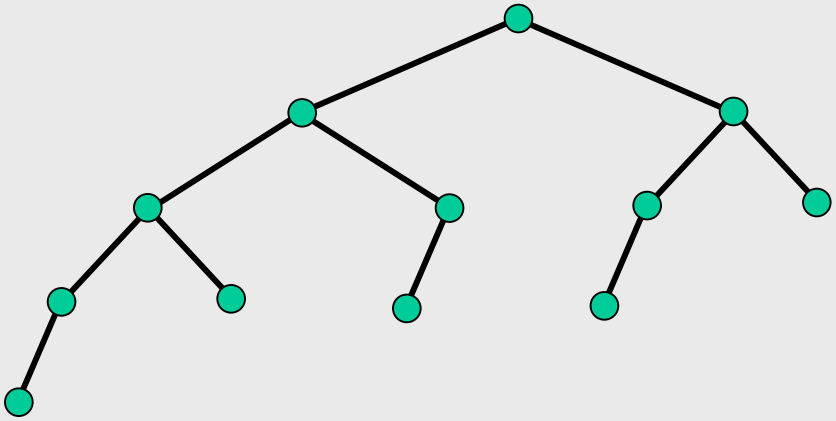
$h = 3$



$h = 4$



$h = 5$



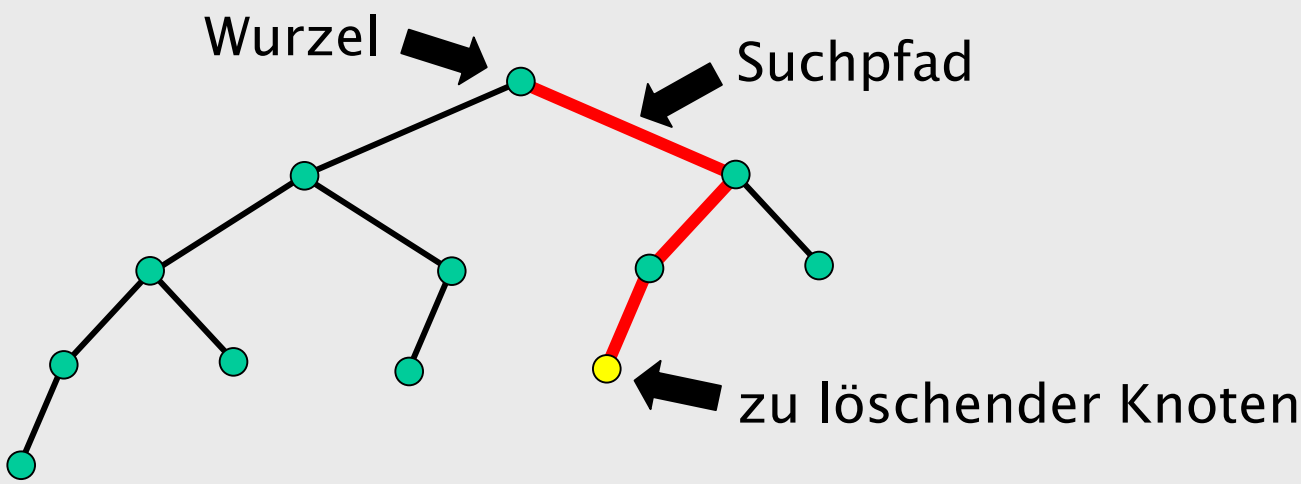
Ausgleichen in AVL-Bäumen

Um die Ausgeglichenheit eines (Teil-)Baumes festzustellen resp. wiederherzustellen, muss man die *Höhe* eines Teilbaumes ermitteln können.

Damit dies genügend schnell geht, wird in jedem Knoten zusätzlich die Höhe seines Teilbaumes gespeichert.

Beim Einfügen oder Löschen ändert sich die Höhe nur für Knoten entlang dem *Suchpfad*. Beim "Abbauen" der Rekursion in `insert` resp. `remove` klettert man aber genau dem Suchpfad nach zurück. Man kann dabei die Höhe wo nötig korrigieren.

Beispiel:



Nach einer einzigen Einfüge- oder Löschoperation ist die Ausgeglichenheit höchstens für Knoten entlang dem Suchpfad verletzt.

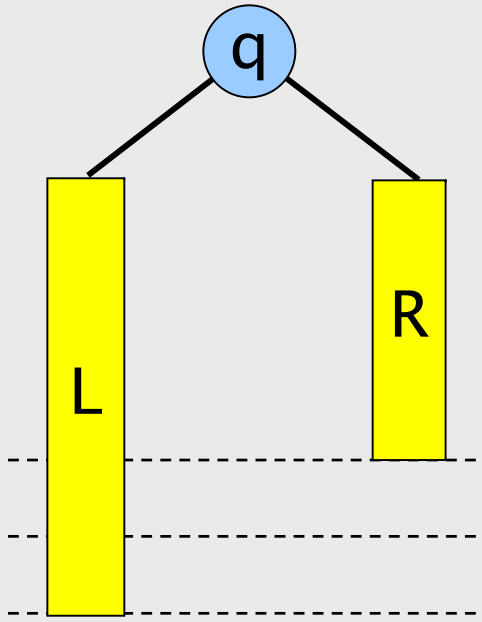
In diesem Fall unterscheiden sich die Höhen der beiden Teilbäume um 2.

Idee: in den Funktionen **`insert`** und **`remove`** zusätzlich vor der **`return`**-Anweisung:

- falls sich die Höhen der beiden Teilbäume **`r->left`** und **`r->right`** um 2 unterscheiden, dies durch sog. Rotationen (s.u.) korrigieren,
- die gespeicherte Höhe des Baumes **`r`** korrigieren.

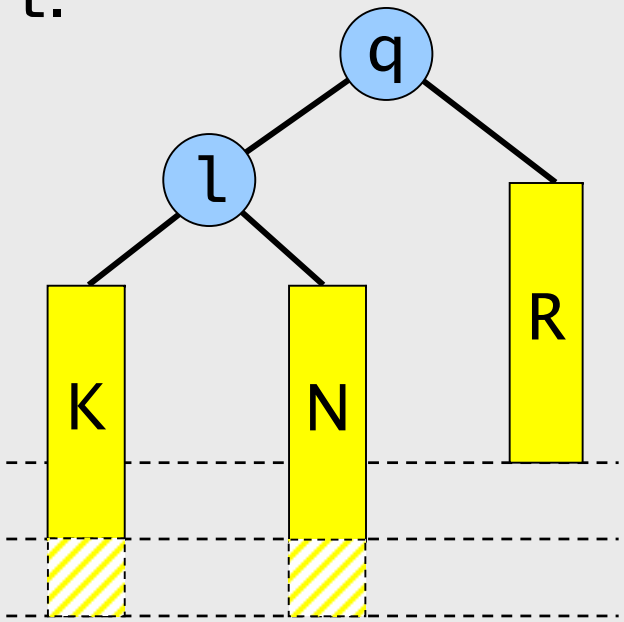
Annahme (oBdA.): der linke Teilbaum ist um 2 höher als der rechte.

In Skizze dargestellt:



Die beiden Rechtecke L und R symbolisieren Teilbäume der gezeichneten Höhe. Dank der Rekursion darf man L und R als ausgeglichen voraussetzen.

Sei nun `l` die Wurzel von L und seien K und N die beiden Teilbäume von `l`.



