Objektorientierte Softwareentwicklung mit C++

Prof. Dr. H. G. Folz

Stand: 20.04.2003

Inhalt

Inhalt 1

Literatur 1

1 Einführung 2

2 Grundelemente der Sprache C++ 4
- 2.1 Ein erstes Programm 4
- 2.2 Zeichenvorrat 5
- 2.3 Kommentare 5
- 2.4 Bezeichner 6
- 2.5 Reservierte Wörter 6
- 2.6 Elementare Datentypen 7
- 2.7 Variablen und Konstanten 7
- 2.8 Operatoren und Ausdrücke 8
- 2.9 Einfache Ein-/Ausgabe 8
- 2.10 Einfache Kontrollstrukturen 10
  - *2.10.1 Verzweigung (if-Anweisung) 10*
  - *2.10.2 Die while-Schleife 11*
- 2.11 Geltungsbereiche und Lebensdauer 13
  - *2.11.1 Lokal: 13*
  - *2.11.2 Global: 13*
  - *2.11.3 Geltungsbereich Klasse: 14*
  - *2.11.4 Geltungsbereich Namensraum (namespace) 14*
  - *2.11.5 using-Direktive 14*
  - *2.11.6 Lebensdauer 14*

3 Elementare Datentypen 15
- 3.1 Ganzzahlige Datentypen 15
- 3.2 Gleitpunkt-Datentypen 16
- 3.3 Der logische Datentyp `bool` 19
- 3.4 Der Datentyp `char` 20
- 3.5 Zeichenketten 22
- 3.6 Die Standardklasse string 22

4 Ausdrücke und Operatoren 25
- 4.1 Arithmetische Ausdrücke 25
- 4.2 Mathematische Funktionen 27
  - *4.2.1 Einige mathematische Funktionen aus der ANSI-C-Bibliothekl 29*
  - *4.2.2 Einschub: Benutzerdefinierte Funktionen 31*
- 4.3 Wertzuweisungen 33
  - *4.3.1 L-Wert und R-Wert 33*
  - *4.3.2 Wertzuweisung als Ausdruck 33*
  - *4.3.3 Wertzuweisung mit Operatoren 34*
  - *4.3.4 Inkrement- und Dekrementoperatoren 34*
- 4.4 Sonstige Operatoren 35
  - *4.4.1 Bit-Operatoren 35*
  - *4.4.2 Der Bedingungs-Operator 36*
- 4.5 Datentypumwandlungen 37
  - *4.5.1 Implizite Typumwandlung 37*
  - *4.5.2 Sichere Standardumwandlungen: 37*
  - *4.5.3 Explizite Typumwandlungen 38*
  - *4.5.4 Sichere Typumwandlungen 38*

5 Anweisungen und Kontrollstrukturen 40
- 5.1 Einleitung 40
- 5.2 Zusammengesetzte Anweisung und Blockstruktur 41
- 5.3 Auswahlanweisungen 42
  - *5.3.1 Die if-Anweisung 42*
  - *5.3.2 Die switch-Anweisung 44*

5.4 Wiederholungsanweisungen 46
5.4.1 Die for-Anweisung 46
5.4.2 Die while-Anweisung 49
5.4.3 Die do-while-Anweisung 51
5.5 Sprünge 53
5.5.1 Die break-Anweisung 53
5.5.2 Die continue-Anweisung 54
5.6 Beispiele für Iterationen 55
5.6.1 Die Exponentialfunktion 55
5.6.2 Der Potenzierungsalgorithmus 57

6 Ein-/Ausgabe 59
6.1 Die iostream-Bibliothek 59
6.2 Vordefinierte iostream-Objekte 59
6.3 Ein-Ausgabe der elementaren Datentypen 60
6.4 Einfache Fehlerbehandlung 61
6.5 Formatierungsmöglichkeiten bei der Ein-/Ausgabe 64
6.5.1 Feldbreite, Ausrichtung und Füllzeichen 64
6.5.2 Formatierung ganzer Zahlen 65
6.5.3 Formatierung von Gleitpunktzahlen 66
6.5.4 Formatierungsmöglichkeiten bei istream 67

7 Funktionen 69
7.1 Einführung 69
7.2 Deklaration und Definition von Funktionen 70
7.3 Funktionsaufruf und Parameterübergabe 71
7.3.1 Funktionstyp 71
7.3.2 Wertaufruf 71
7.3.3 Referenzaufruf 72
7.4 Inline-Funktionen 73
7.5 Überladen von Funktionsnamen 73
7.5.1 Signatur einer Funktion 74
7.5.2 Einschränkungen beim Überladen von Funktionsnamen 74
7.5.3 Regeln zum Auffinden der am besten passenden Funktion 74
7.6 Standard-Argumente 75

8 Zusammengesetzte und benutzerdefinierte Datentypen 77
8.1 Aufzählungstypen 77
8.2 Felder 80
8.2.1 Eindimensionale Felder 80
8.2.2 Felder als Funktionsargumente 82
8.2.3 Mehrdimensionale Felder 84
8.3 Zeichenketten 85
8.3.1 Einführung 85
8.3.2 Ein- und Ausgabe von Zeichenketten 86
8.3.3 Zeichenketten-Funktionen 86
8.4 Strukturen 89
8.4.1 Einführung 89
8.4.2 Strukturen und Funktionen 90
8.4.3 Arrays von Strukturen 92
8.5 Der sizeof-Operator 93

9 Programmstrukturierung 94
9.1 Zerlegung von Programmen in mehrere Quelldateien 94
9.1.1 Stack (Stapel) 94
9.2 Der C++-Präprozessor und bedingte Übersetzung 99
9.2.1 Die wichtigsten Präprozessor-Direktiven 99
9.2.2 Anwendung der Programmstrukturierung 101

10 Dateiverarbeitung 105
10.1 Einführung 105
10.2 Die File-Stream-Klassen 106
10.3 Öffnen und Schließen 107

10.4 Sequentielles Lesen und Schreiben 109
10.5 Blockorientiertes Lesen und Schreiben 111

11 Zeiger 114
11.1 Einführung 114
11.2 Zeiger und Strukturen 119
11.3 Zeiger und Felder 120
11.4 Speicherreservierung zur Laufzeit 123
11.4.1 Der Operator `new` 124
11.4.2 Der Operator `delete` 125
11.4.3 Dynamisch allokierte Felder 126
11.5 Zeigerfelder und Kommandozeilenparameter 128
11.6 Zeiger auf const-Objekte und const-Zeiger 130
11.7 Dynamische Datenstrukturen 131
11.7.1 Einfach verkettete lineare Listen 131
11.7.2 Doppelt verkettete lineare Listen 136

12 Klassen 138
12.1 Aufbau von Klassen 138
12.2 Elementfunktionen 141
12.3 Der this-Zeiger 144
12.4 const-Elementfunktion 144
12.5 Konstruktoren und Destruktoren 148
12.5.1 Konstruktoren 148
12.5.2 Destruktoren 148
12.6 Initialisierung von Attributen 154
12.6.1 Initialisierung von Elementobjekten 154
12.6.2 Initialisierung von Konstanten und Referenzen 155
12.6.3 Objekt-Arrays 156
12.7 Kopierkonstruktor und Zuweisungsoperator 156
12.8 friend-Funktionen und friend-Klassen 161
12.9 Klassenattribute und Klassenmethoden 163

13 Überladen von Operatoren 167
13.1 Einführung 167
13.2 Operatorfunktionen als Elementfunktion oder friend-Funktion 168
13.3 Beispiel: Klasse für rationale Zahlen 170
13.4 Der Index-Operator [ ] 177
13.5 Der Operator ( ) 180
13.6 Intelligente Zeiger: Die Operatoren -> und * 183
13.7 Benutzerdefinierte Typumwandlungen 184
13.8 Eine String-Klasse 185

14 Vererbung 190
14.1 Einführung 190
14.2 public-, protected- und private-Vererbung 193
14.2.1 public-Ableitung 193
14.2.2 private-Ableitung 194
14.2.3 protected-Ableitung 195
14.3 Attribute und Methoden 196
14.4 Konstruktoren und Destruktoren 198
14.5 Statisches und dynamisches Binden 200
14.6 Virtuelle Destruktoren 205
14.7 Vererbung bei Kopierkonstruktor und Zuweisungsoperator 205
14.8 Vererbung und friends 208
14.9 Abstrakte Klassen 210
14.10 Mehrfachvererbung 213

15 Parametrisierte Funktionen und Klassen (Templates) 220
15.1 Parametrisierte Funktionen (Funktions-Templates) 220
15.2 Parametrisierte Klassen (Klassen-Templates) 226
15.2.1 Einführung 226
15.2.2 Klassenattribute und Klassenmethoden in parametrisierten Klassen 231

15.2.3 Vererbung bei parametrisierten Klassen....232
15.2.4 Beispiel: Array 235
15.2.5 Beispiele aus der ANSI-Klassenbibliothek....238

16 Ausnahmebehandlung 242
16.1 Einführung 242
16.2 Ausnahmebehandlung in C++....243
16.3 Standardausnahmen....252

17 Laufzeit-Typinformationen und Typkonvertierungen 256
17.1 Einführung 256
17.2 Typen identifizieren mit `typeid` und `type_info`....256
17.3 Dynamische Typumwandlungen mit `dynamic_cast`....259
17.4 Der Operator `static_cast`....261
17.5 Der Operator `reinterpret_cast` 262
17.6 Der operator `const_cast`....263

18 Namensräume....265

Anhang 270
A.1. Standardheaderdateien von ANSI-C: 270
A.2. Reservierte Wörter 270
A.3. Übersicht über C++-Operatoren....271
A.3.1. Die Operatoren `.*` und `->*` 272
A.4. Das Einbinden von C-Funktionen....273
A.5. Logische Einteilung von Elementfunktionen....273
A.6. Vordefinierte Elementfunktionen bei Klassen 273
A.7. Container-Operationen....274

Literatur

Koenig, Andrew Moo, Barbara	Intensivkurs C++ Addison-Wesley 2003, €34,95 ISBN 3-8273-7029-9
Prinz/Kirch-Prinz	C++. Alles zur Objektorientierten Programmierung Galileo Press GmbH, Bonn 2001, €44,90 ISBN 3-89842-171-6
Schader/Kuhlins	Programmieren in C++ Einführung in den Sprachstandard C++ Version 3.0, Springer Verlag, 4. Auflage, 1997
Breymann, Ulrich	C++ Eine Einführung, Hanser-Verlag, 4. Auflage, 1997 ISBN 3-446-19233-6
Stroustrup, Bjarne	Die C++ Programmiersprache 3. Auflage. Addison, Wesley 1997 ISBN 3-8273-1296-5
Eckel, Bruce	Thinking in C++ Prentice Hall 1998 ISBN 0-13-917709-4 `http://www.BruceEckel.com`
Josuttis, Nicolai	Die C++-Standardbibliothek Addison-Wesley 1996 ISBN 3-8273-1023-7
Breymann, Ulrich	Die C++ Standard Template Library - Einführung, Anwendungen, Konstruktion neuer Komponenten Addison-Wesley 1996

1 Einführung

Nach wie vor ist C++ und auch noch die Programmiersprache C in der Praxis von sehr großer Bedeutung. Insbesondere bei der systemnahen Softwareentwicklung kommt man an C/C++ nicht vorbei. Allerdings ist C++ schwerer zu erlernen und schwerer zu beherrschen als z.B. Java und wird daher nicht mehr als erste Programmiersprache im Studium gelehrt.

Im folgenden sollen die wesentlichen Eigenschaften der Programmiersprache C++ eingeführt und detailliert besprochen werden.

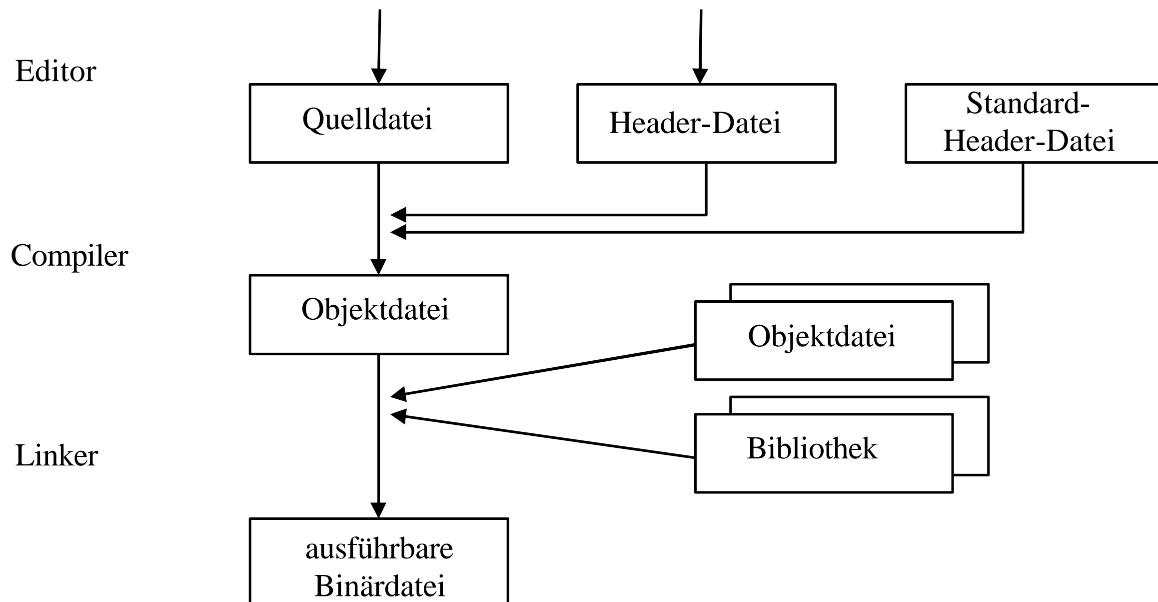
Derzeit (April 2004) ist dieses Skript erst noch in der Entstehungsphase. Es ist zunächst einmal eine Zusammenfassung aus den Skripten zu den früheren Vorlesungen Programmiersprachen 1 und 2 und ist dementsprechend für höhere Semester sicherlich noch zu ausführlich. Außerdem wird natürlich nirgendwo Bezug genommen auf die Programmiersprache Java. Dies soll im Laufe des Jahres anders werden.

- Es sollen neue, weniger triviale Beispiele aufgenommen werden
- Es soll detailliert auf Unterschiede zu Java eingegangen werden
- Das Skript soll kompakter werden.
- Ggf. sollen auch Aufgaben in das Skript aufgenommen werden.

Historie von C++:

1972	Entwicklung von C durch Dennis Ritchie und Ken Thompson zur Implementierung von Unix
1980	erste Version, als “C mit Klassen” entstanden bei AT&T Bell Laboratories entwickelt von Bjarne Stroustrup
1983	C++
1984	Einberufung eines ANSI-Komitees zur Standardisierung der Sprache
1987	Definition des ersten Standards durch das Buch von Stroustrup
Dezember 1996	ANSI-Draft (Aktuelle Version C++3.0)
November 1997	Verabschiedung des endgültigen ANSI-Standards

Entwicklung eines C++-Programmes



2 Grundelemente der Sprache C++

2.1 Ein erstes Programm

```
// hallo.cpp : Allererstes Programm

#include <iostream>

int main() {
    cout << "Hallo Welt!\n";
    return 0;
}
```

Erläuterungen:

`// ...`	einzeiliger Kommentar
`#include`	Einbinden einer sogenannten Header-Datei. d. h. einer Datei, in der sich Definitionen und Deklarationen von Funktionen, Daten, Datentypen und Makros befinden können. Die Datei wird dabei als Quelltext vor dem Übersetzen eingebunden und durch den sogenannten Präprozessor in den Programmquelltext hineinkopiert.
`iostream`	`iostream` oder `iostream.h` enthält die Definitionen für Typen und Funktionen der iostream-Bibliothek, also der Ein-/Ausgabe-Bibliothek.
`main`	Hauptfunktion eines C++-Programms. Alle Anweisungen und Ausdrücke zu `main` stehen zwischen den geschweiften Klammern.
`cout << "..."`	`cout` ist die sogenannte Standardausgabe, ein Objekt aus der `iostream`-Bibliothek, das verschiedene Ausgabefunktionalitäten zur Verfügung stellt. Mit << können Ausdrücke ausgegeben werden.
`\n`	bedeutet, dass ein Linefeed-Zeichen eingefügt wird, also ein Zeilenvorschub.
`return 0`	Mit `return` wird eine Funktion verlassen. Mit dem Verlassen der `main`-Funktion ist auch das Programm beendet. Die Zahl 0 wird als Rückgabewert des zu dem Programm gehörigen Prozesses an das Betriebssystem zurückgegeben. Dabei bedeutet 0, dass das Programm fehlerfrei abgelaufen ist.

Zwischenräume und Leerzeilen

Zwischenräume wie Leerzeichen (Blanks), Tabulatorzeichen oder Zeilenwechsel, werden benutzt, um den Quelltext optisch (d.h. für den menschlichen Leser übersichtlich) zu gliedern.

Notwendig aus der Sicht von C++ ist Zwischenraum nur an wenigen, seltenen Stellen. Es spielt auch keine Rolle, wie viel Zwischenraum eingeschoben wird. Ein einzelnes Leerzeichen ist gleichwertig mit eintausend Leerzeilen.

Unser obiges Beispiel könnte daher genausogut auch so aussehen:

```
#include <iostream>int main(){cout<<"Hallo Welt!\n";return 0;}
```

Generell ist zu beachten, dass ein einheitliches und übersichtliches Layout bei C++-Programmen einzuhalten ist.

2.2 Zeichenvorrat

- Große und kleine Buchstaben werden von C++ als unterschiedlich betrachtet.
- Nationale Sonderzeichen wie z.B. deutsche Umlaute können nicht für Namen innerhalb von C++-Programmen verwendet werden.

<table>
<tr><td>Buchstaben</td><td>a b c d e f g h i j k l m n o p q r s t u v w x y z _
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z</td></tr>
<tr><td>Ziffern</td><td>0 1 2 3 4 5 6 7 8 9</td></tr>
<tr><td>Sonderzeichen</td><td>{ } [ ] ( ) # < > = : + - * / % & | , ; ? ~ ^ " '</td></tr>
</table>

Alle Zeichen werden intern im ASCII-Code dargestellt. Der ASCII-Zeichensatz ist ein 7-Bit-Zeichensatz, der vollständig im ISO-Zeichensatz enthalten ist.

ISO Latin 1 Character Set (ISO 8859-1)

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>0x</td><td>NUL</td><td>SOH</td><td>STX</td><td>ETX</td><td>EQT</td><td>ENQ</td><td>ACK</td><td>BEL</td><td>BS</td><td>HT</td><td>LF</td><td>VT</td><td>FF</td><td>CR</td><td>SO</td><td>SI</td></tr>
<tr><td>1x</td><td>DLE</td><td>DC1</td><td>DC2</td><td>DC3</td><td>DC4</td><td>NAK</td><td>SYN</td><td>ETB</td><td>CAN</td><td>EM</td><td>SUB</td><td>ESC</td><td>FS</td><td>GS</td><td>RS</td><td>US</td></tr>
<tr><td>2x</td><td>SPC</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>3x</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>4x</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>5x</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>^</td><td>_</td></tr>
<tr><td>6x</td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>7x</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>~</td><td>DEL</td></tr>
<tr><td>8x</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>9x</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Ax</td><td></td><td>¡</td><td>¢</td><td>£</td><td>¤</td><td>¥</td><td>¦</td><td>§</td><td>¨</td><td>©</td><td>ª</td><td>«</td><td>¬</td><td>–</td><td>®</td><td>¯</td></tr>
<tr><td>Bx</td><td>°</td><td>±</td><td>²</td><td>³</td><td>´</td><td>µ</td><td>¶</td><td>·</td><td>¸</td><td>¹</td><td>º</td><td>»</td><td>¼</td><td>½</td><td>¾</td><td>¿</td></tr>
<tr><td>Cx</td><td>À</td><td>Á</td><td>Â</td><td>Ã</td><td>Ä</td><td>Å</td><td>Æ</td><td>Ç</td><td>È</td><td>É</td><td>Ê</td><td>Ë</td><td>Ì</td><td>Í</td><td>Î</td><td>Ï</td></tr>
<tr><td>Dx</td><td>Ð</td><td>Ñ</td><td>Ò</td><td>Ó</td><td>Ô</td><td>Õ</td><td>Ö</td><td>×</td><td>Ø</td><td>Ù</td><td>Ú</td><td>Û</td><td>Ü</td><td>Ý</td><td>Þ</td><td>ß</td></tr>
<tr><td>Ex</td><td>à</td><td>á</td><td>â</td><td>ã</td><td>ä</td><td>å</td><td>æ</td><td>ç</td><td>è</td><td>é</td><td>ê</td><td>ë</td><td>ì</td><td>í</td><td>î</td><td>ï</td></tr>
<tr><td>Fx</td><td>ð</td><td>ñ</td><td>ò</td><td>ó</td><td>ô</td><td>õ</td><td>ö</td><td>÷</td><td>ø</td><td>ù</td><td>ú</td><td>û</td><td>ü</td><td>ý</td><td>þ</td><td>ÿ</td></tr>
</table>

2.3 Kommentare

Kommentare dienen der besseren Lesbarkeit von Programmen und sollten generell nicht zu sparsam eingesetzt werden.

C++ bietet hier zwei Möglichkeiten:

Einzeilenkommentar:

```
// der Rest der Zeile ist Kommentar
```

Klassischer C-Kommentar

```
/* Dies ist ein C-Kommentar
   der ueber mehrere Zeilen
   geht
*/
```

2.4 Bezeichner

In einem Programm braucht man zur Bezeichnung von gewissen Größen Namen, auch Bezeichner (engl. identifier) genannt. Bezeichner werden benötigt, um Konstanten, Variablen, Datentypen und Funktionen Namen zu geben, unter denen man diese ansprechen können soll.

Namen können (fast) frei erfunden werden, müssen aber aus Buchstaben, Ziffern und dem Unterstrich-Zeichen (_) zusammengesetzt werden. Das erste Zeichen darf keine Ziffer sein.

Korrekte Beispiele sind:

```
a
DasIstEinZwarZiemlichLangerAberTrotzdemZulaessigerVariablenbezeichner
Das_ist_ein_zwar_ziemlich_langer_aber_trotzdem_zulaessiger_Bezeichner
a0000001
a0000002
_
```

Unzulässige Beispiele sind:

```
1teVariable
Erste-Variable
Erste Variable
DasIstEinLangerUnzulaessigerVariablenbezeichner!
dies+das
```

Regel: Bezeichner dürfen erst verwendet werden, wenn sie vorher definiert wurden.

2.5 Reservierte Wörter

Es gibt in jeder Programmiersprache eine Reihe an reservierten Wörtern, die nicht als benutzerdefinierte Namen verwendet werden dürfen.

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

2.6 Elementare Datentypen

Beispiel:

```
// summe.cpp : Summe zweier Zahlen berechnen

#include <iostream>

int main() {
    int a, b;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    int summe = a + b;
    cout << "Summe = " << summe << endl;
    return 0;
}
```

Beispiel:

```
// kreis.cpp : Flaeche und Umfang eines Kreises berechnen

#include <iostream>

int main() {
    const double PI = 3.141592653589793;
    double radius;
    cout << "Radius = ";
    cin >> radius;
    double flaeche = radius * radius * PI;
    double umfang  = 2 * PI * radius;
    cout << "Flaeche : " << flaeche << endl;
    cout << "Umfang  : " << umfang  << endl;
    return 0;
}
```

Zur Speicherung von Zahlenwerte kennt C++ zwei Grund-Datentypen

`int`	ganze Zahlen wie z.B. `12   1234   -12   -1234`

`double`	Gleitpunktzahlen wie z.B.
	`1.5   -12.123   +3.14   2.`
	`1.2E5   -2.3e-12`

Weitere Varianten dieser Typen und weitere Elementare Datentypen werden wir später kennenlernen.

2.7 Variablen und Konstanten

Zahlenwerte und andere Werte können in sogenannten Variablen abgespeichert werden.

`int a;`	definiert eine Variable mit Namen `a` und Typ `int`.
`double flaeche;`	definiert eine Variable mit Namen `flaeche` und Typ `double`.

Variablen belegen einen Platz im Hauptspeicher und sind nach ihrer Definition noch ohne konkreten Wert.

Initialisierung von Variablen

Variablen können bereits bei ihrer Definition mit einem Wert versehen werden:

```
int summe = 0;
double flaeche = 0.0;
```

Zuweisung von Werten an Variablen

```
int summe = 0; // Initialisierung
summe = 100;   // Zuweisung eines Wertes
```

Konstanten

Variablen sind, wie der Name schon sagt, variabel. Will man nun Daten speichern, die nicht verändert werden sollen, so definiert man Konstanten:

```
const double PI = 3.141592653589793;


PI = 4.0;      // Fehler !!
```

Konstanten müssen bei ihrer Definition initialisiert werden und können danach nie mehr einen anderen Wert erhalten.

2.8 Operatoren und Ausdrücke

Um mit Zahlen zu rechnen oder Variableninhalte zu manipulieren, gibt es in C++ eine Vielzahl von Operatoren. Wir betrachten zunächst nur die arithmetischen Operatoren.

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Operator

Variablen, Konstanten, Ganzzahl- und Gleitpunktzahl-Werte können nun mit Hilfe von Operatoren und Klammern zu Ausdrücken kombiniert werden.

```
a + b

2 * a + b

2 * (a + b)

PI * radius * radius
```

Ausdrücke müssen natürlich gewissen Regeln genügen, über die wir später etwas genauer reden werden.

2.9 Einfache Ein-/Ausgabe

In den beiden Beispielen haben wir einfache Möglichkeiten der Ausgabe und der Eingabe kennengelernt.

```
cout << "Flaeche : " << flaeche << endl;
// Ausgabe eines Textes und eines Variableninhaltes auf cout

cin >> radius;
// Einlesen eines double-Wertes vom Standard-Eingabe-Objekt cin

cin >> a;
// Einlesen eines int-Wertes vom Standard-Eingabe-Objekt cin

cin >> a >> b;
// Einlesen zweier int-Werte vom Standard-Eingabe-Objekt cin
```

Über `cin` und `cout` können jeweils beliebig viele Ausdrücke verarbeitet werden. Die Ausdrücke können sich dabei auch über mehrere Zeilen erstrecken

```
cout << "Flaeche : "
     << flaeche
     << endl;        // endl bedeutet Ausgabe \n und Puffer entleeren
```

Beispiel: Zinsberechnung

Die Formel für die Zinsberechnung sieht folgendermaßen aus

$$\text{Zinsen} = \frac{\text{Kapital} * \text{Zinssatz} * \text{Tage}}{100 * 360}$$

Vorgehensweise bei der Programmerstellung:

- erst nachdenken, dann programmieren!
- Strukturieren des Problems z. B. mit Hilfe von Pseudocode oder Struktogrammen
- Umsetzen in die Programmiersprache
- Testen

Struktogramm:

Eingabe Kapital
Eingabe Zinssatz
Eingabe Anzahl Tage
Berechnung des Zinsbetrags
Ausgabe des Zinsbetrags

Programm:

```
// zins.cpp : Einfache Zinsberechnung

#include <iostream>

int main() {
    const int TAGE_PRO_JAHR = 360;
    const int HUNDERT = 100;
    double kapital, zinssatz, zinsen;
    int tage;
```

```
    cout << "Kapital  = "; cin >> kapital;
    cout << "Zinssatz = "; cin >> zinssatz;
    cout << "Tage     = "; cin >> tage;

    zinsen = kapital * zinssatz * tage / (TAGE_PRO_JAHR * HUNDERT);

    cout << "Die Zinsen betragen DM " << zinsen << endl;
    return 0;
}
```

2.10 Einfache Kontrollstrukturen

Bisher haben wir in unseren Programmen nur sequentielle Abläufe betrachtet. Für realistischere Anwendungen werden aber auch Entscheidungen aufgrund von Bedingungen, sowie Schleifen benötigt.

2.10.1 Verzweigung (if-Anweisung)

Die if-Anweisung, die es in allen Programmiersprachen in ähnlicher Form gibt, dient zur Verzeigung in Alternativabläufe eines Programms.

	Syntax	*Struktogramm-Darstellung*
einseitige Alternative	`if (Bedingung) { Ja-Anweisungen }`	Bedingung / wahr / Ja-Anweisungen
zweiseitige Alternative	`if (Bedingung) { Ja-Anweisungen } else { Nein-Anweisungen }`	Bedingung / wahr / falsch / Ja-Anweisungen / Nein-Anweisungen

Bedeutung:

Wenn die Bedingung erfüllt ist, werden die Ja-Anweisungen ausgeführt, ansonsten, falls vorhanden, die Nein-Anweisungen. Ist nur eine Ja-Anweisung vorhanden, so können die geschweiften Klammern auch weggelassen werden. Dies gilt natürlich auch bei nur einer Nein-Anweisung.

Beispiel: Bestimmung des Maximums zweier eingegebener Zahlen

Ablauf des Programms:

Pseudocode	*Struktogramm*
• Eingabe der ersten Zahl a • Eingabe der zweiten Zahl b • Falls a größer als b ist, setze das Maximum max auf a, sonst auf b • Gebe das Maximum aus	a eingeben b eingeben a > b ? (ja / nein) max = a / max = b max ausgeben

Programmtext:

```
// maximum.cpp : Maximum zweier eingegebener Zahlen

#include <iostream>

int main() {
    int a, b, max;
    cout << "a = ";  cin >> a;
    cout << "b = ";  cin >> b;

    if (a > b)
        max = a;
    else
        max = b;

    cout << "max = " << max << endl;
    return 0;
}
```

2.10.2 Die while-Schleife

Die while-Anweisung ist eine Schleifenanweisung, d. h. abhängig von der Erfüllung einer Bedingung wird an Anweisungsblock immer wieder wiederholt.

Syntax	*Struktogramm-Darstellung*

`while (Bedingung) {`
`    Wiederholungsanweisungen`
`}`
	Bedingung
	Wiederholungsanweisungen

Bedeutung:

1. Auswertung der sogenannten Schleifenbedingung.
2. Ist sie wahr, so wird der Schleifenrumpf, also die Wiederholungsanweisungen einmal ausgeführt, ansonsten wird der Schleifenrumpf übersprungen und mit der Anweisung hinter der while-Anweisung fortgefahren.
3. Gehe zurück zu Punkt 1.

Beispiel: Berechne die Summe der ersten n Zahlen

Pseudocode	*Struktogramm*

• Eingabe der Zahl n
• Setze Zähler i auf 1
• Setze Summe auf 0
• Solange i noch kleiner oder gleich n
  • erhöhe Summe um i
  • erhöhe i um 1
• Gebe die Summe aus
	n eingeben
	i = 1
	summe = 0
	solange i <= n
	  summe = summe + i
	  i = i + 1
	summe ausgeben

Programmtext:

```
// summe2.cpp : Summe der ersten n Zahlen berechnen

#include <iostream>

int main() {
    int i = 1;
    int summe = 0;
    int n;
    cout << "n = "; cin >> n;

    while (i <= n) {
       summe = summe + i;
       i = i + 1;
    }

    cout << "Summe = " << summe << endl;
    return 0;
}
```

Bemerkungen:

- Besteht der Schleifenblock der while-Anweisung nur aus einer Anweisung, so können die geschweiften Klammern auch weggelassen werden.
- Innerhalb einer Schleife können selbstverständlich auch if-Anweisungen und weitere Schleifen vorkommen.

Erweiterung des Beispiels Zinsberechnung:

Das Beispielprogramm Zinsberechnung soll um eine Schleife erweitert werden, so dass der Benutzer des Programms nach Ausgabe des errechneten Zinsbetrages gefragt wird:

```
Noch einmal (j/n) ?
```

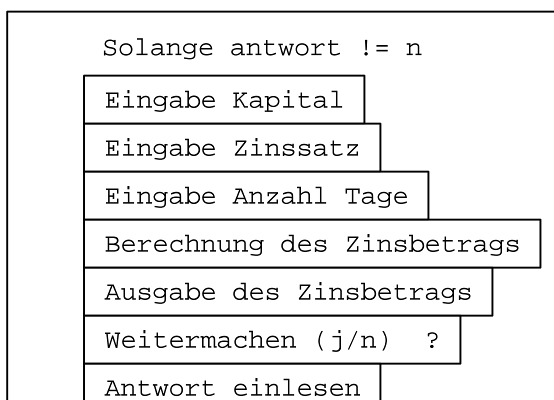
Für die Eingabe einer Antwort benötigen wir noch einen weiteren Datentyp, den Datentyp `char`. Der Datentyp `char` belegt üblicherweise 1 Byte im Speicherplatz und kann ein Zeichen des ANSI/ISO-Zeichensatzes aufnehmen.

Beispiel:

```
char c = 'x';  // char-Konstante
```

Nun zur Erweiterung unseres Zinsbeispiels.

Struktogramm:



Programm:

```
// zins2.cpp : Erweiterung der Zinsberechnung um eine Schleife

#include <iostream>

int main() {
    const int TAGE_PRO_JAHR = 360;
    const int HUNDERT = 100;
    double kapital, zinssatz, zinsen;
    int tage;
    char antwort = ' ';

    while (antwort != 'n') {
        cout << "Kapital  = "; cin >> kapital;
        cout << "Zinssatz = "; cin >> zinssatz;
        cout << "Tage     = "; cin >> tage;

        zinsen = kapital * zinssatz * tage / (TAGE_PRO_JAHR * HUNDERT);

        cout << "Die Zinsen betragen DM " << zinsen << endl;
        cout << "Weitermachen (j/n) ? ";
        cin >> antwort;
    }
    return 0;
}
```

2.11 Geltungsbereiche und Lebensdauer

Eine Deklaration bzw. Definition definiert einen Namen in einem Geltungsbereich (scope).

Bei C++ gibt es vier Geltungsbereiche:

2.11.1 Lokal:

Ein Name, der in einem Block vereinbart wird, ist nur bis zum Ende des Blocks bekannt.
Dazu gehören:

- lokale Variablen in Funktionen
- Funktionsargumente
- Namen, die innerhalb von `for, while, if, do...while` oder `switch` definiert sind.

```
for (int i=0; i < max; i++)
{
...
}  // Geltungsbereich von i endet hier
```

2.11.2 Global:

Namen, die außerhalb von Funktionen und Klassen definiert sind, sind global.
Zwei Varianten:

- *modulglobal*: Der Zusatz `static` zu einer Variablen oder Funktion sorgt dafür, dass diese nur innerhalb der Source-Datei zugreifbar ist.
- *extern*: Fehlt der Zusatz `static`, so ist der Geltungsbereich überall dort, wo der Name deklariert ist

Der Zugriff auf global definierte Variablen kann innerhalb von Funktionen mit dem *Bereichsoperator ::* erfolgen

2.11.3 Geltungsbereich Klasse:

Namen, die innerhalb einer Klasse definiert sind, sind je nach Definition nur innerhalb der Klasse und ihrer Methoden bekannt.

→ später mehr

2.11.4 Geltungsbereich Namensraum (*namespace*)

Namensräume werden vor allem benutzt, um die in Klassen-Bibliotheken verwendeten Namen nach außen abzugrenzen.

Beispiel:

```
namespace A {
  void f(int);
  void f(char);
  class String {…};
}
```

```
namespace B {
  void f(int);
  void f(char);
  class String {…};
}
```

Anwendung:

```
A::String s1="Hallo";
A::f(1);
B::String s2="Welt"; // B:: ist eine eindeutige Qualifikation
// Vorsicht! s1=s2 ist falsch!
B::f(1);
```

Der Zugriff auf Größen, die innerhalb eines Namensraums definiert sind, kann von außerhalb nur mit Hilfe des Bereichsoperators erfolgen. Die ANSI-C++-Standardbibliotheken sind alle im Namensraum `std` definiert.

2.11.5 using-Direktive

Durch eine using-Direktive können sämtliche Namen eines angegebenen Namensraums zugreifbar gemacht werden.

Syntax:

```
using namespace Namespace_Name;

namespace A {
   class X {…};
   void f();
}
void g() {
   using namespace A;
   X x1;       // A::X
   f();        // A::f()
};
```

Voreinstellung für den Standard-Namensraum:

```
using namespace std;
```

2.11.6 Lebensdauer

- Automatische Variablen: bis Blockende bzw. Programmende (optionaler Zusatz: `auto`)
- Statische Variablen: von der Definition bis zum Programmende (Zusatz: `static`)
- Dynamische Variablen: vom Erzeugen (`new`) bis zum Entfernen (`delete`)

3 Elementare Datentypen

C++ enthält eine Reihe von vordefinierten Datentypen, die sogenannten elementaren Datentypen.

3.1 Ganzzahlige Datentypen

Typ	*repräsentiert*	*Größe*	*Wertebereich*
`short`	kurze ganze Zahl mit Vorzeichen	16 Bits	$-2^{15} \ldots 2^{15} - 1$
`int`	ganze Zahl mit Vorzeichen	32 Bits	$-2^{31} \ldots 2^{31} - 1$
`long`	lange ganze Zahl mit Vorzeichen	32 Bits	$-2^{31} \ldots 2^{31} - 1$

Ganzzahlige Datentypen werden intern im Binärformat gespeichert, d. h. als Folge von Nullen und Einsen. Deshalb können die Typen `short`, `int` und `long` höchstens Werte aus den obigen Intervallen aufnehmen. Die Größe von `int` und `long` ist allerdings implementierungsabhängig. Bei 16-Bit-Prozessoren ist `int` üblicherweise nur 16 Bits lang, während `long` bei 64-Bit-Prozessoren 64 Bits lang sein kann.

Beispiel: Ausgabe der Grenzen

```
// limits1.cpp : Ober- und Untergrenzen der ganzzahligen Datentypen

#include <iostream>
#include <climits>

int main() {
    cout << "short " << SHRT_MIN << " ... " << SHRT_MAX << '\n'
         << "int   " << INT_MIN  << " ... " << INT_MAX  << '\n'
         << "long  " << LONG_MIN << " ... " << LONG_MAX << '\n';
    return 0;
}
/* Ausgabe
short -32768 ... 32767
int   -2147483648 ... 2147483647
long  -2147483648 ... 2147483647  */
```

interne Darstellung:

5	0000 0000	0000 0000	0000 0000	0000 0101
-5	1111 1111	1111 1111	1111 1111	1111 1011

Das Negative einer Zahl wird gebildet, indem alle Bits konvertiert werden und dann auf das Ergebnis 1 addiert wird.

Frage: Was kommt heraus, wenn man auf die `short`-Zahl `32767` etwas dazuaddiert?

Ganze Zahlen ohne Vorzeichen

Durch den Zusatz unsigned bei der Definition einer ganzzahligen Variablen kann man erreichen, dass die interne Binärdarstellung immer als positive Zahl interpretiert wird. Dadurch ändern sich die Wertebereiche wie folgt:

Typ	*repräsentiert*	*Größe*	*Wertebereich*
`unsigned short`	kurze ganze Zahl ohne Vorzeichen	16 Bits	$0 \ldots 2^{16} - 1$
`unsigned int`	ganze Zahl ohne Vorzeichen	32 Bits	$0 \ldots 2^{32} - 1$
`unsigned long`	lange ganze Zahl ohne Vorzeichen	32 Bits	$0 \ldots 2^{32} - 1$

Beispiel: Ausgabe der Grenzen

```
// limits2.cpp : Ober- und Untergrenzen der vorzeichenlosen
//               ganzzahligen Datentypen

#include <iostream>
#include <climits>

int main() {
    cout << "unsigned short " << "0 ... " << USHRT_MAX << '\n'
         << "unsigned int   " << "0 ... " << UINT_MAX  << '\n'
         << "unsigned long  " << "0 ... " << ULONG_MAX << '\n';
    return 0;
}
/* Ausgabe
unsigned short 0 ... 65535
unsigned int   0 ... 4294967295
unsigned long  0 ... 4294967295  */
```

Ganzzahlige Literale

Literale sind konstante Ausdrücke, die einen festen Wert repräsentieren. Jedes Literal hat einen eindeutigen zugehörigen Datentyp

Literale vom Typ int:

```
12   -12   0   1234        Dezimaldarstellung
033                        Oktaldarstellung
0x1b                       Hexadezimaldarstellung
```

Literale vom Typ long:

```
123l   1234567L   -45678L  Dezimaldarstellung
07777777                   Oktaldarstellung
0xffffffff                 Hexadezimaldarstellung
```

unsigned Literale:

```
157u   5436U    0456U      unsigned int
12345UL   0xffffffUL       unsigned long
```

3.2 Gleitpunkt-Datentypen

Gleitpunkt-Zahlen können Nachkommastellen und einen Exponent-Anteil haben. Gleitpunktzahlen werden in C++ durch die folgenden drei Datentypen dargestellt:

Typ	*Größe*	*Wertebereich*	*Stellen Genauigkeit*
`float`	32 Bits	$+/-3.4E^{-38} \ldots +/-3.4E^{38}$	7

double	64 Bits	+/-1,7E^{-308} ... +/-1.7E^{308}	15
long double	90 Bits	+/-3.4E^{-4932} ... +/-1.1E^{4932}	19

Auch für diese Datentypen gilt, dass sie von Rechner zu Rechner ggf. unterschiedlich dargestellt sein können. Intern werden Mantisse und Exponent jeweils durch Binärzahlen einer bestimmten Bitbreite dargestellt. Diese Bitbreiten können von Implementierung zu Implementierung variieren.

Beispiel:

Anzahl der Bits			
	float	double	long double
Vorzeichen	1	1	1
Mantisse	23	52	64
Exponent-Vorzeichen	1	1	1
Exponent	7	10	14

Beispiel:

```
// limits3.cpp : Ober- und Untergrenzen der Gleitpunkt-Typen

#include <iostream>
#include <cfloat>

int main() {
    cout << "float:\n"
         << "Anzahl Stellen " << FLT_DIG << '\n'
         << "Minimaler Wert " << FLT_MIN << '\n'
         << "Maximaler Wert " << FLT_MAX << "\n\n";

    cout << "double:\n"
         << "Anzahl Stellen " << DBL_DIG << '\n'
         << "Minimaler Wert " << DBL_MIN << '\n'
         << "Maximaler Wert " << DBL_MAX << "\n\n";

    cout << "long double:\n"
         << "Anzahl Stellen " << LDBL_DIG << '\n'
         << "Minimaler Wert " << LDBL_MIN << '\n'
         << "Maximaler Wert " << LDBL_MAX << "\n\n";
    return 0;
}
/* Ausgabe
float:
Anzahl Stellen 6
Minimaler Wert 1.17549e-38
Maximaler Wert 3.40282e+38

double:
Anzahl Stellen 15
Minimaler Wert 2.22507e-308
Maximaler Wert 1.79769e+308

long double:
Anzahl Stellen 18
Minimaler Wert 3.3621e-4932
Maximaler Wert 1.18973e+4932
 */
```

Bei einer Überschreitung der jeweils maximalen oder minimalen Werte eines Gleitpunktdatentyps stürzt das Programm ab mit einem "floating point overflow".

Gleitpunkt-Literale:

Gleitpunkt-Literale werden in C++ wie folgt beschrieben:

- Vorzeichen (optional)
- Vorkommastellen
- Dezimalpunkt
- Nachkommastellen
- e oder E und ganzzahliger Exponent (optional)
- Suffix f,F oder l,L (optional, zahlen ohne Suffix sind vom Typ double)

Beispiele:

```
-123.789   1.2E6   .5   -1.123e-5          double
1.23f   -123.45e12F                         float
1.234567890e123L                            long double
```

Rechengenauigkeit bei Gleitpunkt-Typen

Das Rechnen mit Gleitpunktzahlen kann Überraschungen mit sich bringen. Dazu schauen wir uns das folgende Beispiel an.

```
// floattest.cpp : Ungenauigkeit der Arithmetik zeigen

#include <iostream>

int main() {
    double x = -1.0;
    while ( x <= 0.0 ) {
       cout << x << ' ';
       x = x + 0.1;
    }
    cout << endl;

    cout << "0.01 - 0.1 * 0.1 = " << (0.01 - 0.1 * 0.1) << endl;
    return 0;
}
```

Die Ausgabe des Programms sieht folgendermaßen aus:

```
-1 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 -1.38778e-16
0.01 - 0.1 * 0.1 = -1.73472e-18
```

Wie lässt sich das erklären?

Da die Mantisse intern binär abgelegt ist, lässt sich ein Dezimalbruch wie `0.1` nicht exakt intern darstellen und deshalb ist `-0.1 + 0.1` nicht genau gleich `0` sondern nur ungefähr. Analog ist auch der Ausdruck `0.01` nicht genau gleich dem Wert des Ausdrucks `0.1 * 0.1`.

Wie lässt sich aber das folgende Beispiel erklären ?

```
// floattest2.cpp : Ungenauigkeit der Arithmetik zeigen

#include <iostream>
```

```
int main() {
    float a = 1.234567E-7,
          b = 1.000000,
          c = -b, x, y;
    x = a + b;
    x = x + c;
    y = a;
    y = y + b + c;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}
/* Ausgabe:
x = 1.19209e-07
y = 1.23457e-07
*/
```

3.3 Der logische Datentyp `bool`

Der Datentyp `bool` kann nur die beiden Wahrheitswerte `true` oder `false` annehmen. Intern wird ein `bool`-Wert üblicherweise in einem Byte gespeichert. Dabei entspricht dann dem Wert `true` die Binärzahl 1 und `false` die Zahl 0.

