Studentische Mitschrift zur Vorlesung

Programmieren I
Einstieg in die Informatik mit C++

Die Vorlesung wurde im Sommersemester 1996 gehalten von Dr. G. Bohlender,
Institut für Angewandte Mathematik, Universität Karlsruhe

Studentische Mitschrift zur Vorlesung

Programmieren I
Einstieg in die Informatik mit C++

Institut für Angewandte Mathematik, Universität Karlsruhe

Zur Entstehung dieser Mitschrift: Die Vorlesung zur Programmiersprache C++ wurde am Institut für Angewandte Mathematik der Universität Karlsruhe erstmals von Dr. Rolf Hammer ausgearbeitet und dann mehrfach im Rahmen des Zyklus "Programmieren I – Einstieg in die Informatik" gehalten:

Semester		Vorlesung	Praktikum
SS	94	Dr. Rolf Hammer	Dipl.-Math. Tim Kersten
SS	95	Dr. Gerd Bohlender	Dipl.-Math. Andreas Wiethoff
WS	95/96	Priv.-Doz. Dr. Walter Krämer	Dipl.-Math. Andreas Wiethoff
SS	96	Dr. Gerd Bohlender	Dipl.-Math. Andreas Wiethoff
WS	96/97	Dr. Kurt Grüner	Dipl.-Math. Stefan Wedner
SS	97	Dr. Gerd Bohlender	Dipl.-Math. Stefan Wedner

Die Mitschrift wurde im Sommersemester 1996 von den Student/innen Martin Häfner[1] und Claudia Biegert[2] verfaßt.

Sie enthält nur den Tafelanschrieb, ohne alle Beispiele (diese sind in den Praktikumsrechnern und im Internet unter

`http://www.uni-karlsruhe.de/~Gerd.Bohlender/ss97.html`

verfügbar). Die Mitschrift wurde von Dr. Gerd Bohlender und Dipl.-Math. Stefan Wedner im Sommer 1997 korrigiert und um einige Tabellen ergänzt. Sie gibt aber nicht den Stoffumfang des aktuellen Semesters wieder.

Karlsruhe, den 30. 6. 97

[1] jetzt: Institut für Angewandte Physik, Universität Karlsruhe
[2] jetzt: Institut für Meteorologie und Klimaforschung, Universität Karlsruhe

Inhaltsverzeichnis

Kapitel 1

Einleitung

Teile der Vorlesung sind auf Folien gehalten worden. Hiervon existiert kein Mitschrieb.

- Algorithmen
- Backus–Naur–Form
- Syntaxdiagramme
- Syntaxdarstellung von Kernighan/Ritchie zur Beschreibung der Grammatik einer Programmiersprache
- Struktogramme
- Flußdiagramme

```
// Ein erstes C++ Programm
// ———————————————————
#include <iostream.h>
int main()
{
    cout << "Mein erstes C++ Programm" << endl;
    return (0);
}
```

Kapitel 2

Grundelemente eines C++ Programms

2.1 Kommentare

// Kommentar bis zum Zeilenende
/* Kommentar evtl. über mehrere Zeilen */

Kommentare können nicht beliebig geschachtelt werden.

Fehlerhafte Schachtelung:

```
/*
/* a = b+c;          */
   cout << a;
*/
```

Mögliche Schachtelung:

```
/*
   a = b+c;          // Addition
   cout << a;        // Ausgabe
*/
```

2.2 Bezeichner (Namen)

Bezeichner werden aufgebaut aus:

- Buchstaben (Klein- und Großbuchstaben werden unterschieden),
- _ (Unterstrich gehört zu den Buchstaben)
- Ziffern

Bezeichner beginnt mit Buchstabe oder _,
es folgen Buchstaben, Ziffern, _

Namen können beliebig lang sein. Sie dürfen aber keine Wortsymbole sein.

Bsp.: abc, Anfang, Ende, x1, _425, ...
verboten: Überlingen, Bad_Herrenalb, Dr._Mabuse, int
aber erlaubt: Int, INT

2.3 Trennzeichen

Leerzeichen
Zeilenendezeichen
Tabulator
Kommentar
Seitenvorschub
} "white space"

müssen stehen zwischen aufeinanderfolgenden Namen und Wortsymbolen.

int i,j; Das Komma ist hier kein Trennzeichen.

⟶ dürfen nicht innerhalb von Namen und Wortsymbolen stehen

2.4 Wortsymbole

asm	**auto**	bool	**break**	**case**
catch	**char**	class	**const**	const_cast
continue	**default**	delete	**do**	**double**
dynamic_cast	**else**	**enum**	**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				

ANSI C Schlüsselworte sind **fett** gedruckt.

Die Schlüsselwörter `bool, false` und `true` bzw. `namespace` und `using` zur Darstellung *boolescher* Werte bzw. zum Einrichten verschiedener Geltungsbereiche sind zur Zeit in vielen Compilern (auch HP-CC 10.01) noch nicht implementiert.

2.5 Interpunktionszeichen

( ) { } ; , ...

2.6 Operatoren

Beispiele:
+ - * /

<= >= < > != == (Vergleichsoperator)
(= ist eine Wertzuweisung!)

2.7 Header–Dateien

– Ein-/Ausgabe
– mathematische Standardfunktionen (sin, cos,...)

$\longrightarrow$ sind nicht im Kern von C++ enthalten
$\longrightarrow$ sind ausgelagert in Standardbibliotheken
$\longrightarrow$ müssen zugänglich gemacht werden mit einer #include–Direktive

z.B. #include <iostream.h> in 1. Spalte beginnend, alles in einer eigenen Zeile, am Anfang des Programms
#include <math.h>

Mathematische Standardfunktionen in math.h

Funktion	Bedeutung
sin(x)	Sinus
cos(x)	Cosinus
tan(x)	Tangens
asin(x)	Arcus Sinus
acos(x)	Arcus Cosinus
atan(x)	Arcus Tangens
atan2(y,x)	atan(y/x)
sinh(x)	Sinus Hyperbolicus
cosh(x)	Cosinus Hyperbolicus
tanh(x)	Tangens Hyperbolicus
exp(x)	Exponentialfunktion e^x
log(x)	natürlicher Logarithmus ln(x)
log10(x)	Logarithmus $\log_{10}(x)$ zur Basis 10
pow(x,y)	allgemeine Exponentiation x^y
sqrt(x)	Quadratwurzel $\sqrt{x}$
ceil(x)	kleinste ganze Zahl $\geq x$
floor(x)	größte ganze Zahl $\leq x$
fabs(x)	Absolutbetrag $\|x\|$
ldexp(x,n)	Skalierung $x * 2^n$
frexp(x, int *exp)	Mantisse von x (im Bereich $1/2 \leq$ frexp < 1); der Exponent (als Potenz von 2) wird in $*exp$ gespeichert
modf(x, double *ip)	Nachkommastellen von x; der ganzzahlige Teil wird in $*ip$ gespeichert
fmod(x,y)	Gleitpunktrest der Division x/y, mit dem gleichen Vorzeichen wie x

Die Argumente x und y sowie das Funktionsergebnis sind vom Typ *double*, das Argument n ist vom Typ *int*. Winkel werden im Bogenmaß angegeben.

2.8 Form eines Programms

Deklarationen

- Variablenvereinbarung
- Typdefinitionen
- Funktionsdefinitionen

Es muß genau eine Funktion namens main definiert sein, dies ist das Hauptprogramm.

Normaler Aufbau eines Programms:

```
#include<iostream.h>
// Definitionen (globale Variablen, Funktionen)
int main ()
  {
    // Definition von lokalen Variablen
    // Anweisungen
    return(0);
  }
// weitere Definitionen
```

Definition von Variablen:

```
int i, j, k;
```

Typische Anweisungen:

a) Wertzuweisung:	Variable = Ausdruck;
b) Eingabe:	cin >> i >> j >> k;
c) Ausgabe:	cout << "Ergebnis: " << i+j+k << endl;

Kapitel 3

Vordefinierte Datentypen

3.1 Ganzzahlige Typen

Systemabhängig (Beipiel: HP CC 10.01)

char	Zeichen (i.a. im ASCII–Code z.B. 'x')
short int	16 bit, d.h. -32768 bis 32767
int	32 bit
long int	32 bit

Präfix	signed	$\triangleq$	vorzeichenbehaftet (default)
	unsigned	$\triangleq$	vorzeichenlos
z.B.	char:		-128 ...127
	unsigned char:		0 ...255

Logische Werte

nach Standardentwurf: bool	false
	true

$\longrightarrow$ stehen bei vielen Compilern (z.B. HP CC 10.01) nicht zur Verfügung, stattdessen werden int verwendet:

0	$\triangleq$	false	
sonst	$\triangleq$	true	(in der Regel die eins)

Gleitkomma- und Pointerausdrücke dürfen ebenfalls als logische Ausdrücke interpretiert werden:

```
double x;
x = 1;
if (x) ...        (≙  if (x != 0.0) ...)
```

3.2 Gleitkommatypen

float
double
long double

Genauigkeit:	float	≤	double	≤ long double
	≥ 6 Stellen		10 St.	10 St.

3.3 sizeof–Operator

liefert Speicherbedarf eines Datentyps in Byte

Einheit: sizeof(char) = 1

Kapitel 4

Literalkonstanten

Wertebereich für den entsprechenden Datentyp

4.1 Ganzzahlige Konstanten

dezimal	Ziffernfolge, die nicht mit 0 beginnt		
oktal	0 gefolgt von oktaler Ziffernfolge (0 ...7): $012 = 1 \cdot 8^1 + 2 \cdot 8^0 = 10(dez)$		
hexadezimal	0x gefolgt von hexadezimaler Ziffernfolge (0 ...9, a, b, c, d, e, f, A,...,F)		
Suffixe	u,U: unsigned	123	mit Vorzeichen
		123u	ohne Vorzeichen
		-1u	$\hat{=}$ 65535 (in 16 bit–Darstellung)
	l,L: long	123	(BC++ 3.1: 16 bit HP CC 10.01: 32 bit)
		123l	(BC++ 3.1: 32 bit HP CC 10.01: 32 bit)

4.2 Gleitkommakonstanten

Eine Gleitkommazahl besteht aus:

- Mantisse (Ziffernfolge, gegebenenfalls Dezimalpunkt)
- Exponentenangabe (e bzw. E, ggf. Vorzeichen) — Exponent zur Basis 10

Beispiele:

$$
\begin{aligned}
1.23e5 &\triangleq 1,23 \cdot 10^5 \\
1e-3 &\triangleq 1 \cdot 10^{-3} \\
1.23 &\triangleq 1,23 \cdot 10^0 \\
.1 &\triangleq 0,1 \\
123. &\triangleq 123,0
\end{aligned}
$$

Suffix:

f,F	:	float
(kein)	:	double
l,L	:	long double

Beispiele:

12e3	Gleitkommazahl double
12e3f	Gleitkommazahl float
0x12e3, 0x12e3f	} ganze Zahl, hexadezimal

0x12e3f bedeutet: $1 \cdot 16^4 + 2 \cdot 16^3 + 14 \cdot 16^2 + 3 \cdot 16^1 + 15 \cdot 16^0$

4.3 Zeichenkonstanten

Zeichenkonstanten werden in Hochkommata geschrieben: 'a', '1', ...