Nun sind die beiden Fälle

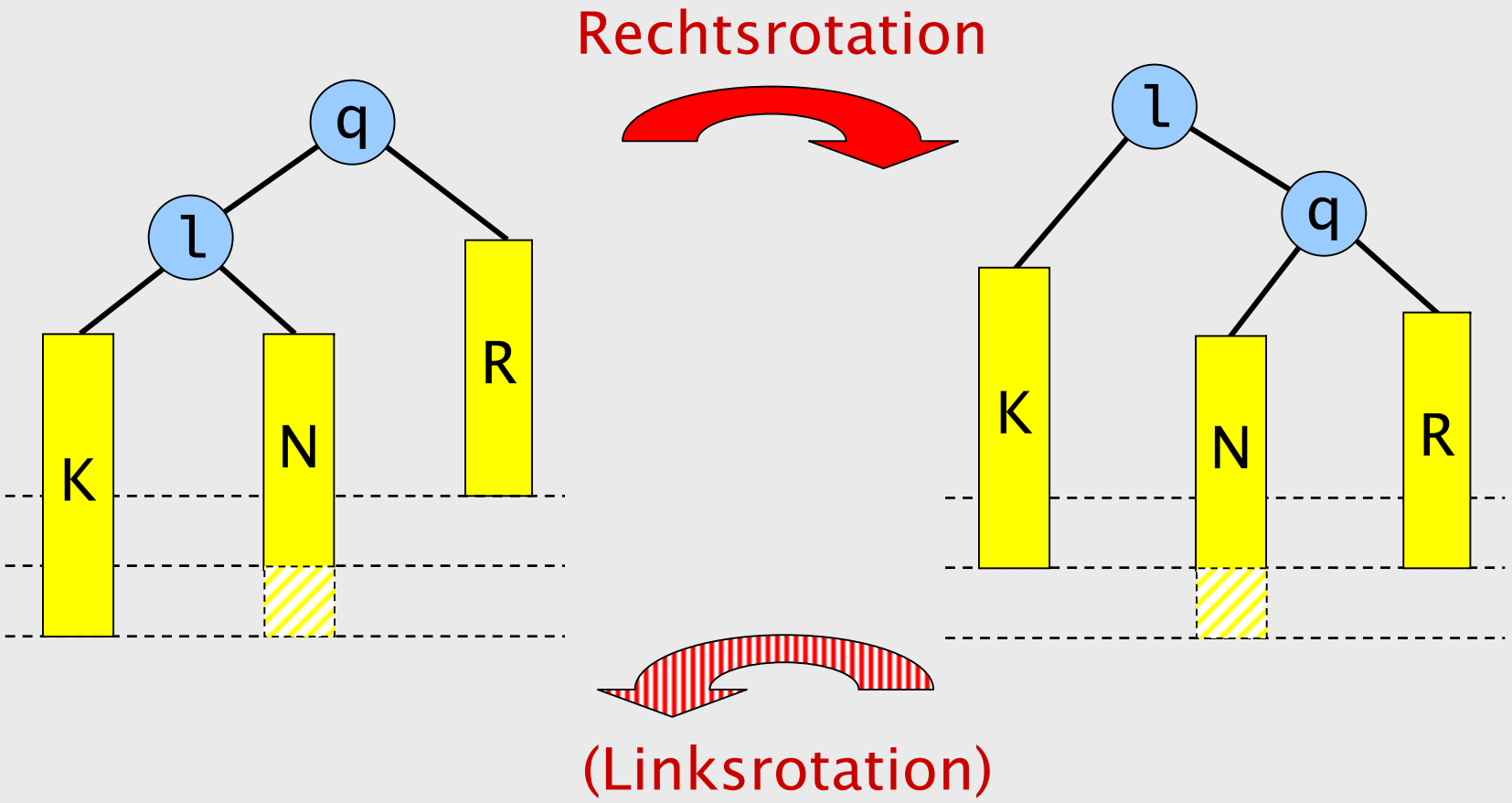
$$\text{Höhe}(K) \geq \text{Höhe}(N)$$

$$\text{Höhe}(K) < \text{Höhe}(N)$$

getrennt zu behandeln.

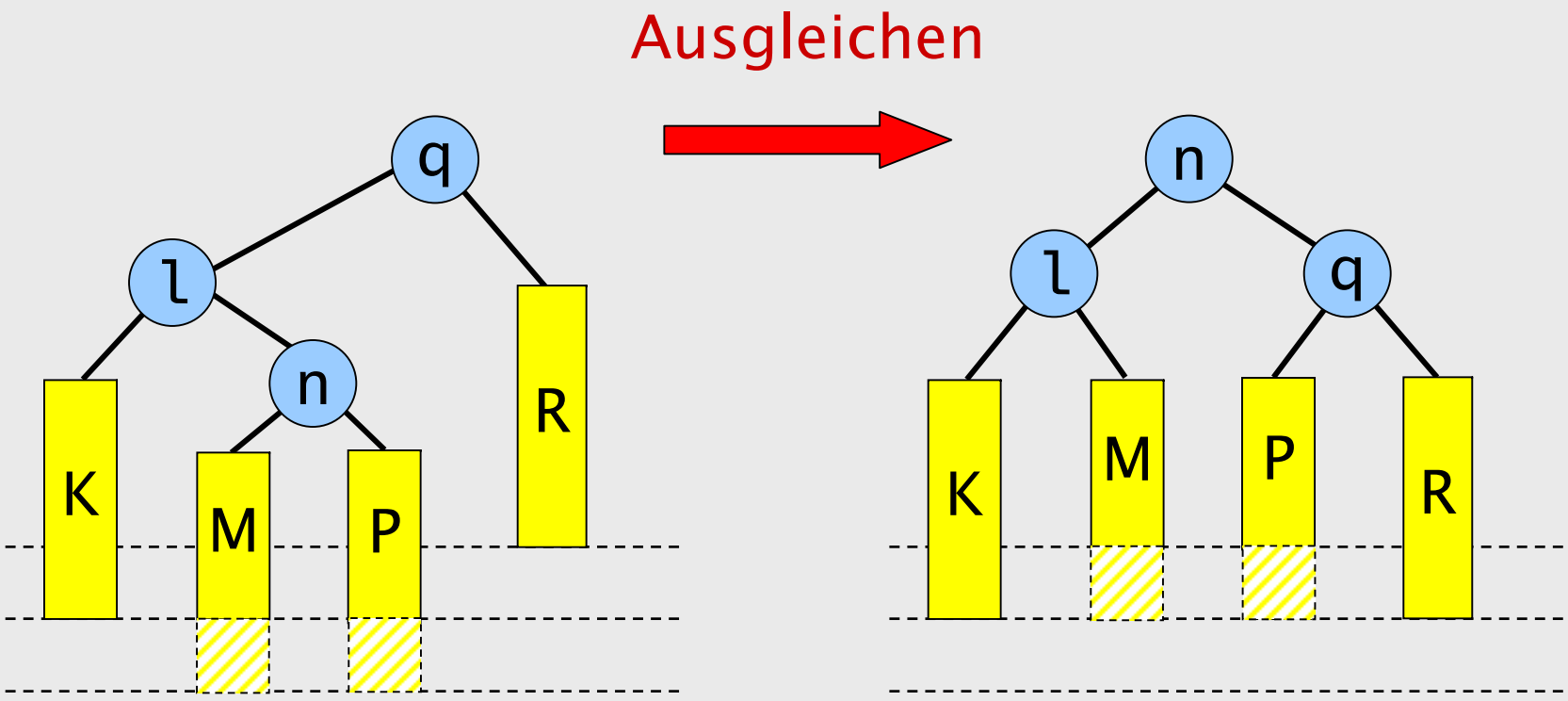
Fall I: Höhe(K) ≥ Höhe(N)

Hier besteht das Ausgleichen in einer *Rechtsrotation:*



Fall II: Höhe(K) < Höhe(N)

Hier besteht das Ausgleichen in einer Linksrotation um (l,n) gefolgt von einer Rechtsrotation um (q,n).



Die Rotationsoperationen bestehen nur aus dem "Umhängen" von zwei Zeigern sowie dem Austauschen der Wurzel:

```
void rotateRight(TreeNode* &r)
{
  TreeNode* l = r->left;
  r->left = l->right;
  l->right = r;
  r = l;
  return;
}
```

Die Linksrotation funktioniert analog. Beide wären noch zu ergänzen um das Nachführen der gespeicherten Höhe.

Durchlaufstrategien

Wie erhält man eine geordnete Aufzählung aller Schlüssel in einem Suchbaum?

Am einfachsten natürlich mit einer rekursiven Funktion:

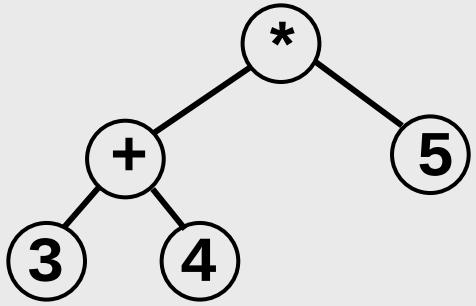
```
void printTree(TreeNode* &r)
{
  if (r) {
    printTree(r->left);
    cout << r->key << endl;
    printTree(r->right);
  }
  return;
}
```

Die Reihenfolge wie die Wurzel und die zwei Teilbäume behandelt werden ist hier wichtig.

Es gibt folgende sinnvolle Strategien, einen (allgemeinen) binären Baum zu traversieren:

- *inorder*-Reihenfolge:
  - linker Teilbaum
  - Wurzel
  - rechter Teilbaum
- *preorder*-Reihenfolge:
  - Wurzel
  - linker Teilbaum
  - rechter Teilbaum
- *postorder*-Reihenfolge:
  - linker Teilbaum
  - rechter Teilbaum
  - Wurzel

Stellt der Baum einen arithmetischen Ausdruck dar wie



dann ergibt

inorder: **`(3+4)*5`** die gewohnte Infix-Schreibweise (Klammern sind nötig!),

preorder: **`*(+(3,4),5)`** die Präfix- oder "Funktions"-Schreibweise (Klammern und Kommas sind nicht nötig wenn die Anzahl Operanden klar ist),

postorder: **`3 4 + 5 *`** die Postfix-Schreibweise, wie z.B. von PostScript und von früheren Taschenrechnern verwendet.

Klassen

Objektorientiertes statt *prozedurales* Programmieren ist eine neuere "Philosophie" der Software-Entwicklung.

Prozedural:

- Das (Unter-) Programm steht im Mittelpunkt.
- Es transformiert Eingabedaten in Ausgabedaten.
- Daten selbst sind "passiv".
- Der Zugriff auf die Daten wird nicht kontrolliert.