Beispiel:

```
// booltest.cpp: bool-Variablen testen

#include <iostream>

int main() {
    int i = 2, j = 3;
    bool b1 = i > j;
    bool b2 = i < j;

    cout << "b1 = " << b1 << endl;
    cout << "b2 = " << b2 << endl;

    return 0;
}
/* Ausgabe:
b1 = 0
b2 = 1   */
```

Vergleichsoperatoren:

Operator	***Bedeutung***
`ausdruck1 == ausdruck2`	Gleichheit zweier Ausdrücke prüfen
`ausdruck1 != ausdruck2`	Ungleichheit zweier Ausdrücke prüfen
`ausdruck1 <  ausdruck2`	Ist *ausdruck1* kleiner als *ausdruck2* ?
`ausdruck1 >  ausdruck2`	Ist *ausdruck1* größer als *ausdruck2* ?
`ausdruck1 <= ausdruck2`	Ist *ausdruck1* kleiner oder gleich *ausdruck2* ?
`ausdruck1 >= ausdruck2`	Ist *ausdruck1* größer oder gleich *ausdruck2* ?

Das Ergebnis eines Vergleichs ist immer ein Ausdruck vom Typ `bool`. Die in der `if`-Anweisung und der `while`-Anweisung erwarteten Bedingungen sollen üblicherweise `bool`-Ausdrücke sein.

Logische Operatoren:

Operator	*Bedeutung*
`ausdruck1 && ausdruck2`	logisches Und angewandt auf zwei `bool`-Ausdrücke
`ausdruck1 || ausdruck2`	logisches Oder angewandt auf zwei `bool`-Ausdrücke
`!ausdruck1`	Verneinung des `bool`-Ausdruckes *ausdruck1*

Die Anwendung dieser Operatoren auf Ausdrücke und das Bilden von zusammengesetzten `bool`-Ausdrücken werden wir etwas später erst genauer besprechen.

Das nachfolgende Beispiel demonstriert die Verwendung von zusammengesetzten logischen Ausdrücken aber am Beispiel des Schaltjahr-Tests.

```
// schaltjahr.cpp : Schaltjahr testen

#include <iostream>

int main() {
    bool programmEnde = false;
    char antwort;
    int jahr;

    while (!programmEnde) {
        cout << "Jahr zwischen 1582 und 3000 eingeben: ";
        cin >> jahr;

        if (jahr < 1582 || jahr > 3000)
            cout << jahr << " liegt nicht zwischen 1582 und 3000!\n";
        else
            if ((jahr % 4 == 0 && jahr % 100 != 0)
              || jahr % 400 == 0)
                cout << jahr << " ist ein Schaltjahr!\n";
            else
                cout << jahr << " ist kein Schaltjahr!\n";

        cout << "Noch einmal (j/n) ";
        cin >> antwort;
        programmEnde = antwort == 'n';
    }
    return 0;
}
```

3.4 Der Datentyp `char`

Für die Darstellung von Zeichen aus dem ASCII- oder ISO-Zeichensatz wird der Datentyp `char` verwendet. Eine `char`-Variable belegt intern 1 Byte. Zeichen werden normalerweise im ASCII- bzw. ISO-Code abgelegt.

Beispiele:

```
char c = 'x';               // Der Kleinbuchstabe x
char escape = '\x1b';       // das escape-Zeichen
char linefeed = '\n';
```

Besondere Zeichenkonstanten

Zeichen	*Bedeutung*	*ASCII-Name*
`\a`	Signalton (nur bei der Ausgabe)	BEL

`\b`	Backspace	BS
`\f`	Seitenvorschub (Formfeed)	FF
`\n`	neue Zeile, Linefeed (Wirkung wie \r und \v)	LF
`\r`	Zeilenrücklauf (Carriage Return)	CR
`\t`	Tabulator	HT
`\v`	Zeilensprung (vertikaler Tabulator)	VT
`\0`	Zeichenketten-Ende	NUL
`\\`	Backslash, auch `\x5C`	
`\'`	das Zeichen ', auch `\x27`	
`\"`	das Zeichen ", auch `\x22`	
ooo	Oktalcode, z. B. `\377`	
\x*hhh*	Hexadezimalcode, z.B. `\x1b`	

Der Datentyp `char` wird auch oft ge- bzw. missbraucht, um kleine Zahlen darzustellen. Seit dem ANSI-Standard werden die Typbezeichnungen `signed char` und `unsigned char` verwendet für 1-Byte-Ganzzahltypen mit bzw. ohne Vorzeichen. Wir werden davon jedoch keinen Gebrauch machen.

Da `char`-Größen intern genau wie ganzzahlige Größen gespeichert sind, ist es kein Problem `char`- und `int`-Ausdrücke miteinander zu vermischen.

```
char eins = 49;      // ASCII-Code 49 entspricht '1'
char zwei = eins + 1;// ASCII-Code 50 entspricht '2'
char a = 'A';
int i = a;           // erlaubt, i hat den Wert 97
```

Regel: Arithmetische Operatoren sollen im Zusammenhang mit `char`-Größen nur in begründeten Ausnahmefällen eingesetzt werden.

Beispiel

```
// chartest.cpp : Druckbare Zeichen ausgeben

#include <iostream>

int main() {
    int i = 0;
    char c = ' ';

    while ( c <= '~' ) {
       cout << c;
       c = c + 1;
       i = i + 1;
       if ( i == 40 ) {
          cout << endl;
          i = 0;
       }
    }
    cout << endl;
    return 0;
}
/* Ausgabe:
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFG
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
pqrstuvwxyz{|}~
*/
```

Erlaubte Operatoren für Zeichen:

`=`	Zuweisung
`==  !=  < > <= >= == !=`	Vergleichsoperatoren verglichen wird jeweils der interne Binärcode

Bemerkung:

Es gibt noch einen Datentyp für Zeichen, namens `wchar_t`, der 2 Bytes belegt und der alle Zeichen des Unicodes aufnehmen kann. Wir werden diesen Datentyp jedoch nicht weiter behandeln.

3.5 Zeichenketten

Es gibt in C/C++ keinen eigenen Datentyp für die Aufnahme von Zeichenketten. Zeichenketten werden in C/C++ als Folge von Zeichen (`char`) betrachtet, die durch das Nullzeichen (`\0`) begrenzt wird. Wir verwenden Zeichenketten vorerst nur als Zeichenkettenkonstanten, d. h. als Folgen von in Anführungszeichen geklammerten Zeichen.

```
cout << "Hallo\n"
```

Interne Darstellung:

Byte	0	1	2	3	4	5	6
Wert	72	97	108	108	111	10	0
Zeichen	'H'	'a'	'l'	'l'	'o'	'\n'	'\0'

Eine Zeichenkette ist also immer um ein Zeichen länger, als Zeichen sichtbar sind. Später werden wir Zeichenketten behandeln als Felder (Arrays) von Zeichen.

Im Programmtext aufeinanderfolgende, nur durch Leerzeichen getrennte Zeichenketten werden durch den Compiler zu einer Zeichenkette zusammengesetzt, d.h.

```
cout << "Dies ist ein sehr "
        "langer Satz";
```

bedeutet das selbe wie

```
cout << "Dies ist ein sehr langer Satz";
```

3.6 Die Standardklasse string

Viele große Probleme entstehen bei C durch den Umgang mit Zeichenketten. Zeichenketten werden als `char`-Arrays, beendet durch ein `'\0'`, dargestellt. Zum Umgang mit `char`-Strings gibt es eine Fülle von Funktionen, die teilweise nicht ohne Risiko einsetzbar sind. In C++ gibt es eine Lösung dafür, eine Standardklasse `string` für den Umgang mit Zeichenketten.

Die Klasse `string` erlaubt eine komfortable und sichere Handhabung von zeichenketten. Bei allen Operationen mit Strings wird der benötigte Speicherplatz automatisch reserviert bzw. freigegeben oder angepasst.

Die Klasse `string` ist in der Header-Datei `<string>` deklariert. Für `string`-Objekte steht eine Fülle von Operationen und Funktionen zur Verfügung, von denen einige nachfolgend vorgestellt werden sollen.

`string`-Objekte anlegen

```
string s1("Hallo");             // string anlegen und mit char-String
string s2 = "Welt";             // initialisieren
string s3;                // Leerstring
string s4(s2);                  // string-Objekt mit anderem string-
Objekt
                          // initialisieren

string ort="66117 Saarbrücken";
string plz(ort,0, 5);     // string mit Teilstring initialisieren
string linie (50, '-');   // eine Zeichenkette, die aus 50 MinusZeichen
                          // besteht
```

Zuweisungen und Verknüpfungen

```
s3 = s1 + " " + s2;             // string-Objekte konkatenieren und
zuweisen
s4 += " " + s2;           // string-Objekt anhängen
```

Eingabe und Ausgabe

```
cout << s3 << endl;             // string ausgeben

for(int i = 0; i < s3.size(); i++) // String zeichenweise ausgeben
   cout << s3[i] << ' ';
cout << endl;

cout << "Zwei Wörter eingeben: ";
cin >> s1 >> s2;          // Wortweises einlesen wie bei char-Strings
cout << s1 << "; " << s2 << endl;

getline(cin, s1);               // Zeile einlesen
getline(cin, s2, '@');    // String bis zu einem Trennzeichen lesen
cout << s1 << "; " << s2 << endl;
```

Vergleiche

```
string a("Albert"), z("Alberta");
string b = a;
if(a == b)          // inhaltlicher Vergleich
   cout << a << " == " << b << endl; // a == b
else
   cout << a << " != " << b << endl;

if (a < z)          // lexikographisch kleiner ?
   cout << a << " < " << z << endl;  // a < z

if(z > a)
   cout << z << " > " << a << endl;  // z > a

if( z != a)
   cout << z << " != " << a << endl; // z != a
```

Zugriff auf einzelne Zeichen

```
string a("Alberta");
a[6] = ' ';               // "Albert "
```

Bei Zugriff über `[]` findet allerdings keine Indexüberprüfung statt, deshalb sollte besser die Funktion `at()` verwendet werden.

```
a.at(10)= ' ';              // FEHLER !
```

Länge eines `string`-Objektes

Die Länge eines string-Objektes kann mit Hilfe der Elementfunktionen `size()` oder auch `length()` ermittelt werden.

```
string hallo = "Hallo Welt";
cout << "Länge: " << hallo.length() << endl; // 10
```

Konvertierung in einen C-String

```
string s1("Hallo");
const char* p = s1.c_str();      // p zeigt auf eine nicht änderbare
                                 // Kopie des Inhaltes von s1
```

Es gibt noch eine gewaltige Menge an `string`-Funktionen zum Suchen, Ersetzen, Einfügen und Löschen in Strings. Hierzu sei auf die Literatur, z. B. Kirch-Prinz, verwiesen.

In einem Beispiel aus PS1 hätte dann unsere Datenstruktur Student so aussehen können:

```
struct Student {
    int matrikelNr;
    string name;
    string vorname;
};
```

Vorteil gegenüber der `char`-String-Version: Name und Vorname können beliebig lang werden, ohne dass man damit einen besonderen programmiertechnischen Aufwand hätte

4 Ausdrücke und Operatoren

Ein Ausdruck steht für einen Wert, der entweder bereits bei der Übersetzung des Programmtextes ermittelt werden kann oder erst zur Laufzeit des Programmes berechnet wird. Ein Ausdruck besteht aus

- Operatoren (z.B. +, -, *, /)
- Operanden (z.B. Konstanten, Variablen, Funktionswerten)
- Interpunktionszeichen (z.B. Klammern)

Bei der Ermittlung eines Werts eines Ausdruckes sind Seiteneffekte möglich, d. h. bei bestimmten Operatoren ist es möglich, dass der Wert eines Operanden verändert werden kann. Jeder Ausdruck hat einen Wert eines bestimmten Datentyps. Dieser wird bestimmt durch die Typen der Operanden und durch die Operatoren.

4.1 Arithmetische Ausdrücke

Zur Bildung von arithmetischen Ausdrücken stehen zunächst die folgenden Operatoren zur Verfügung:

Priorität	***Operator***	***Bedeutung***	***Beispiele***
1	`+` `-`	unäre Vorzeichenoperatoren	`+1` `-a`

2	`*`
	`/`
	`%`
		Multiplikations-, Divisions- und Modulus-Operator	`a * b`
			`2 * -3`
			`3 / 2.0`
			`17 % 4`

3	`+`
	`-`
		Additions- und Subtraktionsoperator	`a + b`
			`2 - 3.14`
			`-a + -b`

C/C++ kennt keinen Potenzierungs-Operator wie andere Programmiersprachen.

Beispiele mit ganzzahligen Operanden

Ausdruck	***Wert***	
`1+2`	`3`	
`230-117`	`113`	
`-34*12`	`-408`	
`23*-12`	`-276`	Der Vorzeichenoperator bindet stärker als *
`23+-8`	`15`	
`23--8`	`31`	
`-1--1`	`0`	
`17 - 4 - 3 * 2`	`7`	

Beispiele für die Anwendung des Modulus-Operators:

Ausdruck	***Wert***	
`11%3`	`2`	
`21%2`	`1`	Test auf gerade Zahl
`22%2`	`0`	Test auf gerade Zahl
`23%50`	`23`	
`-7%2`	`-1`	mathematisch wäre das +1 !
`7%-2`	`1`	mathematisch unmöglich

```
-7%-2          -1
```

Die letzten Ausdrücke entsprechen nicht der Modulus-Funktion in der Mathematik. Der Modulus-Operator ist zunächst nur für ganzzahlige Datentypen definiert. Der Wert wird durch den folgenden Ausdruck ermittelt:

```
a%b == a - (a/b)*b          a, b ganzzahlig
```

Beispiel: Berechnen der Quersumme einer natürlichen Zahl

```
// quersumme.cpp: Berechnen der Quersumme einer ganzen Zahl

#include <iostream>

int main() {
    int zahl, quersumme = 0;
    cout << "Zahl eingeben: ";
    cin >> zahl;

    while(zahl > 0) {
        quersumme = quersumme + zahl % 10;
        zahl = zahl / 10;
    }
    cout << quersumme << endl;
}
```

Operatoren- und Ergebnistyp

Als Operanden für die vier arithmetischen Operatoren können Ganzzahl- und Gleitpunktwerte gemischt werden. Wenn zwei Ganzzahl-Werte verknüpft werden, ergibt sich wieder ein Ganzzahl-Wert als Ergebnis. Wenn einer oder beide Operanden ein Gleitpunktwert ist, ist auch das Ergebnis ein Gleitpunktwert.

Wie sieht das im folgenden Beispiel aus?

```
11.0/4
```

Fest steht lediglich der Typ des Ergebnisses: `double`, d.h. ein Gleitpunktwert.

Hier findet eine implizite Typumwandlung statt: Nachdem beide Operanden unterschiedliche Typen aufweisen (`double` und `int`), wird der einfachere Operand (der Nenner vom Typ `int`) vorher in das Format des komplizierteren Operanden (der Zähler vom Typ `double`) umgewandelt, anschließend wird die normale Gleitpunktdivision durchgeführt. Das Ergebnis ist also

```
2.75
```

Weitere Einzelheiten zur Typumwandlungen werden später diskutiert.

Assoziativität

Bisher wurden nur Ausdrücke mit jeweils einem Operator und zwei Operanden genannt. In einem Ausdruck kann ein Operator aber auch mehrfach vorkommen, wie z.B. in

```
5+4+3
5*4*3
```

Der Wert dieser Beispiel-Ausdrücke liegt eindeutig fest, weil die Operatoren + und * assoziativ (im mathematischen Sinne) sind. Es spielt spielt also keine Rolle, ob zuerst der vordere oder zuerst der hintere Operator ausgerechnet wird. Das Ergebnis ist jeweils das gleiche.

Anders liegen die Verhältnisse in folgenden Beispielen:

```
5-4-3
5/4/3
```

Die Operatoren - und / sind nicht assoziativ, d.h. je nach Reihenfolge der Operator-Anwendung werden verschiedene
Resultate geliefert:

```
5-(4-3) = 5-1 = 4
(5-4)-3 = 1-3 = -2
5/(4/3) = 5/1 = 5
(5/4)/3 = 1/3 = 0
```

Für die Grundrechenarten (und den Modulus-Operator) gilt die

Regel:

Die Operatoren +, -, *, /, % werden von links nach rechts angewendet.

Diese Richtung wird als "Bindungsrichtung" oder "Assoziativität" bezeichnet.

Die o.g. Beispiel-Ausdrücke haben also folgende Werte:

Ausdruck	***Wert***
5-4-3	-2
5/4/3	0

Priorität

Hier gelten die folgenden Regeln:

- Wenn Operatoren unterschiedlicher Prioritäten nebeneinander stehen, werden sie in der Reihenfolge fallender Priorität angewendet. (Punktrechnung geht vor Strichrechnung, Vorzeichenoperatoren vor den anderen)
- Wenn Operatoren gleicher Priorität nebeneinander stehen, werden sie in Richtung der Assoziativität angewendet. Bei den arithmetischen Operatoren also von links nach rechts
- Die von Operator-Assoziativitäten und -Prioritäten vorgegebene Auswertungsreihenfolge eines zusammengesetzten Ausdrucks lässt sich mit runden Klammern beeinflussen.
    - Teilausdrücke in runden Klammern werden immer zuerst von innen nach außen ausgerechnet.
    - Formal spielen runde Klammern damit die Rolle eines Operators höchster Priorität.

4.2 Mathematische Funktionen

Viele mathematische Funktionen (Quadratwurzel, trigonometrische und logarithmische Funktionen, etc.) werden häufig gebraucht. Sie werden als Bibliotheksfunktionen zur Verfügung gestellt und können jederzeit benutzt werden.

Wichtiger Unterschied zwischen Operatoren und Bibliotheksfunktionen:

- Operatoren sind Teil der Sprache C++ und damit z.B. in der Grammatik ausdrücklich genannt.
- Bibliotheksfunktionen werden bei Bedarf aus einer separaten Bibliothek geholt. Sie sind nicht in der Sprachdefinition festgelegt und dem Compiler auch nicht von vorne herein namentlich bekannt.
- Die im Einzelfall nutzbaren Funktionen hängen von den auf dem System bereitgestellten Bibliotheken ab; Wenn eine Bibliothek fehlt, fehlen auch die entsprechenden Funktionen.
- Mit der Sprache "C" ist ein minimaler Umfang von Bibliotheksfunktionen standardisiert worden ("ANSI-C-Standardbibliothek"). Jedes ANSI-kompatible C-Entwicklungssystem muß die betreffenden Bibliotheksfunktionen in genau der beschriebenen Art zur Verfügung stellen.
- Die ANSI-C-Standardbibliothek ist bei C++ vollständig vorhanden und durch zusätzliche Funktionen erweitert worden.
- Bibliotheken lassen sich auswechseln, ohne dass der Compiler berührt wird. Compiler und Bibliotheken können z.B. auch von verschiedenen Herstellern bezogen werden;
- Man kann eigene Bibliotheken erstellen, die in ihrer Verwendung von den vorgegebenen Bibliotheken nicht zu unterscheiden sind.
- Dem Compiler muss die Absicht, Funktionen aus einer bestimmten Bibliothek nutzen zu wollen, kundgetan werden, d. h. die Funktionen sind zu deklarieren.

Beispiel:

```
double x;
x = sqrt(2);   // Quadratwurzel von 2
```

Syntax eines Funktionsaufrufs

Funktionsname (Parameter1, Parameter2, ..., Parametern)

- Die Anzahl und die Typen der Parameter hängen von der betreffenden Funktion ab. Es gibt auch Funktionen ohne Parameter. In diesem Fall wird nach dem Funktionsnamen ein leeres Klammerpaar angegeben.
- Oft wird anstelle des Begriffes "Parameter" auch der Begriff "Argument" verwendet. Beide Ausdrücke sind synonym.
- Der ganze "Aufruf" einer Bibliotheksfunktion hat den Typ, den die Funktion als Ergebnis zurückliefert. Auch dieser hängt von der betreffenden Funktion ab.
- Formal liefert jede Funktion einen Wert eines bestimmten Typs zurück, d. h. ein Funktionsaufruf hat einen Wert eines bestimmten Typs.
- Es gibt auch Funktionen, die keinen Wert zurückliefern. Diese liefern dann den unbestimmten Typ `void` zurück. Eine solche Funktion entspricht dann einer Prozedur in Pascal.
- Funktionen müssen bevor sie verwendet werden entweder explizit definiert werden oder deklariert werden mit einem sogenannten Funktionsprototyp.

Syntax einer Funktionsdeklaration:

Typname Funktionsname (Typ1 Parameter1, Typ2 Parameter2, ..., Typn Parametern);

Beispiel:

```
double sqrt (double x); // Deklaration der sqrt-Funktion
```

Wenn mathematische Funktionen aus einer der Standardbibliotheken verwendet werden sollen, so hat man nur die zugehörigen Header-Dateien in seinen Quelltext zu inkludieren, um alle Funktions-Deklarationen vorzunehmen. Zusätzlich ist beim Übersetzen eventuell noch die dazu benötigte Bibliothek anzugeben. Die ANSI-C/C++-Bibliotheken werden allerdings normalerweise vom Compiler automatisch zu dem übersetzen Programm dazugebunden.

4.2.1 Einige mathematische Funktionen aus der ANSI-C-Bibliothekl

Die folgenden Funktionen sind in der Headerdatei `<cstdlib>` bzw. `<stdlib.h>` deklariert.

Funktionsprototyp	*Bedeutung*
`int  abs(int x)`	Absolutbetrag `| x |`
`long abs(long x)`	Absolutbetrag `| x |`
`int rand()`	Pseudozufallszahl zwischen 0 und RAND_MAX. RAND_MAX ist die größtmögliche Pseudozufallszahl.
`void srand(unsigned int seed)`	initialisiert den Pseudozufallszahlengenerator

Die meisten mathematischen Funktionen sind in der Headerdatei `<cmath>` bzw. `<math.h>` deklariert.

Funktionsprototyp	*Bedeutung*
`double sin(double x)`	`sin x`
`double cos(double x)`	`cos x`
`double tan(double x)`	`tan x`
`double acos (double x)`	`arccos x`
`double asin (double x)`	`arcsin x`
`double atan (double x)`	`arctan x`
`double atan2 (double x, double y)`	`arctan (x/y)`
`double sinh(double x)`	`sinh x`
`double cosh(double x)`	`cosh x`
`double tanh(double x)`	`tanh x`
`double exp(double x)`	e^x
`double log(double x)`	`ln x`
`double log10(double x)`	$\log_{10}$ x // 10-er Logarithmus
`double pow(double x, int y)`	x^y
`double pow(double x, double y)`	x^y
`double sqrt(double x)`	Quadratwurzel
`double floor(double x)`	Größte ganze Zahl <= x
`double ceil (double x)`	Kleinste ganze Zahl >= x

double fabs (double x)	Absolutbetrag | x |

Beispiel:

```
// zufall1.cpp: Testen des Zufallszahlszahlengenerators

#include <iostream>
#include <cstdlib>

int main() {
    int i = 0;

    cout << "RAND_MAX: " << RAND_MAX << endl;

    while (i < 20) {
       cout << rand() << '\t';
       i = i + 1;
    }
    cout << endl;
    return 0;
}
/*
RAND_MAX: 32767
346     130     10982   1090    11656   7117    17595
6415    22948   31126   9004    14558   3571    22879
18492   1360    5412    26721   22463   25047
*/
```

Beispiel:

```
// funktion1.cpp: Testen von Bibliotheksfunktionen

#include <iostream>
#include <cmath>

int main() {
    double x = 1.0;

    cout << "x sqrt(x)           x^1/3\n";

    while (x <= 20.0) {
        cout << x << '\t' << sqrt(x) << "\t\t"
             << pow(x,1.0/3.0) << '\n';
        x = x + 1.0;
    }
    cout << endl;

    return 0;
}
/*
x       sqrt(x)         x^1/3
1       1               1
2       1.41421         1.25992
3       1.73205         1.44225
4       2               1.5874
5       2.23607         1.70998
6       2.44949         1.81712
7       2.64575         1.91293
8       2.82843         2
9       3               2.08008
10      3.16228         2.15443
....
*/
```

4.2.2 Einschub: Benutzerdefinierte Funktionen

Selbstverständlich ist es auch möglich, Funktionen selbst zu erstellen. Dies sollte man immer dann tun, wenn eine Folge von Anweisungen immer wieder vorkommt. Die genauere Besprechung von Funktionen wird erst später erfolgen.

Syntax:

Typname Funktionsname (Typ1 Parameter1, Typ2 Parameter2, ..., Typn Parametern)
{
Folge von Anweisungen und Ausdrücken
}

Beispiel:

Code	Erläuterung

`// das Quadrat einer Zahl`
`// ausrechnen`
`double sqr (double x)` `{`
`   return x * x;`
`}`
	• `sqr` ist der Funktionsname
	• `x` ist ein formaler Parameter
	• `double` ist der Returnwert
	• `return` beendet die Funktion mit dem angegebenen Wert

Aufruf dieser Funktion:

```
double a = 2.5;
double aQuadrat;
aQuadrat = sqr(a);   // 6.25
```

Was passiert hier?

- der Wert `a` wird an die Funktion `sqr` übergeben
- der formale Parameter `x` wird mit dem Wert von `a` belegt.
- der Ausdruck `return x * x;` ermittelt das Quadrat von `x`, beendet die Funktion und gibt dem Funktionsaufruf seinen Wert.

Beispiel:

```
// Bestimme das Minimum zweier Werte
double min (double a, double b) {
   if (a < b)
      return a;
   else
      return b;
}
```

Aufruf dieser Funktion:

```
double x = 1.5, y = 2.5;
cout << min(x, y) << endl;       // 1.5
```

`void`-Funktionen

Es gibt auch Funktionen, die keinen Wert zurückgeben und nur dazu da sind, irgendeine Aktion durchzuführen. Diese kann man als `void`-Funktionen definieren, d. h. als Funktionen ohne Wert.

Beispiel:

```
void menue() {
    cout << "Bitte wählen Sie eine Operaton aus:\n"
         << "+  -  *  /  %\n";
    return;     // verlassen der void-Funktion, ist aber an dieser
}               // Stelle unnötig
```

Regel:

Funktionen müssen vor Ihrer Verwendung entweder deklariert oder definiert werden.

Beispiel:

```
// gerade.cpp : Test auf gerade bzw. ungerade

#include <iostream>

/* Testen, ob eine uebergebene Zahl n gerade ist
   Returnwert: true  falls n gerade
               false falls n ungerade
*/
bool even(int n) {
    return (n % 2) == 0;
}

/* Testen, ob eine uebergebene Zahl n ungerade ist
   Returnwert: true  falls n ungerade
               false falls n gerade
*/
bool odd(int n) {
    return (n % 2) != 0; // alternativ: return !even(n);
}

/* Frage, ob weitergemacht werden soll,
   als Antwort wird nur j oder n akzeptiert
   Returnwert: 'j' oder 'n'
*/
char weitermachen() {
    char antwort = ' ';
    while (antwort != 'j' && antwort != 'n') {
        cout << "Noch einmal (j/n)? ";
        cin  >> antwort;
    }
    return antwort;
}

int main() {
    bool programmEnde = false;
    int zahl;

    while (!programmEnde) {
        cout << "Ganze Zahl eingeben: ";
        cin  >> zahl;
        if (even(zahl))
            cout << zahl << " ist gerade\n";
        if (odd(zahl))
            cout << zahl << " ist ungerade\n";

        programmEnde = (weitermachen() == 'n');
    }
    return 0;
}
```

4.3 Wertzuweisungen

Mit einer Wertzuweisung (engl.: "*assignment*") wird einer Variablen ein neuer Wert zugewiesen. Ein vorher gültiger Wert wird dabei ersatzlos überschrieben.

Eine Wertzuweisung besteht aus einer linken (das Ziel) und einer rechten Seite (die Quelle):

```
variable = ausdruck;
```

Links muss eine Variable stehen, rechts kann ein beliebiger Ausdruck stehen.

Eine Wertzuweisung läuft in zwei Schritten ab, die nacheinander abgewickelt werden:
- der Wert des Ausdrucks auf der rechten Seite wird ausgerechnet;
- dieser Wert wird an die Variable auf der linken Seite zugewiesen;

Nur deshalb macht eine Wertzuweisung der Form

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

Sinn, die aus mathematischer Sicht falsch ist (eine mathematische Variable kann nicht gleichzeitig zwei verschiedene Werte haben).

4.3.1 L-Wert und R-Wert

Auf der linken Seite einer Zuweisung muss ein sogenannter **L-Wert** (engl. *lvalue*) stehen, d.h. normalerweise eine Variable, allgemein jedoch ein Ausdruck, der einen Ort im Speicher darstellt, der einen zugewiesenen Wert aufnehmen kann. Außer Variablen können hier auch Zeiger- und Referenz-Ausdrücke stehen, doch dazu kommen wir erst später.

Rechts darf ein allgemeiner Ausdruck (im Rahmen der Typverträglichkeit) stehen, der sich zu einem Wert ausrechnen läßt.

Ein Ausdruck, der auf der rechten Seite einer Wertzuweisung stehen darf, wird **R-Wert** (engl *rvalue*) genannt.

Beispiel:

```
int a;   // a ist ein L-Wert
2 + 3;   // ein R-Wert
```

4.3.2 Wertzuweisung als Ausdruck

In C++ gilt das Zuweisungszeichen = als Operator niedriger Priorität. Demzufolge ist eine Wertzuweisung nichts anderes als ein Ausdruck. Der "Wert" einer Wertzuweisung ist dabei der zugewiesene Wert (ein R-Wert). Man kann daher eine Wertzuweisung auch auf die linke Seite einer Zuweisung schreiben:

```
a = b = 2;
```

Zunächst wird die hintere Wertzuweisung ausgewertet (`b` bekommt den Wert 2) und das Ergebnis des Ausdrucks (d. h. die 2 als zugewiesener Wert) dann in der vorderen Wertzuweisung verwendet (`a` bekommt auch Wert 2).

Derartige mehrfachen Zuweisungen werden als Kettenzuweisungen bezeichnet. Der Zuweisungsoperator bindet anders als die arithmetischen Operatoren von rechts nach links, ist also rechts-assoziativ.

4.3.3 Wertzuweisung mit Operatoren

Eine Wertzuweisung wird oft eingesetzt, um den Wert einer einzelnen Variablen gegenüber dem vorhergehenden Wert zu modifizieren, wie z.B. in:

```
a = a + 2;      // Wert von a um 2 hochzaehlen
a = a - 1;      // Wert von a um 1 vermindern
a = a / 2;      // Wert von a halbieren
a = a * 10;     // Wert von a verzehnfachen
a = a % 10;     // Wert von a auf a modulo 10 setzen
```

In C/C++ hat man für diese Ausdrücke eine Abkürzung wie folgt gefunden. Statt

```
variable = variable operator ausdruck
```

schreibt man kürzer

```
variable operator= ausdruck
```

Die oben gezeigten Beispiele lassen sich kürzer schreiben als:

```
a += 2;         // Wert von a um 2 hochzaehlen
a -= 1;         // Wert von a um 1 vermindern
a /= 2;         // Wert von a halbieren
a *= 10;        // Wert von a verzehnfachen
a %= 10;        // Wert von a auf a modulo 10 setzen
```

Diese Operatorzuweisungsoperatoren gibt es für alle arithmetischen Operatoren, sowie auch die noch nicht besprochenen Bit-Operatoren und Bitshift-Operatoren. Sie haben die gleiche niedrige Priorität wie der normale Zuweisungsoperator.

4.3.4 Inkrement- und Dekrementoperatoren

Eine weitere Abkürzung wurde für die oft gebrauchten Sonderfälle

```
variable = variable + 1;
variable = variable - 1;
```

geschaffen:

```
variable++;    // Inkrement-Operator
variable--;    // Dekrement-Operator
```

Der Ausdruck `variable++` erhöht den Wert von `variable` um eins und liefert den hochgezählten Wert als Ergebnis zurück. Entsprechend wird der Wert von `variable` durch den Ausdruck `variable--` um eins vermindert.

Beide Operatoren lassen sich in Präfix- und in Postfix-Schreibweise verwenden, d.h. sie können der Variable voran- oder nachgestellt werden.

	Ausdruck	*Bedeutung*	*Wert des Ausdrucks*
Präfix-Schreibweise	`++a`	`a` um eins erhöhen	`a` nach der Erhöhung
	`--a`	`a` um eins ermindern	`a` nach der Verminderung
Postfix-Schreibweise	`a++`	`a` um eins erhöhen	`a` vor der Erhöhung
	`a--`	`a` um eins ermindern	`a` vor der Verminderung

Beispiel:

```
int a = 1;
int b = a++;   // b == 1, a == 2

int c = 1;
int d = --c;   // d == 0, c == 0
```

4.4 Sonstige Operatoren

4.4.1 Bit-Operatoren

Die Bit-Operatoren ermöglichen es, beliebige Bitmuster zu manipulieren. Angewendet wird dies hauptsächlich in der systemnahen Programmierung.

Operator	***Beispiel***	***Bedeutung***
`<<`	`i << 2`	Linksschieben (Multiplikation mit Zweier-Potenzen)
`>>`	`i >> 2`	Rechtsschieben (Division durch Zweier-Potenzen)
`&`	`i & 7`	bitweises UND
`^`	`i ^ 7`	bitweises XOR (Exklusives ODER)
`|`	`i | 7`	bitweises ODER
`~`	`~i`	bitweise Negation
`<<=`	`i <<= 3`	`i = i << 3`
`>>=`	`i >>= 3`	`i = i >> 3`
`&=`	`i &= 3`	`i = i & 3`
`^=`	`i ^= 3`	`i = i ^ 3`
`|=`	`i |= 3`	`i = i | 3`

Der Wert von `a << b` ist das um `b` Bits nach links verschobene und rechts mit 0-Bits aufgefüllte Bitmuster von `a`. Entsprechend ist `a >> b` der Wert von `a` nach der Verschiebung um `b` Bits nach rechts. Wenn `a` einen `unsigned`-Typ hat oder einen nichtnegativen Wert besitzt, wird dabei von links mit 0-Bits aufgefüllt.

Beispiel:

```
int a = 7;     // Bitmuster: 0000 ... 0111
a = a << 1;    // Bitmuster: 0000 ... 1110
               // Wert: 14

int b = 8;     // Bitmuster: 0000 ... 1000
b = b >> 2;    // Bitmuster: 0000 ... 0010
               // Wert: 2
```

Die Operatoren `&` `|` und `^` verknüpfen jeweils zwei ganzzahlige Ausdrücke bitweise.

Der Operator `&` kann dazu verwendet werden, um eine Reihe von Bits auf 0 zu setzen, während der Operator `|` dazu verwendet werden kann, bestimmte Bits zu setzen.

Beispiel:

```
int a = 5, b = 12, c;
c = a & b;     // c ==  4 (0101 & 1100 = 0100)
c = a | b;     // c == 13 (0101 | 1100 = 1101)
c = a ^ b;     // c ==  9 (0101 ^ 1100 = 1001)
```

Die bitweise Negation ist ein unärer Operator, der bei einem ganzzahligen Ausdruck alle Bits umkehrt.

Beispiel:

```
int a = 5, b;
b = ~a;        // b == 10 (~0101 = 1010)
```

4.4.2 Der Bedingungs-Operator

Der Bedingungsoperator ist der einzige dreistellige Operator bei C/C++.

Syntax:

```
(logischer Ausdruck)  ?  Ausdruck1  :  Ausdruck2
```

Zunächst wird der logische Ausdruck ausgewertet. Der Wert des gesamten Ausdrucks ergibt sich nun zu dem Wert von *Ausdruck1*, falls der logische Ausdruck `true` ist andernfalls wird der Wert des Gesamtausdrucks zum Wert von *Ausdruck2*.

Beispiel:

```
min = ( a < b ) ? a : b;
```

bedeutet das selbe wie

```
if (a < b)
    min = a;
else
    min = b;
```

Unser früheres Beispiel zur Implementierung einer Funktion zur Minimumsberechnung sihet mit dem Bedingungsoperator viel einfacher aus.

```
// Bestimme das Minimum zweier Werte
double min (double a, double b) {
   return (a < b) ? a : b;
}
```

Bemerkung

Der Einsatz des Bedingungsoperators trägt nicht gerade zur guten Lesbarkeit eines Programmes bei. Er sollte eher sporadisch, z.B. zum Ersparen einer `if`-Anweisung, eingesetzt werden.

Beispiel:

```
// runden1.cpp : Runden von double-Werten

#include <iostream>
#include <cmath>

int main() {
    bool programmEnde = false;
    char antwort;
    double zahl;
    double gerundeteZahl1, gerundeteZahl2;

    while (!programmEnde) {
        cout << "Gleitpunktzahl eingeben: ";
```

```
        cin  >> zahl;

        // Zur naechstgroesseren ganzen Zahl runden
        gerundeteZahl1 = floor(zahl + 0.5);
        cout << "Zahl gerundet: " << gerundeteZahl1 << endl;

        // Zur betragsmaessig naechstgroesseren ganzen Zahl runden
        gerundeteZahl2 = (zahl > 0) ? floor(zahl + 0.5)
                                    : ceil (zahl - 0.5);
        cout << "Zahl betragsmaessig gerundet: "
             << gerundeteZahl2 << endl;

        cout << "Noch einmal (j/n)? ";
        cin  >> antwort;
        programmEnde = antwort == 'n';
    }
    return 0;
}
/*
Gleitpunktzahl eingeben: 1.5
Zahl gerundet: 2
Zahl betragsmaessig gerundet: 2
Noch einmal (j/n)? j
Gleitpunktzahl eingeben: -1.5
Zahl gerundet: -1
Zahl betragsmaessig gerundet: -2
Noch einmal (j/n)?
...
*/
```

4.5 Datentypumwandlungen

4.5.1 Implizite Typumwandlung

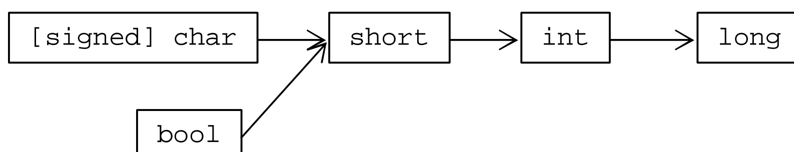
Eine implizite Typumwandlung ist eine Typumwandlung, die vom Compiler ohne zutun des Programmierers durchgeführt wird. Es gibt feste Regeln für die implizite Typumwandlung. Der Compiler wird dabei versuchen, möglichst so zu konvertieren, dass keine Informationen verlorengehen.

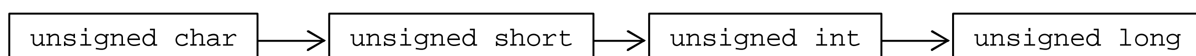
Beispiele:

```
void f(int);
int wert = 3.14159;        // int 3
f(3.14159);                // 3
double x = wert + 3.14159; // wert wird zu double konvertiert

int i = 10;
i *= 2.3;                  // i == 20 oder i == 23 ?

double y = 1/4;            // y == 0.25 oder y == 0 ?
```

4.5.2 Sichere Standardumwandlungen:

Bei diesen Konvertierungen können bei keiner Implementierung Informationen verlorengehen





Die umgekehrten Richtungen werden von den Compilern klaglos akzeptiert. Jedoch ist mit Datenverlust und ggf. sogar mit Laufzeitfehlern zu rechnen.

Implizite Typumwandlungen bei Zeigertypen

Alle Zeigertypen werden implizit in den Typ `void*` konvertiert. Die umgekehrte Richtung ist i. a. falsch.

```
int i = 1;
void* p = &i;        // automatisch konvertiert
int* ip = p;         // ist falsch !
```

4.5.3 Explizite Typumwandlungen

Explizite Typumwandlungen sind Typumwandlungen, die durch den Entwickler explizit vorgenommen werden. Hierfür gibt es bei C++ mehrere Möglichkeiten

C-Casts:

```
int i = 1;
double x =(double) i;
void* p = 0;
char* cptr = (char *) p
```

Funktionale Notation:

```
int i=1;
double x =double(i);
```

Die funktionale Notation ist nur für Typen einsetzbar, die einfache Namen besitzen, also z.B. nicht für `char*`.

Beispiel:

```
typedef char* Pchar;
int*  ip=0;
char* p = Pchar(ip);
```

Bem.:Die funktionale Notation wird für benutzerdefinierte Typumwandlungen benötigt !

Beispiel:

```
complex c = 3.14159; // Datentyp aus der Standardbibliothek
double x = double(c);
```

4.5.4 Sichere Typumwandlungen

1993 wurde eine neue Form für Typumwandlungen eingeführt, die sicherer ist als normale casts.

static_cast

`static_cast` kann nur in Situationen eingesetzt werden, wo auch implizite Umwandlungen möglich sind und bei bestimmten Zeigertypen, allerdings ist es hierbei nicht erlaubt, das Attribut `const` wegzukonvertieren.

```
static_cast<T>(a);  // Wandle den Ausdruck a in den Typ T um

int i = 3, j = 2;
double x = static_cast<double>(i)/j;   // x == 1.5
```

```
void* p = static_cast<void*>(&i);
```

const_cast

`const_cast` kann nur dazu verwendet werden, um das Attribut `const` weg- oder dazuzukonvertieren.

```
const_cast<T>(a);     // Wandle den Ausdruck a in den Typ T um, dabei
                      // darf sich der Typ von a von T nur durch ein
                      // fehlendes oder zusätzliches const unterscheiden

int i = 10;
const int *ip = &i;           // ip Zeiger auf int-Konstante
(*ip)++;                      // falsch !
++*const_cast<int*>(ip);     // korrekt
```

dynamic_cast

`dynamic_cast` ist eine Typumwandlung, die speziell bei Klassenobjekten in Klassenhierarchien eingesetzt wird.

reinterpret_cast

`reinterpret_cast` ist eine Konvertierung unter Umgehung sämtlicher Typprüfungen. Es ist vor allem für sogenannte "schmutzige" Casts vorgesehen. Auch bei `reinterpret_cast` kann `const` nicht wegkonvertiert werden.

```
reinterpret_cast<T>(a); /// Wandle den Ausdruck a in den Typ T um

char ch[] = { 'A', 'B', 'C', 'D' };
float* zf = reinterpret_cast<float*>(ch);
cout << *zf << endl; // 781.035 bei Borland C++
```

Empfehlung: Generell sollten möglichst wenige explizite Typumwandlungen eingesetzt werden, und wenn doch, dann sollte man die neuen verwenden, da sie eine höhere Sicherheit versprechen.

5 Anweisungen und Kontrollstrukturen

5.1 Einleitung

In C++ gibt es eine ganze Reihe an verschiedenen Arten von Anweisungen (engl. *statement*) , die zum Teil so nicht in anderen Programmiersprachen existieren.

- Deklarationsanweisung
- Ausdrucksanweisung
- Zusammengesetze Anweisung
- Auswahlanweisung
- Wiederholungsanweisung
- Sprunganweisung
- Try-Block (wird später erst behandelt)

Deklarationsanweisung

Deklarationen sind Anweisungen und können damit an beliebigen Stellen im Programm stehen. Mit Hilfe von Deklarationen können z. B.

- Variablen und Konstanten definiert und initialisiert werden
- Funktionen deklariert werden
- Funktionen definiert werden
- eigene Datentypen definiert werden (kommt später).

Auch wenn Deklarationen praktisch überall stehen dürfen, so sollte man trotzdem gewisse Reglen einhalten, z.B Variablen und Konstanten nur am Anfang eines Blockes definieren.

Ausdrucksanweisung

Eine Ausdrucksanweisung besteht aus einem gültigen C++-Ausdruck und einem abschließenden Semikolon.

Beispiele:

```
a + b;                      // wenig sinnvoll
sin(0.5) * cos(0.5);

i = 1;                      // sinnvoll
i++;
a = b = c = 1;
b += 3;
;                           // leere Anweisung, kann Sinn machen
```

5.2 Zusammengesetzte Anweisung und Blockstruktur

Syntax:

```
{  Anweisungsfolge  }
```

Eine zusammengesetzte Anweisung fasst mehrere Anweisungen zu einer Einheit, einem sogenannten Block, zusammen. Kennengelernt haben wir Blöcke bereits als Funktionsblock bei der `main`-Funktion oder benutzerdefinierten Funktionen, sowie bei der Zusammenfassung von Anweisungen bei der `if`- und bei der `while`-Anweisung.

Blöcke können geschachtelt werden:

```
{
    int i = 1;
    {
        int j = 2;
        j += i;
    }
}
```

In diesem Beispiel ist `i` sowohl im äußeren als auch im inneren Block zugreifbar, während `j` nur im inneren Block definiert ist.

Geltungsbereich

Namen gelten nicht im gesamten Programm, sondern nur in bestimmten Bereichen: Zu jedem Namen gehört ein Geltungsbereich (engl.: *scope*).