ASCII Tabelle

hex		$00	$10	$20	$30	$40	$50	$60	$70
	dez	00	16	32	48	64	80	96	112
$00	00	NUL	DLE		0	@	P	‘	p
$01	01	SOH	DC1	!	1	A	Q	a	q
$02	02	STX	DC2	”	2	B	R	b	r
$03	03	ETX	DC3	#	3	C	S	c	s
$04	04	EOT	DC4	$	4	D	T	d	t
$05	05	ENQ	NAK	%	5	E	U	e	u
$06	06	ACK	SYN	&	6	F	V	f	v
$07	07	BEL	ETB	’	7	G	W	g	w
$08	08	BS	CAN	(	8	H	X	h	x
$09	09	HT	EM	)	9	I	Y	i	y
$0A	10	LF	SUB	*	:	J	Z	j	z
$0B	11	VT	ESC	+	;	K	[	k	{
$0C	12	FF	FS	,	<	L	\	l	|
$0D	13	CR	GS	–	=	M	]	m	}
$0E	14	SO	RS	.	>	N	^	n	~
$0F	15	SI	US	/	?	O	_	o	DEL

Zeichen-konst. in C	Zeichen	Bedeutung	Verwendung
`\a`	BEL	Bell	Signal, z.B. Pfeifton
`\b`	BS	Backspace	einen Schritt nach links gehen
`\r`	CR	Carriage Return	an Zeilenanfang gehen
	DEL	Delete	Zeichen löschen
	ESC	Escape	leitet Steuerbefehle ein, z.B. für Bildschirmsteuerung
`\f`	FF	Formfeed	Seitenvorschub, neue Seite
`\t`	HT	Horizontal Tab	Tabulator
`\n`	LF	Line Feed	Zeilenvorschub, neue Zeile
	NUL	Null	Füllzeichen ohne Bedeutung
`\v`	VT	Vertical Tab	vertikaler Tabulator
`\\`	\	Backslash	
`\?`	?	Fragezeichen	
`\'`	’	Anführungszeichen	
`\"`	"	Gänsefüßchen	
`\`*ooo*	*ooo*	oktale Zahl	Zeichen mit Wert *ooo* im Oktalsystem
`\x`*hh*	*hh*	hexadezimale Zahl	Zeichen mit Wert *hh* im Hexadezimalsystem

Beispiel: Wird der ASCII-Zeichensatz verwendet, so wird durch
```
cout << '\x51';
```
das Zeichen `Q` ausgegeben.

4.4 Zeichenketten

$\longrightarrow$ in doppelten Hochkommata: "beliebiger String"
Dieser kann mehrere Zeilen umfassen, z.B.

"Zeile1 \n Zeile 2 \n Zeile 3"

Strings (Zeichenketten) werden durch ein spezielles Zeichen, die ASCII–Null beendet, d.h. "abc" besteht aus den vier Zeichen 'a', 'b', 'c', '\0'. Zeilen werden durch \n getrennt.

Beispielprogramm

```
#include<iostream.h>
int main ()
  {
    int celsius, fahrenheit;
    cin >> celsius;
    fahrenheit = 1.8 * celsius + 32;
    cout << "Temperatur in Fahrenheit = " << fahrenheit << endl;
    return (0);
  }
```

endl bewirkt neben der Ausgabe des Zeilenendezeichens zusätzlich noch, daß der Ausgabepuffer geleert wird.

Kapitel 5

Variablen und symbolische Konstanten

5.1 Variablendefinition

Syntax: Typangabe Namensliste;
(mit Komma getrennt)

Beispiele:
```
char ch;
int i, j, k;
double d;
```

Unterscheidung Definition — Deklaration

Definition: Name, Typ, Speicherbedarf einer Größe werden festgelegt; Variable: Speicher wird reserviert; Funktion: Code wird angegeben.

Deklaration: nur Name und Typ werden festgelegt. (Variable: kein Speicher reserviert, Funktion: kein Code angegeben)

Jede Größe kann mehrfach deklariert werden, es muß aber genau eine Definition vorliegen. Eine Variablendeklaration wird durch *extern* markiert (Deklarationen von Funktionsprototypen siehe Abschnitt 11.3).

5.2 Initialisierung von Variablen

implizit: Name (Ausdruck)
explizit: Name = Ausdruck

```
int n=5, i, j, m=2*n;
int k (25);
```

Die Ausdrücke müssen aus Konstanten bzw. vorher deklarierten Variablen bestehen.

5.3 Symbolische Konstanten

zusätzliches Wortsymbol *const* bewirkt, daß der Wert der „Variablen" nicht verändert werden darf → Initialisierung ist erforderlich.

z.B.
```
const int n = 5;
const m = 6;                    //Default–Typ int
const char newline = '\n';
```

Beispiel:
```
const pi = 3.14159;             // liefert pi = 3
cout << pi/4;                   // ergibt 0
```

„/" ist hier die ganzzahlige Division und entspricht *div* bei Pascal.

Kapitel 6

Ausdrücke

6.1 Klassifizierung der Operatoren

- monadisch , dyadisch , triadisch
  1 2 3 Operanden
- Präfix- bzw. Postfix – Operatoren stehen vor bzw. nach dem Operanden
- Links- oder rechtsassoziative Operatoren
  $a \circ b \circ c = \begin{cases} (a \circ b) \circ c & \text{linksassoziativ} \\ a \circ (b \circ c) & \text{rechtsassoziativ} \end{cases}$
  rechtsassoziativ: monadische und Zuweisungsoperatoren
  linksassoziativ: andere dyadische Operatoren
- Prioritäten: 17 Stufen. Die Reihenfolge kann durch Klammern beeinflußt werden.

Liste der Operatoren in C++

Operator	Prio.	Ass.	Bedeutung
::	17	R	globaler Bezugsrahmen (unär)
::		L	lokaler Bezugsrahmen (binär)
$->$, .	16	L	Auswahloperatoren
[ ]		L	Indexoperator
( )		L	Funktionsaufruf und Konstr. eines Werts
++, $--$		R	Inkrement, Dekrement (Postfix: $x++$, $x--$)
`sizeof`		R	Größe eines Objekts (in "Bytes")
++, $--$	15	R	Inkrement, Dekrement (Präfix: $++x$, $--x$)
~		R	Einerkomplement (bitweise Negation)
!		R	logische Negation
+, $-$		R	Vorzeichen (unär)
$*$, &		R	Inhaltoperator, Adreßoperator
( )		R	Typkonversion, z.B. (`long int`) x
`new`, `delete`		R	dynamische Speicherverwaltung
$->*$, $.*$	14	L	Elementauswahl
$*$, /, %	13	L	multiplikative Operatoren
+, $-$	12	L	additive Operatoren (binär)
<<, >>	11	L	Shift-Operatoren, auch Ein-/Ausgabe
<, <=, >, >=	10	L	Vergleichsoperatoren
==, !=	9	L	Vergleich auf Gleichheit, Ungleichheit
&	8	L	bitweises Und
^	7	L	bitweises Exklusives Oder
|	6	L	bitweises Oder
&&	5	L	logisches Und
||	4	L	logisches Oder
?:	3	L	bedingter Ausdruck (ternär)
=,	2	R	Zuweisungsoperatoren
+=, $-=$,			$a+=b$ bedeutet $a=a+b$, wobei a nur
$*=$, /=, %=,			einmal ausgewertet wird, usw.
>>=, <<=,			
&=, ^=, |=			
,	1	L	Komma-Operator, z.B. Liste von Ausdrücken

Prio. = Priorität (17 = höchste, 1 = niederste), Operatoren innerhalb eines Kastens haben gleiche Priorität,

Ass. = Assoziativität (R = von rechts her, L = von links her), z.B. $a+b-c \;\widehat{=}\; (a+b)-c$ und $*p++ = *(p++)$, *nicht* $(*p)++$

6.2 Ganzzahlige Ausdrücke

monadisch: + -
dyadisch: + - * / %

/ entspricht *div* in Pascal
% entspricht *mod* in Pascal. Definition: a % b = a - (a/b) * b

Bei / oder % muß rechte Seite $\neq 0$ sein.

```
int i = 5, j = 2;
double q;
q = i / j;      // int–Division 5 / 2 = 2, wird umgewandelt in 2.0
```

Bei Operatoren werden nur die Typen der Operanden beachtet!

6.2.1 Inkrement- und Dekrementoperatoren

++ : erhöhe den Operanden um 1
−− : erniedrige den Operanden um 1

++, −− gibt es als Präfix- und Postfix–Operatoren.

Präfixoperator:

1. der Operand wird um 1 erhöht oder reduziert
2. die veränderte Variable wird im Ausdruck weiterverwendet

Postfixoperator:

1. der alte Wert wird im Ausdruck verwendet
2. nachträglich wird der Operand um 1 erhöht oder reduziert

Postfixoperator gibt Wert, Präfixoperator Referenz zurück (vgl. Abschnitt 10.2).

Effekt: Verwendung im Ausdruck
Seiteneffekt: Wert der Operanden verändert

Beispiele:

```
int i = 5, j, k;
i++;          // i = i + 1    ⟹   i = 6
++i;          // i = i + 1    ⟹   i = 7
j = ++i;      // i = i + 1 ; j = i   ⟹   i = 8 ; j = 8
k = i++;      // k = i ; i = i + 1   ⟹   k = 8 ; i = 9
```

6.2.2 Bitmanipulation

a, b ganzzahlige Ausdrücke

a & b	bitweises Und
a | b	bitweises Oder
a ^ b	bitweises exklusives Oder
~a	bitweise Negation
a << b	a wird um b Stellen nach *links* geschoben
a >> b	a wird um b Stellen nach *rechts* geschoben

Operanden sollten unsigned sein, sonst z.T. implementierungsabhängige Ergebnisse.

Beispiel: (8-bit Darstellung)

a = 29_{dez}	=	00011101_{bin}			
b = 3_{dez}	=	00000011_{bin}			
a & b	=	00000001_{bin}	=	1_{dez}	
a | b	=	00011111_{bin}	=	31_{dez}	
~a	=	11100010_{bin}	=	226_{dez}	
a << b	=	11101000_{bin}	=	232_{dez}	$(= a \cdot 2^b)$
a >> b	=	00000011_{bin}	=	3_{dez}	$(= a/2^b)$

Beachte:
```
cout << a << 2;        // mit a=5 erhält man 52
cout << (a << 2);      // ergibt 20
```

6.2.3 Zeichenausdrücke

Umwandlung Zahlen — Zeichen (je nach Zeichensatz, i.A. ASCII)

```
char ch;
```

```
int i;
ch = 97;                    // 'a' (Pascal: ch:=chr(97);)
ch++;                       // 'b'
i = ch + 2;                 // i = 100
cout << i << ch;            // 100b
cout << char(i);            // d
```

6.3 Gleitkommaausdrücke

```
+     -     *     /
++    --
```

Standardfunktionen: fabs, sin, cos, sqrt, ...
$\Longrightarrow$ am Programmanfang #include<math.h>

6.4 Logische Ausdrücke

Jeder arithmetische oder Pointerausdruck kann als logischer Ausdruck interpretiert werden:

$0 \triangleq$ falsch
$\neq 0 \triangleq$ wahr (i.d.R. „1")

Speicherung in *int*–Variablen.

6.4.1 Logische Operatoren

a && b	logisches Und
a || b	logisches Oder
!a	logische Negation

Ein logischer Ausdruck wird (unter Beachtung der Priorität) *von links nach rechts* ausgewertet, bis das Ergebnis feststeht, restliche Operanden werden ggf. nicht ausgewertet.

Beispiel:

```
int i = 0, j = 1, k;
k = i && j++;          // i ist immer „falsch"
```

$\Longrightarrow$ rechte Seite wird nicht ausgeführt, j wird nicht erhöht.

6.4.2 Vergleichsoperatoren

```
<      <=      >      >=
==          Gleichheit
!=          Ungleichheit
```

Fehlerquellen:

1. = statt ==
   ...if (i=1) cout << " ist = 1"...
   i = 1 ist eine Wertzuweisung, dies ist logischer Ausdruck mit Wert 1, d.h. cout...wird immer ausgeführt.