Objektorientiert:

- Daten und deren Verwaltung werden zusammengefasst zu *Objekten*.
- Das Objekt bietet kontrollierte Zugriffe auf die Daten an um diese zu lesen oder zu manipulieren.
- Objekte sind Variablen.
- Die Datentypen von solchen Variablen nennt man *Klassen*.

Auch die Mathematik kennt "Datentypen" die aus einer Menge und darauf definierten Operationen bestehen. Beispiele sind Gruppe, Vektorraum, etc.

In C++ sind Klassen als *Erweiterungen des Verbundes* realisiert.

Beispiel "Stack"

Wir wollen einen Datentyp für einen *Stack* (auch: Stapel, Keller) definieren.

Ein Stack

- enthält Datensätze, hier der Einfachheit halber nur **`char`**-Zeichen, und er
- erlaubt die drei Operationen:

  `empty` : abfragen ob der Stapel leer ist

  `push` : einen neuen Datensatz auf den Stack laden

  `pop` : den obersten Datensatz wegnehmen

Ziel: Ein Datentyp **`Stack`**, der die Implementation "versteckt".

Der Stack kann z.B. als *Array* oder als *verkettete Liste* implementiert sein.

Die Implementation soll später ersetzbar sein durch eine effizientere, ohne dass alle Anwendungsprogramme geändert werden müssen.

Nach aussen "sichtbar" sein sollen einzig die Spezifikationen (d.h. die Deklarationen) der drei Funktionen **`empty`**, **`push`** und **`pop`**.

Erste Implementation in C++:

```
struct Stack {
    static const int SIZE = 100;
    char data[SIZE];
    char* top;
    void init()          { top = data; }
    bool empty()         { return top == data; }
    void push(char item){ *top++ = item; }
    char pop()           { return *--top; }
};
```

Die Klasse **`Stack`** hat 7 Elemente, davon 4 *Element-funktionen* (auch "*Methoden*" genannt).

Das Element **`SIZE`** ist ebenfalls speziell weil als **`static`** deklariert.

Die verbleibenden Elemente **`data`** und **`top`** sind herkömmliche Verbund-Komponenten. Nur diese belegen effektiv Speicherplatz pro Variable vom Typ **`Stack`** .

Beispiel für Benützung der Klasse:

```
// Testprogramm: Woerter umkehren
int main()
{
  Stack st;
  st.init();
  while (!cin.eof()) {
    char c = cin.get();
    if (isalnum(c)) st.push(c);
    else {
      while (!st.empty()) {
        cout.put(st.pop());
      }
      if (!cin.eof()) cout.put(c);
    }
  }
}
```

Man beachte:

- Für den Aufruf von Elementfunktionen ist die Syntax

  ```
  st.init();
  ```

  analog zum Zugriff auf andere **`struct`**-Elemente, z.B.

  ```
  st.top;
  ```

- Innerhalb der *Definition* einer Elementfunktion entfällt aber der Präfix **`st.`**, es genügt **`top`**.

Im Aufruf `st.push(c);` ist die Variable `st` vor dem Punkt ähnlich zu verstehen wie ein zusätzliches (implizites) Argument.

Mit einer *gewöhnlichen* Funktion würde man schreiben `push(st, c);`.

Dagegen mit einer *Elementfunktion:*

- Die Funktion `push` "gehört" dem Objekt `st`.
- Ein Vorteil ist: Generische Namen wie `push`, `print` oder `open` können gebraucht werden, ohne dass die Gefahr von Namenskonflikten besteht.

`Stack` bedarf noch mehrerer Verbesserungen!

Eines der Probleme ist:

- Die obligatorische Initialisierung (d.h. der Aufruf von **`init`**) ist unschön, und auch unsicher, da diese vergessen werden kann.
- Man kann dies nicht korrigieren, indem man schreibt

  ```
  char* top = data;
  ```

  weil Verbundkomponenten nicht initialisiert werden dürfen.

Abhilfe: Klassen bieten sog. *Konstruktoren* an.

Ein Konstruktor wird *automatisch aufgerufen*, wenn ein Objekt kreiert wird. Dies geschieht

- beim Eintritt in einen Block, worin eine Variable der Klasse deklariert ist, sowie:
- beim dynamischen Erzeugen mittels **`new`** oder **`new …[…]`**.

Der Konstruktor gleicht einer Elementfunktion bis auf folgende Abweichungen:

- der Konstruktor heisst gleich wie die Klasse.
- der Konstruktor hat keinen Rückgabewert, er wird aber auch nicht als **`void`** deklariert.
- er enthält keine **`return`**-Anweisung.

```
// Stack-Klasse, jetzt mit Konstruktor
struct Stack {
    static const int SIZE = 100;
    char data[SIZE];
    char* top;
    Stack() { top = data; }       // Konstruktor
    bool empty() { return top == data; }
    void push(char item) { *top++ = item; }
    char pop() { return *--top; }
};
```

stack2.cpp

Nächste Verbesserung: Es sollen Stacks mit unterschiedlicher Maximalgrösse möglich sein.

Dazu kann ein *Konstruktor mit Argumenten* verwendet werden.

Wie bei gewöhnlichen Funktionen darf der Funktionsname mehrfach verwendet werden, solange die Signatur jedesmal eine andere ist.

Weil wir jetzt Speicher dynamisch allozieren, müssen wir ihn auch wieder freigeben. Dies macht man im *Destruktor*.

Der Destruktor wird automatisch aufgerufen, wenn ein Objekt gelöscht wird, also:

- beim Verlassen des Blocks, wo die Variable deklariert ist, oder
- beim Freigeben mit **`delete`** oder **`delete[]`**.

Der Destruktor heisst gleich wie die Klasse mit vorangestellter Tilde (**`~`**).

stack3.cpp

```
struct Stack {
  int size;
  char* data;
  char* top;
  Stack() {                  // Default-Konstruktor
    size = 100;
    data = new char[size];
    top = data;
  }
  Stack(int size) {          // Zweiter Konstruktor
    Stack::size = size; // ::  loest Konflikt
    data = new char[size];
    top = data;
  }
  ~Stack() { delete[] data; } // Destruktor
```

Es gibt immer noch Verbesserungsmöglichkeiten:

- Wahrung der *Datenintegrität:*

  Der Benützer der Klasse **`Stack`** hat Zugriff auf die Elemente **`data`** und **`top`**. Er könnte diese fehlerhaft verwenden und damit den Stack verfälschen.

- *Kapselung* der Implementation *(Information hiding):*

  Wenn man dem Benützer Zugriffe auf **`data`** und **`top`** erlaubt, kann man die Klasse **`Stack`** nicht mehr zu Optimierungszwecken re-implentieren.

Die Lösung besteht in einer *Zugriffskontrolle:*

Der Zugriff auf jedes Klassenelement kann in drei Stufen gewährt werden:

- **`private:`** für Klassen-Elemente die nur intern (d.h. in Definitionen von Elementfunktionen) verwendet werden dürfen
- **`public:`** für erlaubte Zugriffe von aussen (solche Elemente bilden die Schnittstelle der Klasse)
- **`protected:`** wie **`private`**, aber zugreifbar auch für abgeleitete Klassen (siehe später).

In unserem Beispiel **`Stack`** sieht dies so aus:

```
struct Stack {
   private:
      int size;
      char* data;
      char* top;
   public:
      Stack() {
      ...
```

Statt **struct** kann das Schlüsselwort **class** verwendet werden.

Unterschied:

- Bei **class** ist **private** vordefiniert,
- bei **struct** dagegen **public**.

Es gibt Situationen wo ein neues Objekt vom Typ **`Stack`** erzeugt wird, ohne dass einer der zwei bis jetzt definierten Konstruktoren aufgerufen wird.

Dies geschieht namentlich bei

- Deklaration mit Initialisierung
- Zuweisung

Hier wird ein Copy-Konstruktor resp. ein Zuweisungs-Operator aufgerufen.

Default- und Copy-Konstruktor, Zuweisungs-Operator und Destruktor müssen nicht explizit programmiert werden.

C++ stellt diese wenn nötig zur Verfügung.

Sobald aber im Konstruktor dynamisch Speicher alloziert wird, sollten alle vier Funktionen explizit programmiert werden.

Der Copy-Konstruktor kann so definiert werden:

```
Stack(const Stack& s) {
      size = s.size;
      data = new char[size];
      top = data;
      char* top0 = s.data;
      while (top0 < s.top) *top++ = *top0++;
}
```

Er wird automatisch aufgerufen bei einer Initialisierung:

```
Stack a;
Stack c = a;
```

Und der Zuweisungs-Operator kann so definiert werden:

```
Stack& operator=(const Stack& s) {
      size = s.size;
      delete[] data;          // !!!
      data = new char[size];
      top = data;
      char* top0 = s.data;
      while (top0 < s.top) *top++ = *top0++;
      return *this;
}
```

Er wird automatisch aufgerufen bei einer Zuweisung:

```
Stack a;
Stack c;
c = a;    // c.operator=(a);
```

stack4.cpp

Andere Anwendung des Stacks:
"Türme von Hanoi" mit Iteration statt Rekursion.