Allgemein gilt: Der Gegenstand einer Definition (bisher: Variable oder Konstante) kann ab der Stelle der Definition bis zum Ende des Blocks benutzt werden, aber nicht vorher im gleichen Block.

Beispiel:

```
{
    int a;          // a gilt ab hier bis zum Block-Ende
    a = 4;          // ok
    b = 2;          // Fehler
    int b;          // b gilt ab hier bis zum Block-Ende
    b = 2*a;        // ok
    int c;          // zulässig, aber nutzlos
}
```

Geltungsbereich eines Namens ist der unmittelbar umgebende Block und alle tiefer geschachtelten Blöcke.

Beispiel:

```
{
    int a;
    a = 1;          // ok
    {
        int b;
        b = 2;  // ok
        a = 3;  // ok
    }
    a = 4;          // ok
    b = 5;          // Fehler -- hier gilt b nicht!
```

```
}
```

Geltungsbereich von `a` sind beide Blöcke, Geltungsbereich von `b` ist nur der innere Block.

Variablen werden beim Verlassen des Blocks, in dem sie definiert sind, wieder zerstört, d. h. der für sie allokierte Speicherplatz wird wieder freigegeben. Der Wert geht verloren und kann nicht mehr restauriert werden. Wird der Block wieder betreten, so wird den Variablen neuer Speicherplatz zugewiesen. Man bezeichnet solche Variablen auch als **lokale Variablen** innerhalb des Blocks.

Redefinition von Namen

Namen können in untergeordneten Blöcken neu definiert werden. Dadurch wird die im äußeren Block gültige Definition überdeckt.

```
{  // aeusserer Block
    int a = 1;              // erstes a, gilt im äußeren Block
    {    // innerer Block
        int a = 2;          // zweites a, gilt im inneren Block
                            // das erste a ist hier nicht erreichbar,
                            // aber es existiert!
        cout << a;          // Wert des zweiten a (2) wird ausgegeben
    }
    cout << a;              // Wert des ersten a (1) wird ausgegeben
}
```

Globale Namen

Ein Name, der außerhalb der `main`-Funktion und aller anderer Funktionen definiert ist, hat als Geltungsbereich das gesamte Programm.

```
int i = 1;

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

Die Variable `i` in diesem Beispiel ist eine globale Variable, d. h. sie ist in allen Funktionen und in allen Blöcken gültig. Sie wird erst mit dem Programmende aus dem Speicher wieder entfernt.

Regel:

Globale Variablen und Konstanten sind generell zu vermeiden!

5.3 Auswahlanweisungen

Mit einer Auswahlanweisung kann ein möglicher Kontrollfluss durch das Programm gewählt werden. C++ kennt zwei Arten von Auswahlanweisungen, die `if`-Anweisung und die `switch`-Anweisung.

5.3.1 Die if-Anweisung

Wir haben die `if`-Anweisung bereits im Abschnitt 2.10 kennengelernt. Es gibt aber noch einiges zusätzlich dazu zu sagen.

Syntax:

```
if (Bedingung) {
    Ja-Anweisungen
}

if (Bedingung) {
    Ja-Anweisungen
}
else {
    Nein-Anweisungen
}
```

Generell ist als Bedingung jeder Ausdruck erlaubt, der einen Wert gleich `0` oder ungleich `0` annehmen kann. Der Wert `0` wird dann automatisch zu `false` konvertiert, jeder Wert ungleich `0` wird zu `true` konvertiert.

Regel:

Es sind nur reine boolsche Ausdrücke zu verwenden. Gegebenenfalls kann mit Hilfe eines Vergleiches auf `== 0` oder `!= 0` jeder Ausdruck zu einem boolschen Ausdruck gemacht werden.

Beliebte Fehlersituationen:

1. Was ist hier falsch ?

```
if (a > 0)
    if (b > 0)
        c = a
else
    c = b;
```

2. Und hier ?

```
if (a > 0);
    cout << a << endl;
```

3. Sieht doch richtig aus, oder?

```
if (a = b)
    cout << "a ist gleich b" << endl;
else
    cout << " a ungleich b"  << endl;
```

Mehrfachverzweigungen mit `if`

Es kommt häufig vor, dass mehrere Alternativen zu unterscheiden sind. Dies kann zu beliebig tief geschachtelten `if`-Anweisungen führen. Wenn man nun jedesmal um 4 Leerzeichen einrückt, dann kann das Programm sehr schnell unübersichtlich werden. Eine Alternative zeigt das folgende

Beispiel:

```
// notenrechner.cpp : Aus Punkten zwischen 0 und 20 Noten berechnen

#include <iostream>

int main() {
```

```
    int punkte;
    double note;

    cout << "Punkte eingeben : ";
    cin >> punkte;

    if (punkte >= 0 && punkte < 8)
        note = 5.0;
    else if (punkte >= 8 && punkte <= 10)
        note = 4.0;
    else if (punkte == 11)
        note = 3.7;
    else if (punkte == 12)
        note = 3.3;
    else if (punkte == 13)
        note = 3.0;
    else if (punkte == 14)
        note = 2.7;
    else if (punkte == 15)
        note = 2.3;
    else if (punkte == 16)
        note = 2.0;
    else if (punkte == 17)
        note = 1.7;
    else if (punkte == 18)
        note = 1.3;
    else if (punkte == 19 || punkte == 20)
        note = 1.0;
    else
        note = 0.0;

    if (note > 0.0)
        cout << "Note: " << note << endl;
    else
        cout << "Ungültige Punktzahl eingegeben!\n";

    return 0;
}
```

5.3.2 Die switch-Anweisung

Syntax:

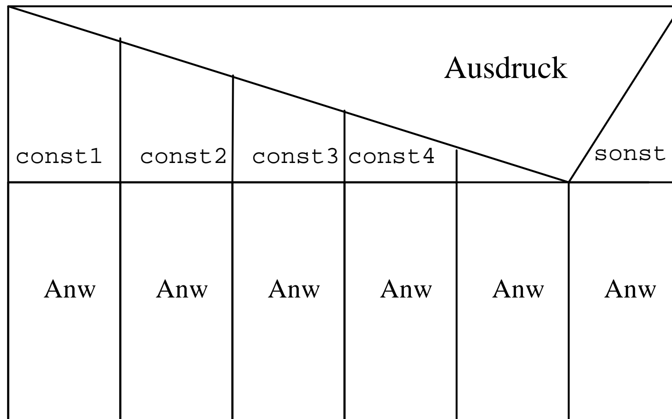
```
switch (Ausdruck) {
    case const1 : Anweisungen break;
    case const2 : Anweisungen break;
    case const3 : Anweisungen break;
    ...
    default:    : ErsatzAnweisungen;
}
```

Bedeutung:

- Der Ausdruck wird ausgewertet und muss ein Ergebnis vom Typ `int` haben oder leicht in `int` konvertierbar sein.
- Dieses Ergebnis wird mit den `case`-Konstanten *`const1`*, *`const2`*, .. verglichen, die zum Einsprung an die richtige Stelle dienen.
- Bei Übereinstimmung werden die zur passenden Konstante gehörigen Anweisungen ausgeführt.
- Die Angabe von `break` dient dazu, die `switch`-Anweisung zu verlassen. Fehlt die Angabe von `break`, so wird mit der nächsten Anweisungsfolge weitergemacht.

- Die `case`-Konstanten *`const1`*, *`const2`*, ... müssen eindeutig und auf `int` abbildbar sein. Üblich sind hier `int`- oder `char`-Literale oder Konstanten.
- Die nach `default` stehenden Anweisungen werden immer dann ausgeführt, wenn der `switch`-Ausdruck einen Wert liefert, der mit keiner der `case`-Konstanten übereinstimmt.

Struktogrammdarstellung:



Beispiel: Noten ausrechnen

```
// notenrechner2.cpp : Aus Punkten zwischen 0 und 20 Noten berechnen
//                     switch-Version

#include <iostream>

int main() {
    int punkte;
    double note;
    cout << "Punkte eingeben : ";
    cin >> punkte;

    if (punkte >= 0 && punkte < 8)
        note = 5.0;
    else if (punkte >= 8 && punkte <= 10)
        note = 4.0;
    else {
        switch (punkte) {
            case 11: note = 3.7; break;
            case 12: note = 3.3; break;
            case 13: note = 3.0; break;
            case 14: note = 2.7; break;
            case 15: note = 2.3; break;
            case 16: note = 2.0; break;
            case 17: note = 1.7; break;
            case 18: note = 1.3; break;
            case 19:
            case 20: note = 1.0; break;
            default: note = 0.0;
        }
    }

    if (note > 0.0)
        cout << "Note: " << note << endl;
    else
        cout << "Ungültige Punktzahl eingegeben!\n";
}
```

Wie man in dem Beispiel sieht, kann man `break` auch absichtlich weglassen.

5.4 Wiederholungsanweisungen

Die mehrfache Ausführung einer Anweisungsfolge wird durch Wiederholungsanweisungen ermöglicht. Wiederholungsanweisungen werden auch als Schleifen bezeichnet. Es gibt drei verschiedene Schleifenarten in C++, die `for`-Anweisung, die `while`-Anweisung und die `do`-Anweisung.

5.4.1 Die for-Anweisung

Syntax

```
for (Initialisierung; Schleifenbedingung; Iterationsanweisung ) {
    Wiederholungsanweisungen
}
```

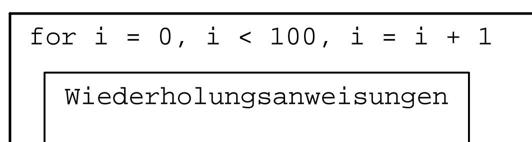
Bedeutung

- Zunächst wird die Initialisierung durchgeführt. Es wird z.B. für eine sogenannte Laufvariable ein Startwert festgelegt. Die Initialisierung wird nur genau einmal durchgeführt.
- Danach wird die Schleifenbedingung geprüft. Sie wird vor jedem Schleifeneintritt ausgewertet und legt fest, ob die Wiederholungsanweisungen ausgeführt werden (`true`) oder nicht (`false`).
- Wenn die Schleifenbedingung den Wert `true` hat, werden zuerst die Wiederholungsanweisungen durchgeführt und dann schließlich die Iterationsanweisung.
- Danach wird wieder zum Prüfen der Schleifenbedingung gesprungen.

Beispiel:

```
// die ersten 100 Zahlen ausgeben
for (i = 0; i < 100; i++) {
    cout << i << endl;
}
```

Struktogrammdarstellung



Die `for`-Anweisung ist äquivalent zur folgenden `while`-Schleife

```
Initialisierung
while ( Schleifenbedingung ) {
    Wiederholungsanweisungen
    Iterationsanweisung
}
```

Das Beispiel sieht dann so aus:

```
// die ersten 100 Zahlen ausgeben
i = 0;
while ( i < 100 ) {
    cout << i << endl;
    i++;
}
```

Die bisher betrachtete Anwendung ist gleichzeit die, die am häufigsten vorkommt. Sie entspricht im wesentlichen auch der `for`-Anweisung, wie man sie von PASCAL her kennt.

Alle drei Ausdrücke im Kopf der Schleife sind optional, d.h. sie können weggelassen werden, nicht aber die trennenden Semicolons.

Voreinstellungen sind:
- Initialisierung: nichts
- Schleifenbedingung: true
- Initialisierungsanweisung: nichts

Daher ist die folgende Endlosschleife syntaktisch korrekt:

```
for( ; ;) ;
```

Lokale Laufvariable

Folgende Konstruktion ist möglich:

```
// rückwärts laufen
for (int i = 100; i > 0; i--) {
    cout << i << endl;
}
// ab hier ist i nicht mehr bekannt!
```

Die Laufvariable `i` ist nur lokal innerhalb der for-Schleife definiert.

Beispiel: Summe der ersten n Zahlen

```
// 1. Version
int n = 100;
int i, summe = 0;

for (i = 1; i <= n; i++)
    summe += i;


// 2. Version
int n = 100;
int summe = 0;

for (int i = 1; i <= n; i++)
    summe += i;


// 3. Version
int n = 100;
int summe;

for (int i = 1, summe = 0; i <= n; summe += i, i++);
```

In dieser 3. Version werden mehrere Initialisierungsanweisungen durch Komma getrennt (der sogenannte Komma-Operator) angegeben und es werden auch mehrere Iterationsanweisungen durch Komma getrennt angegeben. Daher ist im eigentlichen Schleifenrumpf nichts mehr zu tun.

Empfehlung: Vermeiden Sie solche Konstruktionen!

Beispiel: double-Variable hochzählen

```
// double-Variable hochzählen
for (double x = 0.0; x < 1.0; x += 0.1)
    cout << x << '\t' << sin(x);
```

`for`-Schleifen kann man beliebig tief schachteln wie das folgende Beispiel zeigt.

Beispiel: Geschachtelte `for`-Schleifen

```
// einmaleins.cpp : das grosse EinmalEins ausgeben

#include <iostream>
#include <iomanip>

int main() {
    int multiplikand, multiplikator;
    for (multiplikand = 1; multiplikand <= 20; multiplikand++)
    {
        for (multiplikator = 1; multiplikator <= 10; multiplikator++ )
        {
            cout << setw(4) << multiplikand * multiplikator;
        }
        cout << endl;
    }
    return 0;
}
/*
  10  20  30  40  50  60  70  80  90 100
  11  22  33  44  55  66  77  88  99 110
  12  24  36  48  60  72  84  96 108 120
  13  26  39  52  65  78  91 104 117 130
  14  28  42  56  70  84  98 112 126 140
  15  30  45  60  75  90 105 120 135 150
  16  32  48  64  80  96 112 128 144 160
  17  34  51  68  85 102 119 136 153 170
  18  36  54  72  90 108 126 144 162 180
  19  38  57  76  95 114 133 152 171 190
  20  40  60  80 100 120 140 160 180 200
*/
```

Beispiel

```
// dreieck.cpp : ein Zahlendreieck ausgeben

#include <iostream>
#include <iomanip>

int main() {
    int i, j;
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j <= i ; j++ )
        {
            cout << setw(3) << j;
        }
        cout << endl;
    }
    return 0;
}
/*
  0
  0  1
```

```
  0  1  2
  0  1  2  3
  0  1  2  3  4
  0  1  2  3  4  5
  0  1  2  3  4  5  6
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7  8
  0  1  2  3  4  5  6  7  8  9     */
```

Frage: Wie muss man diese Schleife ändern, damit das Dreieck auf der Spitze steht?

Hinweise:

- Die Schleifenbedingung in der `for`-Anweisung kann durchaus auch eine komplexe zusammengesetzte Bedingung sein und muss sich nicht nur auf die Schleifenvariable beziehen.
- Im Schleifenrumpf sollte man es tunlichst vermeiden, die Schleifenvariable selbst zu verändern.

```
for (i = 0; i < 100; i++) {
    ....
    i--;    // versteckte Unendlichschleife
}
```

- Typische Anwendungen der `for`-Anweisung sind beim Suchen in Feldern und beim Sortieren in Feldern zu sehen.

5.4.2 Die while-Anweisung

Die `while`-Anweisung hatten wir bereits im Abschnitt 2.10.2 kennengelernt. Hier noch einmal eine Übersicht.

Syntax	*Struktogramm-Darstellung*

`while (Bedingung) {`
`    Wiederholungsanweisungen`
`}`
	Bedingung
	Wiederholungsanweisungen

Bedeutung:

1. Vor jedem Schleifendurchlauf wird die Schleifenbedingung ausgewertet.
2. Ist sie wahr, so wird der Schleifenrumpf, also die Wiederholungsanweisungen einmal ausgeführt, ansonsten wird der Schleifenrumpf übersprungen und mit der Anweisung hinter der `while`-Anweisung fortgefahren.
3. Gehe zurück zu Punkt 1.

Beispiel: Größter gemeinsamer Teiler zweier ganzer Zahlen n und m

Erster naiver Ansatz: Probiere alle Zahlen bis zum Minimum von n und m als Teiler aus.

```
// ggT nach einem intuitiven Algorithmus bestimmen
long ggT_intuitiv(long n, long m) {
    n = abs(n);
    m = abs(m);
    long teiler = 1;
```

```
    long groessterTeiler = 1;
    while (teiler <= m && teiler <= n) {
        if ( (m % teiler == 0 ) && (n % teiler == 0))
            groessterTeiler = teiler;
        teiler++;
    }
    return groessterTeiler;
}
```

Eine mit diesem Algorithmus durchgeführte Laufzeitmessung ergab folgende Werte:

n	*m*	*ggT(n,m)*	*Laufzeit*
123456	234567	3	0.03 s
1234567	2345678	1	0.291 s
12345678	23456789	1	2.904 s
123456789	234567891	9	33.048 s
1234567891	2147483647	1	301.864 s

(gemessen mit Borland C++ 5.0 auf Pentium 233)

Die durchschnittliche Laufzeit ergibt sich zu etwa `2.44 E-7 * min(n,m)`, wächst also linear . Für noch größere Zahlen, z. B. bei einer beliebig genauen Arithmetik ist das ein absolut inakzeptabler Wert.

Der übliche Algorithmus zum Ermitteln des ggT ist ursprünglich von Euklid (etwa 300 v. Chr.) entwickelt worden und nutzt die folgende Eigenschaft des ggT aus.

Es gilt:

```
ggT(n, m) = ggT(n - k*m, m)      für k ganze Zahl
```

Beispiele:

```
ggT(18, 12) = ggT(6, 12) = ggT(6, 0) = 6
ggT(13, 17) = ggT(13, 4) = ggT(1, 4) = ggT(1, 0) = 1
```

Algorithmus:

```
n und m positiv eingeben
r = m mod n
wiederhole, solange r > 0
    m = n
    n = r
    r = m mod n
Der ggT steht in n
```

Implementierung:

```
// ggT nach dem euklidischen Algorithmus
long ggT (long m, long n) {
   m = abs(m);
   n = abs(n);
   long r = m % n;
   while (r > 0){
```

```
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
```

Die Messung der Laufzeit erfolgte dabei mit der folgenden Funktion:

```
#include <sys/timeb.h>

// aktuelle Uhrzeit in Sekunden und Millisekunden liefern
double getAktuelleZeit() {
    timeb t;
    ftime (&t);
    double uhrzeit = t.time + t.millitm / 1000.0;
    return uhrzeit;
}
```

In `t.time` steht dabei die aktuelle Uhrzeit in Sekunden seit dem 01.01.1970, 0.00 Uhr, in `t.millitm` stehen die Millisekunden.

Anwendung dieser Funktion:

```
     double z = getAktuelleZeit();
     cout << "ggT(1234567891,2147483647):\n"
          << "Wert: " << ggT_intuitiv(1234567891,2147483647);
     cout << "\tZeit: " << getAktuelleZeit() - z << endl;
```

Eine mit dem Euklidischen Algorithmus durchgeführte Laufzeitmessung ergab folgende Werte:

n	*m*	*ggT(n,m)*	*Laufzeit für 10000 Aufrufe*
123456	234567	3	0.19 s
1234567	2345678	1	0.20 s
12345678	23456789	1	0.19 s
123456789	234567891	9	0.211 s
1234567891	2147483647	1	0.513 s

(gemessen mit Borland C++ 5.0 auf Pentium 233)

Die mittlere Laufzeit des Euklidischen Algorithmus liegt bei `c * log min(n,m)` mit einer Konstanten `c > 0`.

5.4.3 Die do-while-Anweisung

Während die `while`-Anweisung eine kopfgesteuerte Schleife ist, ist die `do`-Schleife eine sogenannte fußgesteuerte Schleife.

Syntax	*Struktogramm-Darstellung*

`do {`
`    Wiederholungsanweisungen`
`} while (Bedingung);`
	Wiederholungsanweisungen
	Bedingung

Bedeutung:

1. Zunächst wird der Schleifenrumpf, also die Wiederholungsanweisungen durchgeführt unabhängig davon, ob die Schleifenbedingung erfüllt ist oder nicht.
2. Nach jedem Schleifendurchlauf wird die Schleifenbedingung ausgewertet.
3. Ist sie wahr, so wird der Schleifenrumpf noch einmal ausgeführt, ansonsten wird mit der nächsten Anweisung hinter der `do`-Schleife fortgefahren.

Bei der `do`-Schleife wird also der Schleifenrumpf also immer mindestens einmal durchgeführt. Dies muss bei der `for`- bzw. der `while`-Schleife nicht der Fall sein.

Beispiel:

```
....

char weitermachen() {
    char antwort;
    do {
        cout << "Noch einmal (j/n)? ";
        cin  >> antwort;
    } while (antwort != 'j' && antwort != 'n');
    return antwort;
}

int main() {
    ...
    do {
    ...
    } while (weitermachen() == 'j');
    return 0;
}
```

Äquivalenz zwischen beiden Schleifenformen

Beide Schleifensorten sind gleichwertig: Jede könnte durch die andere ersetzt werden.

Ersetzung `while`- durch `do`-Schleife:

`while`-Schleife	äquivalente `do`-Schleife

`while(Schleifenbedingung) {`
`    Wiederholungsanweisungen`
`}`
	`if(Schleifenbedingung)`
	`    do {`
	`        Wiederholungsanweisungen`
	`    } while(Schleifenbedingung);`

Ersetzung `do`- durch `while`-Schleife:

`do`-Schleife	äquivalente `while`-Schleife

`do {`
`    Wiederholungsanweisungen`
`} while(Schleifenbedingung);`
	`Wiederholungsanweisungen;`
	`while(Schleifenbedingung) {`
	`    Wiederholungsanweisungen`
	`}`

5.5 Sprünge

Motivation:

Manchmal ist es notwendig, eine Schleife aufgrund besonderer Umstände, z.B. aufgrund eines Fehlers vorzeitig abzubrechen. Manchmal ist es auch notwendig, einen Schleifenrumpf vorzeitig zu beenden, um mit dem nächsten Schleifendurchlauf weitermachen zu können. Hier bieten viele Programmiersprachen das Sprachmittel `goto` an. Auch C++ kennt ein `goto`.

Prinzipielle Anwendung:

```
Marke: ...
       ...
       ...
       goto Marke;   // Sprung zur Anweisung hinter Marke
```

Gefährlich ist `goto` insbesondere dann, wenn man mehrere `goto`s einsetzt, die sich u. U. auch noch überlappen. Dann ist ein Programm schnell absolut unverständlich. Deshalb gilt in allen Programierrichtlinien zu C++, und selbstverständlich auch bei uns, auch die

Regel: `goto` ist verboten !!

Für die oben genannten Probleme gibt es mit der `break`- und der `continue`-Anweisung spezielle Lösungen.

5.5.1 Die break-Anweisung

Typische Anwendung:

```
while (Schleifenbedingung) {
    ...
    ...
    if (Fehlerbedingung)
        break; // Sprung zur ersten Anweisung hinter der while-Schleife
    ...    !
}          !
<---------+
```

Die `break`-Anweisung lässt sich auf die `for`-, `while`-, und `do`-Schleife, sowie die `switch`-Anweisung anwenden, nicht jedoch auf die `if`-Anweisung oder auf Blöcke.

Geschachtelte Schleifen:

```
for (...; ...;...) {
    while (Schleifenbedingung) {
        ...
        ...
        if (Fehlerbedingung)
            break;   // Sprung zur ersten Anweisung hinter der while-
              !     // Schleife
    ...       !
    }         !
    ... <-----+
}
```

Bei einem `break` innerhalb einer inneren Schleife wird nur die innere Schleife verlassen.

Beispiel: 100 Messwerte summieren; Negativer Wert zeigt vorzeitiges Ende der Messreihe an.

Lösung ohne `break`:

```
double summe = 0, messwert;

cin >> messwert;
for(n = 0;  (n < 100) && (messwert >= 0);  n++)
{
    cin >> messwert;
    if(messwert >= 0)
         summe += messwert;
}
```

Nachteil dieser Lösung:

- Der Schleifenkopf wird mit Prüfungen für Sonderfälle überladen
- Der erste Messwert muss vorab eingelesen werden, um die Bedingung überhaupt prüfen zu können
- Die Fehlerbedingung muss zweimal getestet werden (für Schleifenabbruch und vor dem Summieren);
- Die eigentliche Schleifenbedingung (der Normalfall) geht unter;

Lösung mit `break`:

```
double summe = 0, messwert;

for(n = 0;  n < 100;  n++)
{
    cin >> messwert;
    if(messwert < 0)
        break;  // Absprung hinter die Schleife
    summe += messwert;
}
```

5.5.2 Die continue-Anweisung

Typische Anwendung:

```
while (Schleifenbedingung) {
    ...
    ...
    if (Bedingung)
        continue;    // Sprung ans Ende des Schleifenrumpfes
    ...      !
} <---------+
```

Die `continue`-Anweisung lässt sich auf die `for`-, `while`-, und `do`-Schleife anwenden.

Geschachtelte Schleifen:

```
for (...; ...;...) {
    while (Schleifenbedingung) {
        ...
        ...
        if (Bedingung)
```

```
            continue; // Sprung ans Ende der while-Schleife
    ...         !
    } <---------+
    ...
}
```

Bei einem `continue` innerhalb einer inneren Schleife wird ans Ende der inneren Schleife gesprungen.

5.6 Beispiele für Iterationen

Iteration

- mehrmaliges Durchlaufen der selben Anweisungskette.
- oft in Verbindung mit der Tatsache, dass Werte der i-ten Ausführung von Werten der (i-1)-ten (und weiteren vorangegangenen) Ausführungen abhängen.

Im folgenden werden wir erste typische Beispiele für iterative Algorithmen zur Berechnung einer Funktion kennenlernen.

5.6.1 Die Exponentialfunktion

Die Exponentialfunktion ist wie folgt definiert:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots$$

Erste naive Vorgehensweise: Berechne jeden Summanden einzeln und dort zunächst den Zähler und dann den Nenner. Summiere auf bis der einzelne Summand eine vorgegebene Schranke unterschreitet.

Implementierung:

```
// exp1.cpp : Berechnung der Exponentialfunktion
//            naiver Ansatz

#include <iostream>
#include <cmath>

double potenz(double x, int n) {
    double produkt = 1;
    for (int i = 0; i < n; i++)
        produkt *= x;
    return produkt;
}

double fakultaet(int n) {
    double wert = 1.0;
    for (int i = 1; i <= n; i++)
        wert *= i;
    return wert;
}

int main() {
    double epsilon = 1e-6;
    double summand = 0.0;
    double wert = 1.0;
    double x;
```

```
    cout << "x = "; cin >> x;
    int i = 0;

    do {
        i++;
        summand = potenz(x, i)/ fakultaet(i);
        wert += summand;
    } while (summand > epsilon);

    cout << "Abbruch nach " << i << " Iterationen!\n";
    cout << "wert = " << wert
         << "   exp(x) = " << exp(x) << endl;
    return 0;
}
```

Wieso ist diese Lösung schlecht?

- Die Fakultätsfunktion und die Potenzfunktion wachsen sehr schnell, so dass ein Überlauf der Arithmetik denkbar ist.

  ```
  13! = 6.22702e+09
  20! = 2.4329e+18
  100! = 9.33262e+157
  170! = 7.25742e+306
  ```

  Bereits `13!` überschreitet die Grenzen von `long` und `171!` überschreitet die Grenzen von `double`.

- Es ist sehr umständlich, für jeden Summanden den Zähler und den Nenner jeweils neu auszurechnen, den es ist sehr einfach aus dem n-ten Summanden den (n-1)-ten Summanden zu ermitteln.

n-ter Summand: $\frac{x^n}{n!}$

(n+1)-ter Summand: $\frac{x^{n+1}}{(n+1)!} = \frac{x^n}{n!} * \frac{x}{n+1}$

Wenn wir diese Eigenschaft ausnutzen, kommen wir zu der folgenden Lösung:

```
// exp2.cpp : Berechnung der Exponentialfunktion
//            zweiter Ansatz

#include <iostream>
#include <cmath>

int main() {
    double epsilon = 1e-6;
    double summand = 1.0;
    double wert = 1.0;
    double x;
    cout << "x = "; cin >> x;
    int i = 0;

    do {
        i++;
        summand = summand * x / i;
        wert += summand;
    } while (summand > epsilon);

    cout << "Abbruch nach " << i << " Iterationen!\n";
    cout << "wert = " << wert
```

```
        << "   exp(x) = " << exp(x) << endl;
    return 0;
}
```

Unterschiede zur ersten Lösung:

- Keine Überläufe bei der Berechnung des Summanden
- Wesentlich weniger Multiplikationen insgesamt
  1. Lösung: Je Schleifendurchlauf 2*i Multiplikationen
  2. Lösung: Je Schleifendurchlauf 1 Multiplikation !

5.6.2 Der Potenzierungsalgorithmus

Aufgabe:

Berechne x^n mit x `double`-Wert und n > 0 `int`-Wert.

Wir haben bei unserer ersten Implementierung der Exponentialfunktion bereits eine Implementierung des Potenzierungsalgorithmus gesehen:

```
double potenz(double x, int n) {
    double produkt = 1;
    for (int i = 0; i < n; i++)
        produkt *= x;
    return produkt;
}
```

Bei diesem Verfahren werden genau n Multiplikationen benötigt. Es geht aber mit weniger Multiplikationen!

Betrachte dazu die Binärdarstellung von n:

$$n = \sum_{i=1}^{k} \alpha_i 2^i = \alpha_0 2^0 + \alpha_1 2^1 + \ldots + \alpha_k 2^k \quad mit\, \alpha_i = 0 \; oder 1$$

Nun gilt:

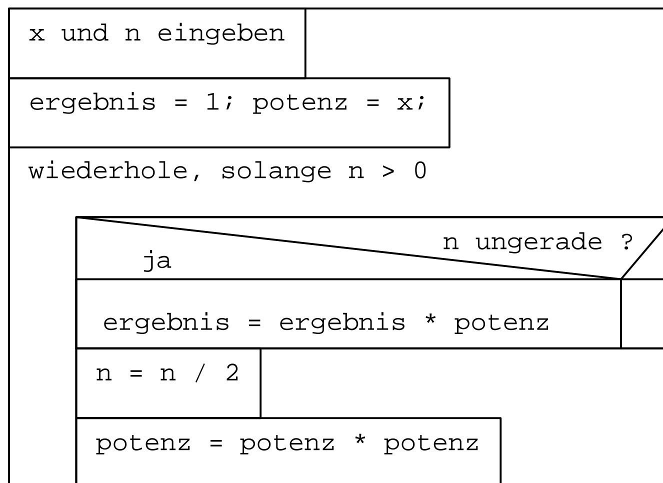
$$x^n = x^{\sum_{i=1}^{k} \alpha_i 2^i} = \prod_{i=1}^{k} x^{\alpha_i 2^i} = x^{\alpha_0 2^0} * x^{\alpha_1 2^1} * \ldots * x^{\alpha_k 2^k}$$

Beispiel:

$$x^{23} = x^{2^0} * x^{2^1} * x^{2^2} * x^{2^4}$$

Die naheliegende Idee ist nun, durch sukzessives Quadrieren die Zweierpotenz-Potenzen von x auszurechnen und ggf. miteinander zu multiplizieren.

Algorithmus:



Durchführung für n = 23

Initialisierung:

```
n = 23; ergebnis = 1; potenz = x
```

Durchlauf	**n**	**n ungerade ?**	**ergebnis**	**potenz**
1	23	ja	x	x^2
2	11	ja	x^3	x^4
3	5	ja	x^7	x^8
4	2	nein	x^7	x^{16}
5	1	ja	x^{23}	x^{32}
6	0	---	---	---

Laufzeit dieses Algorithmus: $c * \log_2 x$

Implementierung:

```
double binPotenz(double x, int n) {
    double ergebnis = 1.0;
    double potenz = x;
    while (n > 0) {
        if ( n % 2 != 0 )        // n ungerade ?
            ergebnis = ergebnis * potenz;
        n = n / 2;
        potenz = potenz * potenz;
    }
    return ergebnis;
}
```

Bemerkung:

Dieser "moderne" Algorithmus wurde bereits 1414 von dem arabischen Mathematiker al-Kashî veröffentlicht und lässt sich auf einen etwa 3000 Jahre alten ägyptischen Algorithmus zur Multiplikation von Zahlen zurückführen (das sogenannte "Fellachen"-Produkt).

6 Ein-/Ausgabe

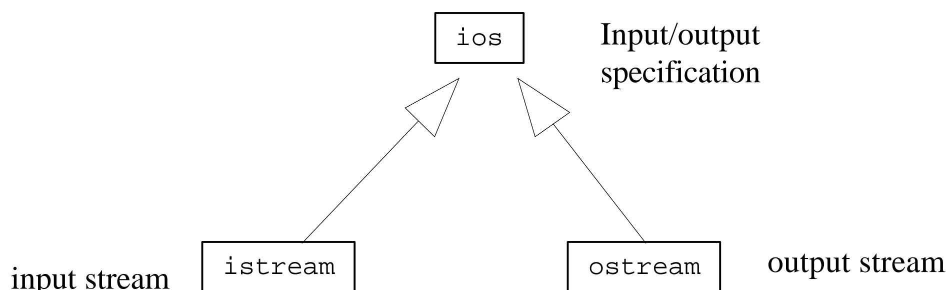
In diesem Abschnitt werden wir zunächst einige weitere Eigenschaften der iostream-Bibliothek von C++ kennenlernen.

6.1 Die iostream-Bibliothek

Die iostream-Bibliothek ist komplett objektorientiert aufgebaut und bietet Datentypen in Form von Klassen für die folgenden Aufgaben:

- Ein-/Ausgabe auf die Standardeingabe bzw- ausgabe
- Dateiverarbeitung
- Einlesen aus und Ausgeben in Zeichenketten
- Formatierte Ein-/Ausgabe
- Fehlerbehandlung

Die wichtigsten Klassen der iostream-Bibliothek sind die folgenden:



Die Klasse `ios` enthält Datenkomponenten, die bei jeder Ein- oder Ausgabe benötigt werden:

- Statusbits für Fehlersituationen
- Formatbits zur Steuerung der Darstellungsart
- Zahlenbasis, links- oder rechtsbündig
- Exponential- und Fixpunktdarstellung (z.B.: 1.0e-2 bzw. 0.01)
- Konstanten für Status- und Formatbits
- Formatparameter

Die Klasse `ostream` ist die Basisklasse für alle Ausgabeklassen, die Klasse `istream` die Basisklasse für alle Eingabeklassen, d. h. alles was in diesen beiden Klassen definiert ist, ist auch genauso bei der Ein- bzw. Ausgabe bei Dateien und Zeichenketten anwendbar.

6.2 Vordefinierte iostream-Objekte

Die folgenden iostream-Objekte sind bereits definiert, wenn ein Programm gestartet wird:

`cin`	Standardeingabe (`istream`)
`cout`	Standardausgabe (`ostream`)
`cerr`	Standardfehlerausgabe (`ostream`, ungepuffert)
`clog`	Standardfehlerausgabe (`ostream`, gepuffert), (wird zur Fehlerausgabe in Dateien benutzt)

- Die Ein-/Ausgabe kann über die Operatoren >> bzw. << erfolgen oder über die sogenannten Elementfunktionen, von denen wir vorerst nur ein paar kennenlernen werden.
- Für alle Standarddatentypen sind die Operatoren << und >> vordefiniert.
- Die bisher besprochenen Funktionalitäten wandeln bei der Eingabe reine Texteingaben in die interne Darstellung der Datentypen um, z. B. bei `int`, und umgekehrt bei der Ausgabe wiederum die interne Darstellung in eine Textdarstellung. Naturgemäß ist dies relativ langsam.

6.3 Ein-Ausgabe der elementaren Datentypen

Ausgabe mit `ostream`:

```
cout << "hallo";     // gibt die Zeichenkette "hallo" auf die
                     // Standardausgabe aus

cout << "Wert:" << j;      //gebe "Wert:" aus und dann j
```

Einlesen mit `istream`:

```
cin >> i;
cin >> i >> j;       // (cin >> i) >> j
                     // lese zuerst i und dann j ein
```

Generell werden elementare Datentypen in einem Standardformat aufbereitet und ausgegeben.

`bool`	als `int`-Wert, d.h. `0` für `false` und ein anderer Wert als `0` für `true`
`char`	abhängig vom Zeichencode: • druckbare Zeichen unverändert • Zeichen mit Codes von 128 aufwärts: zeichensatz-abhängig • Kontrollzeichen (Codes 0-31): systemabhängig
ganzzahlige Werte	Dezimaldarstellung
Gleitkommawerte	Nachkommastellen und Exponent falls notwendig

Generell werden zunächst führende Leerzeichen (Blanks, Tabulatoren, Carriage Return und Linefeed) überlesen. Ganzzahlige Typen werden dabei als Dezimalzahlen, Oktalzahlen (führende `0`) oder Hexadezimalzahlen (führendes `0x`) eingelesen, Gleitpunktzahlen in den üblichen Formaten.

Vorsicht bei der Priorität der << und >> Operatoren!

```
cout << a + b;       // richtig
cout << a & b;       // falsch, da << eine hoehere Prioritaet als & hat
cout << (a & b);     // richtig
cout << (i << 2);    // Vermeiden, da verwirrend
```

Die Formatierung kann über sogenannte Manipulatoren oder Elementfunktionen der Klassen `ostream` bzw. `ios` beeinflusst werden.

Beispiel:

```
// iotest1.cpp: Anwendungen der iostream-Bibliothek

#include <iostream>

int main() {
   int   index   = -23;
   float abstand = 12.345f;
   char  zeichen = 'X';

   cout << "Wert von index   : " << index    << endl;
   cout << "Wert von abstand : " << abstand  << endl;
   cout << "Wert von zeichen : " << zeichen  << endl;

   index = 31;
   cout << "Wert von index dezimal    : " << dec << index << endl;
   cout << "Wert von index oktal      : " << oct << index << endl;
   cout << "Wert von index hexadezimal: " << hex << index << endl;
   cout << "Bitte ganze Zahl eingeben --> "
   if (cin  >> index)      // Eingabe OK ?
      cout << "Wert der Eingabe hexadezimal: " << index << endl;
   else
      cerr << "Fehler: Falsche Eingabe !\n";   return 0;
}
// Ausgabe:
// Wert von index   : -23
// Wert von abstand : 12.345
// Wert von zeichen : X
// Wert von index dezimal    : 31
// Wert von index oktal      : 37
// Wert von index hexadezimal: 1f
// Bitte ganze Zahl eingeben -->123
// Wert der Eingabe hexadezimal: 7b
```

`dec`, `oct` und `hex` sind sogenannte Manipulatoren zum Verändern des Datenstromes (→unten)

Einlesen von Zeichen

Der Operator >> liest nur das nächste nichtleere Zeichen aus dem Eingabestrom.

```
char c;
cin >> c;    // liest das naechste nichtleere Zeichen
```

Sollen auch Leerzeichen gelesen werden, so kann dazu die spezielle Methode `get()` verwendet werden.

```
char c;
cin.get(c);  // lese naechstes Zeichen in c ein
```

Der Ausdruck `cin.get(c)` liefert `true` im Erfolgsfalle und `false` bei Erreichen von `EOF`. Das Gegenstück zu `get()` bei `cout` heißt `put()` und gibt ein Zeichen auf die Standardausgabe aus.

Beispiel: Standardeingabe zeichenweise kopieren

```
char c;
while (cin.get(c))
    cout.put(c);     // ein Zeichen ausgeben
```

6.4 Einfache Fehlerbehandlung

Bisher sind wir immer von der optimistischen Annahme ausgegangen, dass der Benutzer unserer Programme seine Eingaben immer richtig macht.

Problem: Die korrekte Eingabe des Benutzers kann nicht sichergestellt werden. Ein Programm muss auf Fehleingaben reagieren können;

Beispiel: Programm erwartet `int`-Wert:

```
int i;
cin >> i;
```

Aber der Benutzer gibt ein:

```
Hallo
```

Folge: Das Programm kann keinen sinnvollen ganzzahligen Wert lesen;

- iostream-Objekte protokollieren ihren Zustand mit. Dieser Zustand kann über spezielle Elementfunktionen abgefragt werden.
- Jedes iostream-Objekt verwaltet seinen Zustand getrennt und unabhängig von den anderen.
- Im obigen Beispiel merkt sich `cin`, daß eine Eingabe misslungen ist; Diese Information kann nachher abgefragt werden. Es liegt in der Verantwortung des Programms, diese Information auszuwerten und ggf. passend zu reagieren.

Test auf erfolgreiche Eingabe:

```
cin.good();
```

liefert einen bool-Wert zurück:
`true`, wenn die letzte Eingabe erfolgreich war
`false` ansonsten;

Folgendes Programmfragment erkennt Fehleingaben:

```
int i;
cin >> i;
if (cin.good())      // gleichbedeutend mit if (cin)
    cout << "Ok, " << i << " eingegeben" << endl;
else
    cout << "Eingabefehler" << endl;
```

Eine weitere Fehlerquelle bei der Eingabe kann sein, dass die Eingabedaten erschöpft sind, im Klartext, man hat das Ende des Eingabestroms erreicht. Dies kann z.B. dann sein, wenn man die Standardeingabe auf eine Datei umgelenkt hat und diese nun zu Ende gelesen ist.

Abfrage auf Ende der Eingabedatei:

```
cin.eof()
```

liefert einen `bool`-Wert zurück:
`true`, wenn die Eingabedaten verbraucht sind und nichts mehr geliefert wird;
`false` ansonsten

In diesem Fall hat z.B. eine Aufforderung zur Wiederholung der Eingabe keinen Sinn.

Vorsicht: `cin.eof()` zeigt das Ende der Eingabe erst an, wenn der erste erfolglose Leseversuch unternommen wurde, (aber noch nicht, nachdem die letzte Eingabe erfolgreich gelesen worden ist).

Beispiel: Eingabe mit Test auf Fehleingabe und Dateiende

```
int i;
cin >> i;
if (cin.good())
    cout << "Ok, " << i << " eingegeben" << endl;
else if (cin.eof())
    cout << "Eingabedaten erschoepft" << endl;
else
    cout << "Eingabefehler" << endl;
```

I/O-Zustand

Ein iostream-Objekt merkt sich nur den Zustand der letzten I/O-Aktion. Der vorletzte und weiter zurückliegende Zustände werden nicht gespeichert.

Ein fehlerhafter Zustand bleibt so lange gespeichert, bis er ausdrücklich gelöscht wird. Dazu gibt es die folgende Elementfunktion:

```
cin.clear()
```

Wirkung: Das sogenannte `good`-Bit wird auf `true` gesetzt, eventuelle Fehlerbits auf `false`.

Problem: Falls kein korrekter Wert von `cin` gelesen werden konnte, bleiben die fehlerhaften Daten im Eingabestrom stehen und werden beim nächsten Eingabeversuch wieder gelesen. Diese fehlerhaften Daten müssen also zunächst aus dem Eingabestrom entfernt werden.

Dazu gibt es folgende Elementfunktion:

```
cin.ignore(int n, char c)
```

Wirkungsweise:
Es werden maximal n Zeichen überlesen. Das Zeichen c legt fest, bis zu welchem Zeichen im Eingabestrom die Eingabe zu verwerfen ist. Üblicherweise nimmt man hier das Linefeed-zeichen `'\n'`.
`ignore` liest also die Eingabe, bis

- entweder n Zeichen gelesen wurden
- oder ein Zeichen c auftaucht

Im folgenden Beispiel werden maximal 80 Zeichen bzw. der Rest der Zeile ausgelassen.