2. Vergleichsoperatoren sind linksassoziativ

   ```
   int i, j, k = 5;
   ⋮
   if (i <= j <= k) cout << " immer wahr";...
   ```

   i und j sind wahr oder falsch, Ergebnis von (i <= j) ist wahr(1) oder falsch(0) und wird mit k verglichen.

   Beispiel:

   ```
   if (0 <= i <= 9) cout << " i ist Ziffer";...   // liefert falsches Ergebnis
   if (0 <= i && i <= 9) cout << ...              // richtig
   ```

6.5 Konditionaloperator

$\Longrightarrow$ ternärer Operator

Syntax: *Bedingung* ? *Ausdruck1* : *Ausdruck2*

Wirkung: wenn Bedingung wahr ist, ist Ausdruck1 das Ergebnis, sonst Ausdruck2

z.B. `max = (a>b) ? a : b;`

6.6 Zuweisungsoperator

= ist ein Operator, Ergebnis ist der Wert der rechten Seite

Assoziativität *von rechts nach links*

Beispiel: $x = \underbrace{y = 5}_{5}$

Effekt: Wert der rechten Seite wird als Wert des Ausdrucks berechnet (meistens unwichtig).

Seiteneffekt: linke Seite bekommt diesen Wert zugewiesen (wichtig).

Abschreckendes Beispiel:

```
z = (x = (y = 5) * 3) + 20          // y = 5, x = 15, z = 35
```

6.6.1 Kombinierte Zuweisungsoperatoren

Kombination von einem Operator ∘ (einer von + − * / % & | ^ << >>) mit Zuweisungsoperator

a ∘= b (a = a ∘ b)

Beispiele:

```
a += 1;      // a = a+1
b += 27;     // b = b+27
c -= 30;     // c = c-30
a *=2;       // a = a*2
```

6.7 Komma–Operator

Syntax: Ausdruck1, Ausdruck2, ...,Ausdruck n

Wirkung: Die Ausdrücke werden der Reihe nach ausgewertet, das Endergebnis ist der Wert des letzten Ausdrucks

Beispiel:

```
hilf = x, x = y, y = hilf;      //eine Anweisung
```

bewirkt: Vertauschung von x und y

```
hilf = x; x = y; y = hilf;      //drei Anweisungen
```

Vorsicht: $\underbrace{\underbrace{x = 3}_{3}, \underbrace{5}_{5}}_{5}$; //x = 3.0

6.8 Typumwandlungen

ohne Datenverlust kann konvertiert werden:

short int	→	int	→	long int
unsigned short int	→	unsigned int	→	unsigned long int
float	→	double	→	long double

Konversion mit (eventuellem) Datenverlust:

ganze Zahlen	→	Gleitkommazahlen	
Gleitkommazahlen	→	ganze Zahlen	(Abschneiden der Nachkommastellen)

z.B. int i;
i = 5.9; // i = 5
i = -0.6; // i = 0

char wird als Teilbereich der ganzen Zahlen dargestellt:
cout << 'a' << 'a' + 1; // a98

Explizite Umwandlungen:

Syntax: Typname (Ausdruck)
($\underbrace{\text{Typspezifikation}}_{\text{Typecast–Operator}}$) Ausdruck

Beispiel 1:
```
cout << 'a' << char('a'+1);    // ab
cout << 'a' << (char)('a'+1);
```

Beispiel 2:
```
x = (long int) 27;             // hier: ≙ 27l, gilt nicht generell
x = (long int) 70000;          // Vorsicht!
x = (short int) 5.9;           // wird abgeschnitten
```

Kapitel 7

Anweisungen

7.1 Ausdrucksanweisung

Syntax: Ausdruck;

Wirkung: Ausdruck wird ausgewertet, dabei treten i.a. Seiteneffekte auf

Beispiel:
```
i = 27;
i++;
i++, j++;
1;                  // kein Effekt
sin(2.0)*3;         // kein Effekt
```

7.2 Verbundanweisung

Zusammenfassung mehrerer Anweisungen zu einer Anweisung

{Anweisung 1 ... Anweisung n}

z.B. {x = 1; y = 2; z = 3;}

Bemerkung: Zu einer Ausdrucksanweisung gehört immer ein Semikolon.

7.3 Bedingte Anweisung (if–Anweisung)

Syntax: if (Bedingung) Anweisung
if (Bedingung) Anweisung1 else Anweisung2

Wirkung: Bedingung wird ausgewertet; wenn Ergebnis $\neq 0$ ist, wird die Anweisung (bzw. Anweisung1) ausgeführt; wenn Ergebnis $= 0$ ist, dann wird nichts (bzw. Anweisung2) ausgeführt.

Beispiel:
```
if (x>y)
   max = x;
else
   max = y;
```

mehrere Anweisungen mittels Verbundanweisung: { ... }

Frage: Wie wird *else* zu *if* zugeordnet ?

```
if (x>6)
   if (x>10) cout << "groß";
   else cout << "x zwischen 6 und 10";
```

Regel: *else* wird immer dem nächsten davorstehenden *if* zugeordnet; die Zuordnung kann durch Klammerung geändert werden

```
if (x>5)
   { if (x>10) cout << "groß";}
else ...
```

7.4 Auswahlanweisung (switch)

Syntax:

```
switch (Auswahlausdruck)
{ case konstanter Ausdruck 1: Anweisungsliste 1
  case konstanter Ausdruck 2: Anweisungsliste 2
    ⋮
  default: Anweisungsliste    // kann entfallen
}
```

Wirkung:

- Auswahlausdruck wird ausgewertet
- Sprung zur entsprechenden Auswahlmarke bzw. zum default–Zweig
- die entsprechende Anweisungsliste wird abgearbeitet
- in der Regel ist die letzte Anweisung in der Liste eine break–Anweisung, sie bewirkt das Verlassen der switch–Anweisung
- ohne break wird die sequentiell nächste Anweisungsliste abgearbeitet

Bemerkungen

- der Auswahlausdruck muß ganzzahlig sein (oder in ganze Zahlen konvertiert werden können)
- die konstanten Ausdrücke zur Markierung der Fälle müssen sich unterscheiden
- mit break kann die switch–Anweisung verlassen werden

Beispiel:

```
switch (ch)
{ case 'a': cout << "a ist ch";
            break;
  case 'b': case 'c': cout << "ch = b ober c";
}
```

7.5 for–Anweisung

Syntax: for (Init–Anweisung Ausdruck1; Ausdruck2) Anweisung

Init–Anweisung: Ausdrucks- oder Deklarationsanweisung (enden mit Semikolon)

insgesamt immer 2 Semikoli

Beispiel:
```
int i;
for (i=0; i<5; i++) cout << i;      // 01234
for (int i=0; i<5; i++) cout << i;  // 01234
```

Wirkung:
- die Init–Anweisung wird einmal ausgeführt
- solange Ausdruck1 $\neq 0$ ist, wiederhole Anweisung und Ausdruck2

Bemerkung: wenn Ausdruck1 fehlt, gilt Bedingung als immer erfüllt
$\Rightarrow$ Endlosschleife (falls keine Sprunganweisung benutzt wird)

Initialisierungen:
- for (i=0, j=0, k=0; i<5; i++) ...
- for (int n=0; n<=20; n++) ...
- for (i=j=k=0; ...) ...

7.6 while–Anweisung

Syntax: while (Bedingung) $\underbrace{\text{Anweisung}}$
z.B. Ausdruck; oder { }

Wirkung: solange Bedingung $\neq 0$ ist wird die Anweisung wiederholt

Entspricht: for (;Bedingung;) Anweisung

Beispiel: Quersumme einer Zahl berechnen

```
#include <iostream.h>
int main()
{
  int n, Summe=0;
  cout << "Bitte Zahl eingeben: ";
  cin >> n;
  while (n>0)
```

```
    {
      Summe += n % 10;
      n /= 10;
    }
    cout << "Quersumme = " << Summe << endl;
    return 0;
}
```

7.7 do–Anweisung

Syntax: do Anweisung while (Bedingung);

Wirkung: Anweisung wird wiederholt, solange Bedingung wahr ist ($\neq 0$)

Beispiel: positive Zahl einlesen

```
do
{
  cout << "Zahl > 0 eingeben: ";
  cin >> n;
}
while (n <= 0);    // äquivalent while (!(n > 0));
```

7.8 Sprunganweisungen

Syntax:
```
break;
continue;
return Ausdruck_opt;    // siehe Funktion (Abschnitt 10)
goto Marke;
```

```
goto M;
   ⋮
M: cout << "Hier geht es weiter.";
```

7.8.1 break–Anweisung

Die break–Anweisung steht innerhalb von switch oder von Schleifen (for, while, do).

Wirkung: switch–Anweisung bzw. Schleife wird sofort verlassen

Beispiel: Abbruch einer Schleife bei illegalen Eingaben

```
for (i=1; i<=10; i++)
{
  cin >> x;
  if (x<0) break;
  cout << "Quadratwurzel = " << sqrt(x) << endl;
}
//hier geht es nach break weiter
```

7.8.2 continue–Anweisung

- darf in Schleife stehen (for, while, do)
- bewirkt, daß Schleife sofort wiederholt wird (bei for wird Ausdr.2 ausgewertet)

Bsp.:
```
for (i=1; i<=10; i++)
{
  cin >> x;
  if (x<0) continue;
  cout << "Quadratwurzel = " << sqrt(x) << endl;
  // hier geht es weiter nach continue
  // bevor Schleife neu durchlaufen wird,
} // wird Ausdruck2 ausgewertet.
```

Kapitel 8

Abgeleitete Datentypen

8.1 Felder

$\longrightarrow$ feste Anzahl von Komponenten gleichen Typs

8.1.1 Deklaration

innerhalb Variablenliste:

Syntax: Bezeichner [konst.Ausdruck 1] ... [konst.Ausdruck n]
ganzzahliger Ausdruck,
n = Dimension des Feldes

Beispiel: int i, j, feld[10], k;

Feld mit 10 Komponenten mit Indizes
0 bis 9, d.h. feld[10] existiert nicht!

Der Compiler prüft das in der Regel nicht!

Matrizen:

double A[5][5]; // 5 $\times$ 5 Matrix

Speicherung von Matrizen erfolgt zeilenweise (wie in Pascal, nicht wie in Fortran)

Bezeichnung: Typspezifikation Bezeichner [konst.Ausdr.1] ... [konst.Ausdr.n]
$\longrightarrow$Bezeichner vom Typ Typspezifikation [k.A. 1] ... [k.A. n]

z.B. int m[2][2]
m ist vom Typ int[2][2]

8.1.2 Zugriff auf Feldkomponenten

Indexangaben in [ ], für jeden Index ein eigenes Klammerpaar

```
int ein[19], zwei[2][2];

ein[5] = 23;
zwei[1][1] = 24;
```

Achtung:
```
cout << zwei[0,1]; // ausgegeben wird zwei[1]
// äquivalent zu
cout << zwei[1];   // zwei[1] ist Pointer auf zweite Zeile der Matrix
```

Generell bei Matrizen immer: Feldname [Zeilenindex][Spaltenindex]

Statische Felder werden immer ab 0 indiziert, das heißt mathematische Algorithmen mit Indexbereich 1 bis n müssen umformuliert werden auf Indexbereich 0 bis n−1

Abhilfe: Objektorientierte Programmierung in C++, Überladen des Indexoperators [ ]

8.1.3 Initialisierung

Bei Deklaration Angabe von konstanten Ausdrücken, mit denen die Feldkomponenten initialisiert werden.