Statt die Funktion rekursiv aufzurufen, werden die drei "Teilaufgaben" (in umgekehrter Reihenfolge!) auf einen "Aufgabenstack" geladen.

```
int main()
{
  Stack s(200);

  s.push(5);  // Anzahl Scheiben
  s.push(0);  // Turm "von"
  s.push(1);  // Turm "ueber"
  s.push(2);  // Turm "nach"
```

```
while (!s.empty()) {
   // Hole 4 oberste Zahlen
   int nach   = s.pop();
   int ueber  = s.pop();
   int von    = s.pop();
   int anzahl = s.pop();
   if (anzahl == 1) {
     cout << von << " -> " << nach << "\n";
   }
   else {
     // Teilaufgabe 3:  Turm ohne unterste
     // Scheibe von Zwischenplatz
     s.push(anzahl-1);
     s.push(ueber);
     s.push(von);
     s.push(nach);
```

```
            // Teilaufgabe 2:  Unterste Scheibe
            // allein
            s.push(1);
            s.push(von);
            s.push(ueber);
            s.push(nach);

            // Teilaufgabe 1:  Turm ohne unterste
            // Scheibe auf Zwischenplatz
            s.push(anzahl-1);
            s.push(von);
            s.push(nach);
            s.push(ueber);
        }
    }
}
```

stackHanoi.cpp

Verbleibende offensichtliche Probleme von **`Stack`** sind:

- fehlende Überlaufbehandlung
- unbenutzter Speicherplatz

Sie wären lösbar mit verketteter Liste statt Array, aber:

- Speicherplatzverschwendung für die Zeiger.

Eine gute Implementation ist etwas komplizierter.

Die Standardbibliothek bietet eine solche, allerdings:
nicht als Klasse, sondern als Klassen-*Template*.

Vorteil des (später erklärten) Templates: Es definiert nicht nur einen Stack von **`char`** (oder **`int`**), sondern Stacks basierend auf beliebigen (auch eigenen) Datentypen.

Beispiel "Datum"

Wir definieren eine Klasse **`Datum`** mit

- Elementen: **`tag, monat`** und **`jahr`**
- Operationen:

  Vortag, Folgetag, Wochentag, formatierte Ausgabe.

Auf die Elemente **`tag, monat`** und **`jahr`** soll nicht zugegriffen werden dürfen.

Dies verunmöglicht fehlerhafte Datumsberechnungen wie

```
Datum gestern = heute; gestern.tag--;
```

oder

```
stich.tag   = 29;
stich.monat = 2;
stich.jahr  = 2002;
```

```
class Datum
{
  int tag, monat, jahr;
 public:
  // Konstruktoren
  Datum(int, int, int); // Tag, Monat, Jahr
  Datum(int, int);      // T., M., (aktuelles J.)
  Datum(int);           // Tag, (akt. M. und
  Datum();              // heute (default ctor)
  Datum(const char *);  // Datum aus C-String

  // Methoden
  Datum vortag();
  Datum folgetag();
  int wochentag();   // 0: Sonntag, 1: Montag,...
  void ausgabe(int); // Ausg. in versch. Formaten
};
```

Hier wurde eine alternative Syntax verwendet:

- Innerhalb des *Klassenblocks* werden die Element-funktionen nur *deklariert*,
- *definiert* werden sie dann ausserhalb, wobei dann der Bezugsoperator `::` benötigt wird.

Oft ist es bequem, *mehrere Konstruktoren* zu haben. Die ersten vier Konstruktoren kann man aber ersetzen durch einen einzigen, unter Verwendung von *Default-Argumenten*.

Default-Argumente können beim Aufruf ganz oder teilweise (von hinten nach vorn) weggelassen werden.

Die Klasse **Datum** vereinfacht sich damit zu:

```
class Datum
{
  int tag, monat, jahr;
 public:
  // Konstruktoren
  Datum(int t=0, int m=0, int j=0);
  Datum(const char*);  // Datum aus C-String

  // Methoden
  Datum vortag();
  Datum folgetag();
  int wochentag(); // 0: Sonntag, 1: Montag, ..
  void ausgabe();  // formatierte Ausg. auf cout
};
```

Es fehlen noch die *Definitionen*.

Der erste Konstruktor kann etwa so definiert werden:

```
Datum::Datum(int t, int m, int j)
{
  tag   = t!=0 ? t : aktueller_tag();
  monat = m!=0 ? m : aktueller_monat();
  jahr  = j!=0 ? j : aktuelles_jahr();
  // Pruefe Gueltigkeit des Datums
  // ...
}
```

Die übrigen Definitionen lassen wir der Kürze halber weg.

Überladene Operatoren

Neues Beispiel: Eine Klasse **Bruch** für das Rechnen mit Brüchen kann so deklariert werden:

```
class Bruch {
    int zaehler, nenner;
  public:
    Bruch(int z=0, int n=1);
    double wert();
    Bruch operator+(const Bruch& b);
    Bruch operator-(const Bruch& b);
    Bruch operator*(const Bruch& b);
    Bruch operator/(const Bruch& b);
    Bruch operator+=(const Bruch& b);
    // etc.
};
```

Die Elementfunktion **operator+** wird dann so definiert:

```
Bruch Bruch::operator+(const Bruch& b) {
  return Bruch(
     zaehler * b.nenner + nenner * b.zaehler,
     nenner * b.nenner);
}
```

Und der entsprechende Aufruf kann z.B. lauten

```
Bruch c = a.operator+(b);
```

Oder aber, in der schöneren (und symmetrischen!) Schreibweise:

```
Bruch c = a + b;
```

Fast alle Operatoren von C++ können durch Klassenoperationen *überladen* werden, d.h. **a OP b** wird als Kurznotation für **a.operatorOP(b)** verstanden.

In der Definition von **operator+** wird ein Konstruktor aufgerufen und damit ein neues **Bruch**-Objekt erzeugt.

Bei einem Operator **operator*=** ist dies nicht nötig, da der Wert des aktuellen Objektes überschrieben werden darf:

```
Bruch Bruch::operator*=(const Bruch& b) {
  zaehler *= b.zaehler;
  nenner  *= b.nenner;
  return ??;
}
```

Was ist aber der Rückgabewert?

Bei einem Aufruf **a *= b** erwartet man das (veränderte) **a** als Rückgabewert.

Wie spricht man nun aber **a** von innerhalb der Funktionsdefinition an?

a ist ja das *implizite Argument*, das beim Aufruf **a.operator*=(b)** (wofür **a *= b** Kurzschreibweise ist) an die Funktion mitgegeben wird.

Mit **zaehler** und **nenner** kann man zwar **a.zaehler** und **a.nenner** ansprechen, man möchte aber das Objekt als Ganzes.

Zu diesem Zweck stellt C++ einen Zeiger **this** zur Verfügung, der beim Methoden-Aufruf automatisch auf das aktuelle Objekt gesetzt wird.

Die gesuchte Anweisung heisst somit: **return *this;**

Für die Ein- und Ausgabe (z.B. von **cin** resp. auf **cout**) kann man die Operatoren **>>** und **<<** überladen.

Weil das erste Argument von **<<** aber vom Typ **ostream** ist, kann man *keine Elementfunktion* verwenden (dazu müsste die Klasse **ostream** erweitert werden), sondern man braucht eine *gewöhnliche Funktion:*

```
ostream& operator<<(ostream&, Bruch&);
```

Die Funktion braucht Zugriff auf die Klassen-Elemente **zaehler** und **nenner**, die aber **private** sind. Dieses Zugriffsrecht kann beibehalten werden, wenn man die Funktion **operator<<** zum **friend** der Klasse **Bruch** macht.

bruch.cpp

Friends

Eine Friend-Funktion wird im Klassenblock deklariert mit vorangestelltem Schlüsselwort **friend**. Im Beispiel:

```
class Bruch {
    int zaehler, nenner;
  public:
    // wie bisher
    friend ostream& operator<<(ostream&, Bruch&);
};
```

Die Friend-Funktion hat dann Zugriff auf alle, auch **private**, Elemente der Klasse.

Die Friend-Funktion wird dann ausserhalb der Klasse **Bruch** definiert:

```
ostream& operator<<(ostream& s, Bruch& b)
{
  s << b.zaehler << "/" << b.nenner;
  return s;
}
```

Die neue Funktion erlaubt die bequeme Ausgabe von Brüchen wie z.B. **cout << b << endl;**

Die Verkettung funktioniert korrekt, weil **operator<<** das erste Argument **s** wieder zurückgibt (der *Wert* des Ausdrucks **cout << b** ist **cout**), und weil der Operator linksassoziativ ist.

Strings

Ein schönes Beispiel für überladene Operatoren ist auch die Klasse **`string`** aus der Standardbibliothek.