```
int i;
cin >> i;                   // Erster Leseversuch
while(cin.fail())           // Test auf Misserfolg
{
    cin.clear();            // Misserfolg: Zustand zurücksetzen
    cout << "Fehleingabe - bitte wiederholen" << endl;
    cin.ignore(80, '\n');   // Fehleingabe überspringen
    cin >> i; // neuer Leseversuch
}
```

Was im obigen Beispiel noch nicht berücksichtigt wurde, ist der Fall, dass man beim Lesen auf das Dateiende stößt. Dies kann man abschließend nun so lösen:

```
int i;
cin >> i;                    // Erster Leseversuch
while(!cin.good())           // Test auf Misserfolg
{
    if(cin.eof())            // Fortsetzung überhaupt sinnvoll?
        break;
    cin.clear();             // Misserfolg: Zustand zurücksetzen
    cout << "Fehleingabe - bitte wiederholen" << endl;
    cin.ignore(80, '\n');    // Fehleingabe überspringen
    cin >> i; // neuer Leseversuch
}
if(cin.eof())
    // Datenende behandeln
else
    // Eingabe ok
```

Beispiel:

```
// linecount.cpp: Zeilen und Zeichen zaehlen

#include <iostream>

int main() {
    int anzZeilen = 0, anzZeichen = 0;

    for(char c = cin.get(); cin.good(); c = cin.get())
    {
        cout.put(c);
        anzZeichen++;
        if(c == '\n')
            anzZeilen++;
    }
    cout << anzZeichen << " Zeichen\n"
         << anzZeilen  << " Zeilen" << endl;
}
```

6.5 Formatierungsmöglichkeiten bei der Ein-/Ausgabe

Prinzipiell gibt es 2 Möglichkeiten, das Aus- bzw. Eingabeformat zu beeinflussen: Elementfunktionen der Klassen `istream` bzw. `ostream` oder sogenannte Manipulatoren (`<iomanip>`), die in den Datenstrom eingefügt werden.

6.5.1 Feldbreite, Ausrichtung und Füllzeichen

Feldbreite und Füllzeichen setzen

Elementfunktion	***Manipulator***	
`cout.width (10)`	`cout << setw(10)`	setzt die Feldbreite zum Einlesen oder Ausgeben (wird automatisch zurückgesetzt)
`cout.fill ('0');`	`cout << setfill('0')`	Füllzeichen setzen, Voreinstellung `' '`

Ausrichtung des Feldes

`cout.setf (ios::left, ios::adjustfield)`	linksbündige Ausrichtung des auszugebenden Feldes
`cout.setf (ios::right, ios::adjustfield)`	rechtsbündige Ausrichtung des

	auszugebenden Feldes (Voreinstellung)
`cout.setf (ios::internal, ios::adjustfield)`	Vorzeichen linksbündig, Wert rechtsbündig

Beispiel:

```
// ausrichtung.cpp: Ausrichtung und Feldbreite

#include <iostream>
#include <iomanip>

void main() {
int zahl = -17;

    // Feldbreite 10 und Fuellzeichen #
    cout << setw(10) << setfill('#') << zahl << endl;

    // linksbuendig ausgeben
    cout.setf (ios::left, ios::adjustfield);
    cout << setw(10) << setfill('#')<< zahl << endl;

    // Vorzeichen links und Wert rechts
    cout.setf (ios::internal, ios::adjustfield);
    cout << setw(10) << setfill('#')<< zahl << endl;
}
/* Ausgabe
#######-17
-17#######
-#######17
*/
```

6.5.2 Formatierung ganzer Zahlen

Einstellung von Ausgabeformaten:

`cout.setf (ios::showbase)`	Ausgabe mit der Basis, bei oktal fuehrende `0`, bei Hex fuehrendes `0x`
`cout.setf (ios::showpos)`	Ausgabe von positiven Vorzeichen
`cout.setf (ios::uppercase)`	Ausgabe von Buchstaben groß, z.B.bei Hex

Zurücksetzen von Ausgabeformaten:

```
cout.unsetf (ios::showbase );
cout.unsetf (ios::showpos );
cout.unsetf (ios::uppercase )
```

Setzen der Zahldarstellung:

Elementfunktion	***Manipulator***	
`cout.setf (ios::hex)`	`cout << hex`	Hexadezimaldarstellung einschalten
`cout.setf (ios::oct)`	`cout << oct`	Oktaldarstellung einschalten
`cout.setf (ios::dec)`	`cout << dec`	Dezimaldarstellung einschalten

Beispiel

```
// ganzzahl.cpp: Ein-/Ausgabemanipulation bei ganzen Zahlen

#include <iostream>
#include <iomanip>
```

```
void main() {

    // Ausgaben mit Anzeige der Basis
    cout.setf (ios::showbase);
    cout << hex << 100 << " " << oct << 100 << "\n\n";

    // uppercase und lowercase
    cout.setf (ios::uppercase);
    cout << hex << 100 << endl;

    cout.unsetf (ios::uppercase);
    cout << hex << 100 << "\n\n";

    // Ausgaben mit positivem Vorzeichen
    cout.setf (ios::showpos);

    // Mit Manipulator
    cout << dec << 100 << " "
         << oct << 100 << " "
         << hex << 100 << endl;

    // Mit Elementfunktion
    cout.setf(ios::dec); cout << 101 << " ";
    cout.setf(ios::oct); cout << 101 << " ";
    cout.setf(ios::hex); cout << 101 << endl;
}
/* Ausgabe
0x64 0144

0X64
0x64

+100 0144 0x64
+101 0145 0x65
*/
```

6.5.3 Formatierung von Gleitpunktzahlen

Einstellung des Ausgabeformats:

`cout.setf (ios::fixed,      ios::floatfield)`	Dezimaldarstellung
`cout.setf (ios::scientific, ios::floatfield)`	Exponentialdarstellung
`cout.setf (ios::showpoint)`	Dezimalpunkt und abschließende Nullen immer darstellen

Einstellung der Genauigkeit:

Elementfunktion	***Manipulator***	
`cout.precision(4)`	`cout << setprecision(4)`	setzt die Genauigkeit der Ausgabe, Voreinstellung ist 6

Bei der Ausgabe der Nachkommastellen wird gerundet ausgegeben, die Zahl selbst bleibt aber nach wie vor mit ihrer vollen Stellenzahl gespeichert.

Beispiel:

```
// gleitpunkt.cpp: Ein-/Ausgabemanipulation bei Gleitpunktzahlen

#include <iostream>
#include <iomanip>

void main() {
```

```
    cout << "precision normal\n";
    cout << "    2:\t" << setprecision(2) << 321.0 << '\t'
                       << 0.12345678 << endl;
    cout << "    6:\t" << setprecision(6) << 321.0 << '\t'
                       << 0.12345678 << endl;

    cout << "\nprecision showpoint\n";
    cout.setf (ios::showpoint);
    cout << "    2:\t" << setprecision(2) << 321.0 << '\t'
                       << 0.12345678 << endl;
    cout << "    6:\t" << setprecision(6) << 321.0 << '\t'
                       << 0.12345678 << endl;

    cout << "\nprecision fixed\n";
    cout.setf (ios::fixed, ios::floatfield);
    cout << "    2:\t" << setprecision(2) << 321.0 << '\t'
                       << 0.12345678 << endl;
    cout << "    6:\t" << setprecision(6) << 321.0 << '\t'
                       << 0.12345678 << endl;

    cout << "\nprecision scientific\n";
    cout.setf (ios::scientific, ios::floatfield);
    cout << "    2:\t" << setprecision(2) << 321.0 << '\t'
                       << 0.12345678 << endl;
    cout << "    6:\t" << setprecision(6) << 321.0 << '\t'
                       << 0.12345678 << endl;
}
/* Ausgabe
precision normal
    2:  3.2e+02 0.12
    6:  321     0.123457

precision showpoint
    2:  3.2e+02 0.12
    6:  321.000 0.123457

precision fixed
    2:  321.00  0.12
    6:  321.000000      0.123457

precision scientific
    2:  3.21e+02        1.23e-01
    6:  3.210000e+02    1.234568e-01
*/
```

6.5.4 Formatierungsmöglichkeiten bei istream

Für die Klasse `istream` gibt es nur dem Manipulator `ws`, mit dem Leerzeichen überlesen werden können. Das Einlesen von Zeichenketten kann mit der Elementfunktion `width(int)` begrenzt werden.

Beispiel:

```
// eingabe.cpp: Leerzeichen verschlucken und Eingabebreite

#include <iostream>
#include <iomanip>

int main() {
    char name[20];
    cout << "Name: ";
    cin >> ws;        // Leerzeichen entfernen
    cin.width(sizeof(name));     // Maximal 19 Zeichen einlesen
```

```
    cin >> name;     // eventuell zuviel eingegebene Zeichen stehen
                     // noch im Eingabepuffer
    cout << "Eingegeben wurde:" << name << endl;
}
```

Es gibt einige weitere Formatierungsmöglichkeiten. Hierzu sei jedoch auf die aktuelle Literatur verwiesen.

7 Funktionen

7.1 Einführung

Wozu benötigt man Funktionen?

Der Grundidee von Funktionen ist verhältnismäßig einfach: Eine Folge von Anweisungen wird mit einem Namen versehen. Um die Anweisungsfolge auszuführen, muss nur noch ihr Name genannt werden.

Zweck:

- Wiederverwendbarkeit von Code für unterschiedliche aktuelle Daten
- Strukturierung von Programmen, Verbesserung der Lesbarkeit von Programmen
- Verbesserung der Handhabbarkeit großer Programme
- Wiederverwendbarkeit auch über die aktuelle Problemstellung (das aktuelle Programm) hinaus
- Verbesserung der Korrektheit von Programmen

Beispiel

Angenommen ein Programm gibt eine größere Menge von Daten aus. An bestimmten Stellen soll angehalten (z.B. nach jeder Bildschirmseite) und auf eine Eingabe des Benutzers gewartet werden, dann wird das Programm wieder fortgesetzt. Der Benutzer hat so Gelegenheit, die Daten in Ruhe durchzusehen.

Das folgende Programmfragment zeigt den entsprechenden Code:

```
// Daten ausgeben

// Pause, auf Bestaetigung des Benutzers warten
cout << "Press return to continue..." << flush;
cin.get();

// weitere Daten ausgeben

// Pause, auf Bestaetigung des Benutzers warten
cout << "Press return to continue..." << flush;
cin.get();
```

Das Codefragment kommt zweimal in völlig gleicher Form vor. Damit bietet es sich zur Ausgliederung in eine Funktion an. Das doppelte Codefragment wird herausgenommen, mit dem Bezeichner `wait` versehen und in eine Funktion verpackt:

```
// Funktionsdefinition
void wait()
{
    // Pause, auf Bestaetigung des Benutzers warten
    cout << "Press return to continue..." << flush;
    cin.get();
}
```

Um die Funktion auszuführen, wird nur noch ihr Name (gefolgt von leeren runden Klammern) angegeben:

```
// Daten ausgeben
```

```
wait();

// weitere Daten ausgeben

wait();
```

Bei jedem Namen `wait` wird die Funktion aufgerufen, d.h. die ausgegliederten Anweisungen werden zweimal ausgeführt.

7.2 Deklaration und Definition von Funktionen

Definition einer Funktion:

Funktionstyp Funktionsname (Deklaration der formalen Parameter)
{
Folge von Anweisungen und Ausdrücken
}

Eine Funktionsdefinition beginnt mit einem **Funktionskopf**, dem der **Funktionsrumpf** folgt. Der Funktionskopf liefert alle Informationen, die zum Aufruf der Funktion notwendig sind.

- den Funktionsnamen und den Funktionstyp
- die Liste der formalen Parameter

Der Funktionsrumpf einer Funktion ist ein Block, selbst wenn er nur eine einzige Anweisung enthält. Er legt die Implementierung der Funktion fest, d.h. die Anweisungen, die beim Funktionsaufruf ausgeführt werden müssen.

Variablen und Konstanten, die innerhalb des Funktionsrumpfes definiert werden, sind lokal innerhalb der Funktion und können daher nicht direkt von außerhalb einer Funktion angesprochen werden. Die Lebensdauer solcher lokaler Größen beschränkt sich auch auf den Zeitraum des Funktionsablaufs, d. h. mit Beendigung einer Funktion werden die Variablen nd Konstanten wieder freigegeben.

Der Funktionsaufruf (engl. " function call") ist aus syntaktischer Sicht ein Ausdruck, wie arithmetische Ausdrücke oder Aufrufe vordefinierter Funktionen.

Funktionen können nicht an beliebigen Stellen im Programm definiert werden. Sie unterscheiden sich darin von Variablen. Funktionen dürfen nur außerhalb jedes Blocks definiert werden. Sie sind damit die "obersten" Bausteine von Quelltext-Dateien.

Eine Funktion muss vor ihrer Verwendung entweder im Programmtext darüber definiert worden sein oder es muss eine Funktionsdeklaration erfolgt sein.

Deklaration einer Funktion: (Funktionsprototyp)

Funktionstyp Funktionsname (Deklaration der formalen Parameter);

Beispiel:

```
void wait();   // deklariert die Funktion wait
```

Durch einen solchen Ausdruck wird dem Compiler mitgeteilt, dass eine solche Funktion entweder weiter unten im Quelltext oder in einer anderen Programmdatei, die zu dem Programm hinzugebunden werden muss definiert ist. Ist eine solche Funktion tatsächlich nicht

vorhanden, so wird dies erst dem Linker auffallen beim Zusammenbinden des ausführbaren Programmes.

7.3 Funktionsaufruf und Parameterübergabe

Was passiert beim Funktionsaufruf?

- Der Binärcode einer Funktion steht an einer eigenen Stelle im Hauptspeicher
- Wird im Programmcode ein Funktionsaufruf erreicht, so springt der sogenannte Programmzeiger zu der Anfangsadresse der Funktion. Zuvor wird jedoch die Rücksprungadresse, d. h. die Adresse, wo der Programmzeiger zuletzt gestanden hat, im sogenannten Rücksprungstack gespeichert (*push*).
- Die Funktion wird durchlaufen bis zur letzten Anweisung oder bis zu einem `return`.
- Der Programmzeiger verzeigt zu der Rücksprungadresse, die im Rücksprungstack zuletzt abgelegt wurde und entfernt diese aus dem Rücksprungstack (*pop*).
- Der Programfluss geht mit der nächsten Anweisung nach dem Funktionsaufruf weiter.

Dieses Verfahren macht es möglich, dass innerhalb einer Funktion wiederum eine Funktion aufgerufen werden kann und dort wiederum usw. und dass trotzdem gewährleistet ist, dass der Programmablauf immer wieder zu seinem Ausgangspunkt zurückkehrt.

7.3.1 Funktionstyp

Man unterscheidet zwischen Funktionen ohne Rückgabewert, den sogenannten `void`-Funktionen (Prozeduren im Sinne von Pascal) und Funktionen mit Rückgabewert. Als Funktionstyp ist jeder gültige C++-Datentyp erlaubt, also sowohl elementare Datentypen als auch die erst später zu besprechenden zusammengesetzten und benutzerdefinierten Datentypen.

Ein Funktionsaufruf mit Rückgabewert kann innerhalb von Ausdrücken passenden Typs beliebig verwendet werden.

Ein Aufruf einer `void`-Funktion kann nur alleinstehend erfolgen.

7.3.2 Wertaufruf

Bei einem Funktionsaufruf können auch Werte an die Funktion übergeben werden. Die übliche Übergabe von Parametern an Funktionen erfolgt per Wertaufruf (engl. *call by value*).

Beispiel:

```
double sqr (double x) {
   return x * x;
}
```

Mögliche Aufrufe:

```
double y = 1.5;
cout << sqr(y) << endl;
cout << sqr(3.0 * 5.2);
```

In der obigen Funktion wird der Parameter als Wertparameter übergeben, d. h. beim Aufruf der Funktion wird für den formalen Parameter x Speicherplatz allokiert, der mit dem beim Funktionsaufruf übergebenen aktuellen Parameter initialisiert wird. Die so definierte Variable x

ist lokal innerhalb der Funktion und wird demnach mit Beendigung der Funktion wieder freigegeben.

Bei einem Wertaufruf kann als aktueller Parameter ein beliebiger Ausdruck eines zu dem Typ des formalen Parameters passenden Typs übergeben werden. Insbesondere ist auch die Übergabe von R-Werten möglich.

Hat eine Funktion mehrere formale Parameter, so müssen beim Aufruf entsprechend viele und passende aktuellen Parameter übergeben werden.

7.3.3 Referenzaufruf

In der Programmiersprache Pascal gibt es bei Funktionen und Prozeduren auch die Möglichkeit, sogenannte Referenzparameter zu übergeben. In C++ löst man dies durch sogenannte Referenzen.

Eine Referenz definiert einen Aliasnamen für eine bereits deklarierte Variable und benötigt daher keinen eigenen Speicherplatz.

Beispiel:

```
int i = 5;
int &j = i;     // j ist eine Referenz auf int
                // j belegt keinen zusätzlichen Speicherplatz
j = 6 ;
cout << i;      // i == 6, denn i und j belegen den selben Speicherplatz
```

Referenzen werden hauptsächlich verwendet, um Argumente als Referenzparameter an Funktionen zu übergeben.

Einen Referenzaufruf (engl.: *call by reference*) definiert man demzufolge also so:
Beispiel:

```
// inkrementieren
  void incr (int& j)   // j ist der Referenzparameter
  { j++; }

  // in main:
    i = 1;
    incr(i);      // i == 2 !
```

Ist als formaler Parameter eine Referenz angegeben, so wird bei der Übergabe des aktuellen Parameters kein eigener Speicherplatz bereitgestellt. Vielmehr wird der formale Parameter für die Dauer des Funktionsaufrufs als Referenz auf den übergebenen aktuellen Parameter behandelt, d. h. in der Funktion kann der übergebene Parameter direkt verändert werden.

Beispiel:

```
// Vertauschen von Variableninhalten
  void tausche (int& i, int& j) {
    int tmp = j;
    j = i;
    i = tmp;
  }
  //in main:
    tausche(a,b);   //Werte der Variablen a,b werden vertauscht
```

Da in der Funktion die übergebenen Parameter verändert werden können, sind als Parameter auch nur L-Werte erlaubt. Es ginge also im obigen Beispiel folgendes nicht:

```
tausche(2, 3); // Unsinn !
```

Empfehlung: Gehen Sie sparsam mit Referenzparametern um!

7.4 Inline-Funktionen

Durch Angabe des Schlüsselwortes `inline` vor der Funktionsdefinition wird an den Compiler die Anweisung gegeben, überall wo die Funktion aufgerufen wird, den kompletten Funktionsrumpf einzufügen an Stelle des Funktionsaufrufes.

Beispiel:

```
inline int max (int a, int b)
{ return (a > b)? a : b; }
```

Wenn nun im Programm z.B. irgendwo `max(i,j)` auftaucht, ersetzt der Compiler diesen Ausdruck durch: `(i > j)? i : j;`

Konsequenzen:

- Erhöhung der Effizienz
- eventuelle Vergrößerung des Codes
- es werden keine Funktionsmakros mehr benötigt
- Inline-Funktionen sind für Debugger sichtbar im Gegensatz zu Funktionsmakros

Frühere Lösung: Funktionsmakro

```
#define MAX(a,b) (((a)>(b))?(a):(b))
```

Der Ausdruck `MAX(a,b)` mit beliebigen Ausdrücken `a` und `b` wird dann durch den Präprozessor im kompletten Programmtext durch den rechtsstehenden Ausdruck ersetzt.

Nachteil bei Funktionsmakros:

```
MAX(a++,b)###(a++>b)?a++:b       // a wird eventuell 2* inkrementiert
MAX(3,"Hallo")             // nicht typsicher!
```

Bemerkung:

`inline` ist nur eine Empfehlung an den Compiler, d.h. es gibt keine Garantie für das "Inlining". Ab einer gewissen Komplexität wird vom Compiler kein Inlining durchgeführt (z.B.: mindestens eine Schleife oder ab einer gewissen Anzahl Statements).

Empfehlung: Generell keine Funktionsmakros verwenden, statt dessen besser Inline-Funktionen.

7.5 Überladen von Funktionsnamen

Normalerweise sollten unterschiedliche Funktionen auch unterschiedliche Namen haben. Wenn Funktionen jedoch dieselbe Aufgabe auf unterschiedlichen Datentypen ausführen, so kann es durchaus sinnvoll sein, ihnen auch den selben Namen zu geben.

Beispiel:

```
// show.cpp: Ueberladen von Funktionsnamen

#include <iostream>

void show (int wert) {
    cout << "Integer: " << wert << endl;
}
void show (float wert) {
   cout << "Float : " << wert << endl;
}
void show (double wert) {
   cout << "Double : " << wert << endl;
}
void show (char wert) {
    cout << "Char: " << wert << endl;
}
void show (char* wert) {
    cout << "Char*: " << wert << endl;
}

int main() {
   show (12);              // show(int)
   show (3.14159);         // show(double) denn das Literal 3.1459 ist
                          // vom Typ double und nicht float
   show ("Hallo Welt !");  // show(char*)
   return 0;
}
```

7.5.1 Signatur einer Funktion

Funktionsnamen müssen in C++ nicht eindeutig sein. Die Unterscheidung von Funktionen erfolgt aufgrund der Signatur. Die Signatur setzt sich zusammen aus dem Funktionsnamen, sowie aus Anzahl und Typ der Parameter. Der Funktionswerttyp gehört jedoch nicht zur Signatur.

```
show (int) ### show (double)
show (int, int) ### show (int)
```

7.5.2 Einschränkungen beim Überladen von Funktionsnamen

```
void show(int)
int show (int)       // können nicht unterschieden werden

double g(double &);  // mehrdeutig, da double und double&
double g(double);    // nicht unterschiedlich genug

int h (int);         // mehrdeutig, da int und const int
int h (const int);   // nicht unterschiedlich genug

int k(int *);        // mehrdeutig, da int* und int[] nicht
int k(int[]);        // unterschieden werden können
```

7.5.3 Regeln zum Auffinden der am besten passenden Funktion

Beispiel:

```
int f(int, double);
int f(double, int);
```

Welche der beiden Funktionen wird bei `f(1,2)` aufgerufen?

Voraussetzung: Aufruf: `f(p1, ..... ,pn)`
Betrachte alle Funktionen `f` mit `n` Parametern.

1. Exakte Übereinstimmung
   alle Parametertypen passen exakt zusammen bis auf höchstens triviale Typumwandlungen (`double` ### `double&` oder `int[]` ® `int*` oder `int` ® `const int`)

2. Übereinstimmung nach Typausweitung
   z.B.: `short` ### `int` ### `long`
   `float` ### `double`### `long double`
   `char` ### `int`
   `bool` ® `int`

3. Übereinstimmung nach Standardumwandlungen
   z.B.: `int` ### `double`
   `double` ® `int`
   `unsigned int`###`int`

4. Übereinstimmung nach benutzerdefinierten Umwandlungen. Bei benutzerdefinierten Datentypen (Klassen) ist die Definition eigener Typumwandlungen möglich.
   z.B.: `double` ### `complex`
   `char*` ® `string`

5. Übereinstimmung durch variable Parameterlisten (. . .)

Gibt es auf einer Ebene zwei passende Funktionen, so wird der Aufruf als mehrdeutig abgelehnt. Im obigen Beispiel ist dies der Fall.

7.6 Standard-Argumente

Funktionen benötigen oft für den allgemeinen Fall mehr Argumente als für den einfachen, jedoch häufigen, Fall.

Beispiel: Funktion, die eine Zahl wahlweise zur Basis 10, 16, 8, 2 ausgibt.
Voreinstellung soll jedoch 10 sein

```
void show(int value, int base=10); // base wird mit 10 voreingestellt

show(31);            // entspricht: show(31,10)
show(31,16);         // Basis: 16 (Hexadezimal)
```

Standardwerte sind nur von rechts nach links möglich.

```
void f(int a=0, int b, int c=1);  // ist falsch
void f(int a, int b=0, int c=1);  // ist moeglich
```

Standardwerte werden im Funktionsprototyp definiert und nicht in der Funktionsdefinition, es sei denn es wird kein Funktionsprototyp benötigt.

z.B.: `void show (int, int=10);`

Die Regeln zum Überladen von Funktionen erweitern sich natürlich entsprechend. Hat eine Funktion Standardwerte für Argumente, so kommt sie natürlich für Funktionsaufrufe mit unterschiedlichen Parameterzahlen in Frage.

8 Zusammengesetzte und benutzerdefinierte Datentypen

Im nun folgenden Abschnitt werden wir aufbauend auf den elementaren Datentypen daraus neue Datentypen zusammensetzen. Im Einzelnen sind dies zunächst Aufzählungstypen (`enum`), Felder (arrays), Zeichenketten und Strings, sowie Strukturen (`struct`).

8.1 Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche für bestimmte Größen, z. B. kann ein Wochentag die Werte *Sonntag*, *Montag*, ..., *Samstag* annehmen oder ein Farbwert könnte die Farben *rot*, *grün*, *blau*, usw. annehmen. Unsere bisherige Lösung würde so aussehen:

```
int wochentag;      // Sonntag = 0
                    // Montag = 1 usw.

int farbe;          // rot = 0
                    // grün = 1 usw.
```

Anwendung im Programm:

```
if (wochentag == 4)

if (farbe == 5)

wochentag = farbe;  // erlaubt ???
```

Nachteile dieser Lösung:

- Die erlaubten Werte müssen als Kommentar festgehalten werden
- Zugeordnete Zahlen sind nicht eindeutig, z.B. steht `0` sowohl für `rot`, als auch für `Sonntag`
- Die verwendeten Literale haben einen schlechten Dokumentationswert.

Mögliche bessere Lösung

```
int wochentag;

// zugehörige Werte
const int Sonntag = 0;
const int Montag  = 1;
....

int farbe;
// zugehörige Werte
const int rot   = 0;
const int gruen = 1;
....

// Anwendung

if (wochentag == Sonntag)
...
if (farbe == rot)

// aber:

wochentag = gruen;  // kein Syntaxfehler
```

Nachteile dieser Lösung:

- Die Definition der vielen Konstanten ist mühsam

- Zwischen den verschiedenen Wertebereichen gibt es keine Typabsicherung, die durch den Compiler überprüft werden könnte.

Die Lösung dieses Problems sind die sogenannten Aufzählungs- oder Enumerationstypen.

Syntax:

```
enum [Typname] {Aufzählung} [Variablenliste]
```

Bedeutung:

- Durch eine `enum`-Definition wird ein neuer Datentyp definiert, dessen möglicher Wertebereich nur aus den aufgezählten Werten besteht.
- Die Namen für die Werte müssen innerhalb des Geltungsbereichs des `enum`-Typs eindeutig sein.
- Der Typname und die Variablenliste können weggelassen werden. Der Typname sollte aber normalerweise verwendet werden.

Beispiele:

```
enum Farbtyp {rot, gruen, blau, gelb};
enum Wochentag {sonntag, montag, dienstag, mittwoch,
                donnerstag, freitag, samstag};

// Definition von Variablen dieser Typen
Farbtyp farbe1, farbe2=rot;
Wochentag werktag, heute=mittwoch;

// Anwendungen

if (farbe1 == rot)
...
if (heute == sonntag)
...

heute = rot;   // Typkonflikt, Übersetzungsfehler !!
```

Den mit Hilfe von Aufzählungstypen definierten Variablen können ausschließlich Werte aus der dazugehörigen Liste zugewiesen werden, Mischungen von verschiedenen Aufzählungstypen sind nicht erlaubt.

Interne Darstellung

Aufzählungstypen werden intern als natürliche Zahlen dargestellt beginnend mit `0`, d. h. in den obigen Beispielen wird z. B. der Wert `sonntag` intern mit `0` dargestellt, der Wert `montag` mit `1` usw. Daher gelten Aufzählungstypen wie `char` und `int` ebenfalls als ganzzahlige Datentypen und können z. B. in `switch`-Anweisungen verwendet werden.

Abweichung von der Standardvoreinstellung

In begründeten Fällen kann von der Standardvoreinstellung abgewichen werden, wie die Beispiele zeigen.

```
enum Farbtyp {rot=0, gruen=1, blau=2, gelb=4};

enum Interrupt {Video=0x10, Reset=0x19,
                MsDos=0x21, User1=0x80
};
```

Typkonvertierung zu anderen Typen

`enum`-Variablen können automatisch zu `int`-Typen konvertiert werden. Umgekehrt ist eine automatische Konvertierung nicht vorgesehen. Hier muss explizit konvertiert werden.

```
int i = montag;      // ok, wird automatisch konvertiert

Wochentag t = i;     // Fehler! keine automatische Konvertierung

t = static_cast<Wochentag>(i);   // funktioniert, kann aber gefährlich
                                 // sein
t = static_cast<Wochentag>(10);  // undefiniert !!
```

Beispiel:

```
// wochentag.cpp: Aufzaehlungstyp Wochentag

#include <iostream>

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

Wochentag naechsterTag (Wochentag t) {
    if (t == samstag)
        return sonntag;
    else
        return static_cast<Wochentag>(t+1);
}

void ausgeben(Wochentag t) {
    switch ( t) {
        case sonntag   : cout << "Sonntag";
                         break;
        case montag    : cout << "Montag";
                         break;
        case dienstag  : cout << "Dienstag";
                         break;
        case mittwoch  : cout << "Mittwoch";
                         break;
        case donnerstag: cout << "Donnerstag";
                         break;
        case freitag   : cout << "Freitag";
                         break;
        case samstag   : cout << "Samstag";
                         break;
        default        : cout << "Kann nicht vorkommen!";
                         break;
    }
}

int main() {
    Wochentag t;

    for (t = mittwoch; t != dienstag; t = naechsterTag(t)) {
        ausgeben(t);
        cout << endl;
    }
} /*
Mittwoch
Donnerstag
Freitag
Samstag
Sonntag
Montag  */
```

Weitere mögliche Anwendungen:

- Als möglicher Wertetyp für Testmenüs

```
enum FunktionsTyp { Beenden, Addition, Subtraktion, Division,
                    Multiplikation};
```

- Als möglicher Wertetyp für Fehlerbehandlung

```
enum FehlerTyp { OK, Eingabefehler, Bereichsfehler, SchwererFehler};
```

8.2 Felder

Ein **Feld** (Vektor, engl. *array*) besteht aus mehreren Datenelementen gleichen Datentyps, die hintereinander im Speicher abgelegt sind. Die einzelnen Datenelemente heißen Feldelemente. Sie können über einen sogenannten Index angesprochen werden.

8.2.1 Eindimensionale Felder

Definition eines Felds

```
Typangabe Feldname [ Anzahl ];
```

Durch die Typangabe wird ein Datentyp für die einzelnen Feldelemente bestimmt. Die Anzahl muss eine ganzzahlige Konstante oder ein ganzzahliger Ausdruck sein, der nur aus Konstanten besteht.

Beispiel:

```
int tab [10];
```

Hierdurch wird ein Feld `tab` mit 10 Elementen vom Typ `int` definiert. Das Objekt `tab` selbst ist vom Typ `int`-Feld.

tab[0]	
tab[1]	
tab[2]	
tab[3]	
tab[4]	
tab[5]	
tab[6]	
tab[7]	
tab[8]	
tab[9]	

Ein Feld belegt immer einen zusammenhängenden Speicherbereich. Im Falle unseres Feldes `tab` sind das `10 * 4 = 40 Bytes`.

Zugriff auf Feldelemente

Der Zugriff auf einzelne Feldelemente erfolgt mit Hilfe des sogenannten Indexoperators `[ ]`. In C++ beginnt ein Index immer mit 0. Die Elemente des Feldes `tab` sind also:

```
tab[0], tab[1], ..., tab[9]
```

- Der Index des letzten Feldelementes ist immer um 1 niedriger als die Anzahl der Feldelemente.
- Als Index kann jeder `int`-Ausdruck verwendet werden. Es sind auch `enum`-Ausdrücke denkbar.

- Der Indexoperator besitzt einen hohen Vorrang.

Was passiert bei einem Zugriff auf einen nicht zulässigen Index ?

```
tab[10] = 0;          // ??
tab[-1] = 100;        // ??
```

Das Laufzeitsystem von C++ überprüft keine Feldgrenzenüberschreitungen, d. h. ggf. werden einfach Speicherelemente überschrieben, die hinter oder vor dem Feld im Hauptspeicher stehen!

Beispiel:

```
// feldeingabe.cpp: Eingabe eines Feldes

#include <iostream>

int main() {
    const int MAXANZ = 10; // Konstante fuer die Feldgroesse
    int tab[MAXANZ];          // Felddefinition
    int zahl, i, anzahl;   // Zahl, Index, Anzahl

    // Zahlen einlesen:
    cout << "Bis zu 10 Zahlen eingeben\n"
         << "(Abbruch mit einer negativen Zahl):" << endl;
    cin >> zahl;
    for( i = 0;   zahl >= 0; ++i) {
        tab[i] = zahl;
        cin >> zahl;
    }
    anzahl = i;      // tatsaechlich eingegebene Anzahl an Zahlen

    cout << "Eingegeben wurden die folgenden "
         << anzahl << " Zahlen:\n" << endl;

    for( i = 0; i < anz; ++i)
        cout << tab[i] << '\t';
    cout << endl;
}
```

Ein Feld kann mit jedem Datentyp gebildet werden, ausgenommen sind nur spezielle Datentypen wie z. B. `void`.

Initialisierung von Feldelementen

Feldelemente haben nach ihrer Definition normalerweise keine definierten Werte. Man muss also explizit initialisieren:

```
const int MAXANZAHL = 10;
int tab[MAXANZAHL];

for (int i = 0; i < MAXANZAHL; i++)
    tab[i] = 0;
```

Statische Initialisierung

Felder können auch bei ihrer Definition initialisiert werden. Zur Initialisierung wird eine Liste mit den Werten der einzelnen Feldelemente angegeben:

```
double vec [3] = { 1.5, 2.5, 3.5 };
```

vec[0]	1.5
vec[1]	2.5
vec[2]	3.5

Wenn ein Feld bei der Definition initialisiert wird, so muss seine Länge nicht angegeben werden.

```
double vec [] = { 1.5, 2.5, 3.5 };  // automatisch Laenge 3
```

<u>Regel</u>: Wenn die Größe des Feldes angegeben ist, dann sollte die Anzahl der Initialisierungswerte gleich dieser Größe sein.

<u>Lokal definierte Felder</u>

Werden Felder innerhalb von Funktionen definiert, so werden sie erst beim Aufruf der Funktion auf dem Stack erzeugt und beim Verlassen der Funktion auch wieder entfernt.

<u>Zuweisung von Feldern</u>

Es ist nicht möglich ein Feld einem anderen direkt zuzuweisen

```
int tab1[10], tab2[10];
// irgendwie gefüllt

tab1 = tab2;   // FALSCH!!
```

Statt dessen muss die Zuweisung einzeln erfolgen:

```
for (int i = 0; i < 10; i++)
    tab1[i] = tab2[i];
```

8.2.2 Felder als Funktionsargumente

Bei der Übergabe eines Feldes an eine Funktion sind einige Dinge zu beachten:

```
void initFeld (int tab[], int groesse) {
    for (int i = 0; i < groesse; i++)
        tab[i] = rand() % 100;
}
```

- Es erfolgt kein Wertaufruf, statt dessen wird der Funktion die Anfangsadresse des Feldes übergeben
- In der Funktion wird auf den Originalspeicherplätzen des Feldes operiert.
- In der Funktion ist die Feldgroesse nicht bekannt. Sie muss deshalb explizit übergeben werden. Dadurch ist eine Funktion natürlich auch für Felder beliebiger Größe einsetzbar.
- Folgendes ist möglich:

```
void initFeld (int tab[100]) {
    for (int i = 0; i < 100; i++)
        tab[i] = rand() % 100;
}
```

Was passiert dann wohl, wenn man die Funktion so aufruft?

```
int tab[50];
initFeld(tab);
```

Wie erwartet wird dies vom Compiler nicht bemerkt und führt zur Laufzeit direkt oder später zum Absturz. Die Angabe der Arraygröße wird also ignoriert!

- Ein Feld kann nicht direkt der Rückgabewert einer Funktion sein.

Beispiel:

```
// feldtest1.cpp: Eingabe eines Feldes

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>     // Bibliothek fuer Zeitfunktionen

/* Initialisieren des Zufalllsgenerators mit der aktuellen Uhrzeit */
void initZufall () {
    long sek;
    time(&sek);            // in sek steht nun die Uhrzeit in
    srand(sek);            // Sekunden seit 1.1.1970 0.00 Uhr
}

/* Initialisiere ein Feld mit Zufallszahlen zwischen 0 und 100 */
void initFeld (int tab[], const int groesse) {
    for (int i = 0; i < groesse; i++)
        tab[i] = rand() % 100;
}

/* Bestimme den minimalen Wert in einem Feld */
int min (const int tab[], const int groesse) {
    int minWert = tab[0];
    for (int i = 0; i < groesse; i++)
        if (tab[i] < minWert)
            minWert = tab[i];
    return minWert;
}

/* Bestimme den Durchschnittswert der Feldelemente als double-Wert */
double mittelwert (const int tab[], const int groesse) {
    double summe = 0.0;
    for (int i = 0; i < groesse; i++)
        summe += tab[i];
    return summe / groesse;
}

/* Gebe ein Feld auf die Standardausgabe aus */
void ausgabeFeld(const int tab[], const int groesse) {
    for (int i = 0; i < groesse; i++) {
        cout << setw(4) << tab[i];
        if ((i+1) % 10 == 0)
            cout << endl;
    }
    cout << endl;
}

int main() {
    const int TAB1_GROESSE=10;
    const int TAB2_GROESSE=20;
    int tab1[TAB1_GROESSE],
        tab2[TAB2_GROESSE];

    initZufall();
    initFeld(tab1, TAB1_GROESSE);
    initFeld(tab2, TAB2_GROESSE);
    cout << "tab1:\n";
    ausgabeFeld(tab1, TAB1_GROESSE);
    cout << "tab2:\n";
    ausgabeFeld(tab2, TAB2_GROESSE);
```

```
    cout << "Minimum tab1: "
         << min(tab1, TAB1_GROESSE) << endl;
    cout << "Minimum tab2: "
         << min(tab2, TAB2_GROESSE) << endl;

    cout << "Mittelwert tab1: "
         << mittelwert(tab1, TAB1_GROESSE) << endl;
    cout << "Mittelwert tab2: "
         << mittelwert(tab2, TAB2_GROESSE) << endl;
}
/*
tab1:
  29  66  55  49  31  47  22  54  97  50

tab2:
  38  65  78  98  24  76  52  21  25  56
  38  25  74  32  32  15  93  11  93  78

Minimum tab1: 22
Minimum tab2: 11
Mittelwert tab1: 50
Mittelwert tab2: 51.2
*/
```

8.2.3 Mehrdimensionale Felder

Mehrdimensionale Felder werden einfach als Felder von Feldern definiert.

```
double matrix [3][5];      // 3 x 5 - Matrix
                           // 3 Felder mit je 5 double-Werten
```

	0	1	2	3	4
0					
1					
2					

Zugriff auf ein Element:

```
matrix [0][2] = 0.0;       // 2. Element in der 0-ten Zeile
```

Allgemein:

```
Typangabe Feldname [ N ] [ M ];
```

Dadurch wird ein Feld mit N*M Elementen definiert.

```
Feldname [ i ] [ j ];      // i = 0, ..., N-1
                           // j = 0, ..., M-1
```

Beispiel: 3-dimensionales Feld:

```
int matrix3D [3][4][5]; // 3 * 4 * 5 Elemente
```

Intern stehen natürlich alle Feldelemente hintereinander im Hauptspeicher

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 5; j++)
        matrix[i][j] = 10*i + j;
```

Intern:

00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

Ausgabe eines 2-dimensionalen Feldes:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 5; j++)
        cout << matrix[i][j] << '\t';
    cout << endl;
}
```

Initialisierung von mehrdimensionalen Feldern

```
int mat[2][3] = { { 1, 2, 3 },
                  { 4, 5, 6 } };
```

Man kann auch die Feldgrenze für den ersten Index weglassen:

```
int mat1[][3] = { { 1, 2, 3}, { 4, 5, 6 } };
int mat2[][2][3] = { {{1,2,3},{4,5,6}}, {{7,8,9},{10,11,12}} } ;
```

Mehrdimensionale Felder als Parameter von Funktionen

Die Übergabe von mehrdimensionalen Feldern an Funktionen ist problematisch.

```
void matrixAusgeben (int mat[][], int groesse1, int groesse2); //FALSCH!
```

Dies ist zwar wünschenswert, aber leider **falsch!** Statt dessen ist es nur möglich, die erste Dimension offen zu lassen und als Parameter mitzugeben:

```
void matrixAusgeben (int mat[][5], int groesse1);
```

Alternativ ist natürlich auch das möglich:

```
void matrixAusgeben (int mat[3][5]);
```

Dann muss die Funktionen beide Größen selbst kennen und die Flexibilität ist vollständig dahin.

Für die flexibel parametrierbare Übergabe von Feldern an Funktionen werden wir später allerdings viel bessere Lösungen kennenlernen.

8.3 Zeichenketten

Bei C/C++ haben `char`-Felder eine besondere Bedeutung. Sie waren ursprünglich der Ersatz für einen eigenen String-Datentyp, der normalerweise zu einer höheren Programmiersprache gehört.

8.3.1 Einführung

Beispiel:

```
char hallo[] = "hallo";    // intern: h a l l o \0
 // äquivalent zu
char hallo[] = { 'h', 'a', 'l', 'l', 'o', '\0'};
```

Hierdurch wird ein `char`-Feld der Größe 6 (!) angelegt, denn unsichtbar am Ende der Zeichenkette steht das Zeichen `\0`.

8.3.2 Ein- und Ausgabe von Zeichenketten

Die Ausgabe einer Zeichenkette erfolgt wie erwartet

```
char s[] = "Hallo Welt!";
cout << s;     // wird wie erwartet ausgegeben
```

Was passiert aber, wenn man folgendes macht?

```
s[5]='\0';     // intern: H a l l o \0 W e l t ! \0

cout << s;     // Ausgabe: Hallo
```

Es wird also generell nur bis zum `\0` ausgegeben. Dies kann andererseits aber wiederum gefährlich sein, wenn man kein `\0` in seiner Zeichenkette hat.

Die Eingabe von Zeichenketten mit Hilfe des Operators `>>` überliest generell führende Leerzeichen. Außerdem wird nur bis zum nächsten Leerzeichen gelesen, d. h. es wird wortweise gelesen.

```
char s[50];
cin >> s;
```

Eingabestrom: `\t Hallo Welt .....`
Gelesen wird : `Hallo`

Eine mögliche sinnvolle Anwendung dieses Verhaltens ist das wortweise Einlesen eines Eingabestroms:

```
char wort[80];
while (cin >> wort) // Schleife bis EOF
    cout << wort << endl;
```

Sollen jedoch vollständige Zeilen eingelesen werden, so empfiehlt sich der Einsatz der Elementfunktion `getline` von `istream`.

```
char puffer[256];
cin.getline(puffer, 256);
```

Wirkung: Eine Zeile (incl. `'\n'`) wird eingelesen, allerdings maximal 255 Zeichen. Das Linefeed-Zeichen wird allerdings nicht übertragen.

```
cin.getline(puffer, 80, '@');
```

Wirkung: Alle Zeichen bis einschließlich zum angegebenen Trennzeichen werden eingelesen, allerdings maximal 79 Zeichen. Das Trennzeichen wird nicht in `puffer` übertragen.

Beispiel: Zeilenweises Lesen eines Eingabestroms

```
char puffer[256];
while (cin.getline(puffer,256)) // lese bis EOF
    cout << puffer << '\n';
```

8.3.3 Zeichenketten-Funktionen

Die Standardbibliothek beinhaltet eine Bibliothek mit Zeichenketten-Funktionen zum Vergleichen, Kopieren, Verketten und Manipulieren von char-Strings. Die Funktionen sind in der Headerddatei `<cstring>` deklariert.

Die im Folgenden verwendete Schreibweise `char*` ist zumindest in diesem Zusammenhang als gleichbedeutend zur Schreibweise `char[]` zu betrachten und bedeutet `char`-Zeiger.

Länge einer Zeichenkette

```
int strlen ( char* s );
```

- liefert die Länge des `char`-Strings ohne abschliessendes `'\0'`

```
char text[] = "123456";
cout << strlen(text) << endl;    // 6
text[3] = '\0';      // intern: 1 2 3 \0 5 6
cout << strlen(text) << endl;    // 3
```

Kopieren einer Zeichenkette

```
char* strcpy ( char* s1, char* s2 );
```

- kopiert `s2` nach `s1`, inklusive `'\0'`, liefert als Ergebnis `s1`.
- `s1` muss groß genug sein, sonst Speicherplatzverletzung (segmentation error)

```
char meier[] = "Sepp Meier";
char name[30];
strcpy(name, meier);
cout << name << endl;      // Sepp Meier
```

Anhängen einer Zeichenkette (concatenation)

```
char* strcat ( char* s1, char* s2 );
```

- hängt `s2` an `s1` an, liefert als Ergebnis `s1`.
- `s1` muss groß genug sein, sonst Speicherplatzverletzung

```
char hallo[] = "Hallo";
strcat(hallo, " Welt!");
cout << hallo << endl;     // Hallo Welt
```

Vergleich zweier Zeichenketten

```
int strcmp ( char* s1, char* s2 );
```

- vergleicht `s1` und `s2`, liefert:

< 0	falls	`s1 < s2`	(lexikalisch kleiner)
$= 0$	falls	`s1 == s2`	(genau identisch, bis zum '\0')
> 0	falls	`s1 > s2`	(lexikalisch größer)

Beispiel

```
// vergleiche1.cpp: Testen von verschiedenen Zeichenkettenfunktionen

#include <iostream>
#include <cstring>

int main() {
   char text1[20];
   char text2[20];

   do {
       cout << "text1: ";
       cin.getline (text1,20);
       cout << "text2: ";
```

```
        cin.getline (text2,20);

        if (strcmp(text1, text2) == 0)
            cout << "Die Zeichenketten sind gleich!\n";
        else
        if (strcmp(text1, text2) < 0)
            cout << "text1 < text2\n";
        else
            cout << "text1 > text2\n";
    } while (strcmp(text1, "Ende") != 0);
}
```

Eine Variante dieses Beispiels zeigt das folgende Programm:

```
// vergleiche2.cpp: Testen von verschiedenen Zeichenkettenfunktionen

#include <iostream>
#include <cstring>

inline bool istGleich (char s1[], char s2[]) {
    return (strcmp(s1, s2) == 0);
}

inline bool istKleiner (char s1[], char s2[]) {
    return (strcmp(s1, s2) < 0);
}

int main() {
   char text1[20];
   char text2[20];

   do {
        cout << "text1: ";
        cin.getline (text1,20);
        cout << "text2: ";
        cin.getline (text2,20);

        if (istGleich(text1, text2))
            cout << "Die Zeichenketten sind gleich!\n";
        else
        if (istKleiner(text1, text2))
            cout << "text1 < text2\n";
        else
            cout << "text1 > text2\n";
   } while (!istGleich(text1,"Ende"));
}
```

Wie definiert man eigentlich Felder von Zeichenketten?

Ein 5-elementiges Feld von Zeichenketten der Länge 20 definiert man folgendermaßen:

```
char NamensTabelle[5][20];
```

Die i-te Zeichenkette wird dann durch `NamensTabelle[i]` angesprochen. Das Einlesen könnte so funktionieren:

```
for (int i = 0; i < 5; i++) {
        cout << "Name[" << i << "] = ";
        cin >> namensTabelle[i];
    }
```

8.4 Strukturen

8.4.1 Einführung

Eine Struktur in C/C++ ist eine Zusammenfassung mehrerer Datenkomponenten eventuell verschiedenen Typs, die in einem logischen Zusammenhang stehen.

Definition eines Strukturtyps:

```
struct TypName {
   typ1 name1;
   typ2 name2;
   ...
};
```

Durch eine solche Definition wird ein neuer Datentyp definiert, von dem natürlich auch Variablen definiert werden können.

Beispiel:

```
// Punkt im 2-dimensionalen Raum
struct Punkt {
   double x;
   double y;
};

// Definition von Variablen:
Punkt p1, p2;
```

Zugriff auf Strukturkomponenten

Auf einzelne Komponenten einer Struktur kann mit dem Punkt-Operator zugegriffen werden:

```
p1.x = 0.0;    // p1.x ist vom Typ double
p1.y = 0.0;    // p1.y ist vom Typ double
```

Zuweisung und Initialisierung von Strukturen

```
Punkt p1;
p1.x = 1.0;
p1.y = 2.0;

Punkt p2 = p1; // p2 wird mit den Werten aus p1 initialisiert
Punkt p3;
p3 = p1;       // Zuweisung von p1 an p3

Punkt p4 = { 1.5, 2.5 };   // statische Initialisierung
```

Bemerkung: Strukturen können nicht direkt miteinander verglichen werden.