Syntax: Bezeichner [Indexbereich] = {Ausdruck 1,..., Ausdruck n}

z.B. `int feld[4] = {27,3,5,18}` // d.h. feld[0]=27, feld[1]=3,...

Wenn zu wenige Werte angegeben werden, so werden die restlichen Feldkomponenten mit Nullen aufgefüllt. Zu viele Angaben führen zu einem Fehler.

Mehrdimensionale Felder:

$\longrightarrow$ alternativ geschachtelte { } verwenden

z.B.: double m[2][2] = {{5,3.1},{4.5,7}} $\triangleq$ $\begin{pmatrix} 5 & 3.1 \\ 4.5 & 7 \end{pmatrix}$

Bei Angabe einer Initialisierungsliste kann der erste Indexbereich entfallen.

z.B.: int a[ ] = {1,2,3,4,5}
$\triangleq$ Feld mit 5 Komponenten, d.h. a[5]

bei Matrizen:

```
double b[ ][2] = {{1,2},{3,4}}
```
$\Longrightarrow$ 2×2 Matrix mit b[0][0] = 1
b[0][1] = 2
b[1][0] = 3
b[1][1] = 4

Felder können nicht *am Stück* kopiert werden per Wertzuweisung, sondern nur per Schleife komponentenweise, d.h.:

```
int a[5], b[5];
a = b;                  // Fehler: a ist konstanter Zeiger auf Feldanfang
for (int i=0;i<5;i++) a[i] = b[i];          // komponentenweise
```

8.2 Zeichenfelder (Strings)

bekannt: Konstanten "Zeichenkette"
werden abgeschlossen mit zusätzlichen Zeichen \0

Stringvariable: Felder von Zeichen

```
char s[10] = {'a','b','c','\0'};
char n[10] = {'a','b','\0'};
char str1[5], str2[5];
char name[ ] = "Willi";
```
hat Länge 6 (unsichtbares \0)

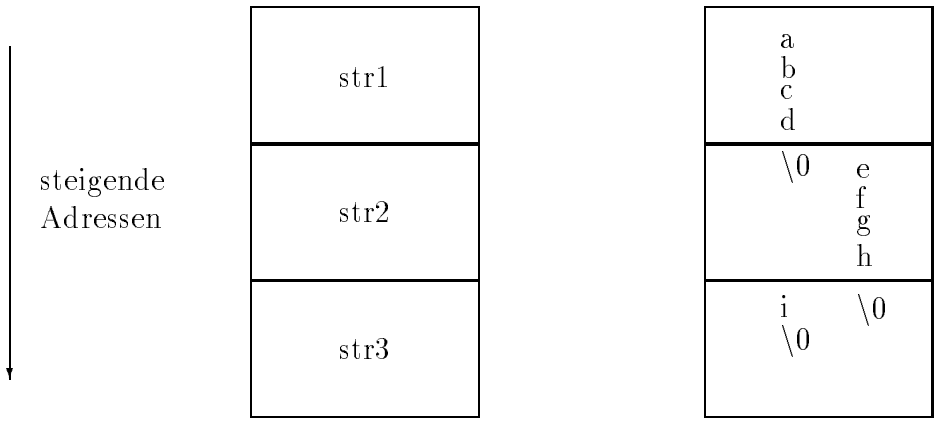
Ein- /Ausgabe mit cin >>..., cout <<...

Wertzuweisung: nicht im Sprachkern
Abhilfe:
- zeichenweise kopieren in Schleife
- Bibliotheksfunktion *strcpy* aus *string.h*

Warnung: Wenn Stringlänge überschritten wird, erfolgt im allgemeinen keine Fehlermeldung, statt dessen wird auf die folgenden Speicherzellen zugegriffen (→ überschrieben)

Beispiel: (HP CC 10.01)

```
char str1[4], str2[4], str3[4];
cin >> str1 >> str3 >> str2;
cout << str1 << endl;
cout << str3 << endl;
cout << str2 << endl;
```



Eingabe: abcd
i
efgh
Ausgabe: abcdefgh
(leerer String)
efgh

8.3 Aufzählungstypen

Syntax: enum Typbezeichner$_{opt}$ {Namensliste} Variablenliste$_{opt}$

Beispiel: enum Anrede {Herr, Frau, Firma};
Anrede $\underbrace{\text{a,b,c}}$;
Variablen vom Typ Anrede

Standardmäßig werden die angegebenen Namen auf ganze Zahlen ab 0 abgebildet. Die Konstanten können wahlweise explizit angegeben werden.

Beispiel: enum Farbe {rot, gelb=2, gruen, blau=6, lila=2};

Mehrfach gleicher Wert und Lücken erlaubt.

Konversion:	enum	$\longrightarrow$	int	:	implizit
	int	$\longrightarrow$	enum	:	explizit

Typecastoperator

```
Farbe f;
int i;
f = gelb;
i = f;              // i = 2
f = (Farbe) 6; //f = blau, intern 6
if (gelb == lila) cout << "ist gleich";
cout << f;      // wird als int ausgegeben
switch (f)
{
  case rot : cout ...
  case gelb: cout ...
}
```

8.4 Typdefinition mit typedef

Syntax: typedef Typspezifikation Typnamen;

Beispiel:
```
typedef long double real;
real a, b, c, matrix[5][5];
```

$\longrightarrow$ sinnvoll in Zusammenhang mit Feldern und Pointern

```
double v1[10], v2[10];
```

alternativ: typedef double $\underbrace{\text{Vektor[10]}}$;
Typname statt Variablenname

Kapitel 9

Pointer (Zeiger)

Pointer = Adressen von Größen (Variablen, symbolische Konstanten, Funktionen)

referenziertes Objekt = die Größe, auf die Pointer zeigt; Anzahl der Bytes hängt vom Typ des Pointers ab

9.1 Deklaration von Pointern

Syntax: Typspezifikation * Bezeichner

Beispiel:
```
int *p ;          // p ist Pointer, der auf int zeigen
                  // darf; referenzierter Typ: int
    ^^^^^
referenzierte Größe, int

int *p, q;        // * gilt nur für nächste Variable in Liste.
                  // d.h. p ist Pointer, q ist int

int *p, *q;       // zwei Pointer

double *d;        // d darf auf double-Variable zeigen

typedef int* intptr;   // Typname
intptr x,y,z;          // 3 Pointer
```

9.2 Wertzuweisung, Initialisierung, Vergleich

Wertzuweisung:

a) (andere) Pointervariable mit gleichem referenziertem Typ

b) NULL oder ganzzahliger Ausdruck mit Wert 0

c) Konversion von ganzzahliger Zahl nach Pointer mit *typecast*;
z.B. int *p = (int*) 0x1234;

d) Beschaffen der Adresse von Größen mittels Adressoperator &

e) Beschaffung von „dynamischem" Speicher mittels *new* (siehe unten)

Adressoperator & liefert die Adresse einer Größe:

Beispiel:
```
int i;          double d;
int* ip;        double *dp;
ip = &i;        dp = &d;

ip = &d;        dp = &i;        // verboten !
ip = NULL;      dp = 0;         // erlaubt
```

Vergleiche: mit ==, !=
erlaubt, wenn referenzierte Typen übereinstimmen (bzw. Vergleich mit NULL, 0)

9.3 Dereferenzierung

Syntax:
$$\underbrace{\texttt{* Pointervariable}}_{\text{referenzierte Größe}}$$

Beispiel:
```
int i, *p, **q;
i = 1;
p = &i;
cout << *p;     // hier wird 1 ausgegeben
*p = 3;
cout << i;      // ergibt 3

q = &p;         // q erhält Adresse des Pointers p
cout << **q;    // ergibt 3
```

9.4 Pointer–Arithmetik

Der Wert eines Pointers darf verändert werden durch Addition/Subtraktion einer ganzzahligen Größe n, dabei wird n mit der Bytezahl des referenzierten Objekts multipliziert (sizeof (ref. Typ)). Die Größe der Datentypen ist systemabhängig.
Bsp.: HP-CC 10.01

```
int         4 Bytes
long int    4 Bytes
float       4 Bytes
double      8 Bytes
```

Bsp.:
```
int *ip;
char *cp;
double *dp;
ip++;          // ip wird um 1*4 erhöht
cp -= 5;       // cp wird um 5*1 verringert
dp--;          // dp wird um 1*8 verringert
```

zusätzlich: Differenz von Pointern

```
int *p, *q;
p = (int *) 0X100;
q = (int *) 0X108;
cout << q - p; // 2 (weil int 4 byte groß ist)
```

Pointer als logische Ausdrücke
(NULL $\triangleq$ 0 $\triangleq$ false; sonst $\triangleq$ true)

```
int *p;
if (p) cout << "p ist nicht NULL";
```

9.5 Pointer und Felder

Der Wert einer Feldvariablen ist ein Pointer auf erste Feldkomponente.

Sei T ein beliebiger Typ, dann vereinbaren wir:

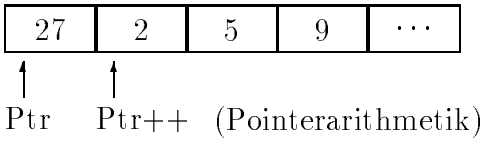
```
T x[10], y;
T* p;
```

äquivalent:

p = & x[0]	$\hat{=}$	p = x;
y = x[0]	$\hat{=}$	y = *x;
x[i]	$\hat{=}$	*(x+i)

$\longrightarrow$ *(x+i) : i-te Komponente vom Typ T; d.h. i ist ein ganzzahliger Index, x ist die Startadresse des Feldes

Feld:



(Pointerarithmetik)

Beispiel: Suche die Länge eines Strings (Ende = \0)

```
{
   char str[10];
   int l = 0;
   cin >> str;
   while (str[l] != '\0') l++;
   cout <<" Laenge " << l;
}
```

mit Pointerarithmetik effizienter:

```
{
   char* p = str;
   while (*p ++);              // referenziertes Zeichen = 0 ?
   cout << " Laenge " << p-str-1;
}
```

Da der Pointer durch den Inkrementoperator nachträglich erhöht wird, wird er einmal zu oft erhöht. Dies muß bei der Berechnung der Stringlänge berücksichtigt werden.

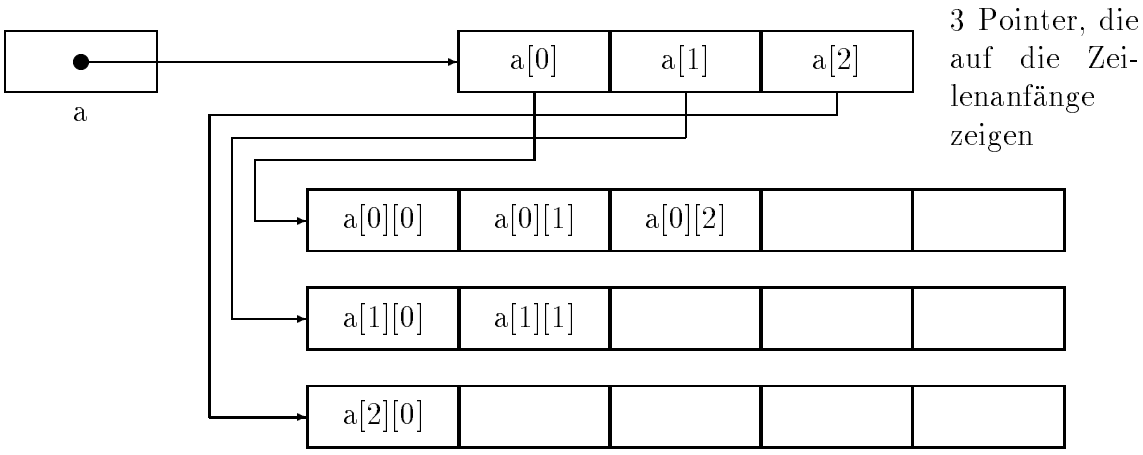
Mehrdimensionale Felder

Beispiel: 3×5 - Matrix

```
double a[3][5];        // a besteht aus 3 Komponenten vom Typ double[5]
```

a[i][j] $\hat{=}$ (a[i])[j]
$\hat{=}$ *((a[i])+j)
$\hat{=}$ *((*(a+i))+j)

- a[i] ist ein Pointer auf Anfang der i. Zeile, d.h. a[i]+j ist ebenfalls ein Pointer
- a+i ist ein Pointer in der Tabelle der Zeilenanfänge
- *(a+i) ist ein Pointer auf den Anfang der i. Zeile
- *(a+i)+j ist ein Pointer auf das j. Element in der i. Zeile
- *((*(a+i))+j) ist das Element a[i][j] selbst



Bei statischen Matrizen liegen die Zeilen unmittelbar hintereinander im Speicher.
Bei dynamisch erzeugten Matrizen gilt dies in der Regel nicht.