Die Klasse enthält mehrere *Konstruktoren*. Die wichtigsten sind hier demonstriert:

```
#include <string> // statt cstring (C-Strings)
string s1("abc"); // String aus C-String
string s2(5,'z'); // erzeugt "zzzzz"
string s3(s2);    // Copy-Konstruktor
string s4;        // Default-K., leerer String
```

Die *Zuweisungsoperatoren* **=** und **+=** (String anhängen) sind ebenfalls überladen. Die rechte Seite kann ein **string**, ein C-String oder ein **char** sein. Mit

```
s += c;
```

wird z.B. ein Zeichen **c** an den String **s** angehängt.

Der Benützer von **string** muss sich nicht um Speicher-Allokation und -Freigabe kümmern. Dies wird von der Klasse selber gehandhabt.

Beim Anhängen an einen String reicht der allozierte Speicherplatz manchmal nicht aus, so dass kopiert werden muss. In diesem Fall wird ein Speicherbereich alloziert, der eine gewisse "Reserve" einschliesst.

Die *effektive* Länge eines Strings kann mit **`s.length()`** abgefragt werden, **`s.capacity()`** liefert dagegen die *allozierte* Länge (die aber selten von Interesse ist).

Für lexikographische Vergleiche wurden die Operatoren **`<`**, **`<=`**, **`==`**, **`!=`**, **`>=`** und **`>`** überladen.

Für den Zugriff auf das i-te Zeichen mittels **`s[i]`** wurde der Operator **`[]`** überladen.

Es existieren weitere nützliche Funktionen. Beispiele sind:

```
string s = "ABCDEFGH";
s.c_str();              // wandelt um in C-String
s.find("CD");           // liefert 2 (Position)
s.find("CC");           // liefert -1
s.substr(4,3);          // liefert EFG
```

Die folgenden Funktionen verändern `s`:

```
s.replace(4,3,"xy");// liefert ABCDxyH
s.erase(2,1);       // liefert ABDxyH
```

string.cpp

Für die Ein- und Ausgabe von Strings wurden die Operatoren **>>** und **<<** überladen, ähnlich wie im Beispiel **Bruch** gezeigt:

```
operator>>(istream&, string&);
operator<<(ostream&, string&);
```

Die für C-Strings existierende Elementfunktion **getline** von **istream** wurde in **string** nachgebildet durch eine gewöhnliche (Friend-)Funktion.

Statt

```
cin.getline(str, 10);
```

heisst es deshalb für **string**s:

```
getline(cin, str);
```

Vererbung

Ein mächtiges Instrument der objektorientierten Programmierung ist die *Vererbung*.

Von einer *Basisklasse* ausgehend, können neue Klassen *abgeleitet* werden, wobei Datenelemente und Elementfunktionen vererbt werden.

Dadurch wird die *Wiederverwendbarkeit* von Programmcode unterstützt.

Als Basisklasse wollen wir eine Klasse **Punkt** verwenden:

```
class Punkt {
  protected:
    double x, y;
  public:
    Punkt(double X=0, double Y=0) { x=X; y=Y; }
    void moveto(double X, double Y) { x=X; y=Y; }
    double abstand(Punkt B) {
        return sqrt(SQR(x-B.x) + SQR(y-B.y)); }
    friend ostream& operator<<(ostream&, Punkt&);
};

ostream& operator<<(ostream& s, Punkt& P)
{
    s << "(" << P.x << "," << P.y << ")";
    return s;
}
```

Wenn nun eine Klasse **Kreis** benötigt wird, kann man diese von **Punkt** ableiten, indem man ein Element **radius** sowie neue Elementfunktionen hinzufügt.

Der Vorteil ist: Die Methoden **moveto** und **abstand** können von der Basisklasse übernommen werden.

Die Syntax der Klassenableitung ist

```
Klassenart neue Klasse : Zugriffsrecht
    Basisklasse { neue Elemente };
```

Die abgeleitete Klasse **Kreis** könnte deshalb etwa so aussehen:

```
class Kreis : public Punkt {
    double radius;
  public:
    Kreis(double X=0, double Y=0, double R=1)
        : Punkt(X,Y) { radius = R; }
    bool istInnen(Punkt P)
        { return abstand(P) < radius; }
    friend ostream& operator<<(ostream&, Kreis&);
};

ostream& operator<<(ostream& s, Kreis& K)
{
    s << "Mittelpunkt (" << K.x << "," << K.y <<
        "), Radius = " << K.radius;
    return s;
}
```

punkt.cpp

Bemerkungen:

- Das Zugriffsrecht nach dem Doppelpunkt, darf fehlen. Es wird dann **`private`** angenommen. Es dient dazu, die Zugriffsrechte auf Elemente der Basisklasse *weiter einzuschränken*. Im Normalfall verwendet man **`public`**.
- Der *Konstruktor* von **`Kreis`** hat eine einzige Anweisung **`radius = R;`**.
- Es gilt aber: der Konstruktor ruft *zuerst implizit* den Konstruktor der Basisklasse auf (und dieser zuerst den der Basis-Basis-Klasse, etc.).
- Und wenn die Basisklasse mehrere Konstruktoren hat? Dann kann man den gewünschten Konstruktor-Aufruf nach der Parameterliste (und einem Doppelpunkt) angeben, wie im Beispiel gemacht.

- Analog dazu ruft der Destruktor *zuletzt* implizit den Destruktor der Basis-Klasse auf.
- Im Beispiel sind keine Destruktoren programmiert, daher werden in beiden Klassen die automatisch erzeugten Destruktoren verwendet.
- In **`istInnen`** wird die Methode **`abstand`** benützt, die von der Basisklasse zur Verfügung gestellt wird.
- In der überladenen Funktion **`operator<<`** werden die von **`Punkt`** geerbten Elemente **`x`** und **`y`** benützt. Das ist möglich, weil diese in **`Punkt`** als **`protected`** deklariert sind.

Objekte in einer Klassenhierarchie sind *polymorph:*
Ein **Kreis** kann immer auch als ein **Punkt** angeschaut werden.

Beispiel:

```
Punkt* arr[4];
arr[0] = new Punkt;  arr[1] = new Kreis;
arr[2] = new Kreis;  arr[3] = new Punkt;
for (int i = 0; i < 4; i++) arr[i]->print();
```

Hier sei **print()**eine zusätzliche Methode von **Punkt**, z.B.

```
void print() {
   cout << "Punkt: " << *this << endl;
}
```

basierend auf dem vorher definierten Operator **<<**.

Die Methode **print** wird an **Kreis** vererbt.

Möchte man stattdessen in **Kreis** die analoge Methode

```
void print() {
    cout << "Kreis: " << *this << endl;
}
```

definieren, dann muss die Basisklasse das *Überschreiben* der Methode erlauben.

Dies geschieht mit dem Schlüsselwort **virtual**:

```
virtual void print() {
    cout << "Punkt: " << *this << endl;
}
```

Eine *virtuelle Methode* wird nur dann vererbt, wenn die abgeleitete Klasse keine Methode mit der *gleichen Signatur* hat.

Im Gegensatz dazu wird eine *rein virtuelle Methode* gar nie vererbt.

- Sie wird nur deklariert, aber *nicht definiert*.
- Sie muss in *jeder* abgeleiteten Klasse definiert werden.
- Sie wird gekennzeichnet durch ein `= 0` in der Deklaration, z.B.:

```
virtual void print() = 0;
```

Eine Klasse, die eine rein virtuelle Methode enthält heisst *abstrakte Basisklasse*.

Beispiel: Man möchte neben Kreisen noch Klassen für weitere geometrische Figuren wie **`Rechteck`** etc. definieren.

- Dann ist **`Punkt`** keine geeignete Basisklasse.
- Besser ist eine abstrakte Basisklasse **`Figur`**.
- Methoden wie **`flaecheninhalt`** oder **`zeichneMich`** sind nun typische *rein virtuelle* Methoden, da sie für die Basisklasse nicht sinnvoll definiert werden können.
- Sie werden stattdessen für *alle* abgeleiteten Klassen definiert.

Von einer abstrakten Basisklasse darf keine *Instanz* gebildet werden. Das bedeutet: Es darf kein Objekt dieser Klasse deklariert werden!

Der Grund liegt darin, dass Objekte keine undefinierten Methoden haben dürfen.

Objekte dürfen also nur von abgeleiteten Klassen gebildet werden.

Dagegen sind *Zeiger* auf eine abstrakte Basisklasse erlaubt. Sie ermöglichen die einheitliche Behandlung von Objekten verschiedener abgeleiteter Klassen.

Beispiel:

```
Figur* figur[N];
// …
for (int i = 0; i < N; i++) {
     figur[i]->zeichneMich();
}
```

Welches ist nun die bei `figur[i]->zeichneMich();` aufgerufene Methode?

Dies hängt vom Typ von `figur[i]` ab, der *erst zur Laufzeit* feststeht. Der Compiler muss also den Aufruf einer noch nicht feststehenden Funktion vorsehen. Man spricht hier von *dynamischer Bindung*.