```
if (p1 == p2)        // Leider falsch !!
```

Statt dessen:

```
if (p1.x == p2.x && p1.y == p2.y)      // richtig!
```

Verschachtelte Strukturen

Strukturen können andere Strukturtypen beinhalten:

```
// Strecke im 2-dimensionalen Raum
```

```
struct Strecke {
   Punkt p1;
   Punkt p2;
};

Strecke strecke1, strecke2;
```

Zugriffe auf verschachtelte Strukturen:

Komponente	Typ
strecke1	Strecke
strecke1.p1	Punkt
strecke1.p1.x	double
strecke1.p1.y	double
strecke1.p2	Punkt
strecke1.p2.x	double
strecke1.p2.y	double

Felder in Strukturen

Felder können auch Komponenten von Strukturen sein

```
struct Student {
   int matrikelNr;
   char vorname[20];
   char nachname[30];
};

Student mueller = { 4711, "Fritz", "Müller" };
Student meier;
meier.matrikelNr = 1234567;
meier.vorname = "Hans";          // leider falsch!!
```

Besser:

```
strcpy(meier.vorname , "Hans");
strcpy(meier.nachname, "Meier");
```

Zuweisen von Strukturen mit Feldern:

```
Student neuMeier;
neuMeier = meier;    // es wird komponentenweise übertragen !
```

8.4.2 Strukturen und Funktionen

Strukturen können
- Argumente von Funktionen sein
- Ergebniswert von Funktionen sein
- per Referenz an Funktionen übergeben werden

```
// punkttest.cpp: Testen der Datenstruktur Punkt

#include <iostream>

// Punkt im 2-dimensionalen Raum
struct Punkt {
   double x;
   double y;
};
```

```
void init (Punkt& p) {
    p.x = 0.0;
    p.y = 0.0;
}

bool istGleich(Punkt p1, Punkt p2) {
    return (p1.x == p2.x && p1.y == p2.y);
}

void read (Punkt& p) {
    cout << "x = "; cin >> p.x;
    cout << "y = "; cin >> p.y;
}

void print (Punkt& p) {
    cout << "( " << p.x << ", " << p.y << " )\n";
}

Punkt addiere (Punkt p1, Punkt p2) {
    Punkt tmp;
    tmp.x = p1.x + p2.x;
    tmp.y = p1.y + p2.y;
    return tmp;
}

int main() {
    Punkt p1, p2;
    init(p1);
    init(p2);
    read(p1);
    read(p2);
    if (istGleich(p1, p2))
       cout << "p1 und p2 sind gleich!\n";
    else
       cout << "p1 und p2 sind ungleich!\n";
    print (p1);
    print (p2);

}
/* Beispielausgabe
x = 1.0
y = 2.0
x = 1.5
y = 2.5
p1 und p2 sind ungleich!
( 1, 2 )
( 1.5, 2.5 )  */
```

Mit Strukturen ist es möglich, die Übergabe von Feldern an Funktionen zu verbessern wie das folgende Beispiel zeigt:

```
// structtest1.cpp: Ein Feld als struct

#include <iostream>
const int maxSize = 100;

struct intArray {
    int size;
    int tab [maxSize];
};

void read (intArray& a) {
   for (int i = 0; i < a.size; i++) {
      cout << "[ " << i << " ] = ";
```

```
        cin >> a.tab[i];
    }
}

void print (intArray a) {
    for (int i = 0; i < a.size; i++)
        cout << a.tab[i] << " ";
    cout << endl;
}

int main() {
     intArray a;
     a.size = 10;
     read(a);
     print(a);
     intArray b = a;  // initialisiere b mit a
     print (b);

}
/* moegliche Ausgabe
[ 0 ] = 1
[ 1 ] = 2
[ 2 ] = 3
[ 3 ] = 4
[ 4 ] = 5
[ 5 ] = 6
[ 6 ] = 7
[ 7 ] = 8
[ 8 ] = 9
[ 9 ] = 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
*/
```

Bei der Funktion `print` wird tatsächlich eine Kopie des Feldes innerhalb der Kopie der Struktur angelegt. Bei sehr großen Feldern in einer Struktur kann dieses Kopieren bei der Übergabe an eine Funktion von Nachteil sein.

```
struct Riesig {
     int tab[1000000];
     ....
};

void print (Riesig r)
{ ... }
```

Wie kann man verhindern, dass nicht kopiert wird, aber die Funktion trotzdem nicht die übergebene Struktur verändern kann?

```
void print (const Riesig& r) // Übergabe als const-Referenz !
{ ... }
```

`r` wird als Original in `print` verarbeitet. Durch den Zusatz `const` wird aber verhindert, dass die Funktion `r` verändern kann.

8.4.3 Arrays von Strukturen

Wie alle C++-Datentypen können natürlich auch Strukturtypen als Bestandteil von Feldern vorkommen.

```
struct Student {
   int matrikelNr;
   char vorname[20];
   char nachname[30];
};

Student teilnehmer[100];
```

Zugriff auf einzelnen Komponenten:

```
teilnehmer[0].matrikelNr = 123456;
strcpy(teilnehmer[0].vorname, "Hans");
strcpy(teilnehmer[0].nachname, "Meier");
```

8.5 Der sizeof-Operator

Manchmal ist es in Programmen notwendig herauszufinden, wie lang Datenelemente eines bestimmten Typs sind. Da dies von Plattform zu Plattform unterschiedlich ist, hat man dafür einen speziellen Operator eingeführt, den `sizeof`-Operator.

Syntax:

```
sizeof (Typ)   oder
sizeof (Ausdruck)
```

Der `sizeof`-Operator liefert die Länge des Datentyps in Bytes zurück. Er kann auch auf Felder angewandt werden.

Beispiel:

```
// sizeoftest1.cpp: Test des Operators sizeof
struct Student {
   int matrikelNr;
   char vorname[20];
   char nachname[30];
};

struct Punkt {
   double x;
   double y;
};

const int maxSize = 100;
struct IntArray {
    int size;
    int tab [maxSize];
};

int main() {
   cout << "sizeof(Student)  = " << sizeof(Student)  << endl;
   cout << "sizeof(Punkt)    = " << sizeof(Punkt)    << endl;
   cout << "sizeof(IntArray) = " << sizeof(IntArray) << endl;
   cout << "sizeof(int)      = " << sizeof(int)      << endl;
   cout << "sizeof(double)   = " << sizeof(double)   << endl;

}/* Ausgabe
sizeof(Student)  = 54
sizeof(Punkt)    = 16
sizeof(IntArray) = 404
sizeof(int)      = 4
sizeof(double)   = 8  */
```

9 Programmstrukturierung

Eine der elementarsten Möglichkeiten, Programmentwicklungszeiten zu reduzieren, ist das Wiederverwenden von bereits entwickelten Programmteilen. Häufig sieht dies auch heute noch in der Praxis so aus, dass Quelltext aus vorhandenen Programmen in andere Programme kopiert und dort modifiziert wird. Dies führt zur Vervielfachung von Programmcode und zu erheblichen Erschwernissen bei der Pflege und Wartung von Programmsystemen. Besser ist es, Programme so zu zerlegen, dass wiederverwendbare Teile, wie z. B. Datentypen und Funktionen in eigene Dateien ausgelagert und zu eigenen Bindemodulen übersetzt werden. Das Wiederverwenden besteht dann im wesentlichen nur noch darin, dass diese Bindemodule zu einem sie verwendenden Programm dazugebunden werden müssen.

9.1 Zerlegung von Programmen in mehrere Quelldateien

Wir werden das Zerlegen an einem konkreten Beispiel durchführen, der Implementierung eines Stack. Wir werden dazu einen Datentyp `DoubleStack` so entwickeln, dass er in verschiedenen Anwendungsprogrammen eingebunden werden kann

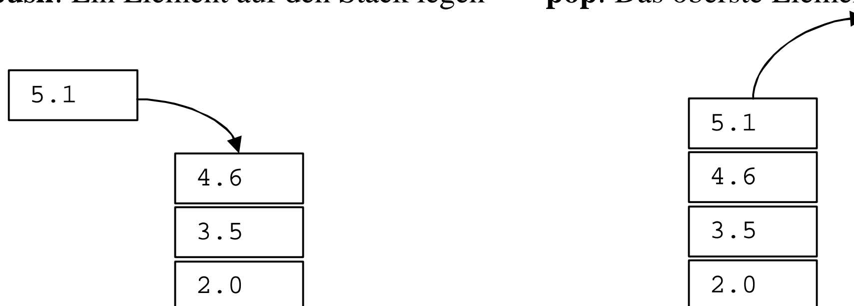
9.1.1 Stack (Stapel)

Stacks sind Datenstrukturen, die Werte aufnehmen können mit der Möglichkeit, Elemente hinzuzufügen oder wegzunehmen. Dabei kann aber immer nur das zuletzt hinzugefügte Element wieder weggenommen werden.

LIFO-Prinzip (*Last in first out*)

Standardfunktionen eines Stack:

push: Ein Element auf den Stack legen **pop**: Das oberste Element vom Stack wegnehmen



Implementierung eines DoubleStack

- mit Hilfe eines double-Feldes und
- eines Stackpointers, d. h. einer Indexvariable, die auf das zuletzt hinzugefügte Element zeigt.

Es gibt auch noch andere, flexiblere Möglichkeiten, einen Stack zu implementieren und daher soll die Anwendung unserer Implementierung so erfolgen, dass nirgendwo die interne Struktur erkennbar ist.

Die Datenstruktur

```
const int MAX_ANZ_ELEMENTE = 100;

struct DoubleStack {
   int anzElemente;               // Anzahl Elemente des Stacks
   double tab[MAX_ANZ_ELEMENTE]; // Implementierungs-Array
};
```

Die Implementierung

```
// Element hinzufuegen
void push (DoubleStack& s, double x) {
   if (s.anzElemente < MAX_ANZ_ELEMENTE)
        s.tab[s.anzElemente++] = x;
}

// oberstes Element wegnehmen
void pop(DoubleStack& s) {
   if (s.anzElemente > 0)
        s.anzElemente--;
}
```

Zusätzlich benötigte Funktionen:

```
// Inhalt des obersten Elementes erfragen
double top(const DoubleStack& s) {
   if (s.anzElemente > 0)
        return s.tab[s.anzElemente - 1];
   else {
        cerr << "Stack leer\n";
        return 0.0;
   }
}

// Ist der Stack leer?
bool empty (DoubleStack& s) {
    return (s.anzElemente == 0);
}

// Ist der Stack voll?
bool full  (DoubleStack& s) {
    return (MAX_ANZ_ELEMENTE <= s.anzElemente);
}

// Wieviel Elemente stehen im Stack?
int size  (const DoubleStack& s) {
    return s.anzElemente;
}

// Stack-Inhalt ausgeben
void print(const DoubleStack& s) {
   for (int i = s.anzElemente - 1; i >= 0; --i)
      cout << s.tab[i] << "   ";
   cout << endl;
}
```

Wir wollen nun diesen neuen Datentyp `DoubleStack` zusammen mit seinen Funktionen in zwei verschiedenen Programmen verwenden, ohne jedoch den Quelltext jedesmal noch einmal neu zu übersetzen.

Gliederung der Quellen

Die Quelldatei wird dazu in eine Header-Datei und eine Implementierungsdatei aufgesplittet.

Quelldatei	***Inhalt***
`doublestack.h`	• Die Definition der Datenstruktur `DoubleStack` • Prototypen für alle obigen Funktionen • Keine Implementierungen
`doublestack.cpp`	• `#include "doublestack.h"` • die Implementierungen sämtlicher in der Header-Datei deklarierten Funktionen

Übersetzung des Datentyps DoubleStack und seiner zugehörigen Funktionen:

```
gcc -c doublestack.cpp
```

Bei fehlerfreier Übersetzung wird eine Ausgabedatei `doublestack.o` erzeugt, die zu allen verwendenden Programmen hinzugebunden werden kann. Dazu muss allerdings auch noch die Header-Datei `doublestack.h` eingebunden werden

Beispiel: Testprogramm für `DoubleStack`

```
// DoubleStackt.cpp: Testprogramm fuer double-Stack

#include "doublestack.h"

enum FunktionsTyp { beenden, PUSH, POP, TOP, EMPTY, FULL, SIZE};

void menue() {
   cout << PUSH   << ": push / "
        << POP    << ": pop / "
        << TOP    << ": top / "
        << EMPTY  << ": empty / "
        << FULL   << ": full / "
        << SIZE   << ": size / "
        << beenden<< ": beenden ->";
}

int main() {
   DoubleStack s1;
   init(s1);
   int funktion;
   double x;
   do {
      cout << "s1 :"; print(s1);
      menue(); cin >> funktion;
      switch (funktion) {
      case PUSH  :  cout << "Wert: "; cin >> x;
                    push(s1, x);
                    break;
      case POP   :  cout << "pop: " << top(s1) << endl;
                    pop(s1);
                    break;
      case TOP   :  cout << "top: " << top(s1) << endl;
                    break;
      case EMPTY :  if (empty(s1))
                       cout << "Stack leer\n";
```

```
                    else
                       cout << "Stack nicht leer\n";
                    break;
      case FULL   :  if (full(s1))
                           cout << "Stack voll\n";
                    else
                       cout << "Stack nicht voll\n";
                    break;
      case SIZE   : cout << "size: " << size(s1) << endl;
                    break;
      case beenden: break;
      default     : cout << "Falsche Funktion!\n";
      }
   } while (funktion != beenden);
}
```

Übersetzung des Testprogramms:

```
gcc -o doublestackt  doublestackt.cpp  doublestack.o
```

Bei fehlerfreier Übersetzung von `doublestackt.cpp` wird zunächst vom Compiler das Bindemodul `doublestackt.o` erzeugt und schließlich der Linker gestartet, der das ausführbare Programm `doublestackt` zusammenbinden soll.

Um nun ein zweites Testprogramm für den Datentyp `DoubleStack` zu entwickeln, ist es nur noch nötig,

- das Testprogramm zu erstellen,
- die Headerdatei `doublestack.h` einzubinden
- das Testprogramm zu übersetzen und
- mit dem bereits übersetzten Bindemodul `doublestack.o` zusammenzubinden.

Beispiel: 2. Testprogramm für `DoubleStack`

```
// DoubleStackt2.cpp: 2. Testprogramm fuer double-Stack

#include "doublestack.h"

int main() {
   DoubleStack s1, s2;
   init(s1);
   push(s1, 3.14);
   push(s1, 2.718);
   s2 = s1;
   pop(s1);
   print(s1);
   print(s2);
}
/* Ausgabe
3.14
2.718   3.14   */
```

Was kann alles in Header-Dateien stehen?

- Datentypdefinitionen z. B. mit `enum` und `struct`
- Makros (möglichst vermeiden)
- Funktionsprototypen

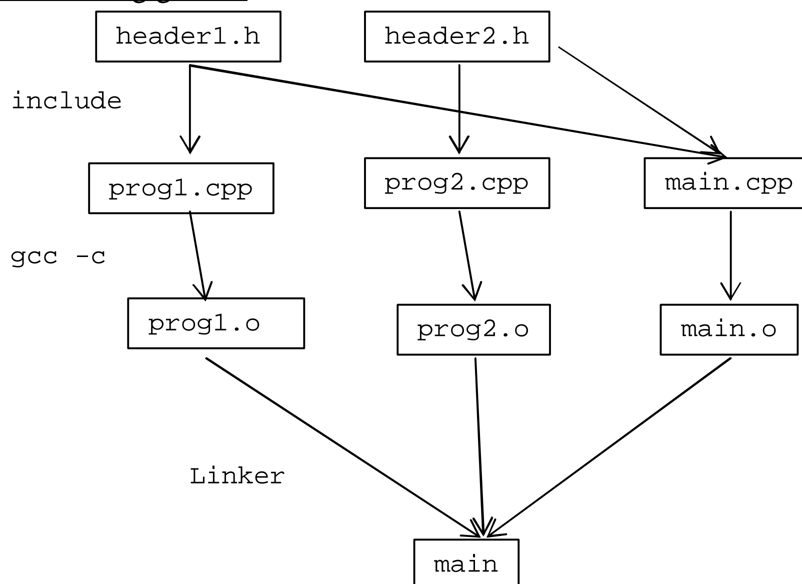
- inline-Funktionen

Was gehört in die Implementierungsdatei?

- Implementierung der Funktionen
- Datentypdefnitionen, die nur zur Implementierung benötigt werden

Ein Programm kann also nunmehr von einer Menge von verschiedenen anderen Programmen abhängig sein. Jede Änderung in einer verwendeten Header-Datei kann dazu führen, dass das Programm neu übersetzt und gebunden werden muss.

Beispiel: Abhängigkeiten



Wird `header1.h` geändert, so müssen `prog1.cpp` und `main.cpp` neu übersetzt werden. Wird `header2.h` geändert, so müssen `prog2.cpp` und `main.cpp` neu übersetzt werden. Wird nur in `prog1.cpp` etwas geändert, so ist nur `prog1.cpp` neu zu übersetzen. in allen Fällen muss das Programm `main` jeweils neu gebunden werden. Natürlich kann man auch immer alles neu übersetzen. Dies führt aber bei sehr großen Programmsystemen zu unnötig hohen Übersetzungs- und Bindezeiten. Um immer nur das ausführen zu müssen, was notwendig ist, kann man die Abhängigkeiten zwischen den verschiedenen Dateien in einem `makefile` beschreiben und die Ausführung dem Werkzeug `make` überlassen. Wie dies funktioniert soll aber ausführlich im Rahmen des Praktikums besprochen werden.

Beispiel für ein makefile:

```
# Makefile fuer das Programm doublestackt

# Makrodefinitionen
LOESCHLISTE = *.o

# Definition der Abhaengigkeiten
#
doublestackt: doublestackt.o doublestack.o
   ${CC} ${CFLAGS} -o  $@ doublestackt.o doublestack.o

doublestackt.o: doublestackt.cpp doublestack.h
   ${CC} ${CFLAGS} -c doublestackt.cpp

doublestack.o: doublestack.cpp doublestack.h
```

```
   ${CC} ${CFLAGS} -c doublestack.cpp

# Bereinigung des Verzeichnisses
loeschen:
   @echo "Folgende Dateien werden geloescht"
   @echo $(LOESCHLISTE)
   rm -f $(LOESCHLISTE)
```

9.2 Der C++-Präprozessor und bedingte Übersetzung

Wir haben bereits besprochen, dass die Übersetzung eines C++-Programms in zwei Stufen abläuft,

- dem Präprozessorlauf
- und der eigentlichen Übersetzung.

Der Präprozessor hat dabei die Aufgabe, vor der eigentlichen Übersetzung noch Textersetzungen im Quelltext vorzunehmen.

Präprozessor-Direktiven

- Schlüsselworte, die der Präprozessor verarbeitet, beginnen alle mit dem Zeichen #.
- Diese Schlüsselworte nennt man auch Präprozessor-Direktiven.
- Jede Präprozessor-Direktive muss in einer eigenen Zeile stehen.

9.2.1 Die wichtigsten Präprozessor-Direktiven

#include

Das Inkludieren von Header-Dateien haben wir bereits kennengelernt. Es gibt dabei zwei Varianten.

`#include <iostream>`	Der Präprozessor sucht die Header-Datei in einem der vordefinierten Include-Verzeichnisse (z. B. `/usr/include`).
`#include "punkt.h"`	Der Präprozessor sucht die Header-Datei in dem aktuellen Arbeitsverzeichnis.

Es ist auch möglich, dem Präprozessor bei der Übersetzung zusätzliche Include-Verzeichnisse mitzugeben.

```
gcc -I $HOME/include prog.cpp -o prog
```

Hier wird vom Präprozessor bei der ersten `include`-Variante auch noch in dem angegebenen Verzeichnis gesucht.

#define

Mit Hilfe von `#define` können Makros als Ersatz für Konstanten definiert werden.

```
#define PI          3.1415926
```

Der Präprozessor ersetzt dann überall im Programmtext das Wort `PI` durch den Rest der Zeile. Diese Art der Verwendung von `#define` ist aber seit es echte Konstanten im Spachumfang von ANSI-C gibt unerwünscht und sollte nicht verwendet werden.

Mit Hilfe von `#define` können auch sogenannte Funktions-Makros definiert werden:

```
#define SQR(a)       ( (a) * (a) )
```

Der folgende Text im Programm

```
z = SQR (x + y);
```

wird durch den Präprozessor ersetzt und sieht dann für den Compiler so aus:

```
z = ((x + y) * (x + y));
```

Auch diese Art von Makros sollen nicht mehr verwendet werden, sind aber, wie auch die anderen Makros, noch sehr häufig in Quelltexten zu finden.

Ein Makro gilt nach seiner Definition als definiert, auch wenn sein Wert leer ist

```
#define DEBUG
```

Ein Makro gilt für den Präprozessor so lange als definiert bis die Definition mit der `#undef`-Direktive zurückgenommen wird.

```
#undef DEBUG
```

#ifdef, #ifndef

Mit Hilfe dieser beiden Präprozessor-Direktiven kann überprüft werden, ob ein bestimmtes Makro definiert oder nicht definiert ist.

Mögliche Anwendung:

```
#ifdef DEBUG
cout << "Irgendein Debugtext";
....
#endif
```

Wirkung: Wenn das Makro `DEBUG` definiert ist, unabhängig von seinem Wert, dann werden die Anweisungen zwischen `#ifdef` und `#endif` dem Compiler weitergegeben, ansonsten nicht. Diese Technik kann verwendet werden, um z.B. zusätzliche Test-Anweisungen in einem Programm unterzubringen und nur durch Setzen eines bestimmten Makros ein- bzw. auszuschalten.

Makro von außen setzen:

```
gcc -DDEBUG -o prog prog.cpp
```

Durch die Option `-D` beim Übersetzen kann explizit ein Makro gesetzt werden, ohne den Programmtext ändern zu müssen.

Variante:

```
#ifdef DEBUG
    #define OUT(a)   cout << a << endl;
#else
    #define OUT(a)
#endif

int main() {
    OUT("Start des Programms");
    ...
```

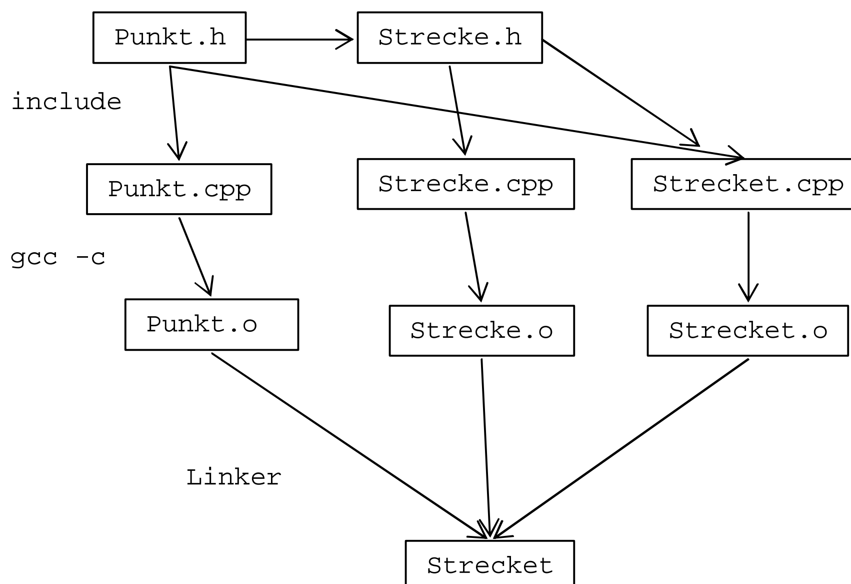
Das Funktionsmakro `OUT` wird entweder durch eine Ausgabe mit `cout` ersetzt oder durch eine leere Anweisung, je nachdem, ob `DEBUG` definiert ist oder nicht.

9.2.2 Anwendung der Programmstrukturierung

Im folgenden Beispiel wird die bedingte Übersetzung angewendet, um zu verhindern, dass Header-Dateien mehrfach in ein Programm inkludiert werden.

Entwickelt werden soll ein Testprogramm für die im letzten Abschnitt bereits betrachtete Datenstruktur `Strecke` und einige ihrer Funktionen. Natürlich werden zur Implementierung der Strecke-Funktionen auch die Datenstruktur `Punkt` sowie deren zugehörige Funktionen benötigt.

Übersicht über die Header- und Implementierungsdateien:



In `Strecket.cpp` wird normalerweise `Punkt.h` zweimal inkludiert. Das Ergebnis wäre, dass der Datentyp `Punkt` zweimal definiert würde, was vom Compliler aber nicht zugelassen werden kann.

Die Headerdatei `Punkt.h`

```
// punkt.h: Definition der Datenstruktur Punkt

#ifndef _PUNKT_H    ← die nächsten Zeilen nur übersetzen, wenn
                      das Makro _PUNKT_H nicht definiert ist
#define _PUNKT_H    ← Makro _PUNKT_H definieren

// Punkt im 2-dimensionalen Raum
struct Punkt {
   double x;
   double y;
};

void   init     (Punkt& p);
bool   istGleich(const Punkt& p1, const Punkt& p2);
void   read     (Punkt& p);
void   print    (const Punkt& p);
double abstand  (const Punkt& p1, const Punkt& p2);

#endif              ← Ende der bedingten Übersetzung
```

Die Implementierungsdatei `Punkt.cpp`

```
// punkt.cpp: Definition der Datenstruktur Punkt

#include "punkt.h"
#include <cmath>
#include <iostream>

void init (Punkt& p) {
    p.x = 0.0;
    p.y = 0.0;
}

bool istGleich(const Punkt& p1, const Punkt& p2) {
    return (p1.x == p2.x && p1.y == p2.y);
}

void read (Punkt& p) {
    cout << "x = "; cin >> p.x;
    cout << "y = "; cin >> p.y;
}

void print (const Punkt& p) {
    cout << "( " << p.x << ", " << p.y << " )\n";
}

double abstand (const Punkt& p1, const Punkt& p2) {
   return sqrt((p1.x - p2.x) * (p1.x - p2.x)
              +(p1.y - p2.y) * (p1.y - p2.y));
}
```

Das Bindemodul `punkt.o` kann nun erzeugt werden mit

```
gcc -c punkt.cpp
```

Die Headerdatei `Strecke.h`

```
// strecke.h: Definition der Datenstruktur Strecke

#ifndef _STRECKE_H
#define _STRECKE_H

#include "punkt.h"

// Strecke im 2-dimensionalen Raum
struct Strecke {
   Punkt p1;
   Punkt p2;
};

void init (Strecke& s);
void read (Strecke& s);
void print (const Strecke& s);
void verschiebe (Strecke& s, Punkt p);
double laenge(const Strecke& s);

#endif
```

Die Implementierungsdatei `Strecke.cpp`

```
// strecke.cpp: Implementierung der Strecke-Funktionen

#include "strecke.h"
#include <iostream>
```

```
void init (Strecke& s) {
    init (s.p1);
    init(s.p2);
}

void read (Strecke& s) {
    cout << "p1:\n"; read (s.p1);
    cout << "p2:\n"; read(s.p2);
}

void print (const Strecke& s) {
    cout << "p1: "; print (s.p1);
    cout << "p2: "; print (s.p2);
}

double laenge(const Strecke& s) {
   return abstand(s.p1, s.p2);
}
```

Das Bindemodul `strecke.o` kann nun erzeugt werden mit

```
gcc -c strecke.cpp
```

Das Testprogramm `Strecket.cpp`

```
// strecket.cpp: Testen der Datenstruktur Strecke

#include <iostream>
#include "punkt.h"       // Punkt-Definitionen einbinden
#include "strecke.h"     // Strecke-Definitionen einbinden
                         // die Punkt-Definitionen werden nicht mehr
                         // eingebunden
int main() {
    Strecke s;
    init(s);
    read(s);
    cout << "S:\n"; print (s);
    cout << "Laenge der Strecke: " << laenge(s) << endl;
}
/* Beispielausgabe
p1:
x = 1.5
y = 1.5
p2:
x = 2.5
y = 2.5
S:
p1: ( 1.5, 1.5 )
p2: ( 2.5, 2.5 )
Laenge der Strecke: 1.41421    */
```

Um nun das Testprogramm `strecket` zusammenzubinden, kann man zunächst das Bindemodul strecket.o erzeugen mit:

```
gcc -c strecket.cpp
```

Danach kann man das Projekt zusammenbinden mit Hilfe von

```
gcc -o strecket strecket.o strecke.o punkt.o
```

Natürlich ist es auch hier möglich, die ganzen Abhängigkeiten und Aktionen in einem Makefile festzulegen und von dem make-Tool organisieren zu lassen.

```
# Makefile fuer das Programm strecket

# Definition der Abhaengigkeiten
#
strecket: strecket.o strecke.o punkt.o
 ${CC} ${CFLAGS} -o  $@ strecket.o strecke.o punkt.o

strecket.o: strecket.cpp strecke.h punkt.h

strecke.o:  strecke.cpp strecke.h punkt.h

punkt.o:  punkt.cpp punkt.h
```

Die fehlenden Aktionen in diesem Makefile werden von make automatisch mit Hilfe eingebauter `make rules` ausgeführt. Im Klartext heißt das: make "weiß" wie aus einer .cpp-Datei eine .o-Datei zu erzeugen ist.

10 Dateiverarbeitung

10.1 Einführung

Bisher haben wir Daten immer nur in Variablen im Hauptspeicher gehalten. Es war, außer durch das Umlenken der Standardausgabe, nicht möglich, Daten dauerhaft auf der Festplatte zu speichern. Wir werden in diesem Abschnitt zunächst den elementaren Umgang mit Dateien kennenlernen.

Eine Datei ist aus der Sicht eines C++-Programms einfach eine beliebig lange Folge von Bytes.

Genauso wie Daten von der Standardeingabe gelesen werden können, können auch Daten aus Dateien gelesen werden, byteweise oder auch blockweise, formatiert oder auch unformatiert. Ebenso verhält es sich mit dem Schreiben in Dateien. Alles, was man beim Schreiben auf die Standardausgabe tun kann, kann man auch für Dateien tun. Dateien, die auf Festplatten gespeichert sind sind normalerweise blockorientiert gespeichert, d. h. das Schreiben und Lesen erfolgt in Blöcken der Größe von z. B. 512 oder 1024 Bytes. Die zu übertragenden Daten werden dabei üblicherweise in einem Puffer im Hauptspeicher zwischengespeichert.

Wie erwähnt ist eine Datei für ein C++-Programm zunächst nur eine Folge von Bytes. Jede Art von Strukturierung ist Aufgabe des Programmierers. Hier unterscheidet man zunächst grob zwei Arten von sequentiellen Dateien, d. h. Dateien, die nur sequentiell gelesen oder beschrieben werden können:

Zeilensequentielle Dateien oder Textdateien

Diese Dateiart entspricht den unter Unix üblichen Textdateien. Eine zeilensequentielle Datei ist unterteilt in beliebig lange Zeilen, die jeweils durch das Zeichen Linefeed (`\n`) beendet werden. Üblicherweise sind die darin enthaltenen Daten auch druckbare Zeichen des ISO oder UNICODES. Die Standardeingabe und die Standardausgabe sind so strukturiert. Der Zugriff erfolgt bei diesen Dateien im wesentlichen durch die Ein-/Ausgabefunktionen und Operatoren, die wir bereits kennengelernt haben.

Satzsequentielle Dateien

Eine satzsequentielle Datei ist üblicherweise untergliedert in einzelne Datensätze, normalerweise alle gleich lang und gleich strukturiert. Sie können auch nicht druckbare Zeichen enthalten. Die Datensätze werden dabei nicht durch bestimmte Trennzeichen beendet sondern alleine durch ihre Satzlänge. Datensätze werden üblicherweise durch Datenstrukturen wie `structs` beschrieben. Satzsequentielle Dateien sind vor allem in der klassischen DV noch sehr weit verbreitet.

Andere Formen der Dateiorganisation

In der klassischen DV noch sehr verbreitet und auch nach wie vor sehr praktisch sind indexsequentielle (ISAM-)Dateien. Hier kann über Schlüsselwerte direkt auf bestimmte Datensätze positioniert werden. ISAM-Dateien werden von C++ jedoch nicht direkt unterstützt.Andere Formen der Dateiorganisation sind z. B. Dateitypen wie HTML- oder XML-Dateien. Hier werden in normalen Textdateien die Gliederungen der Daten über sogenannte "Tags" vorgenommen.

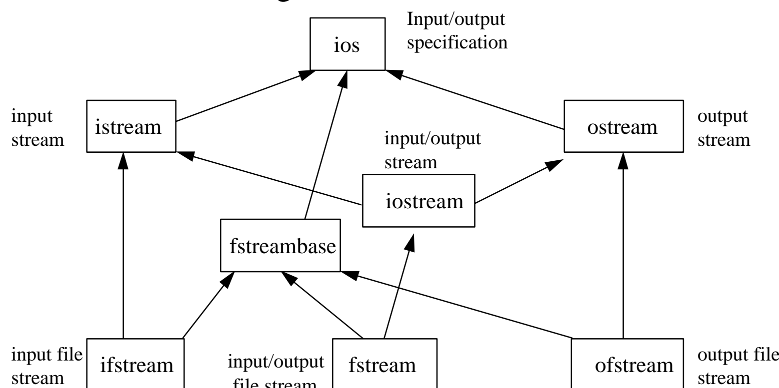
Dateipositionen

Jedes Zeichen in einer Datei hat eine Byte-Position. Das erste Byte hat die Position 0, das zweit die Position 1 usw. Die aktuelle Dateiposition ist die Position des Bytes, das als nächstes gelesen oder geschrieben wird. Mit jedem übertragenen Byte erhöht sich die aktuelle Dateiposition automatisch um 1.

Der Zugriff auf eine Datei erfolgt normalerweise sequentiell, es ist aber auch möglich den Dateizeiger auf bestimmte Positionen innerhalb der Datei zu positionieren (wahlfreier Zugriff).

10.2 Die File-Stream-Klassen

Übersicht über wichtige Klassen der Iostream-Bibliothek:



In der Iostream-Bibliothek werden verschiedene Standardklassen für die Dateiverarbeitung zur Verfügung gestellt. Diese Klassen ermöglichen die komfortable Handhabung von Dateien ohne dass der Entwickler sich um systemspezifische Details und Pufferung kümmern muss. Aus der Übersicht geht hervor, dass diese Klassen von den Standardklassen `istream` und `ostream` abgeleitet sind und daher deren komplette Funktionalität besitzen.

`ifstream`	Klasse zum Lesen aus Dateien
`ofstream`	Klasse zum Schreiben in Dateien
`fstream`	Klasse zum Lesen und Schreiben aus und in Dateien

Die File-Stream-Klassen sind in der Header-Datei `fstream` definiert.

Funktionalitäten

Alle schon von `cin` und `cout` her bekannnten Funktionalitäten:
- formatiertes Lesen und Schreiben mit >> bzw. <<
- zeichenweises bzw. zeilenweises Lesen und Schreiben
- Formatieren mit Manipulatoren und Elementfunktionen
- Statusabfragen

Zusätzliche Funktionalitäten:
- Methoden zum unformatierten Schreiben und Lesen einzelner Zeichen bzw. von Datenblöcken.
- Methoden für das Dateihandling, z. B. das Öffnen und Schließen von Dateien.

10.3 Öffnen und Schließen

Bevor eine Datei bearbeitet werden kann, muss sie geöffnet werden. Dazu ist der Dateiname, in Form einer Pfadangabe erforderlich und ggf. ein sogenannten Eröffnungsmodus.

Definition und Öffnen einer Ausgabedatei:

```
ofstream outfile;
outfile.open("ausgabe",ios::out);
                           ^ Open-Modus
```

Definition und Öffnen einer Eingabedatei:

```
ifstream infile;
infile.open("eingabe",ios::in);
```

Definition und Öffnen einer Ein-/Ausgabedatei:

```
fstream iofile;
iofile.open("datei",ios::in | ios::out);
```

Mögliche Eröffnungsmodi	
`ios::in`	Öffnen zum Lesen (Voreinstellung bei `ifstream`)
`ios::out`	Öffnen zum Schreiben (Voreinstellung bei `ofstream`)
`ios::ate`	Positioniere Schreib-/Leseposition am Dateiende (`at end`)
`ios::app`	Schreiben nur am Dateiende
`ios::trunc`	Dateiinhalt beim Öffnen löschen
`ios::binary`	Schreib- und Lese-Operationen im Binärmodus durchführen, d. h. alle Bytes werden so übertragen wie sie in der Datei stehen.

Schließen von Dateien:

```
infile.close();
outfile.close();
```

Warum sollte man Dateien schließen?

- Beim Schließen wird der Dateipuffer auf die Festplatte geschrieben. Bei einem unvermutetet Abbruch des Programms könnten daher Daten verloren gehen.
- Die Anzahl der Dateien, die ein Programm gleichzeitig offen haben darf ist durch das Betriebssystem begrenzt.
- Guter Stil.

Wird eine Datei einmal nicht vom Programmm selbst geschlossen, so wird das vom Betriebssystem beim Programmende automatisch getan, falls das Programm nicht irregulär endet.

Beispiel:

```
// datei1.cpp: Anwendung von File-Streams

#include <iostream>
#include <fstream>

int main() {
   const int groesse=30;
```

```
    ifstream infile;
    ofstream outfile;
    char eingabe[groesse], ausgabe[groesse];

    cout << "Eingabedatei: "; cin.width(groesse);
    cin >> eingabe;

    infile.open(eingabe);
    if (!infile) {
       cout << eingabe <<" kann nicht geoeffnet werden\n";
       return(1);
    }

    cout << "Ausgabedatei: "; cin.width(groesse);
    cin >> ausgabe;
    outfile.open(ausgabe);
    if (!outfile) {
       cout << ausgabe << " kann nicht geoeffnet werden.\n";
       return(1);
    }

    char c;
    while (infile.get(c)) { // Zeichenweises Lesen
       outfile.put(c);
    }
    infile.close();
    outfile.close();
    return 0;
}
```

Zeilenweise kopieren:

```
char puffer[256];
while (infile.getline(puffer, 256))
    outfile << puffer << '\n';
```

Eine Datei kann direkt beim Anlegen eines File-Streams geöffnet werden. Dabei braucht nur der Dateiname angegeben werden. Für den Eröffnungsmodus werden die voreingestellten Werte verwendet:

```
ifstream infile ("eingabe.txt");  // falls vorhanden zum Lesen öffnen

ofstream outfile("ausgabe.txt");  // zum Schreiben öffnen, ggf. anlegen
                                  // bzw. überschreiben
fstream iofile("einausgabe.txt");  // zum Schreiben und Lesen öffnen
```

Eröffnungsmodi

Der Eröffnungsmodus kann eine Kombination aus verschiedenen Modi sein.

`ios::in | ios::out`	Zum Lesen und Schreiben öffnen; nur bei `fstream` und in Zusammenhang mit dem wahlfreien Zugriff in der Datei.
`ios::out |ios::app`	Zum Schreiben öffnen und automatisch ans Dateiende positionieren. Falls die Datei nicht existiert, wird sie automatisch angelegt.

Fehlerbehandlung

Die Fehlerbehandlung bei Dateioperationen kann genau so erfolgen wie beim Lesen und Schreiben bei der Standardein-/ausgabe. Bei Fehlern wird das `fail`-Bit gesetzt, beim Erreichen des Dateiendes das `eof`-Bit, wenn die Operation gut geht das `good`-Bit.

10.4 Sequentielles Lesen und Schreiben

Wir können nun elementare Datentypen aus Dateien lesen und in Dateien schreiben. Wie aber schreibt und liest man Strukturen?

Man schreibt eigene Funktionen dafür!

Beispiel:

```
// student.h: Definition der Datenstruktur Student

#include <iostream>

struct Student {
    int matrikelNr;
    char name[30];
    char vorname[30];
};

bool read  (istream& in, Student& s);
bool write (ostream& out, const Student& s);
```

Die Funktion `read` erwartet als ersten Parameter ein Objekt vom Typ `istream`. Dieses Objekt kann z. B. die Standardeingabe `cin` sein, aber auch ein beliebiges Objekt vom Typ `ifstream`. Dies hat damit zu tun, dass `ifstream` eine von `istream` abgeleitete Klasse ist. Analog kann bei `write` als Parameter sowohl die Standardausgabe `cout` als auch ein beliebiges `ofstream`-Objekt übergeben werden.

Der Inhalt der Studentendatei soll wie folgt strukturiert sein:

```
1234567;Mueller;Peter;
1111111;Meier;Sepp;
2222222;Schmitt;Carlo;
3333333;Schulze;Egon;
```

Die Einträge sind also jeweils durch ein Trennzeichen voneinander abgetrennt. Einzelne Sätze noch zusätzlich durch ein Linefeed. Die Implementierung der Funktionen `read` und `write` sieht nun so aus:

```
// student.cpp: Implementierung der Student-Funktionen

#include "student.h"
#include <fstream>

bool read  (istream& in, Student& s) {
    in >> s.matrikelNr;
    if (!in.good())
       return in.good();
    in.get();
    in.getline(s.name, 30, ';');
    in.getline(s.vorname, 30, ';');
    in.get();
```

```
    return in.good();
}

bool write (ostream& out, const Student& s) {
    out << s.matrikelNr << ";"
        << s.name       << ";"
        << s.vorname    << ";\n";
    return out.good();
}
```

Mit dem folgenden einfachen Programm füllen wir nun unsere Studentendatei:

```
// studentout.cpp: Anlegen einer Studentendatei

#include "student.h"
#include <fstream>

int main() {
    char antwort;
    Student s;
    ofstream ausgabe("student.dat");
    if (!ausgabe.good()) {
        cerr << "Fehler beim Oeffnen von student.dat\n";
        return 1;
    }

    do {
        cout << "Studentendaten eingeben: \n";
        read (cin, s);         // Lesen von der Standardeingabe
        write (ausgabe, s);    // Schreiben in die Datei
        if (!ausgabe.good()) {
            cerr << "Fehler beim Schreiben in student.dat\n";
            break;
        }

        cout << "Noch einmal ? (j/n) ";
        cin >> antwort;
    } while (antwort != 'n');
    ausgabe.close();
}
```

Das Lesen der Datei erfolgt dann mit folgendem Programm:

```
// studentinp.cpp: Einlesen einer Studentendatei

#include "student.h"
#include <fstream>

int main() {
    char antwort;
    Student s;
    ifstream eingabe("student.dat");
    if (!eingabe.good()) {
        cerr << "Fehler beim Oeffnen von student.dat\n";
        return 1;
    }

    while (read(eingabe, s)) {
        cout << "Matrikel-Nr: " << s.matrikelNr << '\n'
             << "Name       : " << s.name       << '\n'
             << "Vorname    : " << s.vorname    << "\n\n";
    }
```

```
    eingabe.close();
}
```

Beispielausgabe:

```
Matrikel-Nr: 1234567
Name       : Mueller
Vorname    : Peter

Matrikel-Nr: 1111111
Name       : Meier
Vorname    : Sepp
```

10.5 Blockorientiertes Lesen und Schreiben

In den bisherigen Beispielen haben wir Dateien nur mit den Funktionen gelesen und geschrieben, die wir auch schon für die Standardein-/ausgabe verwendet hatten. Damit kann man allerdings nur Textdateien lesen und schreiben. Dateien, in denen die Daten binär abgespeichert sind, muss man anders verarbeiten.

Schreiben von Datenblöcken:

Die Klasse `ostream` und damit auch `ofstream` hat eine Methode (Funktion) `write`, die eine vorgegebene Anzahl Bytes aus dem Hauptspeicher in eine Datei überträgt.