Unterschied: Feld von Pointern, Pointer auf ein Feld

int* a[5]; $\longleftrightarrow$ [ ] hat höhere Priorität als * in der Deklaration

$\Longrightarrow$ a[0] ... a[4] sind vom Typ int*
$\Longrightarrow$ a ist Feld von 5 Pointern

Alternative: Pointer auf ein Feld

```
int (*a)[5];          // *a ist vom Typ int[5]
```

$\Longrightarrow$ a ist ein Pointer auf ein Feld

9.6 Dynamische Speicherverwaltung

Anlegen und Freigeben von Speicher zur Laufzeit

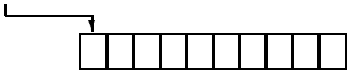
Operator new

Syntax: new Typname
new Typname (Initialisierungsliste$_{opt}$)
new Typname [Ausdruck]

Der Operator *new* liefert einen Pointer vom Typ Typname*

Beispiel:

```
int i,n;
int* ip;
double* dp;
double* feld;
ip = new int;
dp = new double (3.14);
cin >> n;
feld = new double [n];
```



$\longrightarrow$ n double-Zahlen

```
for (i=0;i<n;i++) feld[i] = i;
*ip = 5;
cout << *ip << *dp << endl;          // 53.14
for (i=0;i<n;i++) cout << feld[i];   // 01234 (für n = 5)
```

Bemerkung: Ausgabe von Pointern

```
char* :    Stringausgabe
andere :   Adressenausgabe

char* pc = new char[4];
pc = "aha";
cout << pc << endl;                    // aha

int *pi = new int;
cout << pi << endl;                    // Adresse hexadezimal
cout << (int*) pc;                     // Adresse hexadezimal
```

Löschen von Speicher, der nicht mehr benötigt wird:

Operator delete

Syntax: delete Variablenname (bei new Typname)
delete [ ] Variablenname (bei new Typname[Ausdruck])

ohne [ ] wird bei Feldern nur die erste Komponente freigegeben, die restlichen benötigen weiter Speicherplatz, sind aber i.a. nicht mehr nutzbar.

Beispiel: Anlegen und freigeben von dynamischen Matrizen der Größe n×n

```
#include<iostream.h>
int main ()
  {
    int n,i,j;
    cin >> n;                          // Dimension einlesen
    double **A = new double* [n];  // n Pointer erzeugt (siehe Seite 41)
    for (i=0;i<n;i++) A[i] = new double [n];      // nxn double Zahlen erzeugt
    for (i=0;i<n;i++)
      for (j=0;j<n;j++) A[i][j] = i+j;
    cout << A[0][0];
    ...
    // Matrix löschen:
    for (i=0;i<n;i++) delete [ ] A[i];//i-te Zeile löschen
    delete [ ] A;                      //Hilfsvektoren der Zeilenanfänge löschen
    return 0;
  }
```

9.7 Pointer und der const–Modifizierer

1. Syntax für Zeiger auf eine Konstante:
   const Typ* Variablenname Initialisierer

   ```
   const int *p, a;        // *p ist ein konstanter integer
   *p = 27;                // verboten
   p = &a;                 // erlaubt
   ```

2. Syntax für konstanten Zeiger:
   Typ * const Variablenname Initialisierer ;
   Zeiger ist eine Konstante, referenzierte Größe darf jedoch verändert werden.

   ```
   int i,j;
   int* const p = &i; // Zeiger muß initialisiert werden,
                      // da eine spätere Zuweisung nicht erlaubt ist.
   p = &j;            // verboten
   *p = 27;           // erlaubt
   ```

3. beides:
   const Typ * const Variablenname Initialisierer;
   $\longrightarrow$ sowohl der Pointer als auch die referenzierte Größe sind konstant

9.8 void–Pointer

void–Datentyp : leere Menge, hat keine Elemente
void–Pointer : void* Bezeichner

1. dürfen nicht dereferenziert werden
2. Zuweisung anderer Pointer erlaubt
3. umgekehrt Zuweisung an anderen Pointer nur mit *typecast* erlaubt

Beispiel:

```
int* ip;
void* vp;
vp = ip;            // erlaubt
ip = vp;            // verboten
ip = (int*) vp;     // erlaubt
     typecast
```

4. Vergleiche mit beliebigen Pointern erlaubt

5. void–Pointer verwendet im Zusammenhang mit Speicherverwaltung von C mittels *malloc*

Kapitel 10

Funktionen

10.1 Funktionsdefinition

Syntax: Typspezifikation Funktionsname (formale Argumentliste)
{Deklarationen und Anweisungen}

- Formale Argumentliste: Typ_1 Name_1,... Typ_n Name_n
- Liste kann leer sein, ( ) müssen aber trotzdem stehen

Beispiel: int main ( ) { Hauptprogramm }

Beispiel: Funktion für x^n

```
double power (double x, int n)
   {
      int i,m;
      double y;
      m = (n>=0) ? n : -n;          //m = |n|
      y = 1.0;
      for (i=0;i<n;i++) y *=x;
      if (n>=0) return y;
      else return 1/y;
   }
```

Bemerkungen

1. für jedes Argument muß eigene Typspezifikation stehen:
   int f(int a, b...) // verboten
   int f(int a, int b...) // richtig

2. Feldtypen sind nicht als Ergebnistyp zugelassen:
   int[10] f( ) //verboten

3. Rückgabe des Funktionsergebnisses erfolgt über:
   return $\underbrace{\textit{Ausdruck}}$;
   vom Typ der Funktion
   → es sollte in jedem Fall einmal eine return–Anweisung durchlaufen werden, mehrere return–Anweisungen sind erlaubt
   → wird kein return durchlaufen, dann ist das Funktionsergebnis undefiniert
   → return beendet den Funktionsaufruf

4. Funktionen können nur auf äußerster Ebene deklariert werden (nicht geschachtelt wie in Pascal)

10.2 Wert- und Referenzparameter

Werteparameter: Typangabe Name
⟶ stehen für Ausdruck, aktuelles Argument wird nicht verändert

Referenzparameter: Typangabe $\underbrace{\&}$ Name
kennzeichnet Referenz–Argument
⟶ aktuelles Argument kann verändert werden, formales Argument = Synonym für das aktuelles Argument

Beispiel: Funktion liest Zeichen, liefert als Ergebnis logischen Ausdruck (wahr, falls Zeichen eine Ziffer ist)

```
int get_digit (char & digit)
   {
      cout << "Ziffer eingeben : ";
      cin >> digit;
      if (digit >= '0' && digit <= '9') return 1;
      else return 0;                                  // keine Ziffer
   }
```

Bemerkungen

1. Felder sind immer Referenzparameter, Angabe von & unzulässig
   falsch : int f (int & x[5]) ...
   richtig: int f (int x[5]) ...
   Referenzparameter

2. in C gibt es keine Referenzargumente, nur in C++
   Abhilfe: Pointer
   int f (int *p);
   {...*p = 1;...}
   → Aufruf mittels Adreßoperator:
   int i;
   f (&i);

10.3 void–Funktionen

⟶ entsprechen den Prozeduren in Pascal

Beispiel: Variablen vertauschen

```
void swap (double& x, double& y)
  {
    double help;
    help = x; x = y; y = help;
    return;            // Ergebnistyp void = leere Menge
  }
```

⟶ return ohne Ausdruck beendet Funktion, kann am Funktionsende auch entfallen

- void–Funktionen haben meist Referenzparameter

Zugriff auf Argumente der Kommandozeile des Hauptprogramms

```
int main (int paramcount, char* paramstr [ ])
  {
    for (int i=1;i<paramcount;i++) cout << paramstr[i] << " ";
```

```
    return (0);
}
```

Aufruf: param 1 2 3 $\Longrightarrow$ Ausgabe: 1 2 3
Bemerkung: In paramstr[0] wird der Programmname (param) abgespeichert.

10.4 const–Parameter

Schutz des Arguments vor Veränderung (z.B. bei Feldern)

Beispiel:
```
void p (const double x[10])
  {   x[0] = -0.5;        // verboten
  ...
```

→ x ist ein Referenzargument und ist trotzdem vor Veränderung geschützt

10.5 Funktionsaufruf

innerhalb eines Ausdrucks vom entsprechenden Typ:

Syntax: Funktionsname (aktuelle Argumentliste)
mit Komma getrennt

Wertaufruf

Es wird eine lokale Variable angelegt und mit dem Wert des entsprechenden aktuellen Arguments initialisiert. Arbeit mit lokaler Variablen: Änderungen haben keinen Einfluß auf das entsprechende aktuelle Argument.

Beispiel:
```
int f (int x)
  {
    x += 5;
    cout << x;          // veränderter Wert ausgegeben
  }
...
int a = 3;
f(a);                   // Ausgabe: 8
```

```
cout << a;                  // Ausgabe: 3
```

Bemerkung: Aufruf von Funktionen ohne Argumente

```
void Meldung ()             // Klammern nötig
  {   cout << "hallo";   }

void main ()
  {
    Meldung ();             // korrekt, Ausgabe von "hallo"
    Meldung;                // syntaktisch korrekt, kein Effekt
  }
```

Meldung ohne Klammern ist die Adresse der Funktion.

10.6 Funktionen mit default–Werten

Default–Werte (Standardwerte, Ersatzwerte) für Funktionsparameter. Sie werden beim Funktionsaufruf eingesetzt, wenn kein aktuelles Argument angegeben ist. Die Angabe als Initialisierung erfolgt mit dem = Ausdruck.

Beispiel: Volumen berechnen

```
int volumen (int h=1, int b=1, int t=1)
  {   return (h*b*t);   }
```

Aufrufe:	volumen(3,2,4)	$\longrightarrow$	24
	volumen(5,4)	$\longrightarrow$	20 : h=5, b=4, t=1
	volumen(5)	$\longrightarrow$	5
	volumen( )	$\longrightarrow$	1

Wird ein Defaultwert angegeben, dann auch für die darauffolgenden Argumente.

10.7 Inline–Funktion

Normalfall: Code für Funktion steht getrennt vom Hauptprogramm. Beim Aufruf erfolgt Unterprogrammsprung mit Parameterübergabe; am Ende Rücksprung, Ergebnisrückgabe

$\Longrightarrow$ kostet Zeit

inline: bewirkt, daß nach Möglichkeit der Aufruf eingespart wird und Code direkt ins Hauptprogramm kopiert wird

$\Longrightarrow$ geht schneller, braucht mehr Speicher für Code

Beispiel:
```
inline int min(int x, int y)
{
  return (x>y) ? y : x;
}
```

10.8 Pointer statt Referenzparameter

In ANSI–C gibt es keine Referenzparameter, als Ersatz stehen Pointer zur Verfügung.