Funktions-Templates

Zum Vertauschen zweier Ganzzahlen definierten wir die Funktion

```
void swap(int& i, int& j)
     { int h = i; i = j; j = h; return; }
```

Mit einem *Funktions-Template* (einer "Funktions-Schablone") kann man jetzt vom Typ `int` abstrahieren:

```
template <typename Irgendwas>
void swap(Irgendwas& i, Irgendwas& j)
     { Irgendwas h = i; i = j; j = h;
                                 return; }
```

Zu beachten:

- Statt dem Schlüsselwort **typename** kann das veraltete (und verwirrende!) **class** verwendet werden.

Mögliche Aufrufe sind nun:

```
int   i=1, j=2;  swap(i, j);
float a=1, b=2;  swap(a, b);
Punkt A(3,4), B; swap(A, B);
```

Bemerkung: die Standardbibliothek enthält bereits ein solches Funktions-Template **swap**.
(Man unterscheide also **std::swap** und **::swap**).

swap.cpp

Klassen-Templates

Statt einer einzelnen Funktion kann auch eine ganze Klasse mit einem Datentyp parametrisiert werden. Dazu wird ein *Klassen-Template* benutzt.

Eine mathematische Anwendung wäre ein Klassen-Template **`Polynom`** für Polynome mit Koeffizienten aus beliebigen Datentypen.

Die Idee ist, dass hier **`float`** oder **`int`** ebenso verwendet werden können wie **`Bruch`** oder wie eine Klasse für komplexe Zahlen aus der Standardbibliothek.

Vorausgesetzt wird nur, dass die verwendete Klasse bestimmte Operationen besitzt, etwa **`+ - * /`** für das Beispiel **`Polynom`**.

Als einfaches Beispiel soll nun die Klasse **Bruch** zu einem Klassen-Template erweitert werden, das z.B. **short** oder **unsigned int** für Zähler und Nenner zulässt.

Die *Deklaration* geschieht wie beim Funktions-Template:

```
template <typename T>
class Bruch {
      T zaehler, nenner;
    public:
      Bruch(T z=0, T n=1);
      double wert();
      //…
};
```

Ein syntaktischer Unterschied ergibt sich aber beim *Gebrauch* der beiden Arten von Templates:

- Beim Funktions-Template wird der Typ auf Grund der aktuellen Parameter automatisch erkannt.
- Beim Klassen-Template geht dies nicht. Hier muss der Typ explizit angegeben werden.
  Beispiel:

```
Bruch<short> a, b(3), c(15,4);
Bruch<short> d = a;
```

Komplexe Zahlen

In ähnlicher Weise funktioniert ein Klassen-Template für komplexe Zahlen aus der Standardbibliothek.

Es unterstützt komplexe Zahlen über **`float`**, **`double`**, **`long double`** und **`int`**.

Beispiel:

```
#include <complex>
complex<float> i(0,1), n(-1);
cout << i*i    << endl;      // -1
cout << sqrt(n) << endl;     // +-i
```

complex.cpp

Arithmetik-Operatoren: **`+ - * /`**

Vergleichs-Operatoren: **`== !=`**

Ein-/Ausgabeoperatoren: **`>> <<`**

Elementfunktionen:

```
real()  // Realteil
imag()  // Imaginaerteil
norm()  // Betragsquadrat
abs()   // Betrag
arg()   // Argument
polar() // Polarform
```

Unterstützte Standardfunktionen:

```
sin() cos() tan() sinh() cosh() tanh()
sqrt() exp() log() log10()
```

Konstruktor, Zuweisungs- und Eingabeoperator akzeptieren auch *reelle* Zahlen.

Anwendungsbeispiel: Mandelbrot-Menge

```
#include <complex>
#include <ifmwindow>
using namespace std;

int main()
{
  const int    nMax     = 150;
  const double maxNorm  = 1e2;
  const int    width    = wio.xmax() - wio.xmin();
  const int    height   = wio.ymax() - wio.ymin();
  const int    maxColor = wio.number_of_colors()-2;

  double x0 = -2., y0 = -2., h = 4./width;
```

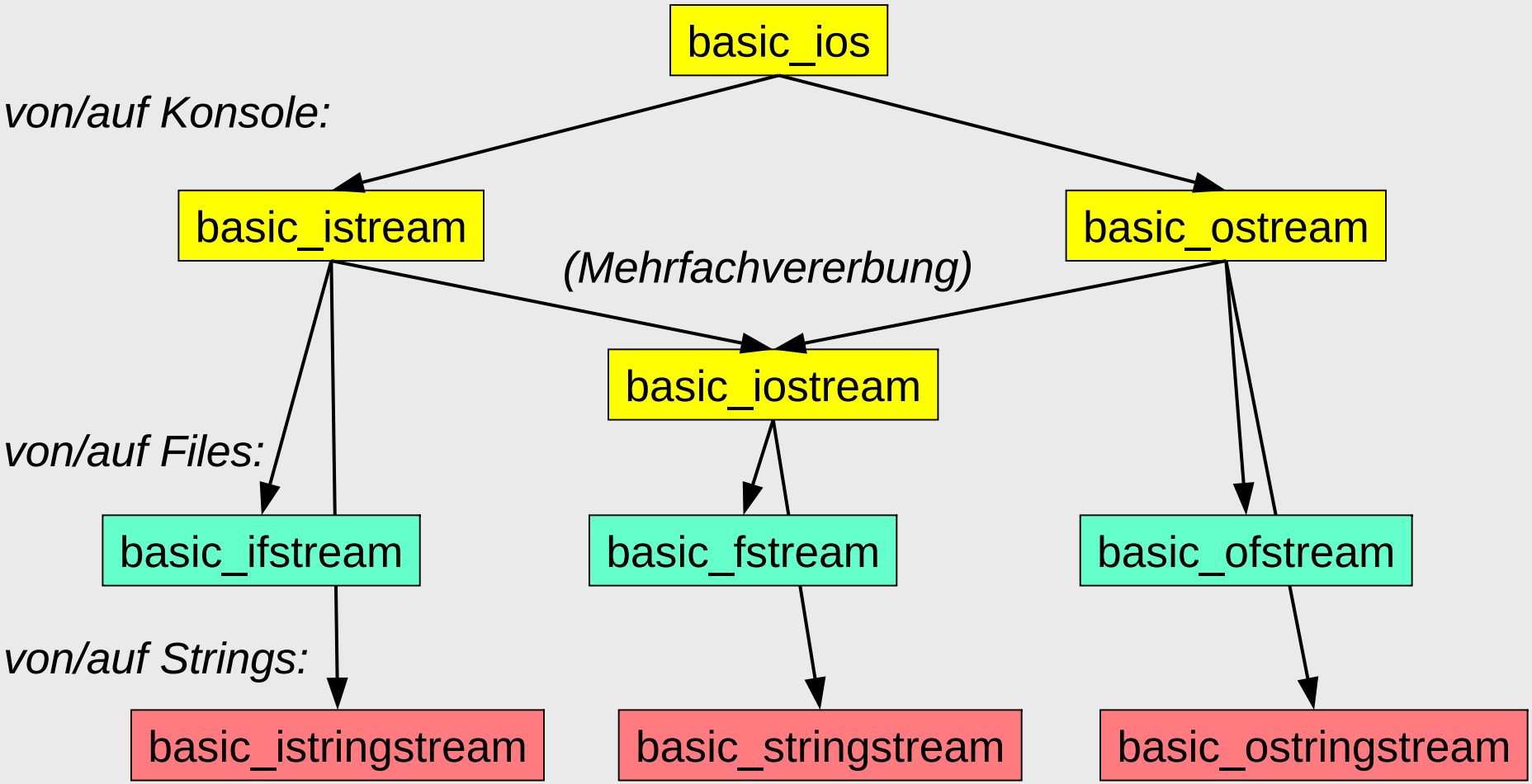
mandelbrot.cpp

```
while (true) {
  int col, row, n;
  for (row = 0; row <= height; row++) {
    for (col = 0; col <= width; col++) {
      complex<double> c(x0+col*h, y0+row*h),
                      z(0,0);

      for (n=0; n<nMax && norm(z)<maxNorm; n++) {
           z = z*z+c;
      }

      wio << color(n*maxColor / nMax)
          << Point(col,row);
    }
    wio << flush;
  }
```

```
    while (!wio.check_key() &&
           !wio.check_mouse_click());// wait for event
    if (wio.check_key()) break;

    int button = wio.get_mouse_click(col, row);

    switch (button) {
          // L: zoom in, M: set center, R: zoom out
      case 1: h /= 2.;  x0 += col*h;
                        y0 += row*h; break;
      case 2: x0 -= (width /2.-col)*h;
              y0 -= (height/2.-row)*h; break;
      case 3: x0 -= col*h;
              y0 -= row*h;  h *= 2.; break;
    }
  }
  return 0;
}
```

Beispiel: Ein-/Ausgabeströme

Die Standardbibliothek definert die folgende Hierarchie von (Template-) Klassen für Text-Ein/Ausgabe:



Alle Template-Klassen **basic_X** können für die beiden Datentypen **char** und **wchar_t** benutzt werden.