```
ostream& write (const char* buf, int n);
```

- `buf` ist dabei die Anfangsadresse eines Hauptspeicherbereiches, der nicht notwendig ein `char`-Feld sein muss, sondern einen beliebigen Inhalt haben kann.
- `n` ist die Anzahl der zu übertragenden Bytes.
- Als Rückgabewert gibt `write` eine Referenz auf den Ausgabestrom zurück.

Lesen von Datenblöcken:

Die Klasse `istream` und damit auch `ifstream` hat eine Methode (Funktion) `read`, die eine vorgegebene Anzahl Bytes aus einer Datei in einen Hauptspeicherbereich überträgt.

```
istream& read (char* buf, int n);
```

- `buf` ist dabei die Anfangsadresse eines Hauptspeicherbereiches, der nicht notwendig ein `char`-Feld sein muss, sondern einen beliebigen Inhalt haben kann.
- `n` ist die Anzahl der zu übertragenden Bytes.
- Als Rückgabewert gibt `read` eine Referenz auf den Eingabestrom zurück.

Anwendung

Was wir im vorigen Abschnitt mit der normalen Ein-/Ausgabe gemacht haben, werden wir nun mit der blockorientierten Ein-/Ausgabe umsetzen.

```
// studentblockout.cpp: Anlegen einer Studentendatei

#include "student.cpp"
#include <fstream>

int main() {
    char antwort;
```

```
    Student s;
    ofstream ausgabe;
    ausgabe.open("student2.dat", ios::out | ios::binary);
    if (!ausgabe.good()) {
        cerr << "Fehler beim Oeffnen von student.dat\n";
        return 1;
    }

    do {
        cout << "Studentendaten eingeben: \n";
        read (cin, s);
        ausgabe.write (reinterpret_cast<char *>(&s), sizeof(Student));
        if (!ausgabe.good()) {
            cerr << "Fehler beim Schreiben in student.dat\n";
            break;
        }

        cout << "Noch einmal ? (j/n) ";
        cin >> antwort;
    } while (antwort != 'n');
    ausgabe.close();
}
```

Die erzeugte Datei hat folgendes Aussehen. Links der Hexadezimalcode und rechts, das was druckbar ist:

```
00000000 87D6 1200 4D65 6965 7200 0000 0D00 0000 ....Meier.......
00000016 81D5 4000 0D00 0000 ECFE 1200 282F 4000 ..@.........(/@.
00000032 B40D 5365 7070 0000 0000 0000 48B3 4000 ..Sepp......H.@.
00000048 60CF 4000 00FF 1200 282F 4000 C00B 4100 `.@.....(/@...A.
00000064 47F4 1000 4D75 656C 6C65 7200 0D00 0000 G...Mueller.....
00000080 81D5 4000 0D00 0000 ECFE 1200 282F 4000 ..@.........(/@.
00000096 B40D 5065 7465 7200 0000 0000 48B3 4000 ..Peter.....H.@.
00000112 60CF 4000 00FF 1200 282F 4000 C00B 4100 `.@.....(/@...A.
00000128 8EE8 2100 4265 636B 656E 6261 7565 7200 ..!.Beckenbauer.
00000144 81D5 4000 0D00 0000 ECFE 1200 282F 4000 ..@.........(/@.
00000160 B40D 4672 616E 7A00 0000 0000 48B3 4000 ..Franz.....H.@.
00000176 60CF 4000 00FF 1200 282F 4000 C00B 4100 `.@.....(/@...A.
```

Um blockorientiert schreiben zu können, muss man die Datei im Binärmodus öffnen.

```
ausgabe.open("student2.dat", ios::out | ios::binary);
```

Das eigentliche Schreiben ist dann etwas schwieriger nachzuvollziehen:

```
ausgabe.write (reinterpret_cast<char *>(&s), sizeof(Student));
```

`&s` ist die Anfangsadresse der `Student`-Struktur `s` im Hauptspeicher. Der Aufruf der Cast-Operators sorgt dafür, dass die Struktur von der Funktion `write` wie ein char-Feld verarbeitet wird. Die Länge dieses `char`-Feldes ist natürlich genau so groß wie ein `Student`-Objekt Bytes hat.

Das Einlesen sieht dann so aus:

```
// studentinp.cpp: Einlesen einer Studentendatei

#include "student.cpp"
#include <fstream>

int main() {
```

```
    char antwort;
    Student s;
    ifstream eingabe;
    eingabe.open("student2.dat", ios::in | ios::binary);
    if (!eingabe.good()) {
        cerr << "Fehler beim Oeffnen von student2.dat\n";
        return 1;
    }

    while (eingabe.read(reinterpret_cast<char*>(&s), sizeof(Student)))
    {
        cout << "Matrikel-Nr: " << s.matrikelNr << '\n'
             << "Name       : " << s.name       << '\n'
             << "Vorname    : " << s.vorname    << "\n\n";
    }
    eingabe.close();
}
```

Auch hier ist es nötig, die Datei im Binärmodus zu öffnen. Auch hier muss das übergebene `Student`-Objekt zu einem `char`-Feld konvertiert werden, damit die Daten hineingelesen werden können.

Noch offene Punkte bei der Iostream-Bibliothek

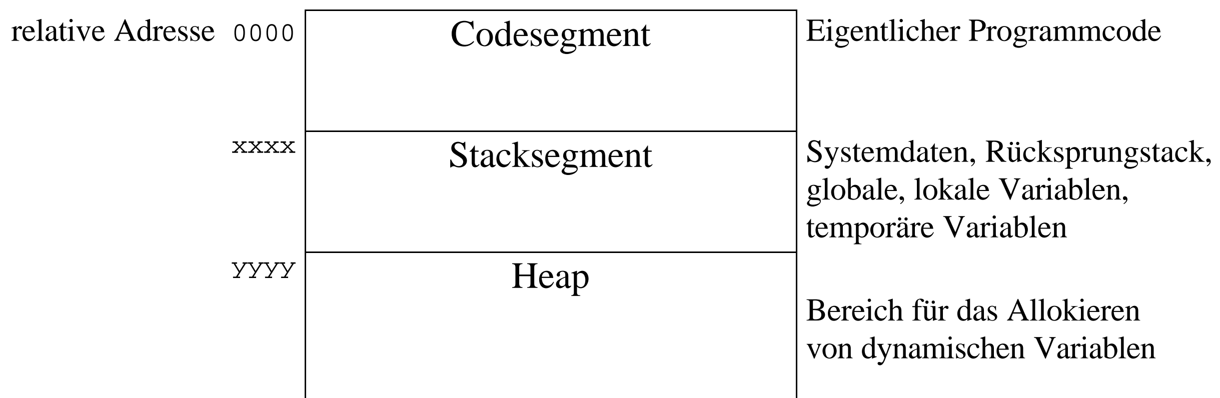
- Es ist möglich, den Dateizeiger in Dateien frei zu positionieren mit Funktionen wie `tellg` und `seekp`.
- Es gibt Datenströme zum Einlesen aus char-Feldern und zum Schreiben in `char`-Felder (`istrstream`, `ostrstream`, `strstream`)
- Es gibt Datenströme zum Einlesen aus `string`-Objekten und zum Schreiben in `string`-Objekte.

11 Zeiger

Als systemnahe Programmiersprache, in der ein Großteil der systemnahen Software implementiert wird, hat C++ nicht nur die Möglichkeit, mit Datenelementen umzugehen, sondern auch mit deren Adressen.

11.1 Einführung

Speicherorganisation eines geladenen Programmes



- Jedes C++-Objekt liegt an einem bestimmten Ort innerhalb des dem Programm vom Betriebssystem zugewiesenen Hauptspeicherbereiches.
- Die konkrete Lage wird vom Compiler festgelegt.
- Der Hauptspeicher ist untergliedert in Maschinenworte und diese wiederum in Bytes.
- Die Adresse eines Objektes ist die Byte-Nummer, ab der das Objekt gespeichert ist.
- Der zugewiesene Hauptspeicher beginnt ab der Adresse 0 und wird dem Programm vom Betriebssystem als sequentielle Byte-Folge dargestellt.

Definition: Zeiger

Ein **Zeiger** (engl. Pointer) ist ein Ausdruck, der die Adresse und den Typ eines Objektes (Datenelement oder auch Funktion) repräsentiert.

Der Adressoperator

Zum Ermitteln der Adresse eines Objektes wird der **Adressoperator** `&` angewandt:

```
int i = 1;

cout << "Wert von i    : " << i  << '\n'
     << "Adresse von i: " << &i << endl;

/* Ausgabe z. B.
Wert von i    : 1
Adresse von i : 0x0012ff88
*/
```

Der Ausdruck `&i` (lies: *Adresse von i*) ist ein konstanter Zeiger auf die `int`-Variable `i` und enthält die Adresse an der `i` innerhalb des Speichers des ablaufenden Prozesses zu finden ist.

Vorsicht Verwechselungsgefahr!:

- & in einer Typdefinition bezeichnet einen Referenztyp:

```
int& i = ...;    // i hat den Typ "int-Referenz"
```

- & in einem Ausdruck bezeichnet den Operator, der eine Adresse liefert

```
...&i...         // Adresse von i
```

- Der Adressoperator wird vorwiegend zur Wertzuweisung an Zeiger und zur Übergabe von Adressen als Funktionsparameter benötigt.

Der Verweisoperator

Eine Variable, die eine Adresse aufnehmen kann, definiert man mit Hilfe **des Verweis-Operators `*`**

```
int *ip = &i;
```

Damit wird eine Variable `ip` definiert, die den Typ `int*` (sprich: *Zeiger auf `int`*) hat. `ip` kann somit die Adresse einer `int`-Variablen speichern. In einer Deklaration bzw. Definition bedeutet der Operator `*` immer "Zeiger auf".

Zeigertypen

Zu jedem Datentyp von C++ gibt es einen assoziierten Zeigertyp

```
int        →  int*
char       →  char*
double     →  double*
....       →
Punkt      →  Punkt*
.....
int *      →  int **
```

Zusätzlich gibt es den sogenannten *generischen* Zeigertyp

```
void *
```

Ein Zeiger vom Typ `void*` kann auf jede beliebige Hauptspeicheradresse verweisen unabhängig vom Typ des dort befindlichen Objektes.

Eine Variable von einem Zeigertyp belegt zumindest auf 32-Bit-Maschinen 4 Bytes und kann als Wert eine Adresse des zugehörigen Inhalttyps aufnehmen. Zu jedem Basistyp `T` gibt es also einen davon abgeleiteten Zeigertyp `T*` und wie bereits besprochen einen Referenztyp `T&`.

```
int a, *ptr = &a, &r = a;  // int, int-Zeiger und int-Referenz
```

Unterschiede zwischen Referenz und Zeiger

- Sowohl eine Referenz als auch ein Zeiger verweisen jeweils auf ein Objekt im Hauptspeicher.
- Eine Referenz ist kein eigenständiges Objekt und kann nicht ohne referenziertes Objekt existieren.

- Ein Zeiger ist ein eigenständiges, isoliertes Objekt, das auch ohne referenziertes Objekt lebensfähig ist.
- Ein Zeiger hat eine eigene Adresse und kann durch das Zuweisen einer neuen Adresse jederzeit auch auf andere Datenobjekte verweisen.

Beispiel: Inhalt und Adresse eines Zeigers:

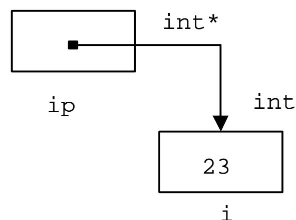
```
    int *ip = &i;
    cout << "Wert von ip    : " << ip  << '\n'
         << "Adresse von ip: " << &ip << endl;

/* Ausgabe z. B.
Wert von ip    : 0x0012ff88
Adresse von ip: 0x0012ff84
*/
```

Begriffsbestimmung

Adresse	Ort eines Objektes im Hauptspeicher
Zeiger	Variable mit einer Adresse als Wert
Zeigertyp	Typ eines Zeigers
Basistyp	Typ des Objektes an einer Adresse
referenziertes Objekt	Objekt, auf das ein Zeiger verweist

Beispiel:

```
int i = 23;
int* ip = &i;
```



Adresse	Wert von `ip`, Ort von `i` im Speicher
Zeiger	Variable `ip`
Zeigertyp	`int*`
Basistyp	`int`
referenziertes Objekt	`i`

Vorbelegung von Zeigern

Standardmäßig hat eine Zeigervariable bei ihrer Definition einen undefinierten Wert. Zeiger sollten jedoch wie jede andere Variable auch entweder initialisiert werden oder frühzeitig mit einem Wert versehen werden. Wenn einem Zeiger noch keine konkrete Adresse zugewiesen werden kann, so sollte er mit der nie vorkommenden Adresse 0 initialisiert werden.

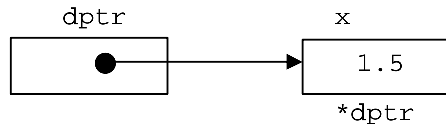
```
char *cp = 0;
int * ip = NULL;     // alternativ möglich, aber unnötig
```

Der Wert `NULL` ist ein Makro, das in den unterschiedlichsten Headerdateien definiert ist und häufig als Nullwert für Zeiger verwendet wird. Die Zahl 0 hingegen ist automatisch als Nullwert für alle Zeigertypen zugelassen.

Zugriff auf Objekte über Zeiger

Um das Objekt, auf das ein Zeiger verweist, mit Hilfe des Zeigers anzusprechen, wird ebenfalls der Operator `*`, hier allerdings in einer anderen Rolle, als Dereferenzierungs-Operator, benutzt.

```
double x = 1.5;
double *dptr;
dptr = &x;
```




```
double y = *dptr; // Inhalt der Adresse, die in dptr steht
// danach steht in y der Wert 1.5
```

Allgemein gilt: Ist `p` ein Zeiger, so ist `*p` (lies: *Inhalt von* `p`) das Objekt auf das `p` zeigt.

Der Dereferenzierungs-Operator * hat wie der Adressoperator & einen hohen Vorrang gegenüber anderen Operatoren.

Vorsicht!

1. Was ist wohl der Unterschied zwischen den beiden folgenden Definitionen?

```
int* ip1, ip2, ip3;
int *ip1, *ip2, *ip3;
```

2. Es besteht eine gewisse Verwechselungsgefahr:

- * in der Typdefinition bedeutet Adresstyp

  ```
  int* ip;
  ```

- * in einem Ausdruck liefert als unärer Operator das von einem Zeiger referenzierte Objekt

  ```
  ...*ip...       // von ip referenziertes Objekt
  ```

- * in einem Ausdruck steht als binärer Operator für die Multiplikation

  ```
  ...2*3...       // Multiplikation
  ```

L-Werte

Der *-Operator liefert ein Objekt, d. h. einen L-Wert als Ergebnis. Ein Ausdruck mit *-Operator kann in daher jedem Zusammenhang eingesetzt werden, in dem ein Objekt des Basistyps stehen kann.

```
double x = 1.5;
double *dptr;
dptr = &x;

*dptr = 2.0;   // danach ist x = 2.0 !
```

Damit sind sogar Auswüchse möglich wie der folgende korrekte Ausdruck:

```
*dptr = *dptr**dptr;
```

Zuweisungen zwischen Zeigern

Adressen können zugewiesen werden:

```
int* ip1;
int* ip2;
```

```
ip1 = 0;        // Zuweisung Nullzeiger an ip
ip2 = ip1;      // Zuweisung ip1 (Wert = Nullzeiger) an ip2
```

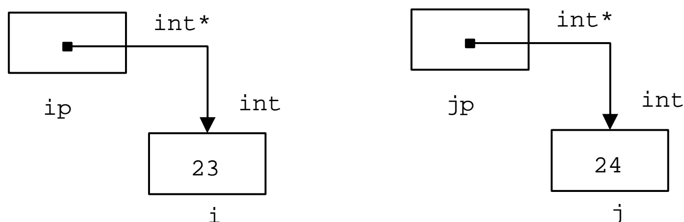
Werte unterschiedlicher Zeigertypen (z.B. `int*` und `double*`) sind inkompatibel, d.h. sie können einander nicht zugewiesen werden. Ausnahme ist der Nullzeiger, der jedem Zeiger zugewiesen werden kann. Ausnahme ist ebenso eine Zuweisung an `void`-Zeiger.

```
int i = 1;
int* ip = &i;
double x = 2.0;
double* dp = &x;

ip = dp;        // FALSCH! nicht typverträglich
dp = ip;        // FALSCH! nicht typverträglich

void* p;
p = ip;         // OK
ip = p;         // Die Umkehrung gilt aber nicht!
```

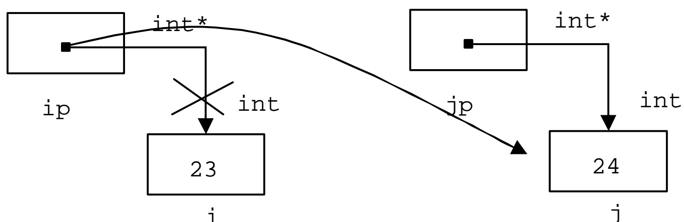
Die Zuweisung zwischen unterschiedlichen Zeigertypen ist allerdings immer durch explizite Typkonvertierungen zu erzwingen. Allerdings muss man dann schon genau wissen, was man tut.

<u>Beispiel:</u>

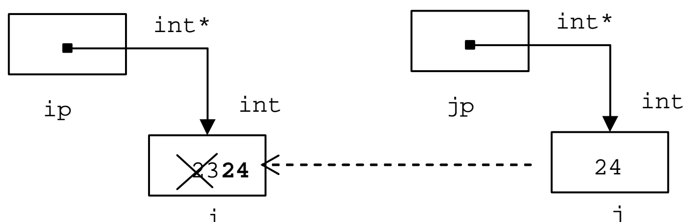
```
int i = 23;
int j = 24;
int* ip = &i
int* jp = &j;
```




```
ip = jp;        // Zuweisung zwischen Zeigern
```




```
*ip = *jp;      // Zuweisung zwischen referenzierten Objekten
```



Vergleiche zwischen Zeigerausdrücken

Adressen können auf Gleichheit verglichen werden:

```
int* ip1;
int* ip2;
...
if(ip1 == ip2)   // falls beide Zeiger den gleichen Wert haben...
   ...
```

Werte unterschiedlicher Zeigertypen (z.B. `int*` und `double*`) sind nicht vergleichbar. Ausnahme ist der Nullzeiger, mit dem jede Adresse verglichen werden kann. Auch mit `void`-Zeigern kann immer verglichen werden.

Wie bei Zuweisung ist der Vergleich zwischen Zeigern und zwischen den referenzierten Objekten möglich:

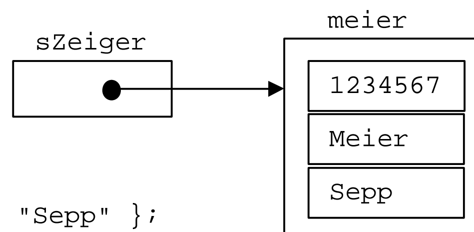
```
int i = 23;
int j = 24;
int* ip = &i
int* jp = &j;

if(ip == jp)... // Vergleich zwischen Zeigern

if(*ip == *jp)...   // Vergleich zwischen referenzierten Objekten
```

Größenvergleiche mit `>` `<` `>=` `<=` von Adressen sind zwar möglich, im Allgemeinen aber wenig sinnvoll.

11.2 Zeiger und Strukturen

Zeiger auf Strukturen funktionieren wie alle anderen Zeiger:

```
struct Student {
    int matrikelNr;
    char name[30];
    char vorname[30];
};

Student meier = { 1234567, "Meier", "Sepp" };
Student *sZeiger;

sZeiger = &meier;          // sZeiger zeigt nun auf die Struktur meier
```



Zugriff auf einzelne Komponenten:

Typ	*Zugriff über struct-Variable*	*Zugriff über Zeiger mit Operator `*`*	*Zugriff über Zeiger mit Operator `->`*
`int`	`meier.matrikelNr`	`*sZeiger.matrikelNr`	`sZeiger->matrikelNr`
`char*`	`meier.name`	`*sZeiger.name`	`sZeiger->name`
`char*`	`meier.vorname`	`*sZeiger.vorname`	`sZeiger->vorname`

Strukturen, Strukturelemente und Adressen :

Typ	*Ausdruck*
`Student*`	`&meier`
`int*`	`&(meier.matrikelNr)`
`char**`	`&(meier.name)`
`char*`	`&(meier.name[0])`

11.3 Zeiger und Felder

Wie bereits mehrfach erwähnt, bezeichnet in C++ der Name eines Feldes zugleich die Anfangsadresse des Feldes im Hauptspeicher. Genauer gesagt: Ein Feldname ist ein Zeiger auf das erste Vektorelement.

```
char hallo[] = "Hallo Welt!";
```

`hallo` ist ein char-Zeiger auf das Element `hallo[0]`, d. h. es bedeutet das selbe wie der Ausdruck `&hallo[0]`.

```
cout << hallo;        // gleichbedeutend mit cout << &hallo[0];
```

Ein Feldname ist allerdings keine Zeigervariable, sondern eine Konstante, der man nicht einfach eine andere Adresse zuweisen kann. Es ist aber möglich, einen Feldnamen einer Zeigervariablen zuzuweisen.

```
char *cPtr;
cPtr = hallo;         // cPtr zeigt jetzt auch auf das char-String
cout << cPtr;         // "Hallo Welt" ausgeben
```

Beispiel:

```
// textPtr.cpp: char-Vektoren und char-Zeiger

#include <iostream>

int main() {
   cout << "Demo mit char-Vektoren und char-Zeigern.\n\n";
   char text[] = "Guten Morgen!",
        name[] = "Hans!";

   char *cPtr = "Hallo ";           // cPtr auf "Hallo "
                                    // zeigen lassen.
   cout << cPtr << name << '\n'
        << text << endl;

   cout << "Der Text \"" << text
        << "\" beginnt bei der Adresse " << static_cast<void*>(text)
        << endl;

   cout << text + 6         // Was passiert hier?
        << endl;

   cPtr = name;             // cPtr auf name zeigen lassen.

   cout << "Das ist das " << *cPtr << " vom " << cPtr
        << endl;

   *cPtr = 'k';
   cout << "Der Hans der " << cPtr << endl;
}
/* Beispiel für eine Ausgabe:
Demo mit char-Vektoren und char-Zeigern.

Hallo Hans!
Guten Morgen!
Der Text "Guten Morgen!" beginnt bei der Adresse 0x0012ff7c
Morgen!
Das ist das H vom Hans!
Der Hans der kans!   */
```

Um statt dem `char`-String die Adresse des ersten Zeichens auszugeben, muss man den Zeiger zu einem `void`-Zeiger konvertieren.

Adressierung von Feldelementen

Der Zugriff auf einzelne Feldelemente ist in C++ eng mit der Zeigerarithmetik verbunden. Zeigerarithmetik heißt, dass es möglich ist, zu Zeigerwerten ganze Zahlen hinzuzuaddieren oder davon zu subtrahieren.

Beispiel:

```
int tab[4] = { 0, 10, 20, 30 };
```

Wir wissen bereits, dass `tab` ein Zeiger auf das Feldelement `tab[0]` ist. Der Ausdruck `tab+1` zeigt nun auf das nächste Feldelement `tab[1]`, d. h. die Adresse, die in `tab` steht wird um `sizeof(int)` erhöht und nicht um `1` wie man vermuten könnte. Allgemein gilt:

`tab + i` zeigt auf das Feldelement `tab[i]`
`*(tab + i)` ist das Feldelement `tab[i]`

Natürlich kann man diese Art der Adressierung auch direkt mit Zeigervariablen durchführen.

Beispiel:

```
int* iPtr = tab;     // iPtr zeigt auf tab[0]
```

Dann ist `iPtr` genau wie `tab` ein Zeiger auf das Feldelement `tab[0]`. Folglich zeigen dann auch `iPtr+1`, `iPtr+2`, `...` auf die Feldelemente `tab[0], tab[1], ...`

Für eine beliebige ganze Zahl `i` sind dann die folgenden Ausdrücke äquivalent:

`&tab[i]   tab + i   iPtr + i`	Adresse des i-ten Feldelementes
`tab[i]   *(tab + i)   *(iPtr + i)   iPtr[i]`	i-tes Feldelement

D.h. auch bei normalen Zeigern ist die Schreibweise `iPtr[i]` erlaubt. Dies liegt schlicht und einfach daran, dass der Compiler den Ausdruck `tab[i]` sowieso intern zu `*(tab + i)` übersetzt, was soviel heißen soll wie "Gehe von der Anfangsadresse `tab` um `i` Objekte weiter, und nimm das dortige Objekt!". Das selbe gilt natürlich dann auch für den Ausdruck `iPtr[i]`.

Zeigerarithmetik

Mit Zeigervariablen und mit den in ihnen enthaltenen Adressen ist Arithmetik in beschränktem Umfang möglich:

- Zeigervariablen dürfen anderen Zeigervariablen gleichen Typs zugewiesen werden.
- Zeigervariablen dürfen verglichen werden (`==, !=, <, >, <=, >=`).
- Zeigervariablen dürfen inkrementiert und dekrementiert werden (`++, --`).
- Zeigervariablen dürfen um ganzzahlige Werte erhöht pder vermindert werden (`ptr +/- int-Ausdruck`).
- Der Abstand zwischen zwei Zeigern darf ermittelt werden (`ptr1 - ptr2`). Das Ergebnis ist dann aber immer vom Typ `int` und keinesfalls eine Adresse. Das Ergebnis gibt

allerdings nicht die Anzahl an Bytes zwischen beiden Adressen an sondern vielmehr die Anzahl an Datenelementen zwischen beiden Adressen.

Beispiel: Suche in einer Tabelle

```
// studenttab.cpp: Suchen in einer struct-Tabelle

#include <iostream>

struct Student {
    int matrikelNr;
    char name[30];
    char vorname[30];
};

// Suchen in einem Feld mit Hilfe von Zeiger-Arithmetik
int suchen1( Student *stab, int anzahl, int suchNr) {
    Student *sPtr;
    for (sPtr = stab; sPtr < stab + anzahl; ++sPtr)
        if (sPtr->matrikelNr == suchNr)
            return sPtr - stab;  // Anzahl Objekte zwischen den
                                 // beiden Adressen
    return -1;
}

// Suchen in einem Feld mit Hilfe von Indizes
int suchen2( Student *stab, int anzahl, int suchNr) {
    int i;
    for (i = 0; i < anzahl; ++i)
        if (stab[i].matrikelNr == suchNr)
            return i;
    return -1;
}

int main() {
    Student stab[] = { { 1111111, "Meier", "Sepp" },
                       { 2222222, "Mueller", "Peter" },
                       { 3333333, "Schmidt", "Hans" },
                       { 4444444, "Hoffmann", "Jakob" } };
    cout << "suchen1(3333333): " << suchen1(stab, 4, 3333333) << '\n'
         << "suchen1(5555555): " << suchen1(stab, 4, 5555555) << '\n';

    cout << "suchen2(3333333): " << suchen2(stab, 4, 3333333) << '\n'
         << "suchen2(5555555): " << suchen2(stab, 4, 5555555) << '\n';

}/*
suchen1(3333333): 2
suchen1(5555555): -1
suchen2(3333333): 2
suchen2(5555555): -1
*/
```

Beispiel: Die GNU-Implementierung von `strcmp`, `strcpy` und `strlen`

```
int strcmp(const char *s1, const char *s2)
{
  while (*s1 == *s2)
  {
    if (*s1 == 0)
      return 0;
    s1++;
    s2++;
  }
```

```
  return *(unsigned const char *)s1 - *(unsigned const char *)(s2);
}

char * strcpy(char *to, const char *from)
{
  char *save = to;

  for (; (*to = *from); ++from, ++to);
  return save;
}

int strlen(const char *str)
{
  const char *s;

  if (str == 0)
    return 0;
  for (s = str; *s != '\0'; ++s);
  return s-str;
}
```

Spezialitäten

Was bedeutet wohl der Ausdruck

```
*ptr++
```

Der Operator * hat die gleiche Priorität wie der Operator ++. Die Auswertung erfolgt aber von rechts nach links, d. h. sie erfolgt so:

```
*(ptr++)
```

Der Wert dieses Ausdrucks ist also der Inhalt der Speicherstelle auf die `ptr` zunächst zeigt. Der Zeiger `ptr` wird allerdings in diesem Ausdruck erhöht.

11.4 Speicherreservierung zur Laufzeit

Alle bisher verwendeten Datenelemente werden automatisch allokiert und am Ende ihrer Lebensdauer automatisch freigegeben.

- Lokale Variablen:
  - Allokieren beim Ausführen der Definition
  - Freigeben beim Austritt aus dem unmittelbar umgebenden Block

Beispiel:

```
int main() {
    int i = 1;                  /* i allokieren */
    while(i <10)
    {
        int j = 2*i + 3;        /* j allokieren */
        cout << j;
    }                           /* j freigegeben */
    cout << i;
}                               /* i freigegeben */
```

- Globale und statische Variablen:
  - Allokieren beim Start des Programmlaufes

- Freigeben am Ende des Programmlaufes

Der Entwickler hat nur mittelbar einen Einfluss auf die Lebenszeit der Variablen. Diese wird von der Blockstruktur bestimmt. Fehler (fehlendes Allokieren, vergessenes Freigeben) sind ausgeschlossen, weil der Compiler automatisch korrekten Code erzeugt.

Dynamische Speicherreservierung

Nun ist zur Zeit der Übersetzung oft noch nicht bekannt, wie viele Daten ein Programm wirklich benötigt. In solchen Fällen ist es notwendig, dynamisch, d. h. zur Laufzeit des Programms, Speicherplatz reservieren zu können. Dies geschieht mit Hilfe des Operators **`new`**.

Dynamisch reservierter Speicher wird nicht automatisch freigegeben, sondern muss explizit durch Aufruf des Operators **`delete`** freigegeben werden. Die Verantwortung für das Reservieren und wieder Freigeben obliegt also vollständig dem Programmierer.

Dynamisch reservierter Speicher wird aus dem sogenannten Heap oder Freispeicher des Prozesses angefordert.

11.4.1 Der Operator `new`

Problem: Dynamische Datenstrukturen können nicht im Quelltext definiert werden, denn Art und Anzahl dynamischer Objekte ergeben sich erst zur Laufzeit, liegen zur Übersetzungszeit also noch nicht fest.

Folge: Mangels Definition haben dynamische Objekte keine Namen

Lösung: Ein dynamisches Objekt wird namenlos allokiert und die Adresse des neuen Objektes wird zurückgeliefert. Um auf die namenlose Variable zugreifen zu können, verwendet man einen Zeiger.

Syntax:

```
ptr = new Typ;       // ptr Zeiger auf Typ
```

Wirkungsweise:

Der Operator `new` allokiert im Heap ein Objekt des angegebenen Typs und liefert die Adresse darauf zurück. Nach dem Aufruf von `new` hat die dynamisch erzeugt Variable noch keinen sinnvollen Wert.

Beispiel:

```
int* ip;        // Zeiger auf int, nicht initialisiert
ip = new int;   // Allokieren eines neuen int-Objektes
```

Das neue Datenelement kann nun über seinen Zeiger ganz normal verwendet werden:

```
cin >> *ip;     // einlesen eines Wertes
(*ip)++;        // um 1 hochzählen
cout << *ip;    // ausgeben des hochgezählten Wertes
```

Allokieren mit Initialisierung

Zumindest beim Allokieren von elementaren Datentypen ist es möglich, `new` als Parameter einen passenden Wert mitzugeben, mit dem die dynamische Variable initialisiert werden soll.

```
int* ip;             // Zeiger auf int, nicht initialisiert
ip = new int(5);     // Allokieren eines neuen int-Objektes,
                     // initialisieren mit dem Wert 5
```

11.4.2 Der Operator `delete`

Ein mit `new` allokiertes Objekt bleibt unabhängig vom weiteren Programmablauf erhalten, bis es explizit freigegeben wird. Es liegt nun in der Verantwortung des Programmierers, dafür zu sorgen, dass nicht mehr benötigter Speicher wieder freigegeben wird. Dies geschieht mit dem Operator `delete`.

Syntax:

```
delete ptr;    // ptr Zeiger auf dynamisch allokiertes Datenelement
```

Wirkungsweise:

Der allokierte Speicherbereich wird wieder freigegeben. Hat `ptr` den Wert `0` bzw. `NULL`, dann passiert nichts.

Beispiel:

```
int* ip;
ip = new int;
...
delete ip;     // dynamisches int-Objekt wird freigegeben
```

Mögliche Fehler:

- Der Wert des Zeigers wird durch `delete` nicht verändert, d. h. in `ip` steht noch die Adresse des gerade freigegebenen Objektes. Ein Zugriff darauf kann zu unkalkulierbaren Ergebnissen führen.
- Auch ein Freigeben eines bereits freigegebenen Speicherbereiches kann fatale Folgen haben.
- Mit `delete` kann man keine automatischen Variablen freigeben!

```
int i = 1;
int* ip = &i;
delete ip;  // FEHLER!
```

- Dynamisch allokierter Speicher wird nie freigegeben, ein häufiges Problem von bekannten Betriebssystemen.
- Der Zeiger wird vom System automatisch freigegeben, der dynamisch allokierte Speicher bleibt.

```
{
    int* ip = new int;
    ....
}     // ← ip wird automatisch freigegeben, die dynamische Variable
      //    wird zur Speicherleiche
```

- Einem Zeiger, der auf ein dynamisch allokiertes Objekt zeigt, wird ein anderer Wert zugewiesen. Damit wird der ursprünglich allokierte Objekt nicht mehr referenziert.

```
int* ip = new int;
...
ip = new int;     // Das erste int ist nicht mehr referenziert!
```

Empfehlung: Ein Zeiger sollte nach dem Aufruf von `delete` immer explizit auf `0` gesetzt werden.

```
int* ip;
ip = new int;
...
delete ip;
ip = 0;  // Zeiger ins Nichts auf 0 zurücksetzen
```

Fehlerbehandlung bei `new`

Falls nicht mehr genug Speicherplatz auf dem Heap vorhanden ist, wird standardmäßig eine Ausnahme von `new` ausgelöst, die vom Programm aufgefangen werden kann. Bei älteren Compilern wird von `new` noch im Fehlerfalle der Nullzeiger zurückgegeben, d. h. hier muss streng genommen nach jedem `new` überprüft werden, ob es gut gegangen ist.

11.4.3 Dynamisch allokierte Felder

Zum Allokieren und Freigeben von Feldern gibt es eigene Varianten des `new` bzw. `delete`-Operators. Zum Allokieren wird der `new[]`-Operator und zum Freigeben der `delete[]`-Operator verwendet.

Beispiel:

```
int *iPtr = new int[100];  // iPtr zeigt auf das erste Element eines
                           // 100-elementigen int-Feldes
for (int i = 0; i < 100; ++i)
    iPtr[i] = 0;
...
delete[] iPtr;             // Freigeben des kompletten Feldes
```

Vorsicht! Ein `delete iPtr` würde nur das Element `iPtr[0]` freigeben und der Rest des Feldes wäre nicht mehr referenzierbar.

Beispiel: Zur Laufzeit dimensionierbares Feld

```
int feldGroesse;
cout << "Größe des Feldes? ";
cin >> feldGroesse;
int *tab = new int[feldGroesse];
```

Im folgenden werden wir die Beispieldatenstruktur `DoubleStack` aus Abschnitt 9.1.1 noch einmal in überarbeiteter Form entwickeln. Nachteile der damaligen Lösung waren das fest dimensionierte Feld in der Struktur und die Verwendung einer globalen Konstante für die Feldgröße. Zusätzlich werden bei dieser Implementierung statt Referenzen Zeiger benutzt, nicht weil dies einfacher wäre, sondern eher um den Umgang mit Zeigern zu demonstrieren.

Die Datenstruktur

```
// doublestack.h Struktur zur Implementierung eines double-Stack
// 2. Version: dynamisch allokiertes Array
struct DoubleStack {
   int anzElemente;      // Anzahl Elemente des Stacks
   int groesse;          // Groesse des internen Feldes
   double* tab;          // Implementierungs-Array
```

```
};
```

Die Implementierung

```
// DoubleStack.cpp  Funktionen zur Implementierung eines Double-Stack
// Implementierung des dynamisch allokierten Arrays

#include "doublestack.h"

// Element hinzufuegen
void push (DoubleStack* sp, double x) {
   if (sp->anzElemente < sp->groesse)
       sp->tab[sp->anzElemente++] = x;
}

// oberstes Element wegnehmen
void pop(DoubleStack* sp) {
   if (sp->anzElemente > 0)
       sp->anzElemente--;
}

// Inhalt des obersten Elementes erfragen
double top(const DoubleStack* sp) {
   if (sp->anzElemente > 0)
       return sp->tab[sp->anzElemente - 1];
   else {
       cerr << "Stack leer\n";
       return 0.0;
   }
}

// Ist der Stack leer?
bool empty (DoubleStack* sp) {
    return (sp->anzElemente == 0);
}

// Ist der Stack voll?
bool full  (DoubleStack* sp) {
    return (sp->groesse <= sp->anzElemente);
}

// Wieviel Elemente stehen im Stack?
int size  (const DoubleStack* sp) {
    return sp->anzElemente;
}

// Stack-Inhalt ausgeben
void print(const DoubleStack* sp) {
   for (int i = sp->anzElemente - 1; i >= 0; --i)
      cout << sp->tab[i] << "   ";
   cout << endl;
}

// Stack initialisieren
// internes Feld allokieren
void init(DoubleStack* sp, int groesse) {
    sp->anzElemente = 0;
    sp->groesse = groesse;
    sp->tab = new double[sp->groesse];
    for (int i = 0; i < sp->groesse; i++)
        sp->tab[i] = 0.0;
}

// Stack komplett leermachen und internes Feld löschen
void clear(DoubleStack* sp) {
```

```
    delete[] sp->tab;
    sp->tab = 0;
    sp->groesse = 0;
    sp->anzElemente = 0;
}
```

Ein Testprogramm

```
// DoubleStackt2.cpp: 2. Testprogramm fuer double-Stack

#include "doublestack.h"

int main() {
   DoubleStack *sp1;
   sp1 = new DoubleStack;
   init(sp1, 10);
   push(sp1, 3.14);
   push(sp1, 2.718);
   print(sp1);
   pop(sp1);
   print(sp1);
   clear(sp1);
}
/* Ausgabe
2.718   3.14
3.14   */
```

11.5 Zeigerfelder und Kommandozeilenparameter

Zeiger bieten die Möglichkeit, mit umfangreichen Daten einfach und effizient umzugehen. So ist es zum Beispiel beim Sortieren von großen Objekten effizienter, nicht die Objekte im Speicher zu verschieben, sondern nur die Adressen darauf.

Definition von Zeigerfeldern

```
Student* studentPtr[10];   // studentPtr ist ein Feld von Student-Zeigern
```

Das Feld `studentPtr` besteht aus 10 Zeigern vom Typ `Student*`. Jedem dieser Zeiger kann nun die Adresse eines `Student`-Objektes zugewiesen werden.

```
Student meier   = { 1111111, "Meier", "Sepp" };
Student mueller = { 2222222, "Mueller", "Peter" };

studentPtr[0] = &meier;
studentPtr[1] = &mueller;
studentPtr[2] = new Student;

studentPtr[2].matrikelNr = 3333333;
strcpy(studentPtr[2].name, "Schulze");
strcpy(studentPtr[2].vorname, "Egon");

for (int i = 3; i < 10; i++)
    studentPtr[i] = 0;
```

Die einzelnen Objekte, auf die die Zeiger eines Zeigerfeldes verweisen, müssen nicht zusammenhängend im Speicher liegen. Üblicherweise werden solche Objekte zur Laufzeit mit `new` dynamisch angelegt und später auch wieder mit `delete` aus dem Speicher entfernt. Die Reihenfolge der Objekte ist dann alleine durch die Zeiger gegeben.

Beispiel: Feld von `char`-Zeigern

Die nachstehende Funktion gibt zu einer Fehlernummer einen entsprechenden Fehlertext aus einem Feld von `char`-Strings aus.

```
// displayError.cpp : Zu einer Fehlernummer eine Fehlermeldung
// ausgeben

#include <iostream>

void displayError(int errorNr) {
    static char* errorMsg[] = {
        "Ungültige Fehlernummer",
        "Fehler 1: Zu viele Daten",
        "Fehler 2: Nicht genügend Speicher",
        "Fehler 3: Keine Daten vorhanden" };
    if (errorNr < 1 || errorNr > 3)
        errorNr = 0;
    cerr << errorMsg[errorNr] << endl;
}
```

Das Schlüsselwort `static`

Das dort vorkommende Schlüsselwort `static` bedeutet, dass das Feld `errorMsg` beim ersten Durchlaufen der Funktion `displayError()` angelegt und initialisiert wird und nach dem Verlassen der Funktion erhalten bleibt. Beim Wiedereintritt in die Funktion haben `static`-Variablen noch den Wert, den sie beim letzten Durchlauf zuletzt besaßen. `static`-Variablen werden erst mit Programmende aus dem Speicher entfernt. Bei der objektorientierten Software-Entwicklung werden üblicherweise keine lokalen oder globalen `static`-Variablen eingesetzt.

Kommandozeilenparameter

Bisher haben wir die Funktion `main()` immer ohne Parameter verwendet. Es ist aber möglich, auch der Funktion `main()` Parameter mitzugeben. Diese werden beim Start des Programms über die Kommandozeile mitgegeben. Die Parameter von main sehen wie folgt aus:

```
int main(int argc, char* argv[])
{ ...          }
```

In `argc` wird die Anzahl der Argumente übergeben, die in der Kommandozeile angegeben wurden. Dabei zählt der Programmname mit , so dass `argc` immer mindestens den Wert `1` hat.

Der Parameter argv ist ein Feld von `char`-Zeigern:

`argv[0]`	zeigt auf den Programmnamen
`argv[1]`	zeigt auf das erste eigentliche Argument, also das erste Wort hinter dem Programmnamen.
`argv[2]`	zeigt auf das zweite Argument.
`....`	
`argv[argc - 1]`	zeigt auf das letzte Argument.
`argv{argc]`	ist der Nullzeiger

Üblicherweise werden die Parameternamen `argc` und `argv` benutzt, aber es können auch beliebige andere Namen gewählt werden.

Unter verschiedenen Betriebssystemen kann noch ein dritter Parameter für `main()` deklariert werden. Es handelt sich hierbei um ein Feld von `char`-Zeigern auf die Definitionen der Umgebungsvariablen.

Beispiel:

```
// argtest.cpp: Test der Kommandozeilenparameter

#include <iostream>

int main (int argc, char* argv[] )
{
    cout << "Programmname: " << argv[0] << endl;
    if (argc < 2)
        cout << "Keine Parameter uebergeben!\n";
    else
        for (int i = 1; i < argc; i++)
            cout << "argv[" << i << "]: " << argv[i] << endl;
}/*
Start mit: argtest
Programmname: argtest
Keine Parameter uebergeben!

Start mit: argtest hallo du da!
Programmname: argtest
argv[1]: hallo
argv[2]: du
argv[3]: da!       */
```

11.6 Zeiger auf const-Objekte und const-Zeiger

Mit Hilfe eines normalen Zeigers kann sowohl lesend als auch schreibend auf ein Objekt zugegriffen werden. Wie definiert man aber nun Zeiger auf Konstanten oder auch konstante Zeiger?

Zeiger auf Konstanten:

```
const int max = 100; // int-Konstante

int *ip;

ip = &max;           // FEHLER! Wird vom Compiler nicht übersetzt !

const int *ipc;      // ein Zeiger auf eine int-Konstante
ipc = &max;          // o.k.
```

- Über einen Zeiger auf ein konstantes Objekt darf man natürlich nicht den Inhalt der Konstanten verändern.
- Normale Zeiger dürfen nicht auf konstante Objekte zeigen. Zeiger auf Konstanten dürfen aber durchaus auf nicht-konstante Objekte zeigen. Allerdings ist auch dann eine Änderung nicht erlaubt.

```
int i = 5;
const int *ipc = &i;       // o.k.
*ipc = 6;                  // nicht erlaubt !!
```

Beispiel:

```
char* strcpy (char *p, const char *q);
```

Der `char`-String, auf den `q` zeigt darf innerhalb der Funktion nicht verändert werden

Konstante Zeiger

Umgekehrt kann es auch vorkommen, dass man einem Zeiger eine Adresse zugewiesen hat, die nicht überschrieben werden darf.