Beispiel:
```
void swap (int* pn, int* pm)
{ int help = *pn; *pn = *pm; *pm = help; }
```

Aufruf:
```
int i = 3, j = 4;
swap(&i, &j);          // & ist der Adressoperator
```

10.9 Pointer und Funktionen

Funktionsergebnis vom Typ Pointer:

```
int* f (...)...
```

() hat höhere Priorität als *, d.h. das Funktionsergebnis ist vom Typ int*, also ein Pointer.

„Gegenteil:" Pointer auf eine Funktion mit Ergebnistyp integer

```
int (*f) (...)...
```

Die Klammerung ändert Reihenfolge, d.h.

- *f ist eine Funktion vom Typ int (...)
- f ist ein Pointer, der auf eine solche Funktion zeigt

Beispiel: Wertetabelle ausgeben

```
void Wertetabelle (double (*f) (double), double x, double y, int n)
                   formaler Parameter für Pointer auf Funktion
  {
    int i; double h;
    h = (y-x)/(n-1);
    for (i=0;i<n;i++) cout << (*f)(x+i*h) << endl;          // siehe unten
  }
...
Wertetabelle (&sin, 0, 1, 10);
Wertetabelle (cos, -1, -1, 100);    // cos entspricht &cos
```

f ist ein Pointer, (*f) ist die Funktion mit Rückgabetyp double, (x+i*h) ist die Argumentliste für Funktion.

10.10 Überladen von Funktionen

Mehrere Funktionen dürfen den gleichen Namen haben, wenn sie an der Argumentliste unterschieden werden können.

Beispiel:
```
void swap (int& x,int& y) ...
void swap (double& x, double& y) ...
```

Aufruf:
```
int i,j;
double a,b;
swap (i,j);          // 1. Funktion
swap (a,b);          // 2. Funktion
```

Beachte:

1. Funktionsergebnis wird nicht zur Unterscheidung herangezogen
2. Wert- und Referenzaufruf werden nicht unterschieden

Kapitel 11

Blockstruktur und Rekursion

11.1 Blockstruktur

Block: {...}

Lokaler Geltungsbereich

Deklaration eines Namens (Variable oder Konstante), nicht Funktion, innerhalb eines Blockes möglich.

Gültigkeit : ab Deklaration bis Blockende...}

Beispiel:
```
int main ()
{
  double x = 1.0;
    {
      // untergeordneter Block
      double y = 4.0;
      cout << x << y;       // 1 4
      int x = 5;
      cout << x << y;       // 5 4
    }       // hier endet die Gültigkeit von y
  cout << x << y;           //1 (altes x), y führt zu Fehler
  return 0;
}
```

Globaler Geltungsbereich

Deklaration eines Namens außerhalb aller Blöcke (wie Funktionen)

Beispiel:
```
int dim = 10;
void f()
  {... // Zugriff auf dim erlaubt
  }
...
int main ()
  {... // ebenso Zugriff erlaubt
  }
```

Wenn globale Größe verdeckt ist, kann trotzdem auf sie zugegriffen werden mit dem Operator ::

```
int i = 1;
int main ()
  {
    int i = 2;
    cout << i << ::i;          // Ausgabe: 2 1;
    ...
```

Lebensdauer einer Größe:

Dauer, in der die Größe gespeichert bleibt

Automatische Größe:

- Deklaration innerhalb eines Blocks
- Lebensdauer bis Blockende
- keine automatische Initialisierung

Statische Größen:

- Deklaration außerhalb aller Blöcke oder mit Wortsymbol *static*
- Lebensdauer während gesamter Laufzeit des Programms
- werden automatisch mit 0 initialisiert

- Initialisierung nur einmal, sobald der Programmablauf erstmals auf die Definition trifft
- Wert bleibt statisch im Speicher erhalten

Beispiel:

```
int x;
int main ()
{
  int y;
  cout << x << y ;
          0   undefiniert
}
```

Beispiel:

```
for (int i=1; i<=3; i++)
{
  static double x=3;
  double y=3;
  ++x;
  ++y;
  cout << x << y << endl;
}
// i=1: x=4; y=4;
// i=2: x=5; y=4;
// i=3: x=6; y=4;
```

Beispiel:

```
void Aufruf ()
{
  static int n=1;
  cout <<   n++   << "-ter Funktionsaufruf" << endl;
    nachträglich erhöhen um eins
  ⋮
}
```

```
Hauptprogramm:  Aufruf ();   // 1-ter Funktionsaufruf
                Aufruf ();   // 2-ter Funktionsaufruf
                Aufruf ();   // 3-ter Funktionsaufruf
```

Bemerkung:

```
int i = 42;
for (int i=1; i<3; i++)
{
}   // Lebensdauer von i endet am Schleifenende
cout << i;   // ergibt 42
```

$\Longrightarrow$ zur Zeit noch *andere* Definition, insbesondere auch bei unserem Compiler (HP CC)

```
{
   for (int i ...)
   {   }
}   // erst hier endet Lebensdauer von i
```

11.2 Rekursion

mathematisches Beispiel:

$$x^n = \begin{cases} x \cdot x^{n-1} & \text{falls} \quad n \geq 1 \\ 1 & \text{falls} \quad n = 0 \end{cases}$$

$$\longrightarrow x^4 = x \cdot x^3 = x \cdot x \cdot x^2 = x \cdot x \cdot x \cdot x^1 = x \cdot x \cdot x \cdot x \cdot 1$$

Rekursion: Funktion wird innerhalb ihrer Definition bereits aufgerufen

beachte: es muß Abbruchbedingung existieren, damit nur endlich viele Aufrufe erfolgen

Beispiel: größter gemeinsamer Teiler ggT zweier ganzer Zahlen

$$ggT(x,y) = \begin{cases} x & \text{falls} \quad y = 0 \\ ggT(y, x \bmod y) & \text{falls} \quad y \neq 0 \end{cases}$$

in C++:

```
int ggT (int x, int y)
{
  if (y==0) return x;   // hier wird Funktion mit return beendet
  return ggT (y, x % y); // else überflüssig
}
```

Bei jedem Aufruf werden lokale Größen (insbesondere Parameter der Funktion) neu angelegt im Sinne der Blockstruktur, z.B. 3 rekursive Aufrufe von ggT: es existieren 3 Werte von x und y im Speicher, nur der jeweils zuletzt angelegte ist zugänglich.

Bemerkung: im Zusammenhang mit Rekursionen sind normalerweise *static* Variablen nicht sinnvoll

ggT (15,123)
ggT (123,15)
ggT (15,3)
ggT (3,0) → y=0 → 3

11.3 Funktionsprototypen und indirekte Rekursion

bisher: Funktionsdeklaration immer auch Funktionsdefinition
jetzt: separate Funktionsdeklaration

Syntax: Typspezifikation Funktionsname (Argumentliste);
→ es fehlt der Rumpf

Beispiel:
```
int power (double x, int n);
≙ int power (double, int);      // Namen einfach weglassen
≙ int power (double y, int k);  // Namen egal
```

Bemerkung: Namen der Argumente gelten als Kommentar

→ Ergebnistyp und Argumentliste müssen zwischen allen Deklarationen und Definitionen übereinstimmen

→ Deklarationen einer Funktion dürfen mehrfach auftreten, aber genau *eine* Definition

→ Programm kann bereits compiliert werden, wenn nur Deklaration existiert

→ Programm kann nur ausgeführt werden, wenn auch die Definitionen aller

Funktionen bekannt sind

→ nach der Deklaration darf die Funktion in anderen Funktionen aufgerufen werden

Kapitel 12

Klassen/Objektorientierte Programmierung

12.1 Einleitung in Objektorientierte Programmierung

Klasse: faßt Daten und "Methoden" (Elementfunktionen) zusammen

$\longrightarrow$ abstrakte Datentypen

- Darstellung nach außen: sichtbarer Teil einer Klasse
- technische Details der Realisierung für den Benutzer ohne Bedeutung

12.2 Deklaration und Definition von Klassen

Syntax: class Name {Elementliste$_{opt}$};

Elementliste: Datenelemente und Funktionen/Operationen, die diese Daten bearbeiten

zusätzliche Angabe von *private:* oder *public:*

private: solche Elemente sind nur innerhalb der Klasse zugänglich
public: für Benutzer der Klasse zugänglich

in class: erst einmal sind alle Elemente private, Angaben gelten "bis auf Widerruf" in Klasse

Definition der Elementfunktionen

1.Variante: inline

Beispiel:
```
class point
  {
    private: double x,y; // private optional
    public:
      double get_x ()   { return x; }
      double get_y ()   { return y; }
      void move (double dx, double dy)
        { x += dx; y += dy; }
  };
```

2. Variante: in Klasse nur Deklaration, Definition wird nachgetragen

Beispiel:
```
class point
  {
    private: double x,y;
    public:
      double get_x ();          // Deklaration
      double get_y ();          // Semikolon nötig
      void move (double dx, double dy);
  };

// nachträglich: Definition der Methoden
// Bezug zur Klasse wird durch Geltungsbereichoperator hergestellt
// Ergebnistyp Klassenname :: Funktionsname (...) {...

double point :: get_x ()    { return x; }
double point :: get_y ()    { return y; }
void point :: move (double dx, double dy)
  { x += dx; y = += dy; }
```

üblich:

- kurze Funktion ohne Schleifen: inline (1.Variante)
- sonst: 2.Variante

12.3 Zugriff auf Komponenten

Klassenelemente = Daten und Methoden

a.) *innerhalb* der Klasse
Zugriff auf alle Komponenten erlaubt, einfach Komponentenname angeben

b.) *außerhalb* der Klasse
Zugriff nur auf public–Komponenten

Syntax: Klassenname. Komponente

Beispiel:
```
point p, q[3];
// Punkt p, Vektor von 3 Punkten q
cout << "x–Koordinate von p = " << p.get_x () << endl;
p.move (1,1);
for (int i=0;i<3;i++)
   q[i].move (1,1);            // alle Punkte q[i] verschieben
cout << p.x << q[0].y;     // Fehler! Verboten!
// direkter Zugriff auf diese Komponenten führt zu Fehler
// denn sie sind private
```

Im Beispiel fehlt die Initialisierung der Daten. Abhilfe bieten die Konstruktoren.