Dazu wurden die Abkürzungen definiert:

```
typedef basic_X<char>    X;
typedef basic_X<wchar_t> wX;
```

Also z.B.:

```
typedef basic_istream<char>    istream;
typedef basic_istream<wchar_t> wistream;
```

Die bereits bekannten **cin, cout, cerr** sind nun Variablen mit den Definitionen:

```
istream cin;
ostream cout, cerr;
wistream wcin;
wostream wcout, wcerr;
```

Analog können wir jetzt z.B. einen Strom für Texteingabe von einem File definieren:

```
fstream fin;
```

Die Operatoren und Methoden **>>, getline** etc. stehen zur Verfügung nachdem man das File “geöffnet” hat:

```
fin.open("meinText.txt", ios::in);
```

```
// Zaehle Woerter in Textfile
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  ifstream fin;
  fin.open("countWords.cpp", ios::in);
  if (!fin.good()) {
     cerr << "File not readable." << endl;
     return 1;
  }
  int count = 0;
  while (!fin.eof()) {
     string s;  fin >> s; count++;
  }
  count--; // abschliessendes 'newline' Zeichen
  cout << count << " words." << endl;
  return 0;
}
```

countWords.cpp

Numerische Parameter

Zur Parametrisierung einer Klasse kann auch ein *numerischer Wert* verwendet werden.

Beispiel: *Restklassen modulo n.*

```
template <int n>
class Restklasse {
  private:
    int wert;  // Wert der Restklasse
        // Normalisieren (interne Hilfsfkt.)
    Restklasse<n> normal() {
        if (wert %= n < 0) wert += n;
        return *this;
    }
```

```
  public:
        // Konstruktor:
    Restklasse<n> (int W=0) {
        wert=W; normal(); }
        // ueberladene Operatoren:
    Restklasse<n> operator+ <> (
        const Restklasse<n>& b) {
        return Restklasse<n>(wert+b.wert);
    }
        // etc.
        // Ausgabe-Operator:
    friend ostream& operator<< (
        ostream& s, Restklasse<n>&);
};
```

Testprogramm:

```
int main()
{
     Restklasse<10> a(5), b(8), c;
     c = a + b;  cout << c << endl;

     Restklasse<7> d(5), e(8), f;
     f = d + e;  cout << f << endl;

     // f = a + d; // Typen-Fehler!

     return 0;
}
```

restklasse.cpp

Container-Klassen

Eine wichtige Anwendung haben die Klassen-Templates in den sog. *Container-Klassen.*

Container-Klassen sind "Sammelbehälter" für Daten vom gleichen Typ, der aber nicht zum vornherein spezifiziert wird.

Die *Standardbibliothek* bietet Container-Klassen für Mengen, Stacks, Warteschlangen, dynamische Arrays etc., die bis vor kurzem noch in einer Extra-Bibliothek STL *(standard template library)* untergebracht waren.

Einige wichtige Container-Klassen sind:

Klasse	Beschreibung
`vector`	Wie Array, aber mit dynamischem Vergrössern/Verkleinern. Einfügen/Löschen am Ende in O(1), sonst in O(n).
`deque`	"double ended queue": Wie `vector`, aber veränderbar an beiden Enden. Einfügen/Löschen am Anfang/Ende in O(1), sonst in O(n).
`list`	Doppelt verkettete Liste. Einfügen/Löschen an beliebiger Position in O(1).
`set`	Enthält Elemente immer sortiert und nur je einmal.
`stack`	Hat nur Methoden `empty()`, `push()`, `pop()`, `top()` und `size()`.
`queue`	Erlaubt nur Zugriff auf erstes und letztes Element. Löschen nur vorne, Einfügen nur hinten.
`priority_queue`	Wie `queue`, aber sortiert nach `<`, mit grösstem Element zuvorderst.

Vektoren

Wir betrachten nun die spezielle Container-Klasse **vector**, die als verbesserte Form des Arrays benutzt werden kann.

Einen Vektor für 5 Ganzzahlen erzeugt man z.B. so:

```
#include <vector>
vector<int> v(5);
```

Beispiele mit weiteren Konstruktoren:

```
vector<char> u;              // leerer Vektor
vector<float> v(5, 1.0); // Anzahl und Wert
int a[] = {1,4,9,16,25};
vector<int> w(a, a+5);   // Vektor aus Array
```

vector.cpp

Auf die Elemente kann man mit dem überladenen Operator `[]` zugreifen. Beispiele:

```
v[2] = 2.0;
float f = v[j-1];
```

Zum Durchlaufen eines Vektors benutzt man aber besser die sog. *Iteratoren*.

Iteratoren sind Objekte verschiedener in der Header-Datei `#include <iterator>` definierter Klassen.

Dank den überladenen Operatoren `*` und `++` kann man Iteratoren praktisch wie *Zeiger* auf Vektor-Elemente verwenden.

Vorteil der Iteratoren: Zusammen mit den Methoden **`begin()`** und **`end()`** der jeweiligen Container-Klasse ermöglichen sie das sequentielle Durchlaufen des Containers mit einer **`for`**-Schleife.

Beispiel:

```
#include <vector>
#include <iterator>
vector<int> v = ...;
vector<int>::const_iterator i;
for (i = v.begin(); i != v.end(); i++) {
     cout << (*i) << " ";
}
```

Mit den Methoden **`rbegin()`**und **`rend()`**kann auch eine **`for`**-Schleife in umgekehrter Reihenfolge gebildet werden.

Durch Verwendung von Iteratoren vermeidet man das Überschreiten des Indexbereichs!

Iteratoren gibt es für *alle* Container-Klassen der Standardbibliothek.

Generisches Programmieren

Die Verwendung von Templates erlaubt es, von den Datentypen zu abstrahieren, was auch als *generisches Programmieren* bezeichnet wird.

Die Container-Klassen unterstützen diese Philosophie. Sie erlauben es, bereits implementierte Algorithmen auch für neue Klassen zu verwenden.

Weil die Container-Klassen zu einem grossen Teil identische Methoden und Funktionen haben, ist sogar der Container-Typ relativ leicht nachträglich austauschbar (z.B. gegen einen effizienteren oder einen flexibleren).

Die Container-Klassen **vector**, **deque**, **list** und **set** besitzen (unter anderem) alle die Methoden:

- `empty()` // Abfrage ob Container leer
- `size()` // aktuelle Grösse
- `max_size()` // maximal mögliche Grösse
- `begin()` // liefert Iterator auf erstes El.
- `end()` // … auf Position nach letztem El.
- `rbegin()` // liefert Rückwärts-Iterator
- `rend()` // liefert Rückwärts-Iterator

Die Klassen **vector**, **deque** und **list** haben zusätzlich:

```
• assign()      // bei Iterator Element ersetzen
• insert()      // ... Element einfügen
• erase()       // ... Element löschen
• clear()       // ganzen Container leeren
• push_back()   // Element hinten anhängen wobei
                // die Grösse des Containers
                // automatisch angepasst wird,
                // wenn nötig mit Re-Allokation
• pop_back()    // letztes Element löschen
```

Zusätzlich zu den Methoden gibt es auch viele Funktionen für Container-Klassen. Diese werden deklariert mit **`#include <algorithm>`**.

Eine kleine(!) Auswahl davon ist:

```
for_each()    // Funktionsaufruf mit jedem Element:
              // for_each(c.begin(),c.end(),drucke);
sort()        // sortiert Elemente mit Quicksort
stable_sort()// sortiert, unter Beibehaltung
              // der Reihenfolge bei Gleichheit
find()        // sequentielle Suche
binary_search() // binäre Suche, setzt Sortiertheit
                // voraus
merge()       // Mischen zweier sortierter Container
```

Weitere Bemerkungen:

Die Standardbibliothek bietet leider keine Container-Klasse für *binäre Bäume*.

Dafür existiert ein Container **`multimap`**, der Paare (Schlüssel, Wert) effizient verwalten kann. Er ist nicht als Baum, sondern als *Hash-Tabelle* implementiert.

Die *sortierten* Container funktionieren für beliebige Klassen, wo der (mit einer linearen Ordnungsrelation) überladene Operator **`<`** existiert.

Beispiel: Einen Vektor von Zahlen sortieren

```
vector<int> a;
while (…) { … ; a.push_back(…); }
a.sort(a.begin(), a.end());
```

Intern wird der Operator `<` verwendet.

Weil `int` keine Klasse ist, kann dieser nicht überladen werden. Man kann aber als drittes Argument von `sort` eine Vergleichsfunktion angeben:

```
bool myLess(const int& p, const int& q)
                { return p%10 < q%10; }
a.sort(a.begin(), a.end(), myLess);
```

sortNumbers.cpp

Zum Vergleich: Einen Vektor von **Person**en sortieren

Man kann genau gleich vorgehen:

```
vector<Person> a;
while (…) { … ; a.push_back(…); }
a.sort(a.begin(), a.end());
```

Es wird wiederum der Operator **<** verwendet, ein solcher wird also für **Person** vorausgesetzt.