```
char *const p = "Hallo";   // p ist konstanter Zeiger auf char

p[3]='x';                  // richtig
p="Welt";                  // falsch
```

Das Objekt, auf das der Zeiger verweist, darf verändert werden, nicht aber der Inhalt des Zeigers selbst, also die Adresse des Objektes.

Konstante Zeiger auf Konstanten

```
const char * const p ="Hallo";   // konstanter Zeiger auf konstantes
                                 // char-String
```

Hier darf weder der Zeigerinhalt, noch der Inhalt des referenzierten Objektes verändert werden.

Bemerkung:

Ein Feld entspricht vom Typ her einem konstanten Zeiger.

11.7 Dynamische Datenstrukturen

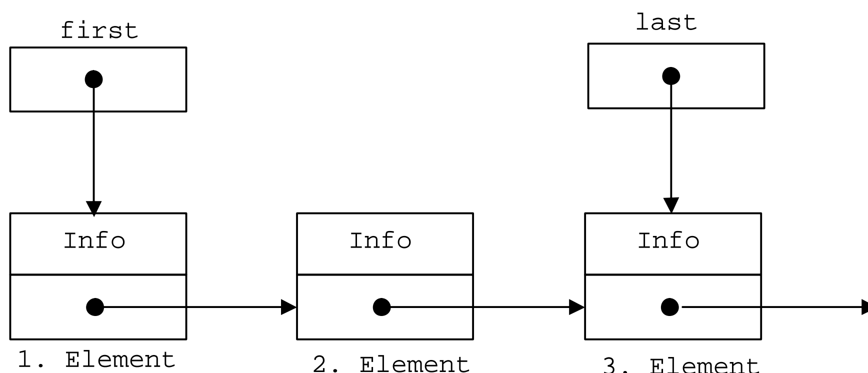
Der Nachteil von Feldern ist, dass die Größe des Feldes bei der Definition bekannt sein muss. Über den Mechanismus des dynamischen Allokierens von Speicher können Felder zwar "vergrößert" werden. Dies ist aber recht ineffizient. Ein häufiges Problem in der Praxis ist, dass Datenmengen im Hauptspeicher effizient zu verwalten sind, deren Größe extrem variieren können. Um dieses Problem zu lösen, setzt man dynamische Datenstrukturen ein.

In der Informatik sind eine Reihe von dynamischen Datenstrukturen bekannt, z. B. einfach- und doppelt verkettete Listen, Binärbäume und eine Reihe anderer Arten von Bäumen.

11.7.1 Einfach verkettete lineare Listen

Einfach verkettete lineare Listen bestehen aus einzelnen Listenelementen. Jedes Listenelement trägt zum einen einen Inhalt, der beliebigen Typs sein kann, sowie einen Vorwärtsverweis, normalerweise einen Zeiger, auf das nächste Listenelement der Liste. Der Vorwärtsverweis des letzten Listenelementes ist leer, also der Nullzeiger.

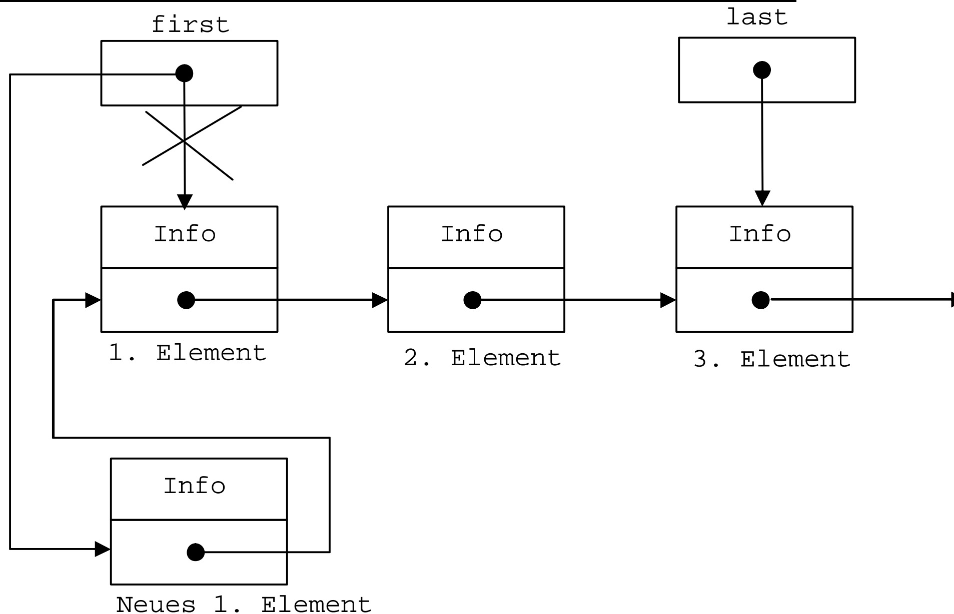
Organisation einer einfach verketteten linearen Liste



Jedes Listenelement besitzt, abgesehen vom ersten und letzten Element, genau einen Vorgänger und genau einen Nachfolger. Es gibt eine Reihe von elementaren Operationen, die man mit linearen Listen leicht durchführen kann.

Um den Zugriff auf die Liste zu organisieren, benötigt man normalerweise nur einen Verweis auf das erste Listenelement. Um das Anfügen und Löschen von Listenelementen am Ende der Liste zu vereinfachen, benutzt man üblicherweise zusätzlich einen Verweis auf das letzte Listenelement.

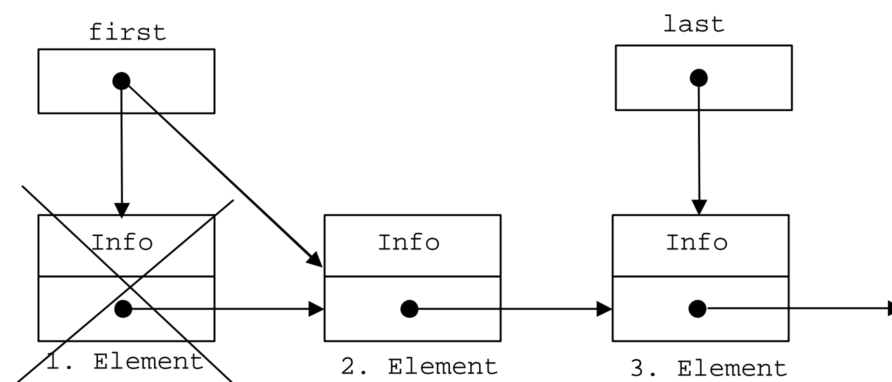
Einfügen eines neuen Elementes am Listenanfang (`push_front`)



Ablauf dieser Operation:

- Lege ein neues Listenelement an mit einem Verweis auf das bisher erste Listenelement
- Lasse den Verweis auf das bisher erste Element auf das neue Element zeigen.

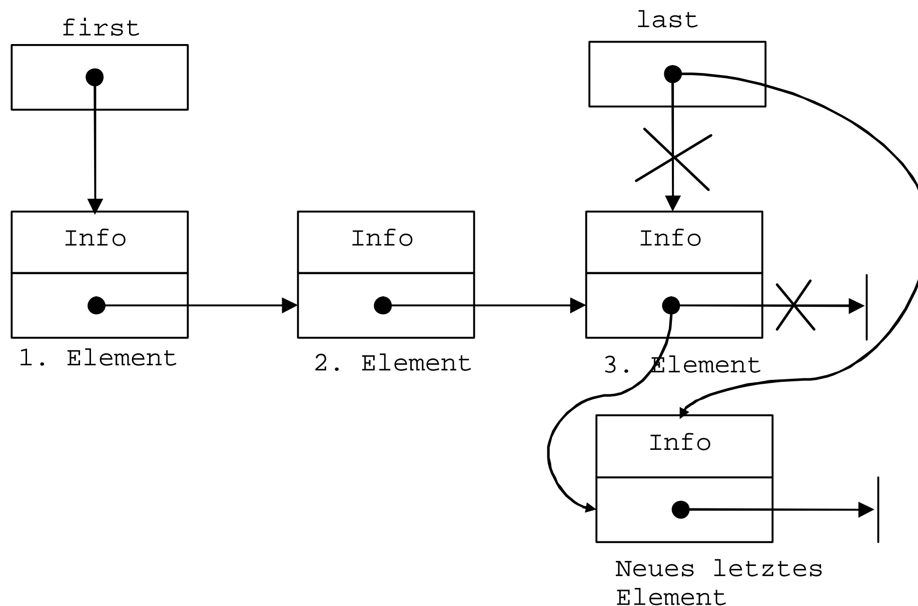
Löschen des ersten Listenelementes (`pop_front`)



Ablauf dieser Operation:

- Lasse den Verweis auf das bisher erste Element auf das zweite Element zeigen.
- Lösche das bisher erste Element

Anhängen eines Listenelementes am Ende der Liste (`push_back`)



Ablauf dieser Operation:

- Lege ein neues Listenelement an mit einem leeren Verweis
- Lasse den Verweis auf das bisher letzte Element auf das neue Element zeigen.
- Lasse den Verweis auf das bisher letzte Element auf das neue Element zeigen.

Implementierung in C++

Datenstruktur für das Listenelement:

```
typedef int InhaltTyp;  // Definiere den Typnamen InhaltTyp

// Datenstruktur fuer ein Listenelement
struct ListElement {
    InhaltTyp inhalt;   // Inhalt des Listenelements
    ListElement* next;  // Vorwärtsverweis
};
```

Bemerkung:
Mit Hilfe von `typedef` besteht in C++ die Möglichkeit, vorhandenen Typen neue Namen zu geben. Dies macht z. B. dann einen Sinn, wenn der Typ einen komplizierten Namen hat.

```
typedef const int* CINTPTR; // CINTPTR ist nun ein anderer Name für
                            // const int*
```

In einer `typedef`-Definition nimmt der neue Datentyp immer die Position eines Variablennamens ein. Es wird aber kein neuer Datentyp definiert, es wird lediglich ein neuer Name für einen bereits existierenden Datentyp eingeführt.

Datenstruktur für die Liste:

```
// Datenstruktur fuer die Organisation der Liste
struct Liste {
    ListElement* first; // Verweis auf das erste Element
    ListElement* last;  // Verweis auf das letzte Element
    int length;         // Anzahl Listenelemente
};
```

Aufbauen einer Liste:

```
// Leere Liste erzeugen
Liste l;
l.first = l.last = 0;
l.length = 0;

// Element anhängen
l.first = new ListElement;
l.last = l.first;
l.first->inhalt = 5;
l.first->next = 0;

// Element einfügen
ListElement* tmp = new ListElement;
tmp->inhalt = 10;
tmp->next = l.first; // zeigt aufs erste Listenelement

l.first = tmp;       // eingehängt
```

Natürlich ist dieses Verfahren etwas zu mühsam. Wir werden für all diese Operationen Funktionen implementieren.

Elementare Listenoperationen

```
// Erzeugen eines Listenelementes
ListElement* createListElement(InhaltTyp t, ListElement* p) {
    ListElement* tmp = new ListElement;
    tmp->inhalt = t;
    tmp->next = p;
    return tmp;
}

// Einfuegen am Anfang der Liste
void push_front(Liste& l, InhaltTyp t) {
    l.first = createListElement(t, l.first);
    if (l.last == 0)     // war die Liste leer?
        l.last = l.first;
    l.length++;
}

// Loeschen des ersten Elementes
void pop_front (Liste& l) {
    if (l.length == 0) return;  // Liste leer ?
    if (l.first == l.last) {    // Nur ein Listenelement
        delete l.first;
        l.first = l.last = 0;  // danach ist die Liste leer
    } else {
        ListElement* tmp = l.first;  // erstes merken
        l.first = l.first->next;     // first auf das zweite Element
        delete tmp;                  // altes Erstes loeschen
   }
   l.length--;
}

// Anfuegen am Listenende
void push_back (Liste& l, InhaltTyp t) {
    if (l.first == 0) {  // leere Liste
        l.first = createListElement(t);
        l.last = l.first;
    } else {
        l.last->next = createListElement(t);
        l.last = l.last->next;
   }
   l.length++;
```

```
}

// Loeschen des letzten Listenelementes
void pop_back  (Liste& l) {
    if (l.length == 0) return; // leere Liste ?
    if (l.first == l.last) {    // Nur ein Listenelement
        delete l.first;
        l.first = l.last = 0;
    } else {
        ListElement* tmp = l.first;
        // vorletztes Element suchen
        while (tmp->next != l.last)
            tmp = tmp->next;
        l.last = tmp;// tmp zeigt nun auf das vorletzte Listenelement
        delete l.last->next;   // letztes loeschen
        l.last->next = 0;
    }
   l.length--;
}
```

Ausgeben des Listeninhaltes

Beim Ausgeben des Listeninhaltes muss ein `ListElement`-Zeiger sich durch die Liste durch"hangeln" bis der Vorwärtszeiger den Wert 0 hat.

```
// Ausgeben des Listeninhaltes
void print (Liste& l) {
    ListElement* tmp = l.first;
    while (tmp != 0) {
        cout << tmp->inhalt << '\t';
        tmp = tmp->next;
    }
    cout << endl;
}
```

Löschen der kompletten Liste

Da alle Listenelemente dynamisch allokiert wurden, ist das Löschen der Liste nicht so trivial. Jedes Element muss einzeln gelöscht werden. Dazu muss man ebenfalls mit einem zeiger durch die Liste laufen.

```
// Alle Listenelemente loeschen
void eraseAll(Liste& l) {
    ListElement *p, *q;
    p = l.first;
    while (p != 0) {
        cout << p->inhalt << "\t";
        q = p;
        p = p->next;
        delete q;
    }
    cout << endl;
    l.first = l.last = 0;
    l.length = 0;
}
```

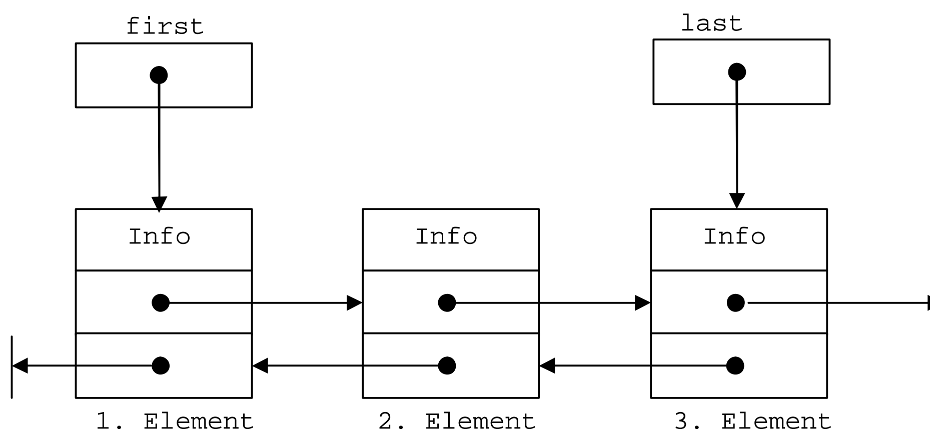
Es ist eine ganze Reihe an weiteren Operationen der Liste denkbar, wie z. B. das Einfügen oder Löschen eines Elementes an einer beliebigen Stelle der Liste oder das Suchen eines bestimmten Wertes in der Liste usw. Viele dieser Operationen sind aber einfacher zu realisieren mit doppelt verketteten Listen.

11.7.2 Doppelt verkettete lineare Listen

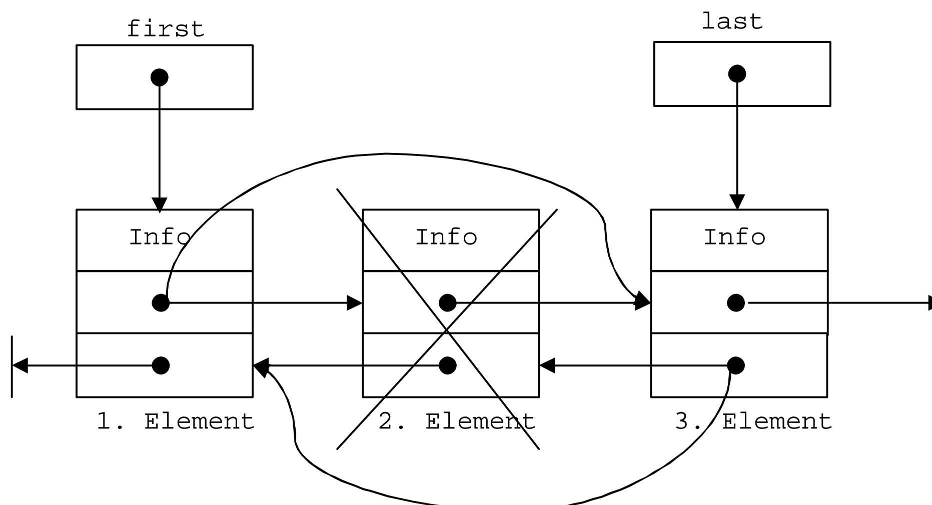
Bei doppelt verketteten Listen hat jedes Listenelement zusätzlich zum Vorwärtszeiger auf das nächste Listenelement auch noch einen Rückwärtszeiger, der auf das voorhergehende lement zeigt. Beim erstem Listenelement ist dann der Rückwärtszeiger ein Nullzeiger und beim letzten Listenelement ist der Vorwärtszeiger ein Nullzeiger.

Vorteile von doppelt verketteten Listen sind natürlich, dass man auch rückwärts durch die Liste laufen kann. Außerdem ist das Löschen von Listenelementen in der Liste wesentlich einfacher als bei einfach verketteten Listen. Ein Nachteil ist jedoch, dass man bei Listenoperationen einen wesentlich höheren Verwaltungsaufwand hat, da schließlich immer beide Zeiger korrekt zu versorgen sind.

Organisation einer doppelt verketteten linearen Liste



Löschen in einer doppelt verketteten Liste



Ablauf der Operation: Lösche i-tes Element

- Lasse den Vorwärtszeiger des (i-1)-ten Elementes auf das (i+1)-te Element zeigen
- Lasse den Rückwärtszeiger des (i+1)-ten Elementes auf das (i-1)-te Element zeigen
- Lösche das i-te Element
- Zu berücksichtigen sind natürlich Sonderfälle wie das Löschen des ersten bzw. letzten Elementes.

Datenstruktur für die Liste:

```
// linlist2.h: Typdefinitionen der doppelt verketteten linearen Liste

typedef int InhaltTyp;   // Definiere den Typnamen InhaltTyp

// Datenstruktur fuer ein Listenelement
struct DListElement {
    InhaltTyp inhalt;      // Inhalt des Listenelements
    ListElement* next;     // Vorwärtsverweis
    ListElement* previous; // Rückwärtsverweis
};

// Datenstruktur fuer die Organisation der Liste
struct DListe {
    DListElement* first; // Verweis auf das erste Element
    DListElement* last;  // Verweis auf das letzte Element
    int length;          // Anzahl Listenelemente
};
```

Die Umsetzung der einzelnen Operationen sind Aufgabe in der aktuellen Übung.

12 Klassen

12.1 Aufbau von Klassen

Klassendefinition

```
class Klassenname {
public:
   // oeffentliche Merkmale

protected:
   // geschuetzte Merkmale

private:
   // private (geheime) Merkmale
}; // durch Semikolon beendet
```

- Durch eine solche Definition wird ein Datentyp mit dem Namen "Klassenname" definiert.
- Die Angaben `public`, `protected`, `private` sind jeweils optional und können beliebig oft und in beliebiger Reihenfolge vorkommen.

`public` Alle nach außen exportierten Merkmale. Eigentlicher Schnittstellenteil, sollte normalerweise nur aus Elementfunktionen bestehen. Attribute sollten immer geschützt oder privat sein.

`protected` Ein geschütztes Merkmal kann nur verwendet werden von:
- klasseneigenen Methoden,
- `friend`-Klassen und `friend`-Funktionen,
- Methoden von abgeleiteten Klassen.

private Ein privates Merkmal kann nur verwendet werden von:
- klasseneigenen Methoden,
- `friend`-Klassen und `friend`-Funktionen,

`private` ist implizit voreingestellt.

```
class X {
   int i;       // automatisch private
public:
   void f();
};
```

Empfehlung: Möglichst die Reihenfolge `public`, `protected`, `private` einhalten.

Bemerkung

`structs` und `unions` sind auch Klassen, allerdings mit dem Unterschied, dass dort die Voreinstellung `public` statt `private` ist.

```
struct X {
   int i;       // automatisch public
public:
   void f();
};
```

Empfehlung:

`struct` und `union` sollten nur bei typischen C-Datenstrukturen verwendet werden und nicht zur Definition von Klassen im objektorientierten Sinn.

Beispiel:

```
// punkt1.cpp: Definition eines Punktes mit X- und Y-Koordinate
//             1. Version

#include <cmath>

class Punkt {
public:
   Punkt ()  { x = 0.0; y = 0.0; }     // Konstruktoren
   Punkt (double, double);

   double getX() { return x; }
   double getY() { return y; }
   void   verschiebe (double, double); // Verschieben eines Punktes
   void   skaliere   (double);         // Skalieren
   double abstand    (Punkt);          // Abstand zu einem anderen Punkt
private:
   double x, y;                        // x- und y-Koordinate
};                                     // Ende des Klassenblocks

// Implementierung der Elementfunktionen
Punkt::Punkt (double xwert, double ywert) {
   x = xwert;
   y = ywert;
}

void Punkt::verschiebe (double a, double b) {
   x += a;
   y += b;
}

void Punkt::skaliere (double faktor) {
   x *= faktor;
   y *= faktor;
}

double Punkt::abstand (Punkt p) {
   return sqrt((x - p.x) * (x - p.x)
              +(y - p.y) * (y - p.y));
}
```

Testprogramm zu `punkt1.cpp`

```
void ausgabePunkt (Punkt p) {
   cout << "x = " << p.getX() << ", y = " << p.getY() << endl;
}

int main() {
   Punkt p1(1.0,2.0);
   Punkt p2 = Punkt(1.0,5.0);
   Punkt* pp = new Punkt(2.1,3.2);  // Dynamisch einen Punkt erzeugen
   ausgabePunkt(p1);
   ausgabePunkt(p2);
   ausgabePunkt(*pp);
   cout << "Abstand zwischen p1 und p2 : " << p1.abstand(p2) << endl;
   cout << "Abstand zwischen p1 und *pp: " << p1.abstand(*pp) << endl;
   delete pp;  // Dynamisches Objekt wieder entfernen
}
```

Attribute und Methoden

- Attribute oder Methoden sind automatisch innerhalb der ganzen Klasse bekannt, auch wenn die Definition bzw. Deklaration erst am Ende des Klassenblocks erfolgt.
- Innerhalb des sogenannten Klassenblocks sind die Attribute üblicherweise unter `private` oder `protected` definiert.

Regel: Nichtkonstante Attribute dürfen nie unter `public` definiert werden !

- Methoden können unter `public` definiert sein (öffentliche Schnittstelle), aber auch unter `private` oder `protected` (Implementierungsfunktion)
- `getX`, `getY`: Methoden, die innerhalb der Klassendefinition angegeben sind, sind dadurch automatisch `inline`-Funktionen.
- `verschiebe`, `skaliere`, `abstand`: innerhalb der Klasse nur deklariert, die eigentliche Definition erfolgt außerhalb der Klasse.

Eine Klasse definiert einen eigenen Datentyp. Betrachte dazu:

```
class X {int i;};
class Y {int j;};    // rein formal gleich strukturiert

X x1, x2;
Y y1;
x1 = x2; // ok
x1 = y1; // nicht erlaubt, da Typkonflikt; nur möglich, wenn explizit
         // eine Typumwandlung zwischen Y und X existiert
```

Konstruktoren

Die Methode `Punkt()` ist ein sogenannter *Konstruktor* und hat als solcher die Aufgabe, Objekte vom Typ `Punkt` zu erzeugen und zu initialisieren.

```
Punkt p1(1.0,2.0);   // Definition eines Punkt-Objektes, dabei wird
                     // der Konstruktor mit den angegebenen Parametern
                     // aufgerufen

Punkt* pp = new Punkt(2.1,3.2);
   // Dynamisches Erzeugen eines Punktes, dabei wird durch new genügend
   // Speicherplatz für die Datenstruktur Punkt allokiert und
   // anschließend der Konstruktor aufgerufen.

delete pp;     // entfernt den dynamisch erzeugten Punkt
```

- Ein Konstruktor ist keine Funktion im eigentlichen Sinne, denn ein Konstruktor hat keinen Rückgabetyp.
- Ein Konstruktor ohne Parameter ist ein sogenannter *Standardkonstruktor (Default-Konstruktor).*
- Ist kein Konstruktor explizit definiert, so wird vom Compiler ein *Standardkonstruktor* erzeugt, der nur das Objekt erzeugt aber nicht initialisiert.
- Konstruktoren können überladen werden, d. h. es kann beliebig viele Konstruktoren einer Klasse geben, die sich jeweils durch Anzahl und Typ der Parameter unterscheiden müssen.

Im Beispiel sind zwei Konstruktoren definiert, ein Standardkonstruktor und ein Konstruktor mit zwei `double`-Parametern. Durch die Verwendung von Standardwerten für die Parameter können beide Konstruktoren zusammengefasst werden.

```
Punkt (double=0.0, double=0.0);
```

Dieser Konstruktor kann mit zwei, mit nur einem oder auch ohne Parameter aufgerufen werden.

```
Punkt p1(1.0, 2.0);  // x = 1.0, y = 2.0
Punkt p2(5.0);       // x = 5.0, y = 0.0
Punkt p3;            // x = 0.0; y = 0.0
```

12.2 Elementfunktionen

Die Elementfunktionen, auch *Methoden* oder *Operationen* genannt, einer Klasse werden innerhalb des Klassenblocks deklariert bzw. bereits komplett definiert.

Eine Deklaration besteht aus einem Funktionsprototyp:

```
void  verschiebe (double, double);
```

oder die Funktion wird direkt im Klassenblock definiert:

```
double getY() { return y; }
```

Bei vollständiger Definition innerhalb des Klassenblocks sind Elementfunktionen automatisch *inline-Funktionen.*

Empfehlung: Größere Elementfunktionen sollten aus Gründen der Lesbarkeit immer außerhalb des Klassenblocks definiert werden.

Definition einer Elementfunktion außerhalb des Klassenblocks:

```
Typ klassenname::funktionsname(.....)
{.......};
```

```
void Punkt::verschiebe (double a, double b) {
    x += a;
    y += b;
}

inline void Punkt::skaliere (double faktor) {
    x *= faktor;
    y *= faktor;
}
```

- Elementfunktionen haben freien Zugriff auf alle `public`, `protected` und `private`-Merkmale ihrer Klasse. Globale Funktionen haben nur Zugriff auf `public`-Merkmale.
- Methoden können wie globale Funktionen innerhalb einer Klasse überladen werden. Auch hier dient zur Unterscheidung die Signatur der Funktion.
- Methoden mit gleicher Signatur aber verschiedener Klassenzugehörigkeit sind verschieden.

Implementierung von Klassen

Aufteilung in Klassendefinition und Klassenimplementierung.

Headerdatei (z.B.: `Punkt.h`)

- Klassendefintion (`class Punkt {.....}`)
- `inline`-Elementfunktionen
- Aufzählungstypen, Konstanten, Makros
- `include`-Anweisungen

- auf keinen Fall: Implementierung von nicht-`inline`-Elementfunktionen

Implementierungsdatei (z.B.: `Punkt.cc, Punkt.cpp`)

- Inkludieren der Headerdatei
- Implementierung der nicht-`inline`-Elementfunktionen
- `friend`-Funktionen
- wird zu `Punkt.o` übersetzt

Vorwärtsdeklaration oder vollständige Klassendeklaration:

Wenn die Klasse `Punkt` in einem Programm verwendet werden soll, so gibt es prinzipiell 2 Möglichkeiten:

`#include "Punkt.h"` : die komplette Klassenschnittstelle ist zugänglich

- Vorwärtsdeklaration: `class Punkt;`
  Es ist nur erlaubt, Pointer oder Referenzen auf Punkt-Objekte zu definieren,
  z.B.: `Punkt *PP;`
  Beim Übersetzen großer Programmsysteme bringt die Vorwärtsdeklaration Zeitvorteile beim Binden

Zusätzlich ist natürlich das Bindemodul `Punkt.o` zu dem Programm dazuzubinden.

Wir betrachten das in unserem Studenten-Beispiel:

Beispiel:

```
// student2.h: 2. Version der Klasse Student
//             Headerdatei

#ifndef _STUDENT2_H
#define _STUDENT2_H

#include <iostream>
#include <string>

using namespace std;

class Student {
public:
    // Konstruktoren
    Student();
    Student(int mnr, string n, string vn);
    void init()                 { matrikelNr = 0;       }

    // Inline-Elementfunktionen
    void setMatrikelNr (int m)  { matrikelNr = m;  }
    void setName (string n)     { name = n;        }
    void setVorname (string vn) { vorname = vn;         }

    int    getMatrikelNr() { return matrikelNr;    }
    string getName()       { return name;          }
    string getVorname()          { return vorname; }

    // sonstige Elementfunktionen
    void eingabe();
    void ausgabe();

private:
    int matrikelNr;
```

```
    string name;
    string vorname;
};

#endif
```

```
// student2.cpp: 2. Version der Klasse Student
//               Implementierungsdatei

#include "student2.h"

// Konstruktoren
Student::Student() {
    matrikelNr = 0;
    // name und vorname sind automatisch initialisiert
}

Student::Student(int mnr, string n, string vn) {
    matrikelNr = mnr;
    name = n;
    vorname = vn;
}

// Elementfunktionen
void Student::eingabe() {
    cout << "Matrikelnr: "; cin >> matrikelNr;
    cout << "Name: "; cin >> name;
    cout << "Vorname: "; cin >> vorname;
}

void Student::ausgabe() {
    cout << matrikelNr << '\t'
         << name       << ", "
         << vorname    << endl;
}
```

```
// student2t.h: 2. Version der Klasse Student
//              Testprogramm


#include "student2.h"

int main() {
    Student meier;
    meier.eingabe();
    meier.ausgabe();
    meier.setVorname("Sepp");
    meier.ausgabe();
    Student* mueller = new Student(7654321, "Mueller", "Gerd");
    mueller->ausgabe();
    delete mueller;
}
/*
Matrikelnr: 1234567
Name: Maier
Vorname: Hans
1234567 Maier, Hans
1234567 Maier, Sepp
7654321 Mueller, Gerd
*/
```

12.3 Der this-Zeiger

Implizit wird jeder Elementfunktion als erstes Argument ein Zeiger auf das zugehörige Objekt übergeben. Auf dieses unsichtbare Argument kann mittels des Schlüsselwortes `this` zugegriffen werden.

```
void Punkt::skaliere (double faktor) {
    x *= faktor;     // bedeutet eigentlich this->x *= faktor
    y *= faktor;     // bedeutet eigentlich this->y *= faktor
}
```

Implizit ist `this` als

```
Punkt *const this;
```

definiert. Der Zeigerwert kann also nicht verändert werden.

Beispiel:

```
class X{
public:
   X(int i) {
      this->i = i;
   }
private:
   int i;
};
```

Bemerkung:

Es gibt häufig Anwendungen, wo eine Elementfunktion einen Zeiger oder eine Referenz auf das aktuelle Objekt selbst zurückgeben muss.

```
return *this;  // Objektinhalt bzw. Referenz auf das Objekt
return this;   // Adresse des Objektes
```

Wichtig ist dies z. B. beim "Verpointern" von Listen und Bäumen.

12.4 const-Elementfunktion

So wie konstante Größen einfachen Typs definiert werden können, kann man natürlich auch konstante Objekte definieren:

```
const Punkt P1 (1.0, 5.0);
```

Wie sieht das dann aber mit dem Aufruf von Elementfunktionen aus? Da Elementfunktionen ein Objekt verändern können, darf das nicht ohne weiteres erlaubt werden.

Lösung: Elementfunktionen, die das Objekt nicht verändern, werden mit dem Zusatz `const` gekennzeichnet.

Beispiel

```
// punkt2.cpp: Definition eines Punktes mit X- und Y-Koordinate
//          - zusaetzlich const-Elementfunktionen

class Punkt {
public:
   Punkt (double=0.0, double=0.0);     // Konstruktor
   double getX() const { return x; }
```

```
   double getY() const { return y; }
   void   verschiebe (double, double); // Verschieben eines Punktes
   void   skaliere   (double);          // Skalieren
   double abstand    (Punkt) const;     // Abstand zu einem anderen Punkt
private:
   double x, y;                         // x- und y-Koordinate
};

double Punkt::abstand (Punkt p) const {
   return sqrt((x - p.x) * (x - p.x)
               +(y - p.y) * (y - p.y));
}
```

- Bei einem konstanten Objekt dürfen nur `const`-Elementfunktionen aufgerufen werden.
- Das Schlüsselwort `const` steht zwischen der Argumentliste und dem Funktionsblock.
- `const` muss sowohl im Funktionsprototyp, als auch bei der Funktionsimplementierung stehen.
- Der Zusatz `const` gehört zur Signatur einer Elementfunktion. Beim Überladen von Elementfunktionen wird also unterschieden zwischen
  - Funktionsname
  - Anzahl und Typ der Parameter
  - `const`-Zusatz

  Es kann also zwei Elementfunktionen mit gleichem Namen, gleicher Parameterzahl und gleichem Parametertyp geben, die sich nur im `const`-Zusatz unterscheiden.
- Wird eine Elementfunktion als `const` deklariert, die ein Datenelement modifiziert, so wird dies bereits vom Compiler abgelehnt.
  Woran erkennt er das ?
  → statt `Punkt* const this` wird `const Punkt *const this` übergeben

- Die Überprüfung durch den Compiler hat allerdings Grenzen, betrachte z. B.

  ```
  class X{ char *str; ... };
  ```

  In einer `const`-Elementfunktionen wäre es möglich, den Inhalt des char-Strings, auf den `str` zeigt zu verändern, ohne, dass dies vom Compiler überprüfbar wäre.

Regel: `const` soll bei Elementfunktionen hinzugefügt werden, die den Objektinhalt nicht verändern.

Erweiterung im ANSI-Standard

Es gibt Situationen, wo auch `const`-Elementfunktionen bestimmte Attribute verändern können sollten, z. B. wenn die Attribute für die Konstantheit des Objektes nicht relevant sind. Zu diesem Zweck wurde das Schlüsselwort `mutable` (=veränderlich) eingeführt.

Beispiel:

```
class Daten {
public:
   Daten() {anzahl=0;}
   void zeigeDaten() const {anzahl++,.....}
private:
   mutable int anzahl;     // anzahl zaehlt nur die Häufigkeit des
   ...                     // Aufrufs von zeigeDaten() ist aber für
};                         // die Konstantheit des Objekts nicht relevant
```

`mutable`-Attribute dürfen von `const`-Elementfunktionen verändert werden.

Bemerkung: Ältere Compiler kennen das Schlüsselwort `mutable` noch nicht !

Beispiel:

```
// doublestack2.h: Klasse fuer double-Stack
// 2.Version: Festes Implementierungsarray

#include <iostream>
#include <cassert>

const int MAX_ANZ_ELEMENTE = 100;  // der Einfachheit halber noch global

class DoubleStack {
public:
   DoubleStack()            { anzElemente = 0; }
   bool empty () const      { return (anzElemente == 0); }
   bool full  () const      { return (MAX_ANZ_ELEMENTE <= anzElemente); }
   void push  (double x);  // Wert in Stack ablegen
   void pop   ();          // Wert aus Stack herausholen
   double top () const;    // Zuletzt abgelegter Wert
   int size   () const      { return anzElemente; }
   void print() const;     // Ausgabe auf cout (nur fuer Testzwecke)
private:
   int anzElemente;    // Anzahl Elemente des Stacks
   double tab[MAX_ANZ_ELEMENTE]; // Implementierungs-Array
};
```

```
// doublestack2.cpp: Klasse fuer double-Stack
// 2.Version: Festes Implementierungsarray

#include "doublestack2.h"

void DoubleStack::push (double x) {
   assert(!full());
   tab[anzElemente++] = x;
}

void DoubleStack::pop() {
   assert(!empty());
   anzElemente--;
}

double DoubleStack::top() const {
   assert(!empty());
   return tab[anzElemente - 1];
}

void DoubleStack::print() const {
   for (int i = anzElemente - 1; i >= 0; --i)
      cout << tab[i] << "   ";
   cout << endl;
}
```

Nachteile dieses Beispiel:

- Verwendung des Literals 100
- Festes Implementierungsarray

Interaktives Testprogramm zur Klasse `DoubleStack`

```
// doublestack2t.cpp: Testprogramm fuer double-Stack 2. Version

#include "doublestack2.h"

enum FunktionsTyp { beenden, push, pop, top, empty, full, size};

void menue() {
   cout << push   << ": push / "
        << pop    << ": pop / "
        << top    << ": top / "
        << empty  << ": empty / "
        << full   << ": full / "
        << size   << ": size / "
        << beenden<< ": beenden ->";
}

int main() {
   DoubleStack s1;
   int funktion;
   double x;
   do {
      cout << "s1 :"; s1.print();
      menue(); cin >> funktion;
      switch (funktion) {
      case push  :  cout << "Wert: "; cin >> x;
                    s1.push(x);
                    break;
      case pop   :  cout << "pop: " << s1.top() << endl;
                    s1.pop();
                    break;
      case top   :  cout << "top: " << s1.top() << endl;
                    break;
      case empty :  if (s1.empty())
                       cout << "Stack leer\n";
                    else
                       cout << "Stack nicht leer\n";
                    break;
      case full  :  if (s1.full())
                       cout << "Stack voll\n";
                    else
                       cout << "Stack nicht voll\n";
                    break;
      case size   : cout << "size: " << s1.size() << endl;
                    break;
      case beenden: break;
      default     : cout << "Falsche Funktion!\n";
      }
   } while (funktion != beenden);
   return 0;
}
```

Einsatz des assert-Makros

Im obigen Beispiel wurde das Makro `assert` (Headerdatei: `<cassert>`) verwendet. Dieses Makro ist bereits im ANSI-C-Sprachstandard vorgesehen und sieht meist in etwa wie folgt aus:

```
#ifdef NDEBUG
   #define assert(p)  ((void)0)
#else
   #define assert(p)  ((p) ? (void)0 : _assert(#p, __FILE__, __LINE__))
#endif
```

Anwendung:

```
assert (Zusicherung);
```

Eine Zusicherung ist dabei eine Bedingung, die an dieser Programmstelle erfüllt sein muss. Man kann Zusicherungen z.B. dazu verwenden, um am Anfang von Funktionen eine Vorbedingung zu überprüfen, die erfüllt sein muss, um die Funktion korrekt ausführen zu können.

```
void DoubleStack::pop() {
   assert(!empty());
   anzElemente--;
}
```

Was passiert, wenn die Zusicherung verletzt wird?

Dann wird das Programm an der Stelle abgebrochen mit einer Meldung über die verletzte Zusicherung.

```
!empty(), file: doublestack3.cpp, line: 26
```

Vorteil gegenüber einer normalen if-Anweisung:

Wenn das Makro `NDEBUG` definiert ist und das kann über Compiler-Schalter mitgegeben werden, dann werden alle Stellen im Programm, wo ein `assert` steht, durch 0 ersetzt.

12.5 Konstruktoren und Destruktoren

12.5.1 Konstruktoren

- Spezielle Methoden zum Erzeugen und Initialisieren von Klassenobjekten.
- Aufruf bei:
  - Definition von Objekten
  - Übergabe von Objekten als Funktionsargument
  - Rückgabe eines Objektes als Funktionswert
  - bei `new`

Name des Konstruktors gleich *Klassenname*.

Konstruktoren können überladen werden. (Unterscheidung durch Anzahl und Typ der Parameter

Konstruktoren haben keinen Ergebnistyp (`return` ohne Argument ist möglich).

Innerhalb eines Konstruktors können andere Elementfunktionen aufgerufen werden.

Ist kein Konstruktor explizit definiert, so wird vom Compiler ein Standardkonstruktor ohne Parameter erzeugt.

Gibt es bereits Konstruktoren mit Argumenten, so wird nicht automatisch ein Standardkonstruktor erzeugt.

- Konstruktoren werden normalerweise `public` vereinbart.

12.5.2 Destruktoren

- Jede Klasse hat genau einen Destruktor. Dieser Destruktor wird immer dann automatisch aufgerufen, wenn ein Objekt aus dem Speicher entfernt werden soll.
- Wird kein expliziter Destruktor definiert, so erzeugt der Compiler einen Standarddestruktor

- Ein Destruktor ist genau dann explizitzu definieren, wenn innerhalb des Objekts zusätzlich Speicherplatz allokiert wurde.
- Der Destruktor einer Klasse heißt *~Klassenname*

```
class X {
public:
   X() {...};
  ~X() {...};
...
};
```

- Ein Destruktor wird nie explizit aufgerufen (auch wenn manche Compiler das erlauben).
- Der Destruktor sollte immer public sein !

Beispiel:

```
// doublestack3.h: Klasse fuer double-Stack
// 3. Version : Dynamisch erzeugtes Array

#include <iostream>
#include <cassert>

class DoubleStack {
public:
   DoubleStack(int=100);
  ~DoubleStack()        { delete [] tab; }
   bool empty () const     { return (anzElemente == 0); }
   bool full  () const  { return (maxAnzElemente <= anzElemente); }
   void push  (double x); // Wert in Stack ablegen
   void pop   ();         // Wert aus Stack herausholen
   double top () const;   // Zuletzt abgelegter Wert
   int size   () const  { return anzElemente; }
   void print() const;   // Ausgabe auf cout (nur fuer Testzwecke)
private:
   int anzElemente;      // Anzahl Elemente des Stacks
   double *tab;          // Implementierungs-Array
   int maxAnzElemente;   // Größe des Arrays
};

DoubleStack::DoubleStack(int n) {
   assert(n > 0);
   tab = new double[n];
   maxAnzElemente = n;
   anzElemente = 0;
}
```

Lebensdauer von Objekten

Globales Objekt	Der Destruktor wird bei Programmende automatisch aufgerufen
Statisches Objekt	Der Destruktor wird bei Programmende automatisch aufgerufen
Lokales Objekt	Der Destruktor wird bei Verlassen des Blockes, in dem das Objekt definiert wurde automatisch aufgerufen.
Dynamisch erzeugtes Objekt	Der Destruktor wird implizit von `delete` aufgerufen, allerdings nicht automatisch bei Programmende.

Bemerkung:

Bei Abbruch eines Programms mittels `exit()` werden für konstruierte statische und globale Objekte die Destruktoren aufgerufen, bei Abbruch mittels `abort()` jedoch nicht.

Aufgabe

```
#include <iostream>
int main () {
   cout<<"Hallo Welt!\n";
   return 0;
}
```

Wie kann man das Programm so ändern, dass es auf dem Bildschirm folgendes ausgibt:

```
Initialisieren.....
Hallo Welt!
Aufräumen.....
```

aber ohne `main()` zu verändern ?

Das folgende Beispiel stellt eine erste Container-Klasse dar. In der C++-Standardbibliothek gibt es eine Fülle von verschiedenen Arten von Containern, wie. z.B. Listen, Bäume assoziative Arrays.

Beispiel: Eine Klasse für eine `int`-Menge

Methoden:

`IntMenge(int m)`	Konstruktor: Erzeuge eine Menge mit maximal `m` Elementen
`~IntMenge()`	Destruktor: Speicher für die Menge freigeben
`bool istElement(int t)`	Ist `t` ein Element der Menge ?
`void einfuege(int t)`	Füge `t` in die Menge ein *Vorbedingung*: Menge nicht voll
`void loesche(int t)`	Löschen eines Elementes aus der Menge
`bool leer()`	Ist die Menge leer?
`bool voll()`	Ist die Menge voll, d.h. enthält sie genau `m` Elemente?