12.4 Konstruktoren

mit den bisherigen Methoden:

```
class complex
  {
    double re,im;
    ...
    public: void init (double a=0, double b=0)
              { re = a; im = b;}   // bei void kann return entfallen
  };
...
complex x,y,z;
x.init ();        // x ≙ (0,0)        ≙ 0 + i · 0
```

y.init (3.14); // y $\hat{=}$ (3.14,0) $\hat{=} 3.14 + i \cdot 0$
z.init (1,2); // z $\hat{=}$ (1,2) $\hat{=} 1 + i \cdot 2$

Wunsch: das Ganze automatisch

Lösung: Konstruktor

$\longrightarrow$ spezielle Elementfunktion mit dem Namen der Klasse und ohne Ergebnistyp

Beispiel:

```
class complex
  {
    double re,im;
    ...
    public: complex (double a=0, double b=0)
              { re = a; im = b;}
  };
...
complex x;            // Standardkonstruktor ohne Argumente
                      // x ≙ (0,0)       ≙ 0 + i · 0
complex y (3.14);     // y ≙ (3.14,0)    ≙ 3.14 + i · 0
complex z (1,2);      // z ≙ (1,2)       ≙ 1 + i · 2
```

complex () ist nicht erlaubt: sieht aus wie die Deklaration einer komplexen Funktion

Bemerkungen

- wenn ein Konstruktor ohne Argumente bzw. mit default–Argumenten existiert, wird er als Standardkonstruktor bezeichnet; er wird bei jedem Klassenobjekt aufgerufen, das nicht explizit initialisiert wurde
- Konstruktoren dürfen keinen Ergebnistyp haben, auch nicht void
- mehrere Konstruktoren erlaubt (Überladen oder mittels default–Argumenten)

- ohne Standardkonstruktor werden die Datenkomponenten nicht automatisch initialisiert

Expliziter Aufruf des Konstruktors

Konstruktor wird als Funktion aufgerufen und liefert Wert, der z.B. zur Initialisierung von Variablen verwendet werden kann

Beispiel:
```
complex z    = complex (1,2);  // complex: Aufruf des Konstruktors
complex z    = 3.14;           // Konstr. mit Arg. wird aufgerufen
complex v[3] = {complex (1,2), complex (3,4), complex (0,7)};
complex w[3]= {complex (), complex (5), complex ()};
complex u[2] = { 3.14 , 2.1828};
                 (3.14,0)
```

12.5 Destruktoren

Spezielle Elementfunktionen mit dem Namen ~Klassenname ohne Ergebnistyp und ohne Argumente. Dient zur Freigabe von dynamisch angelegtem Speicherplatz.

Bsp.: String anlegen und wieder vernichten

```
class string
{
  int l;    // maximale Länge
  char* s;    // Pointer auf Anfang der Zeichenkette
  public:
    string (int len, char fill);   // gewünschte Länge
    ~string ();    // Destruktor
}
string :: string (int len, char fill = ' ')    // Konstruktor
{
  int i;
  l = len;
  s = new char[len+1];
  for (i=0;i<len,i++) s[i]=fill;
  s[len] = '\0';
}
```

```
string :: ~string ()              // Destruktor
{
  delete[ ] s;
}
```

Bemerkungen:

- nur 1 Destruktor pro Klasse erlaubt
- Destruktor wird automatisch aufgerufen, sobald die Lebensdauer einer Klassenvariablen (Instanz) endet, z.B. am Blockende
- fehlt der Destruktor, so generiert der Compiler einen Standarddestruktor (im Beispiel werden *l* und *s* freigegeben, aber nicht die Zeichenkette)

12.6 Strukturen (auch bei C)

Eine Struktur (*struct* statt *class*) entspricht einer Klasse.

Unterschiede:

- bei *struct* standardmäßig alles *public*
- bei *class* standardmäßig alles *private*
- class und struct *mit* Methoden gibt es erst bei C++, nicht bei C
- struct *ohne* Methoden (= *record* bei Pascal) gibt es auch in C

12.7 friend–Funktionen

Eine *friend*–Funktion gehört *nicht* zur Klasse, sie erhält aber trotzdem auf die *private*–Komponenten der Klasse Zugriff.

Syntax: friend Funktionsdeklaration // innerhalb der Klasse

Bsp.: komplexe Addition

```
class complex
{
```

```
    double re, im;
    public: complex ();
    friend  complex  add (complex, complex);    // add ist Funktionsname
            Ergebnistyp  Typen der formalen Parameter
};
...
complex add (complex a, complex b)    // hier nicht complex :: add ...
{
    complex c;    // Hilfsvariable
    c.re = a.re + b.re;    // Zugriff erlaubt, obwohl private
    c.im = a.im + b.im;
    return c;
}
```

ohne friend kann der Zugriff erreicht werden:

- generelle Freigabe der Zugriffe auf Komponenten mit *public*
- Zugriff über zusätzliche *public*–Funktionen getre(), setre() und getim(), setim()

Beides verstößt gegen die Philosophie der Datenkapselung (Zugriff auf interne Daten sollte möglichst beschränkt werden).

Vorteil: Datenabstraktion, Sicherheit vor unbeabsichtigten Änderungen der Daten

Ergänzungen:

1. ```
   class Y
   {
     friend class X
   ...
   }
   ```

   friend class X bewirkt, daß alle Funktionen der Klasse X friend – Funktionen der Klasse Y sind. Die Umkehrung muß nicht gelten.

2. friend Ergebnistyp Klassenname :: Funktionsname (...);
   damit wird eine Elementfunktion aus einer anderen Klasse als befreundet deklariert
   ```

Kapitel 13

Benutzerdefinierte Operatoren

Fast alle Operatorsymbole von C++ können für benutzerdefinierte Typen verwendet werden: *Überladen*

Ausnahme: sizeof , :: , .

2 Varianten:
a) als Methode der Klasse
b) als friend

Syntax: Als spezieller Funktionsname wird verwendet:
operator Operatorsymbol

Anmerkung: die Standardoperatoren für Standardtypen (int, double, ...) dürfen nicht verändert werden

13.1 Binäre Operatoren

1. Variante als Komponente der Klasse

```
class complex
{
  double re, im;
  ...
  public:
    complex operator + (complex x);    // beachte: ein Argument!
};
```

```
complex complex :: operator + (complex x)
{
  complex z;
  z.re = re + x.re;
  z.im = im + x.im;
  return z;
}
```

Aufruf:

a)
```
complex a,b,c;
c = a . operator + (b);
```
(a: Klasse, operator +: Methode)

b)
```
c = a + b;
```

2. Variante mit friend–Funktion

```
class complex
{
  double re, im;
  ...
  friend complex operator + (complex x, complex y);    // zwei Argumente!
};
complex operator + (complex x, complex y);
{
  complex z;
  z.re = x.re + y.re;
  z.im = x.im + y.im;
  return z;
}
```

Aufruf:

a)
```
c = operator + (a,b);
```
b)
```
c = a + b;
```

2 Alternativen:

zweistelliger Operator als

- Komponentenfunktion (Methode) der Klasse mit *einem* Argument, aufrufendes Objekt ist erster Operand.
- friend–Funktion der Klasse mit *zwei* Argumenten

13.2 Überladen der Ein- und Ausgabe

iostream.h enthält die Klassen

istream für Eingabeströme
ostream für Ausgabeströme

cin, cout sind Instanzen dieser Klassen

13.2.1 Überladen der Eingabe mit >>

>> : linker Operand istream, rechter Operand beliebigerTyp

1. Variante: als Methode der Klasse

scheitert, weil kein Operator >> in die Klasse istream eingefügt werden kann

2. Variante: als friend–Funktion

```
class beliebigerTyp
{
  ...
  friend istream& operator >> (istream& is, beliebigerTyp& x);
  // wichtig: Referenzparameter, damit sich Eingabestrom beim Lesen ändert
};
istream& operator >> (istream& is, beliebigerTyp& x)
{
  // Eingabe der Komponenten der Klasse beliebigerTyp
  // z.B. complex: is >> x.re >> x.im;
  return is;
}
```

Aufruf:

```
beliebigerTyp a;
operator >> (cin,a);
cin >> a;
```

Eingabe mehrerer Operanden:

```
cin >> a >> b;
```

damit dies funktioniert, muß der Operator >> als Ergebnis den veränderten Zustand des Eingabestroms (Referenz) zurückliefern

13.2.2 Überladen der Ausgabe mit <<

```
class beliebigerTyp
{
  friend ostream& operator << (ostream&, beliebigerTyp);
  // beliebigerTyp Wertaufruf, kann Ergebnis eines Ausdrucks sein
};
ostream& operator << (ostream& os, beliebigerTyp x)
{
  // Ausgabeoperation für Datenkomponenten
  // z.B. complex: os << x.re << x.im;
  return os;
}
```

Aufruf:

```
beliebigerTyp a;
operator << (cout,a);
cout << a;
```

mehrere Ausgaben, gemischt mit anderen Typen:

```
complex a;
double b;
...
cout << a << b;
```

Bemerkungen:

1. Operatoren wie + ...werden am besten als Methode formuliert
2. Ein-/Ausgabe müssen als friend–Funktionen formuliert werden
3. Operatoren mit gemischtem Operandentyp (z.B double und complex) i.a. als friend–Funktionen

13.3 Überladen des Zuweisungsoperators =

Ein Zuweisungsoperator = wird für jede Klasse automatisch generiert.

Wirkung:

- komponentenweise Kopie
- keine Konversion, Zuweisung nur innerhalb der Klasse
- nur Kopie der Komponenten, *nicht* der per *new* zur Laufzeit erzeugten Größen (flache Kopie)

Konversion double → complex: Zuweisungsoperator nötig

```
class complex
{   ...// wie bisher
  public:
    complex& operator = (double x)
    {
      re = x;
      im = 0.0;
      return *this;
    }
};
```

Aufruf:

```
complex a,b;
double d;
a.operator = (d);
a = d;
a = b = d;    // b = d: complex
```

Zu jeder Klasse gibt es einen automatisch erzeugten Pointer *this*, der auf die aufrufende Instanz der Klasse zeigt.

13.4 Überladen des Indexoperators [ ]

Der Indexoperator kann überladen werden, er wird *nicht* automatisch erzeugt
→ selbst definieren als Elementfunktion mit einem Operator

Beispiel: Polynom maximal dritten Grades

```
const maxgrad = 3;
class polynom
{
  double coeff [maxgrad+1];
  ...
  public:
    double& operator [ ] (int i)
    {
      if (i<0) || (i>maxgrad) exit(1);    // exit braucht stdlib.h
      return coeff [i];
    }
};
```

Aufruf:

```
polynom p;
p.operator [ ] (0) = 1.7;    //Ergebnistyp des Operators ist double&
p[0] = 1.7;                  // gleichwertig
cout << p[0] << p[1];
```

13.5 Vergleichsoperatoren

Variante 1: Komponentenfunktion (Methode) mit einem Operanden
Variante 2: friend–Funktion mit zwei Operanden

Beispiel: (Variante 1)

```
class complex
{ ...
  public:
  // Ergebnis ersatzweise int
  int operator == (complex z)
  {
    return z.re == re && z.im == im;
```

```
    }
};
```

13.6 Unäre Operatoren (einstellig)

1. Variante: Komponentenfunktion (Methode) ohne Operand

2. Variante: friend–Funktion mit einem Operand

$\longrightarrow$ dadurch wird der entsprechende Präfix–Operator definiert

Beispiel: Unäres – für komplexe Zahlen

```
class complex
  {
    ...
    public: complex operator – ();
  };
...
complex complex :: operator – ()
  {
    complex z;
    z.re = -re;
    z.im = -im;
    return (z);
  }
```

Aufruf:

```
complex a,b;
...
a = b.operator – ();          // ...oder...
a = -b;
```

Beispiel: Konjugiert komplexe Zahl mit ˜

```
class complex
  {
    ...
    public:
      friend complex operator ˜ (complex z);
  };
complex operator ˜ (complex z)
  {
```

```
    complex y;
    y.re = z.re;
    y.im = -z.im;
    return y;
  }
```

Aufruf:
```
a = operator ~ (b);        // gleichbedeutend mit ...
a = ~b;
```

Inkrement– und Dekrementoperatoren

++, −−

- Der Präfix–Operator wird definiert als Komponentenfunktion ohne Argument, wobei der Funktionstyp üblicherweise eine Referenz ist.
- Der Postfix–Operator wird definiert als Komponentenfunktion mit Dummy–Argument vom Typ *int*, wobei der Funktionstyp üblicherweise ein Wert ist.