Dieser kann jetzt nach eigenem Bedarf programmiert werden. Beispiel:

```
bool operator<(const Person& p, const Person& q)
    { return p.telNr < q.telNr; }
```

sortPersons.cpp

Mehrdimensionale Arrays

Einen 2D-Array (z.B. eine Matrix) kann man als Vektor von Vektoren auffassen.

Damit haben wir jetzt (endlich) mehrdimensionale dynamische Arrays.

Beispiel:

```
int m = ..., n = ...;
vector<float> v(n);
vector<vector<float> > A(m, v);
for (int i=0; i < m; i++)
   for (int j=0; j < n; j++) A[i][j] = i*j;
```

Die Matrix kann nachträglich in der Grösse verändert werden:

Eine Zeile anhängen:

```
A.push_back(v);
```

Eine Spalte anhängen:

```
for (int i = 0; i < m; i++)
    A[i].push_back(0.);
```

matrix.cpp

Ausnahmebehandlung

Der *Exception*-Mechanismus dient zum Abfangen von Fehlern oder anderen Ausnahmebedingungen.

Er besteht aus drei Teilen:

- Mit **`throw`** "wirft" man eine Ausnahme. Praktisch heisst dies: Man verlässt solange alle Schleifen und Funktionen (ähnlich wie mit **`break`** resp. **`return`**) bis man
- einen **`try`**-Block antrifft. Falls nun einer der
- dazugehörigen **`catch`**-Ausdrücke auf die Ausnahme passt, wird der entsprechende Block ausgeführt. Einen **`catch`**-Block nennt man *Handler*.

Ein Handler kann die Ausnahme nochmals werfen, damit sie von einem Handler auf einer höheren Stufe (weiter-) behandelt werden kann.

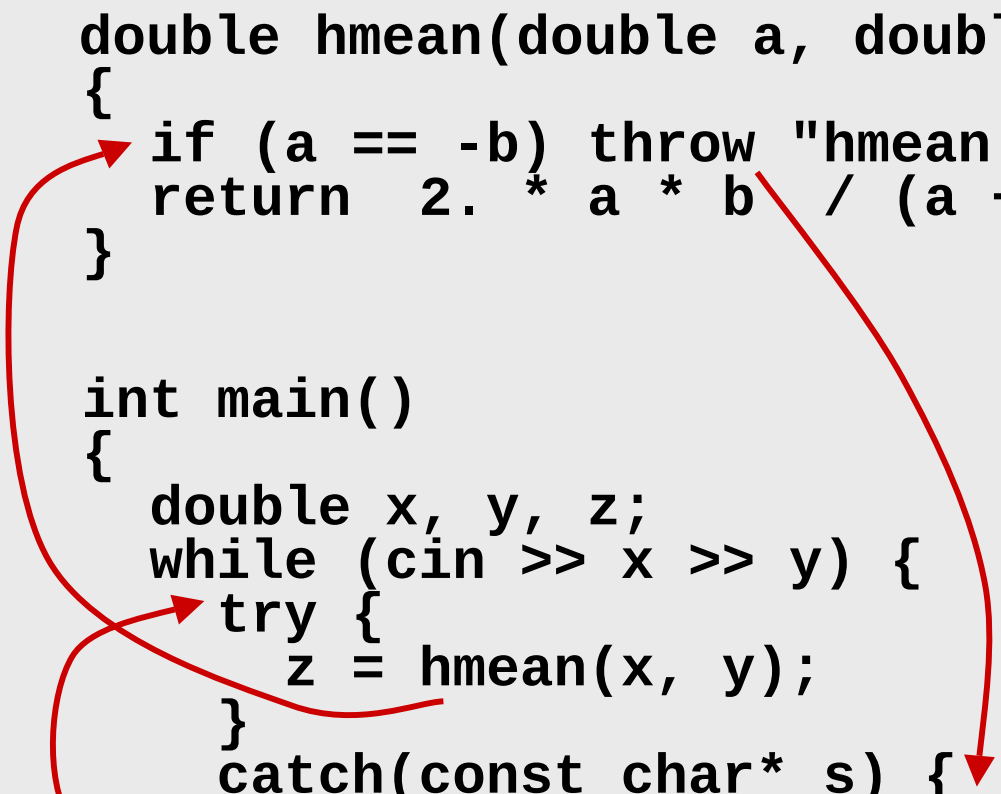
Die Syntax der Ausnahmebehandlung ist:

```
try {
  …
  throw Ausnahme ; // auch in Schleife, Fkt.
  …
}
catch ( Datentyp  Variable ) {
  // Ausnahmebehandlung
}
```

Der Ausdruck **Ausnahme** ist von beliebigem Datentyp. Falls dieser mit **Datentyp** übereinstimmt (oder davon abgeleitet ist!), wird der Handler ausgeführt.

Bemerkungen:

- Einem **`try`**-Block können mehrere **`catch`**-Blöcke (Handler) folgen.
- Ein Handler **`catch (...)`** fängt Ausnahmen von jedem Datentyp. Wichtig: Drei Punkte (keine Metasprache!).
- Ausnahmen für die kein passender Handler gefunden wird, lösen eine neue Ausname vom Typ **`bad_exception`** aus (die dann von einem Default-Handler aufgefangen wird).
- Ein geeigneter Datentyp sind *C-Strings*. Der dazu passende Handler beginnt mit: **`catch (const char* s)`**. Siehe folgendes Beispiel:



```
double hmean(double a, double b)
{
  if (a == -b) throw "hmean: a==-b nicht erlaubt";
  return  2. * a * b  / (a + b);
}


int main()
{
  double x, y, z;
  while (cin >> x >> y) {
    try {
      z = hmean(x, y);
    }
    catch(const char* s) {
      cout << s << "\nBitte 2 andere Zahlen!\n";
      continue;
    }
    cout << "Harmonisches Mittel: " << z
         << "\nBitte 2 neue Zahlen oder 'q'!\n";
  }
  return 0;
}
```

hmean.cpp

Neben C-Strings werden vor allem *Objekte* verwendet. Hierzu definiert man sogenannte *Fehlerklassen*.

Es ist sogar üblich mit *leeren* Fehlerklassen zu arbeiten. Die Information über die Art des Fehlers befindet sich dann nicht im Objekt, sondern in dessen Datentyp. Darum ist es erlaubt, nur `catch ( Datentyp )` statt `catch ( Datentyp Variable )` zu schreiben.

Hierarchische Fehlerklassen haben den Vorteil, dass man wahlweise einen allgemeinen oder mehrere spezielle Handler verwenden kann.

Das folgende Beispiel verdeutlicht dies:

```
// Berechnung von log(x) zur Basis y:
#include <iostream>
#include <math>
#include <climits>
using namespace std;

class Matherr{};                        // Basisklasse
class Singularity : public Matherr{}; // drei
class Base        : public Matherr{}; // abgel.
class Domain      : public Matherr{}; // Klassen

double logb(double x, double y) {
   if (x < 0)                    throw Domain();
   if (fabs(x) < eps)            throw Singularity();
   if (y<0||fabs(y)<eps||fabs(y-1)<eps)
                                 throw Base();
   return log(x)/log(y);
}
```

```
int main()
{
  try {
    double x, y;
    cout << "Argument: " ; cin >> x;
    cout << "Basis: " ;    cin >> y;
    cout << "log = " << logb(x,y) << endl;
  }
  //catch(Matherr)  { cout << "Fehler in logb!\n";}
  catch(Domain)     { cout << "Nicht im "
                         "Definitionsbereich!\n"; }
  catch(Singularity){ cout << "Singularitaet!\n"; }
  catch(Base)       { cout << "Nicht erlaubte "
                                "Basis!\n"; }
  catch(...)        { cout << "Andere Ausnahme\n";}
  return 0;
}
```

matherr.cpp

Die Klasse **`exception`** ist in **`#include <exception>`** vordefiniert. Sie enthält eine virtuelle Methode **`what()`** welche eine Fehlerbeschreibung liefert.

Davon abgeleitet ist z.B. die recht nützliche Fehlerklasse **`bad_alloc`** (definiert in **`#include <new>`**).

Ausnahmen dieser Klasse werden vom **`new`**-Operator geworfen, falls die gewünschte Allokation nicht möglich ist. Um den Programmabbruch zu vermeiden, kann man diese Ausnahmen abfangen und evtl. die Methode **`what()`** aufrufen.

Auch hierzu nochmals ein Beispiel:

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

int main(int argc, char** argv)
{
  if (argc != 2) {
    cerr << "Usage: a.exe nBytes\n"; exit(1);
  }
  int n = atoi(argv[1]);
  char* p;
  try { p = new char[n]; }
  catch(bad_alloc) {
    cout << "Soviel gibt's nicht\n"; }
  cout << "Adresse " << hex << (int)p << endl;
  return 0;
}
```

new.cpp

Danke für Ihr Interesse

und

viel Spass und Erfolg mit C++!