Beispiel: ++ für complex

```
class complex
  {
    ...
    complex& operator ++ ()        // Präfixoperator
      {
        re ++;
        im ++;
        return *this;
      }
    complex operator ++ (int d)    // Postfixoperator
      {
        complex temp = *this;      // alten Wert abspeichern
        re ++;
        im ++;
        return temp;               // alten Wert zurückgeben
      }
  };
```

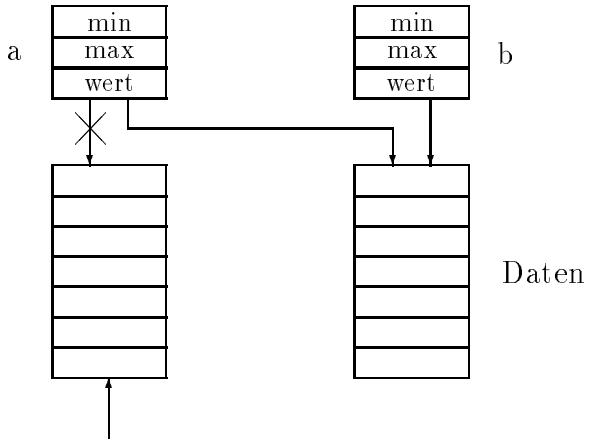
Aufruf der Operatoren:

```
complex a,b,c;
```

```
b = a.operator ++ ();      // Präfix
b = ++a;                   // a wird erhöht, b bekommt veränderten Wert
b = a.operator ++ (0);     // Postfix
b = a++;                   // a wird erhöht, b bekommt alten Wert
```

13.7 Flache Kopie, Zuweisungsoperator, Copy–Konstruktor

```
class Feld
  {
    int min, max;
    double* wert;       // Pointer auf Daten
    public:...
  } a,b;                // 2 Variablen vom Typ Feld vereinbart
...
a = b;
```



Probleme:

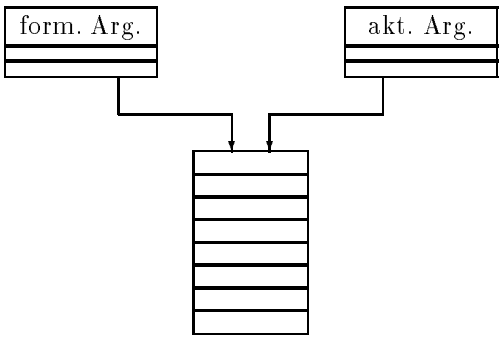
- der Speicher, auf den a.wert gezeigt hat, geht verloren (Speicherlücke)
- a.wert, b.wert zeigen auf gleichen Speicher, d.h. nach a[5] = 23; hat auch b[5] den wert 23
- irgendwann endet Lebensdauer von z.B. a; wenn b weiterlebt, dann wird bei Zugriffen auf b eventuell nicht definierter Speicher angesprochen
  $\Rightarrow$ kann zum Systemabsturz führen

Abhilfe: Zuweisungsoperator, der nicht nur min, max und wert kopiert, sondern auch die Daten, auf die *wert* zeigt.

```
Feld& Feld :: operator = (Feld& x)
  {
    if (wert == x.wert) return *this;
    // Spezialfall Zuweisung x=x
    if (wert != NULL) delete[] wert;
    min = x.min;
    max = x.max;
    if (max - min >= 0)
      {
        wert = new double [max -min+1];
        for (int i=0;i<=max-min;i++)
          wert[i] = x.wert[i];      // wert ist normales Feld
      }
    else wert = NULL;
    return *this;
  }
```

$\Longrightarrow$ gleiches Problem bei Funktionsargument mit Wertaufruf

Beispiel: `void TuWas (Feld x)  // Werteparameter`
`{ x[min] = 999; }`



aktuelles Argument wird verändert *trotz* Werteaufruf

Abhilfe: Copy–Konstruktor
$\rightarrow$ Anlegen einer ordentlichen Kopie

Form: Argument (Referenz) vom Klassentyp

Beispiel:
```
Feld :: Feld (Feld& x)    // Argument: Referenz!
{
  min = x.min;
  max = x.max;
  if (max−min >= 0)
  {
    ...// wie oben
  }
  else wert = NULL;    // bei Konstruktor kein return
}
```

Kapitel 14

Abgeleitete Klassen

14.1 Vererbung

Von einer bestehenden Klasse können weitere Klassen abgeleitet werden. Sie "erben" alle Komponenten der "Basisklasse" (Daten, Methoden) und haben gegebenenfalls weitere Komponenten.

Zugriffsrechte:

private–Komp.:	dürfen nur verwendet werden innerhalb der Klasse und in friend–Funktionen, sonst nicht, auch nicht in abgeleiteten Klassen
public–Komp.:	allgemein zugänglich
protected–Komp.:	innerhalb der Klasse, innerhalb der abgeleiteten Klasse und in friend–Funktionen

$\Rightarrow$ "private < protected < public "

Syntax:

```
class abgeleiteteKlasse : Zugriffsrechte Basisklasse
{
  // neue Komponenten der abgeleiteten Klasse
}
```

Angabe der Zugriffsrechte:

bewirkt *Einschränkung* der Zugriffsrechte von außerhalb auf Komponenten der Basisklasse

public, protected, private

(keine Angabe: class private, struct public)

Zugriffsrechte von außen sind Minimum der beiden Angaben

a) in der Basisklasse

b) bei der Ableitung

Von einer abgeleiteten Klasse können weitere Klassen abgeleitet werden:

Mensch → Student → Kursteilnehmer

14.2 Verdeckte Funktionen / Virtuelle Funktionen

In der abgeleiteten Klasse kann man eine Größe mit dem gleichen Namen wie in der Basisklasse vereinbaren. Damit wird die Größe in der Basisklasse verdeckt, sie ist nicht ohne weiteres zugänglich und wird durch die neue Größe ersetzt.

Man kann auf die verdeckte Größe zugreifen durch Qualifizierung:
Instanzname. Basisklassenname :: Komponentenname

Wird in der Basisklasse auf eine Funktion der Klasse zugegriffen, dann wird der Funktionsaufruf fest eincodiert.
Wird in einer abgeleiteten Klasse diese verdeckt (neu definiert), dann wird in der Basisklasse auf die verdeckte Funktion zugegriffen und nicht auf die neu definierte in der abgeleiteten Klasse.

Abhilfe:

Angabe von *virtual* bei der Funktion in der Basisklasse

Wird in einer abgeleiteten Klasse eine Funktion mit identischer Parameterliste und gleichem Ergebnistyp definiert, dann wird diese Funktion verwendet.

Bemerkungen

- anderer Ergebnistyp in abgeleiteter Klasse: Fehler
- andere Parameterliste: keine Verdeckung, sondern Überladung

- "rein virtuelle Funktion":
  in Basisklasse nur Deklaration
  spezielle Form, z.B. virtual int f (int) $\underbrace{=0}$;
  Markierung als dummy–Fkt.

14.3 Mehrfachvererbung

Eine Klasse kann von mehreren Basisklassen abgeleitet werden.

Syntax: class Klassenname : Zugriff Basisklasse, Zugriff Basisklasse, ...
```
{
  // eigene Komponenten
}
```

Bei Mehrdeutigkeiten muß mit dem Namen der Basisklasse qualifiziert werden.

Syntax: Instanzname. Basisklassenname :: Komponentenname

Beispiel:
```
class Vater {public char* Name;};
class Mutter {public char* Name;};
class Kind : public Mutter, Vater { ...};
Kind Egon;
Egon. Mutter :: Name = "Erika";
Egon. Vater :: Name = "Erwin";
```

Kapitel 15

Templates

Dabei handelt es sich um parametrisierte Funktionen und Klassen.

15.1 template–Funktionen

Zweck: Funktionsschablone definieren, die für verschiedene Datentypen verwendet werden kann (bisher nötig: mehrere überladene Funktionen)

Syntax: template <class Typname > Funktionsdeklaration/-definition
beliebiger Name

Typname wirkt als *formaler Parameter* !

Beispiel:
```
template <class T> void swap(T&x, T&y)
  {
    T hilf;          // Hilfsvariable vom Typ T
    hilf = x;
    x = y;
    y = hilf;
  }
```

Aufruf:
```
int x,y;
double a,b;
swap (x,y);
swap (a,b);
swap(a,x);          // ergibt einen Fehler
```

$\longrightarrow$ gemischte Typen als Argumente hier nicht erlaubt !

```
complex c,d;
swap(c,d);
```

Bezeichnungen:

template–Funktion = parametrisierte Funktion
= Funktionsschablone
= generische Funktion

Ablauf: Beim Aufruf werden die aktuellen Argumente analysiert; für deren Typ wird eine "Funktionsinstanz" generiert und zur Laufzeit aufgerufen.

wichtig: Alle Operatoren (bzw. Funktionen), die in der template–Funktion aufgerufen werden, müssen für den Datentyp definiert sein, der beim Aufruf angegeben wurde.

Regeln:

- es dürfen mehrere Parameter für Typen verwendet werden, z.B.
```
template <class T, class S> T f(T x, S y, T z)
  {
    T lokaleVariable;...
  }
```
- Typparameter erlaubt als formaler Parameter, Ergebnistyp, lokale Variablen der template–Funktion, nicht außerhalb (z.B. im Hauptprogramm)
- jeder template–Parameter muß als formales Argument der template–Funktion auftreten
  Beispiel: template <class T> T f $\underbrace{(\,)}_{\text{hier muß T vorkommen}}$ {...} $\longrightarrow$ Fehler !

15.2 template–Klassen

Zweck: z.B. Def. einer Klassenschablone für

- Vektoren mit beliebigem Komponententyp
- Listen

Syntax: template <class Typname> Klassendeklaration

Beispiel: Vektoren mit beliebigen Typ:

```
template <class T>
class vector
  { private:   int size;
               T* comp;    // Pointer auf 1. Komponente des Vektors
    public:    T& operator [ ] (int i);
               operator* (vector x);
  }
```

Instanzen der template-Klasse bilden: Klassenname <konkreter Typname>

Beispiel:
```
vector <int> x,y;
vector <double> u,v;
vector <complex> c,d;
```

Typname *vector* kann innerhalb der Klassendeklaration so verwendet werden, außerhalb in Definitionen zu einer Klasse muß $vector < T >$ geschrieben werden.

Beispiel:
```
// Definition von *
template <class T>
  T vector <T> :: operator * (vector<T> x)
    {
      int i;
      T sum = comp[0] * x.comp[0];
      for (i=1;i<size;i++)
         sum += comp[i] * x.comp[i];
      return sum;
    }
```

Literaturverzeichnis

[1] **Schader/Kuhlins**, *Programmieren in C++*, 4.Aufl., Springer 1996, DM 48,–

[2] **Glaeser, G.**, *Von Pascal zu C/C++*, Markt und Technik Verlag, DM 69,–

[3] **Kernighan/Ritchie**, *Programmieren in C++*, Hauser Verlag München 1990, DM 58,–

[4] **Stroustrup,B.**, *Die C++ Programmiersprache*, 2.Aufl., Addison Wesley 1992, DM 89,90

[5] **Alex/Bernör**, *UNIX, C und Internet*, DM 58,–

[6] **Achtert**, *Das große Buch zu C++*, Data Becker, DM 99,–

[7] **Duden Informatik**

[8] **Sedgewick**, *Algorithmen in C++*, Addison Wesley 1991

[9] **Ottmann/Widmayer**, *Algorithmen und Datenstrukturen*, BI 1990

[10] **Engeln–Müllges/Reutter**, *Numerische Mathematik für Ingenieure*, BI 1990

[11] **Engeln–Müllges/Reutter**, *Formelsammlung zur Numerischen Mathematik mit C-Programmen*, BI 1989