```
// intmenge1.h: Eine Menge von Integern (1. Version)

#include <cassert>

typedef int Typ;            // Elementtyp

class IntMenge {
public:
   IntMenge (int m);              // Menge von maximal m Elementen
  ~IntMenge();                    // Menge destruieren

   bool istElement(Typ t) const; // Ist t ein Element der Menge ?
   void einfuege  (Typ t);       // t einfuegen in die Menge
   void loesche   (Typ t);        // t loeschen aus der Menge
   bool leer      () const;      // Ist die Menge leer ?
   bool voll      () const;      // Ist die Menge voll ?
   int  anzahl    () const;      // Anzahl Elemente

   // Iterator zur IntMenge
   typedef const Typ* iterator;   // eigener Iteratortyp
   iterator begin() const {  return tab;          }
   iterator end  () const {  return tab + aktAnzahl; }
```

```
private:
   Typ *tab;                     // Implementierung als Integer-Array
   int aktAnzahl, maxAnzahl;     // Aktuelle Anzahl, maximale Anzahl
};

inline bool IntMenge::leer() const
{   return aktAnzahl == 0; }

inline bool IntMenge::voll() const
{   return aktAnzahl >= maxAnzahl; }

inline int IntMenge::anzahl() const
{   return aktAnzahl; }
```

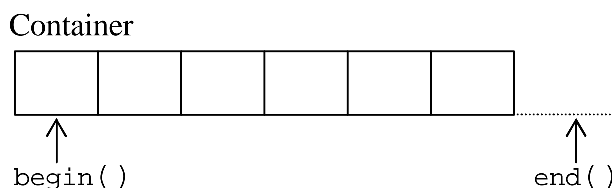
```
// intmenge1.cpp: Implementierung einer int-Menge

#include <iostream>
#include "intmenge1.h"

// Konstruktor: Legt Array mit m Elementen an
// Vorbedingung: M >= 1
IntMenge::IntMenge (int m ) {
   assert(m >=1);
   aktAnzahl = 0;
   maxAnzahl = m;
   tab = new Typ[maxAnzahl];
}

// Destruktor: Entfernen des int-Arrays
IntMenge::~IntMenge() {
   delete [] tab;
}

// Einfuegen eines Elements in die Menge
// Vorbedingung: Menge ist nicht voll
void IntMenge::einfuege(Typ t) {
   assert(!voll());
   if (istElement(t)) return;
   int i = aktAnzahl++;
   Typ tmp;
   tab[i] = t;

   while (i > 0 && tab[i-1] > tab[i])    {
      tmp = tab[i];       // t richtig einsortieren
      tab[i] = tab[i-1];
      tab[i-1] = tmp;
      i--;
   }
}

// Loeschen eines Elements aus der Menge
void IntMenge::loesche(Typ t) {
   int l = 0, r = aktAnzahl - 1;  // linke und rechte Grenze
   int m, i;
   while (l <= r) {
      m = (l + r)/2;
      if (t < tab[m])
         r = m - 1;
      else if (t > tab[m])
         l = m + 1;
      else {
         for (i = m+1; i < aktAnzahl; i++)
            tab[i-1] = tab[i];
         aktAnzahl--;
```

```
         return;
      }
   }
}

// Ist t Element der Menge ?
bool IntMenge::istElement(Typ t) const {
   int l = 0, r = aktAnzahl - 1;  // linke und rechte Grenze
   int m;
   while (l <= r) {
      m = (l + r)/2;
      if (t < tab[m])
         r = m - 1;
      else if (t > tab[m])
         l = m + 1;
      else
         return true;      // gefunden
   }
   return false;     // nicht gefunden
}
```

Zum Durchlaufen der einzelnen Mengenelemente wurde eine spezielle Technik angewandt, sogenannte *Iteratoren*. Diese Technik wird in dieser Art vor allem in der *Standard Template Library (STL)* angewandt. Innerhalb der Klasse wird ein Typ `iterator` definiert, in unserem Beispiel ist das einfach der Typ int-Zeiger.

Spezielle Iterator-Werte

`begin()`: Iterator, der auf das erste Element des Containers zeigt
`end()` : Iterator, der hinter das letzte Element des Containers zeigt, in unserem Beispiel einfach der Wert 0.



Die Ausgabe der Menge kann damit z. B. so erfolgen

```
void ausgabe(IntMenge& m) {
   IntMenge::iterator it;
   for (it = m.begin(); it != m.end(); ++it)
      cout << *it << "  ";
   cout << endl;
}
```

Bemerkung:

In der STL gibt es viele verschiedene Varianten von Iteratoren und eine Fülle von Algorithmen, die mit Hilfe von Iteratoren leicht anwendbar sind. Auch wenn die Anwendung in unserem Beispiel als etwas überdimensioniert erscheint, so ist der Vorteil vor allem darin zu sehen, dass Iteratoren für alle Arten von Containern gleich eingesetzt werden.

Testprogramm zur Klasse `IntMenge`:

```
// intmeng1t: Testprogramm zur Klasse IntMenge
//
#include <iostream>
```

```
#include "intmenge1.cpp"

extern "C" int rand();     // ANSI-C Standardfunktion fuer Pseudo-
                           // Zufallszahlen

// Zufallszahl aus dem Intervall 0..u
int randint (int u) {
   int r = rand();
   if (r < 0) r = -r;
   return 1 + r%u;
}

// Menge ausgeben
void ausgabe(IntMenge& m) {
   IntMenge::iterator it;
   for (it = m.begin(); it != m.end(); ++it)
      cout << *it << "  ";
   cout << endl;
}

enum FunktionsTyp { beenden, einfuege, istElement, loesche, ausgeben};

void menue() {
   cout << einfuege   << ": einfuegen / "
        << istElement << ": ist Element / "
        << loesche    << ": loeschen / "
        << ausgeben   << ": ausgeben / "
        << beenden    << ": beenden ";
}

int main() {
   const int MAX=1000;
   int anzahl;
   cout << "Maximale Anzahl ? "; cin >> anzahl;

   IntMenge m(anzahl);
   while (!m.voll()) {
      m.einfuege(randint(MAX)); // Zufallszahl einfuegen
   }
   cout << "Ausgabe der Menge: \n"; ausgabe(m);

// Interaktiver Testrahmen
   int funktion;
   Typ t;
   do {
      menue(); cin >> funktion;
      switch (funktion) {
      case einfuege  :  cout << "Wert: "; cin >> t;
                        m.einfuege(t);
                        break;
      case istElement:  cout << "Wert: "; cin >> t;
                        if (m.istElement(t))
                           cout << t << " enthalten !\n";
                        else
                           cout << t << " nicht enthalten !\n";
                        break;
      case loesche   :  cout << "Wert: "; cin >> t;
                        m.loesche(t);
                        break;
      case ausgeben  :  ausgabe(m);
                        break;
      case beenden   :  break;
      default        :  cout << "Falsche Funktion!\n";
      }
   } while (funktion != beenden);
```

```
   return 0;
}
```

12.6 Initialisierung von Attributen

Bisher wurde noch nicht besprochen, wie z.B. Attribute, die als `const` oder als Referenz definiert sind, initialisiert werden. Ebenso ist noch unklar, wie z.B. ein Attribut, das selbst ein Objekt einer Klasse ist, konstruiert wird.

12.6.1 Initialisierung von Elementobjekten

Objekte können wiederum Objekte von anderen Klassen enthalten, aber auch Zeiger und Referenzen auf Objekte.

Beispiel: Elementobjekte

```
//strecke.cpp: Definition einer Strecke im 2-dimensionalen Raum

#include "punkt3.h"

class Strecke {
public:
   Strecke (double x1, double y1, double x2, double y2);
   Punkt getP() const { return p; }
   Punkt getQ() const { return q; }
   void verschiebe (double, double);
   double laenge(){ return p.abstand(q); }
private:
   Punkt p, q;
};

Strecke::Strecke (double x1, double y1, double x2, double y2)
    : p(x1,y1), q(x2,y2) {}

void Strecke::verschiebe (double a, double b){
   p.verschiebe(a,b);
   q.verschiebe(a,b);
}
```

```
// strecket.cpp: Testprogramm fuer Strecke

#include <iostream>
#include "strecke.h"

void ausgabePunkt (Punkt p) {
   cout << "x = " << p.getX() << ", y = " << p.getY() << endl;
}

void ausgabeStrecke (Strecke s){
    cout << "p: "; ausgabePunkt(s.getP());
    cout << "q: "; ausgabePunkt(s.getQ());
    cout << "Länge: " << s.laenge() << endl;
}

int main(){
    Punkt p1(1.0,2.0);
    const Punkt p2(1.0,5.0);
    Strecke s(1.0,2.0,2.0,3.0);
    ausgabeStrecke(s);
    s.verschiebe(2.0,3.0);
    ausgabeStrecke(s);
    return 0;
```

```
}
```

Problem: Konstruktion von Elementobjekten.
Ist ein Attribut selbst ein Objekt einer Klasse, so muss dieses Elementobjekt bereits fertig konstruiert sein, wenn die erste Anweisung des Konstruktorrumpfes durchlaufen wird.

Lösung: Elementinitialisierungsliste

```
Strecke::Strecke (...) : p(x1,y1), q(x2,y2) {}
```

###
Initialisierung von Elementobjekten

Mit Hilfe der Elementinitialisierungsliste (oder auch Initialisierungsliste) können Attribute eines Objektes initialisiert werden, bevor der eigentliche Konstruktorrumpf durchlaufen wird.

Die Elementinitialisierungsliste wird bei der Definition des Konstruktors angegeben, nicht bei der Deklaration. Sie steht durch ":" getrennt hinter der Argumentliste

Jedes Attribut darf nur einmal in der Elementinitialisierungsliste auftauchen.

Es können Attribute jeden beliebigen Typs angegeben werden

```
class X {
public:
   X (int j, double z) : i(j), y(z) {}
private:
   int i;
   double y;
};
```

Sind die Attribute von elementarem Datentyp oder Zeigertyp, so werden sie einfach initialisiert (ähnlich der Initialisierung bei der Definition einer Variablen). Ist ein Attribut ein Objekt, so wird in der Initialisierungsliste ein Konstruktor der zugehörigen Klasse aufgerufen und ggf. mit Parametern versorgt.

Besitzt ein Elementobjekt einen Standardkonstruktor, so kann der Konstruktoraufruf in der Initialisierungsliste weggelassen werden.

Ablauf der Konstruktion: Zunächst werden die Initialisierungen in der Reihenfolge durchgeführt, in der die Elemente in der Klassendeklaration deklariert wurden, dann erst der Konstruktorrumpf.

Regel: Die Initialisierungsliste ist immer in der Reihenfolge zu erstellen, die der Deklarationsreihenfolge innerhalb der Klasse entspricht.

Ablauf der Destruktion: wird bei dem übergeordneten Objekt der Destruktor aufgerufen, so werden zunächst die Destruktoren der Elementobjekte aufgerufen und zwar in genau der umgekehrten Reihenfolge wie beim Konstruktor.

12.6.2 Initialisierung von Konstanten und Referenzen

Attribute können auch als `const` deklariert sein oder als Referenz. Wie bekannt, müssen Konstanten und Referenzen bei der Definition initialisiert werden

Beispiel:

```
// constref.cpp: Initialisierung von Konstanten und Referenzen
```

```
class ConstRef {
public:
   ConstRef (int ii);
private:
   int i;
   const int ci;
   int & ri;
};

ConstRef::ConstRef (int ii) {
   i = ii;
   ci = ii;    // falsch !!
   ri = i;     // falsch !!
}
```

besser:

```
ConstRef::ConstRef (int ii): ci(ii), ri(i)
{   i = ii;  }
```

Konstanten und Referenzen müssen in derElementinitialisierungsliste initialisiert werden.

Bemerkung: Initialisieren ist meistens schneller als Zuweisen, d. h. im Allgemeinen dem Vorbelegen von Attributen durch Zuweisungen vorzuziehen.

12.6.3 Objekt-Arrays

Beispiel: `Punkt ptab[100];`

Wie erfolgt die Konstruktion der einzelnen Elemente ?
Wie können den Konstruktoren jeweils Parameter übergeben werden ?

Antwort: Arrays von Objekten können nur definiert werden, wenn die Klasse einen Standardkonstruktor besitzt, das heißt einen Konstruktor, der ohne Parameter auskommt.

Bei der Definition eines Objekt-Arrays wird für jedes Array-Element automatisch der Standardkonstruktor aufgerufen.

Ebenso ist folgendes möglich:

```
Punkt *pp = new Punkt[100];
   .
   .
delete [] pp;
```

12.7 Kopierkonstruktor und Zuweisungsoperator

Objekte können mit Hilfe eines anderen Objektes konstruiert werden. Ebenso können Objekten Objekte des gleichen Typs zugewiesen werden. Was passiert dabei eigentlich genau?

Beispiel:

```
Punkt p1(3.0,5.0);
Punkt p2(p1);        // impliziter Aufruf des Kopierkonstruktors
Punkt p3 = p1;       //      "        "      "        "
Punkt p4;
p4 = p1;             // impliziter Aufruf des Zuweisungsoperators
```

Eine Klasse besitzt immer einen sogenannten Kopierkonstruktor, der dazu da ist, ein Objekt zu konstruieren und mit Hilfe eines zweiten Objekts zu initalisieren. Dieser wird üblicherweise automatisch durch den Compiler erzeugt. Es gibt allerdings auch Situationen, wo es notwendig ist, den Kopierkonstruktor explizit zu implementieren.

Definition des Kopierkonstruktors:

```
Punkt::Punkt (const Punkt & p);
```

Aufruf des Kopierkonstruktors

```
Punkt p2(p1);
```

Wirkungsweise: `p2` wird erzeugt und mit Hilfe der Attribute von `p1` elementweise initalisiert. Dabei wird bei Elementobjekten jeweils wiederum der zugehörige Kopierkonstruktor aufgerufen.

In folgenden Situationen erfolgt der Aufruf des Kopierkonstruktors:

1. Explizite Initialisierung bei der Definition

```
Punkt p2(p1);
Punkt p3 = p1;
```

2. Übergabe eines Objekts als Funktionsargument

```
void f (Punkt P);
```

Beim Aufruf wird der Kopierkonstruktor aufgerufen, um eine lokale Kopie anzulegen

3. Rückgabe eines Objekts als Funktionswert

```
Punkt schnittpunkt (Strecke s1, Strecke s2) {
   Punkt p;
   ...
   return p;   // Kopie von p erzeugen
}
```

Wann ist es notwendig, einen Kopierkonstruktor explizit selbst zu definieren ?

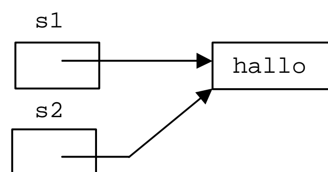
Beispiel:

```
// string1.cpp: Eine einfache String-Klasse

#include <cstring>
#include <iostream>

class String {
public:
   String (char *s=0);
  ~String ();

   const char* c_str() { return str; }
// weitere Funktionen ...

private:
   char *str;
};

String::String(char *s){
   if (s == 0){
      str = new char[1];
```

```
        *str = '\0';
    } else {
        str = new char[strlen(s)+1];
        strcpy(str,s);
    }
}

String::~String () { delete [] str; }

int main(){
    String s1="Hallo";
    String s2("Welt!");

    cout << s1.c_str() << ", " << s2.c_str() << endl;
    return 0;
}
```

Beispiel:

```
String s1 ("Hallo");
String s2 = s1
```



s2 ist eine flache Kopie von s1, denn der interne Zeiger `str` zeigt jeweils auf die selbe Adresse.

Betrachte die folgende Situation:

```
void tue_nichts ( String s) {}
tue_nichts (s1);
cout << s1.c_str()<<endl;
```

Was passiert ?

- Beim Funktionsaufruf von `tue_nichts` wird eine lokale Kopie von `s1` erzeugt, die auf dasselbe `char`-String verweist wie `s1.str`.
- Beim Verlassen der Funktion wird das `char`-string mit `delete[]` freigegeben !

Konsequenz: Der Kopierkonstruktor ist explizit zu definieren !

Betrachte nun die folgende Situation:

```
String s1 = "Hallo", s2;
s2 = s1;
```

s2 existiert bereits, deshalb wird nicht der Kopierkonstruktor aufgerufen, sondern der sogenannte Zuweisungsoperator, d. h. `s2 = s1` ist in Wirklichkeit `s2.operator=(s1)`.

Der Compiler erzeugt auch automatisch einen Zuweisungsoperator für die Klasse. Der Zuweisungsoperator weist ein Objekt einem anderen elementweise zu, kopiert also flach. Dabei wird für Elementobjekte jeweils deren Zuweisungsoperator aufgerufen. Auch der Zuweisungsoperator muss expliziert definiert werden, wenn innerhalb des Objektes Speicher allokiert wird.

Definition des Zuweisungsoperators:

```
String& String::operator=(const String & s);
```

Beispiel:

```
// string2.cpp:Eine einfache String-Klasse erweitert um
// Kopierkonstruktor und Zuweisungsoperator

#include <cstring>
#include <iostream>

class String {
public:
   String (char *s=0);
   String (const String& s);
  ~String ();

   String& operator=(const String& s);
   const char* c_str() { return str; }
// weitere Funktionen ...

private:
    char *str;
};

// Kopierkonstruktor
String::String(const String& s) {
    str = new char[strlen(s.str)+1];
    strcpy(str,s.str);
}

// Zuweisungsoperator
String& String::operator=(const String& s) {
   if (this == &s)
      return *this;
   if (strlen(str) != strlen(s.str)) {
      delete [] str;
      str = new char[strlen(s.str)+1];
   }
   strcpy(str,s.str);
   return *this;
}
```

Regel: Bei Klassen mit dynamisch allokiertem Speicherplatz sind Kopierkonstruktor und Zuweisungsoperator explizit zu definieren!

Variante: Wenn Kopierkonstruktor und/oder Zuweisungsoperator nicht erwünscht sind, so genügt es, sie unter `private` zu deklarieren. Dadurch ist ein Aufruf von außerhalb der Klasse nicht möglich!

Beispiel:

```
// doublestack4.h: Klasse fuer double-Stack
// 4. Version : Dynamisch erzeugtes Array,
//              Kopierkonstruktor, Zuweisungsoperator

class DoubleStack {
public:
   DoubleStack(int=100);
   DoubleStack(const DoubleStack&);                // Kopierkonstruktor
   DoubleStack& operator= (const DoubleStack&);    // Zuweisungsoperator
  ~DoubleStack()           { delete [] tab; }

   ...
private:
```

```
   int anzElemente;        // Anzahl Elemente des Stacks
   double *tab;            // Implementierungs-Array
   int maxAnzElemente;     // Größe des Arrays
   double* tab_copy(double*, int); // Kopieren des internen Arrays
};

// Kopieren des Double-Arrays
double* DoubleStack::tab_copy(double* t, int new_size) {
   double *dp = tab;
   if (maxAnzElemente != new_size)
     // nur neu allokieren bei anderer Groesse
      dp = new double[new_size];
   maxAnzElemente = new_size;
   for (int i = 0; i < maxAnzElemente; i++)
      dp[i] = t[i];
   return dp;
}

// Kopierkonstruktor
DoubleStack::DoubleStack(const DoubleStack& s) : maxAnzElemente(0) {
   tab = tab_copy (s.tab, s.maxAnzElemente);
   anzElemente = s.anzElemente;
}

// Zuweisungsoperator
DoubleStack& DoubleStack::operator= (const DoubleStack& s) {
   if (this == &s) // Ist dies eine Zuweisung an mich selbst ?
      return *this;
   if (maxAnzElemente != s.maxAnzElemente)
      // nur neu allokieren bei anderer Groesse
      delete [] tab;
   tab = tab_copy (s.tab, s.maxAnzElemente);
   anzElemente = s.anzElemente;
   return *this;
}
```

<u>Beispiel:</u>

```
// intmenge2.h: Eine Menge von Integern (2. Version)
//          Erweitert um Kopierkonstruktor und Zuweisungsoperator

typedef int Typ;            // Elementtyp

class IntMenge {
public:
   IntMenge (int m);                  // Menge von maximal m Elementen
   IntMenge (const IntMenge&);        // Kopierkonstruktor
   ~IntMenge();                       // Menge destruieren
   IntMenge& operator= (const IntMenge&); // Zuweisungsoperator
....

// Kopierkonstruktor
IntMenge::IntMenge (const IntMenge& m)
   : aktAnzahl(m.aktAnzahl),
     maxAnzahl(m.maxAnzahl)
{
   tab = new Typ[maxAnzahl];
   for (int i=0; i < maxAnzahl; i++)
      tab[i] = m.tab[i];
}
// Zuweisungsoperator
IntMenge& IntMenge::operator=(const IntMenge& m){
   if (this == &m)   // Zuweisung an sich selbst ?
      return *this;
   if (maxAnzahl != m.maxAnzahl){
```

```
      delete [] tab;
      maxAnzahl = m.maxAnzahl;
      tab = new Typ[maxAnzahl];
   }
   aktAnzahl = m.aktAnzahl;
   for (int i=0; i < maxAnzahl; i++)
      tab[i] = m.tab[i];
   return *this;
}
```

Vordefinierte Elementfunktionen bei Klassen

Beispiel: `class Leer {};`

Welche Funktionen sind bereits vordefiniert ?

- *Standardkonstruktor* `Leer();`
  Deklarieren von Objekten, keine Initialisierung.

- *Standardkopierkonstruktor* `Leer(const Leer&);`
  Elementweises Kopieren aller nicht-statischen Datenelemente, dabei ggf. Aufruf des Kopierkonstruktors von Elementobjekten.

- *Destruktor* `~Leer();`
  Wird tatsächlich nur generiert, wenn die Klasse von einer Klasse mit Destruktor abgeleitet wird, hat standardmäßig nur die Funktion, ggf. den Destruktor der Basisklasse aufzurufen beim Destruieren eines Objektes.

- *Zuweisungsoperator* `Leer& operator = (const Leer&);`
  Elementweises Zuweisen aller nicht-statischen Datenelemente, dabei ggf. Aufruf des Zuweisungsoperators von Elementobjekten.

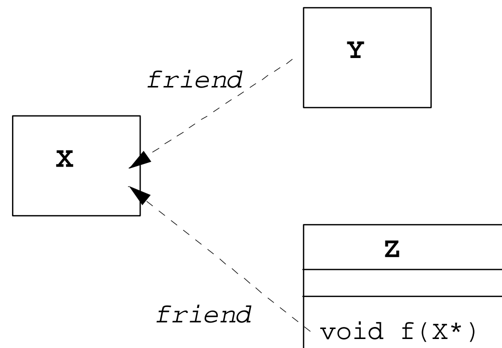
12.8 friend-Funktionen und friend-Klassen

Es gibt bei C++ Situationen, wo es notwendig ist, dass eine globale Funktion auf private Daten eines Objektes direkt zugreifen muss. Ebenso kann es vorkommen, dass zwei Klassen so eng miteinander arbeiten, dass private Informationen unmittelbar zugänglich sein müssen. Um dieses Problem zu lösen, hat man in C++ das `friend`-Konzept geschaffen.

Eine Funktion, die nicht Elementfunktion einer Klasse X ist, kann dennoch auf `private`- oder `protected`- Merkmale von X zugreifen, wenn sie als `friend` von X spezifiziert ist. Eine Klasse, die als `friend`-Klasse von X deklariert ist, hat ebenso das Recht auf private Merkmale zuzugreifen.

Beispiel:

```
// friend1.cpp: friend-Funktionen und -Klassen

class X; // Vorwärtsdeklaration

class Y {
public:
   void f(X* xp);
   void g(X* xp);
};

class Z {
public:
   void f(X* xp);
   void g(X* xp);
};

class X {
   friend class Y;
   friend void Z::f(X*);
   friend void f(X*);
public:
   X(int a):i(a) {}
private:
   int i;
};

void Y::f(X* xp) { cout << "Y::f() " << xp->i << endl; }
void Y::g(X* xp) { cout << "Y::g() " << xp->i << endl; }
void Z::f(X* xp) { cout << "Z::f() " << xp->i << endl; }
void Z::g(X* xp) { cout << "Z::g() " << xp->i << endl; }
                                       // Zugriff auf i nicht erlaubt

void f(X* xp) { cout << "f() " << xp->i << endl; }

int main() {
   X* xp = new X(1);
   Y y1;
   Z z1;
   y1.f(xp);
   y1.g(xp);
   z1.f(xp);
// z1.g(xp);  kein friend
   f(xp);
   return 0;
}
```



Alle *friends* müssen explizit in der Klassendefinition deklariert werden und sind damit als Bestandteil der Klassenschnittstelle zu betrachten.

Eine Hauptanwendung von `friends` ist das Überladen von Operatoren.

Beispiel:

Die Klasse `String` soll um eine Ein-/Ausgabe mittels Streams erweitert werden, d.h. folgendes soll möglich sein:

```
String s;
cin >> s;       // s von der Standardeingabe einlesen
cout << s;      // s auf die Standardausgabe ausgeben
```

Die einzige Möglichkeit der Realisierung besteht in der Implementierung der folgenden globalen Funktionen:

```
istream& operator>>(istream& in, String& s);
ostream& operator<<(ostream& out, const String& s);
```

denn:

```
cin  >> s;     // entspricht operator>> (cin, s)
cout << s;     // entspricht operator<< (cout, s)
```

Beispiel:

```
// string3.cpp: Eine einfache String-Klasse
//     erweitert um Ein-/Ausgabe ueber Streams

class String {
public:
   ...
   // Ein-/Ausgabe ueber Streams
   friend istream& operator>>(istream&, String&);
   friend ostream& operator<<(ostream&, const String&) ;
};

// Eingabe-Operator (einfache Loesung)
istream& operator>>(istream& in, String& s) {
   const int String_size = 256;
   char inBuf[ String_size ];
   in >> inBuf;
   s = inBuf; // String::operator=(String(char*));
   return in;
}

// Ausgabe-Operator
ostream& operator<<(ostream& out, const String& s) {
   return out << s.str;
}
```

12.9 Klassenattribute und Klassenmethoden

Bisher haben wir eine Klasse betrachtet als eine Menge von Objekten, wobei jedes Objekt einen eigenen Satz an Attributen und Methoden besitzt. Oft gibt es aber die Notwendigkeit, Attribute oder Methoden zu definieren, die nur genau einmal je Klasse existieren, sogenannte *Klassenattribute* bzw. *Klassenmethoden.*

Definition:

- Ein *Klassenattribut* ist ein Attribut, das nur der Klasse selbst zugeordnet sind. Es existiert bereits, wenn noch kein einziges Objekt erzeugt wurde.

- Eine *Klassenmethode* ist eine Elementfunktion, die nur der Klasse selbst zugeordnet ist. Sie ist bereits aufrufbar, wenn noch kein einziges Objekt existiert.

Klassenattribute bzw. Klassenmethoden werden durch das Schlüsselwort `static` gekennzeichnet. Klassenmethoden bekommen keinen `this`-Zeiger übergeben, können daher auch nur auf Klassenattribute zugreifen.

```
class X {
...
   static int max;
   static int getMax();
...
};
int X::max=100;      // Initialisieren eines Klassenattributes
int X::getMax() { return X::max; }
                     // Definition einer Klassenmethode
```

Aufruf einer Klassenmethode: `i = X::getMax();`

Klassenatttribute sind vorhanden, sobald zur Laufzeit die Klassendefinition durchlaufen ist und existieren bis zum Programmende.

Beispiel: Erweiterung der Klasse `DoubleStack`

```
class DoubleStack {
public:
   DoubleStack(int=DoubleStack::stacksize); // Vorbelegung mit
                                            // der Klassenkonstanten
   ...
private:
   static const int stacksize;              // Klassenkonstante
   ...
};

const int DoubleStack::stacksize=100;       // Initialisierung der
                                            // Klassenkonstanten
```

Die Initialisierung muss bereits bei der Klassendefinition erfolgen und muss demzufolge auch in der Header-Datei stehen.

Erweiterung im ANSI-Standard:

Konstante Klassenattribute, auch *klassenspezifische Konstanten* genannt, können auch direkt innerhalb des Klassenblocks initialisiert werden. Im obigen Beispiel würde das so aussehen:

```
class DoubleStack{
   ...
private:
   static const int stacksize = 100;
   ...
};
```

Dies ist allerdings beschränkt auf integrale Datentypen (`char, short, int, long`) und Aufzählungstypen.

Aufgabe: Eine Klasse soll zählen, wieviele Objekte von ihr gerade existieren.

Lösung: Definiere ein Klassenattribut `objektAnzahl` zum Zählen der gerade existierenden Objekte. Bei jedem Konstruktoraufruf ist `objektAnzahl` zu erhöhen, bei jedem Destruktoraufruf zu vermindern.

```
// zaehl1.cpp: Zaehlen von Objekten

class Zaehl {
public:
   Zaehl(int);
```

```
    Zaehl(const Zaehl&);
   ~Zaehl();
    void setInhalt (int);          // ObjektInhalt veraendern
    int  getInhalt() const;        // ObjektInhalt zurueckgeben
    static int getObjektAnzahl(); // ObjektAnzahl zurueckgeben
private:
    int inhalt;                    // Inhalt des Objektes
    static int objektAnzahl ;      // Zaehlt die Anzahl Objekte der Klasse
};

// Static-Elemente initialisieren
int Zaehl::objektAnzahl = 0;
int Zaehl::getObjektAnzahl()     { return Zaehl::objektAnzahl; }
```

```
// Standardkonstruktor
Zaehl::Zaehl(int i = 0) : inhalt(i) {
    ++Zaehl::objektAnzahl;
    cout << "Zaehl::Konstruktor       " << inhalt
         << " Anzahl Objekte: "         << objektAnzahl << endl;
}

// Kopierkonstruktor
Zaehl::Zaehl(const Zaehl& z) : inhalt(z.inhalt) {
    ++Zaehl::objektAnzahl;
    cout << "Zaehl::Kopierkonstruktor " << inhalt
         << " Anzahl Objekte: "         << objektAnzahl << endl;
}

Zaehl::~Zaehl() {
    --Zaehl::objektAnzahl;
    cout << "Zaehl::Destruktor        " << inhalt
         << " Anzahl Objekte: "         << objektAnzahl << endl;
} // Destruktor

inline int Zaehl::getInhalt() const
{ return inhalt; }

inline void Zaehl::setInhalt (int i)
{ inhalt = i; }
```

```
// Testprogramm
void f( Zaehl z) {}        // Impliziter Aufruf des Kopierkonstruktors

Zaehl g () {
    Zaehl z(100);
    return z;              // Impliziter Aufruf des Kopierkonstruktors
}

int main(){
    cout << "main: Beginn\n";
    Zaehl z1(1), z2(2);
    Zaehl *zp;
    for (int i = 0; i < 10; ++i)
       zp = new Zaehl(10*i);
    delete zp;
    cout << "Vor  f()\n";
    f(z1);
    cout << "Nach f()\n";
    cout << "Vor  g()\n";
    z2 = g();
    cout << "Nach g()\n";
    cout << "main: Ende\n";
    return 0;
```

```
}
```

Ein weiteres, in der Praxis häufig vorkommendes, Beispiel sind Klassen, die zur Laufzeit nur ein Objekt besitzen dürfen. Diese Klassen modelliert man nach dem sogenannten Entwurfsmuster "Singleton".

Beispiel: Die Singleton-Klasse

```
// Singleton.h: Klasse, die nur ein Objekt besitzen kann

class Singleton {
public:
    static Singleton* instance();

    void setValue(int v)  { value = v;      }
    int  getValue() const { return value;   }

protected:
    Singleton();

private:
    static Singleton* uniqueInstance;
    int value;
};

Singleton* Singleton::uniqueInstance = 0;
```

```
// Singleton.cpp: Klasse, die nur ein Objekt besitzen kann

#include "singleton.h"

// Implementierung
Singleton::Singleton() {
    value = 0;
}

Singleton* Singleton::instance() {
    if (uniqueInstance == 0)
        uniqueInstance = new Singleton;
    return uniqueInstance;
}
```

```
// SingletonT.cpp: Testprogramm für die Singleton-Klasse

#include "singleton.cpp"
#include <iostream>


int main() {
//    Singleton s;   nicht zugreifbar
    Singleton *sp1;
    sp1 = Singleton::instance();
    sp1->setValue(5);
    sp1 = 0;
    Singleton *sp2 = Singleton::instance();
    cout << "Singleton: value: " << sp2->getValue() << endl;
}
```

13 Überladen von Operatoren

Wie bereits am Beispiel der Operatoren >> und << für die Ein- bzw. Ausgabe kennengelernt, ist es in C++ möglich, die vorhandenen Operatoren auch für benutzerdefinierte Datentypen zu verwenden.

13.1 Einführung

Fast alle C++-Operatoren können überladen werden.

Überladbare Operatoren

Operator	*Bedeutung*
`+ - * / %`	Arithmetische Operatoren (- und + auch unär: `-a`, `+a`)
`++ --`	Inkrement und Dekrement (als Präfix (z.B.: `++a`) und Postfix (z.B.: `a++`))
`< > <= >= != ==`	Vergleichsoperatoren
`! && ||`	Logische Operatoren
`^ & |` ###	Bitoperatoren
`<< >>`	Bitshiftoperatoren
`= += -= *= /= %=` `^= &= |= <<= >>=`	Zuweisungsoperatoren
`[]`	Indexoperator
`()`	Funktionsaufrufoperator
`-> ->*`	Zugriffsoperatoren
`,`	Kommaoperator
`& *`	Adress- und *-operator, `z.B.: &a *ptr`
`new new[]` `delete delete[]`	Speicher allokieren bzw. deallokieren

Nicht überladbare Operatoren

Operator	*Bedeutung*
`.`	Elementzugriff z.B.: `a.f()`
`.*`	Elementzugriff über Pointer, z. B. `a.*ptr;`
`::`	Bereichsoperator, z. B. `zaehl::getObjektAnz()`
`?:`	Bedingungsoperator
`sizeof`	Größe von Objekten

Das Überladen eines Operators erfolgt durch die Definition einer Operatorfunktion:

```
Funktionstyp operator <Operator-Symbol> (Parameterliste) {......}
```

Regeln:

- Eine Operatorfunktion muss entweder Elementfunktion einer Klasse sein oder mindestens einen Parameter vom Typ Klasse, Referenz auf Klasse, Aufzählung oder Referenz auf Aufzählung haben.
- Operatorfunktionen können nicht für elementare Datentypen definiert werden.
- Die Anzahl der Operanden kann nicht verändert werden, d.h. binäre Operatoren bleiben binär, unäre Operatoren unär.
- Die Priorität eines Operators kann nicht verändert werden.

- Die Assoziativität, d. h. die Reihenfolge der Zusammenfassung z.B. von links nach rechts oder von rechts nach links, kann nicht verändert werden.
- Es können nur existierende Operatoren überladen werden, d.h. es können keine neuen Operatoren erfunden werden.
- Mit Ausnahme des Zuweisungsoperators (`operator=()`) werden alle Operatorfunktionen an abgeleitete Klassen weiter vererbt.
- Bei manchen Operatoren gibt es spezielle Vorschriften für Rückgabetyp und Argumente.

Beispiel: Eine Klasse für komplexe Zahlen

```
class Complex {
public:
   Complex (double r=0, double i=0) : re(r), im(i) {}
   Complex& operator+=(Complex);
   ...
   friend Complex operator+ (Complex, Complex);
   friend bool    operator==(Complex, Complex);
   ..
private:
   double re, im; // Real- u. Imaginaerteil
};

Complex& Complex::operator+= (Complex c) {
   re += c.re;
   im += c.im;
   return *this;
}
Complex operator+(Complex c1, Complex c2) {
   return Complex(c1.re + c2.re, c1.im + c2.im);
};

bool operator==(Complex c1, Complex c2) {
   return (c1.re == c2.re && c1.im == c2.im);
};
```

Bemerkung:

In der ANSI-C++-Standardbibliothek ist eine Klasse `complex` definiert, die alle wesentlichen benötigten Funktionen beinhaltet (`<complex>`).

13.2 Operatorfunktionen als Elementfunktion oder friend-Funktion

Ein Operator kann entweder mit Hilfe einer Elementfunktion einer Klasse oder einer globalen Funktion, die i. a. als friend-Funktion einer Klasse deklariert ist, überladen werden.

Binäre Operatoren

Sei `@` ein binärer Operator, d.h. ein Operator mit zwei Operanden. Dann hat der Ausdruck

```
a @ b
```

zwei mögliche Interpretationen:

`a.operator@()`	Elementfunktion
`operator@(a,b)`	globale Funktion (i.a. als `friend` deklariert)

Im ersten Falle ist es zwingend erforderlich, dass der Operand a ein Objekt der Klasse ist, wie das Beispiel des Operators += bei der Klasse Complex zeigt. Im zweiten Fall kann a auch von einem anderen Typ sein kann, wie z.B. bei den Operatoren >> und <<.

Beispiel:

```
Complex c1(1,0), c2;
Double x = 1.5;
c2 = x + c1;    // der Aufruf wird umgesetzt zu
                // operator+(Complex(x),c1)
```

Unäre Operatoren

Ein unärer Operator wirkt nur auf einen Operanden. Es gibt zwei verschiedene Arten von unären Operatoren, solche bei denen der Operator vor dem Operanden steht, also Präfix-Operatoren, wie z.B.

```
+a   -a   ~a   &a   *a   !a   ++a   --a
```

und Operatoren, bei denen der Operator hinter dem Operanden stehen kann, also Postfix-Operatoren

```
a++   a--
```

Auch hier gibt es zwei mögliche Interpretationen:

`a.operator~();`	Elementfunktion
`operator~(a);`	globale Funktion

Sonderfall: ++ --
Der Inkrement- bzw. der Dekrementoperator ist sowohl als Präfix- als auch als Postfixoperator überladbar.

`++a`	`a.operator++();`	Präfix
`a++`	`a.operator++(int);`	Postfix, dabei ist das int-Argument nur ein Dummy-Argument und dient ausschließlich der Unterscheidung zwischen Präfix und Postfix

Wann sind Operatoren als Elementfunktionen, wann als globale Funktionen zu implementieren?

Regeln:

- Ein Operator ist als Elementfunktion zu überladen, wenn der erste Operand ein Objekt der zugehörigen Klasse sein muss.
- Unäre Operatoren sollten als Elementfunktionen überladen werden.
- Ist der erste Operand kein Objekt dieser Klasse, so ist der Operator als Nichtelementfunktion zu implementieren.
- Bestimmte Operatoren müssen Elementfunktionen sein, z.B.: = [ ] ( )

13.3 Beispiel: Klasse für rationale Zahlen

Im Folgenden wird eine Klasse `Ratio` für den Umgang mit rationalen Zahlen entwickelt. In fast jedem Buch sind ähnliche Lösungen zu dieser Problematik zu finden. Wir werden eine eigene Variante dazu entwickeln.

Die Klasse sollte folgende Möglichkeiten bieten:

- Arithmetik
  `+ - * /`
- Vergleiche
- Ein-/Ausgabe über Streams

```
Ratio a;        // a = 0
cin >> a;       // Eingabe: 1/3
Ratio b(2,5);   // 2/5
Ratio c = a + b;
cout << c;      // 11/15
c += a;
```

Attribute

```
private:
   long zaehler;
   long nenner;
```

Für `Ratio`-Objekte sollen immer folgende Bedingungen erfüllt sein:

- Zähler und Nenner müssen gekürzt sein
- Der Nenner ist positiv (`> 0`)
- Die Zahl `0` wird durch `0/1` dargestellt

Konstruktor

Man braucht folgende Möglichkeiten der Konstruktion:

```
Ratio a(1, 2); // 1/2
Ratio b(2);    // 2/1
Ratio c;       // 0/1
```

Deklaration:

```
class Ratio {
public:
   Ratio (long z=0, long n=1);   // Konstruktor
   ...
```

Implementierung:

```
Ratio::Ratio (long z, long n) {
   asserts (n != 0); // nenner == 0 fuehrt zur Ausnahme
   if (n > 0) {
      zaehler = z; nenner = n;
   } else {
      zaehler = -z; nenner = -n;
   }
   kuerzen();
}
```

Da in der Klasse nicht dynamisch Speicher allokiert wird, brauchen Kopierkonstruktor, Zuweisungsoperator und Destruktor nicht explizit definiert zu werden.

Das Kürzen der rationalen Zahl wird mit Hilfe des klassischen Euklidischen Algorithmus realisiert:

```
// Euklidischer Algorithmus
long Ratio::ggT (long a, long b){
   asserts (a >= 0 && b > 0);
   long r = b;
   while (r > 0){
      r = a % b;
      a = b;
      b = r;
   }
   return a;
}

// Kuerzen von zaehler und nenner
Ratio& Ratio::kuerzen (){
   long teiler = ggT(abs(zaehler), nenner);
   if (teiler > 1) {
      zaehler = zaehler / teiler;
      nenner  = nenner  / teiler;
   }
   return *this;
}
```

Unäre Operatoren + −

```
// Unaere Operatoren + und -
Ratio Ratio::operator+ () {
   return *this;
}

Ratio Ratio::operator- () {
   return Ratio(-zaehler, nenner);
}
```

Zuweisungsoperatoren

Der Operator = ist nicht explizit zu implementieren, da hier der standardmäßig vom Compiler generierte Zuweisungsoperator ausreicht.

```
Ratio& Ratio::operator+= (const Ratio& a) {
   zaehler = zaehler * a.nenner + nenner * a.zaehler;
   nenner  = nenner * a.nenner;
   kuerzen();
   return *this;
}

Ratio& Ratio::operator*= (const Ratio& a) {
   zaehler *= a.zaehler;
   nenner  *= a.nenner;
   kuerzen();
   return *this;
}
// -= und /= analog
```

Binäre Operatoren + - * /

Die binären Operatoren für die Arithmetik sind als friends zu realisieren, da Ausdrücke wie

```
a + 1    1 + a
```

möglich sein sollten.

1. Version

```
Ratio operator+ (Ratio a, Ratio b) {
   Ratio tmp;
   tmp.zaehler = a.zaehler * b.nenner + a.nenner * b.zaehler;
   tmp.nenner = a.nenner * b.nenner;
   tmp.kuerzen();
   return tmp;
};
```

Nachteile: Beim Aufruf wird zweimal der Kopierkonstruktor aufgerufen, außerdem wird ein temporäres Objekt benötigt und beim `return` wird der Kopierkonstruktor erneut aufgerufen !

2. Version

```
Ratio operator+ (const Ratio& a, const Ratio& b) {
   Ratio  tmp = a;
   return tmp += b; // Aufruf des += Operators
}
```

Auch hier wird ein temporäres Objekt benötigt und beim `return` wird der Kopierkonstruktor aufgerufen.

3. Version

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

Unterschied: Nur beim `return` wird ein Konstruktor aufgerufen.

Vergleichsoperatoren

Die Operatoren == und != sind einfach zu implementieren:

```
bool operator== (const Ratio& a, const Ratio& b) {
  return (a.zaehler==b.zaehler && a.nenner==b.nenner);
}

bool operator!= (const Ratio& a, const Ratio& b) {
  return (a.zaehler!=b.zaehler || a.nenner!=b.nenner);
}
```

Aber wie implementiert man <, >, <=, >= ?

1. Version

```
bool operator<(const Ratio& a, const Ratio& b) {
   Ratio tmp = a - b;
   return tmp.zaehler<0;
};
```

Hier wird ein temporäres Objekt zur Realisierung benötigt. Es geht aber einfacher, wie die nächste Variante zeigt.

2. Version

```
bool operator< (const Ratio& a, const Ratio& b) {
   return (a.zaehler * b.nenner - a.nenner * b.zaehler) < 0;
}
// >  <= >= analog
```

Inkrement und Dekrement

Unterschied zwischen der Präfix- und der Postfix-Version:

```
Ratio a(1,3);  // 1/3
cout << "a++: " << a++ << endl;  // a++: 1/3
cout << "a  : " << a   << endl;  // a  : 4/3
```

Der Wert des Ausdrucks `a++` ist der Wert von `a` vor der Inkrementierung. Der Ausdruck `a++` ist kein L-Wert.

```
Ratio b(1,3);  // 1/3
cout << "++b: " << ++b << endl;  // ++b: 4/3
cout << "b  : " << b   << endl;  // b  : 4/3
```

Der Wert des Ausdrucks `++b` ist der Wert von `b` nach der Inkrementierung. Der Ausdruck `++b` ist ein L-Wert.

```
// ++a:
Ratio& Ratio::operator++() {
   zaehler += Nenner;
   return *this;
};

// a++:
Ratio Ratio::operator++ (int) {
   Ratio tmp = *this;
   zaehler += Nenner;
   return tmp; //Rueckgabe des Wertes vor dem Inkrementieren
};
```

Vollständige Klassendefinition

```
// ratio.h:  Eine Klasse für rationale Zahlen

class Ratio {
public:
   Ratio (long z=0, long n=1);   // Konstruktor

   long getZaehler ()  { return zaehler; } // Zugriffsfunktionen
   long getNenner  ()  { return nenner;  }

   // Zuweisungen
   Ratio& operator+=(const Ratio&);
   Ratio& operator-=(const Ratio&);
   Ratio& operator*=(const Ratio&);
   Ratio& operator/=(const Ratio&);

   // Konvertierungsfunktionen
   long   toLong()   { return zaehler / nenner; }
   double toDouble() { return double(zaehler) / double(nenner); }

   // Unaere Operatoren
   Ratio operator+  ();
   Ratio operator-  ();
```

```
     // Inkrement- und Dekrementoperatoren
   Ratio& operator++ ();         // ++a
   Ratio& operator-- ();         // --a
   Ratio  operator++ (int);      // a++
   Ratio  operator-- (int);      // a--

   // Binaere Operatoren
   friend Ratio operator+  (const Ratio&, const Ratio&);
   friend Ratio operator-  (const Ratio&, const Ratio&);
   friend Ratio operator*  (const Ratio&, const Ratio&);
   friend Ratio operator/  (const Ratio&, const Ratio&);

   // Vergleichsoperatoren
   friend bool operator== (const Ratio&, const Ratio&);
   friend bool operator!= (const Ratio&, const Ratio&);
   friend bool operator<  (const Ratio&, const Ratio&);
   friend bool operator>  (const Ratio&, const Ratio&);
   friend bool operator<= (const Ratio&, const Ratio&);
   friend bool operator>= (const Ratio&, const Ratio&);

   // Ein-/Ausgabeoperatoren
   friend ostream& operator<< (ostream&, const Ratio&);
   friend istream& operator>> (istream&, Ratio&);

   // Absolutbetrag
   friend Ratio abs (const Ratio& a);

private:
   long zaehler;
   long nenner;
   Ratio& kuerzen ();
   static long ggT(long, long);
   static long abs(long a) { return (a < 0) ? -a : a; }
};

// Ratio.cpp: Implementierung von Rationalen Zahlen
// Implementierung der Klasse RATIO

#include "ratio.h"

// Euklidischer Algorithmus
long Ratio::ggT (long a, long b) {
   assert (a >= 0 && b > 0);
   long r = b;
   while (r > 0){
      r = a % b;
      a = b;
      b = r;
   }
   return a;
}

// Kuerzen von zaehler und nenner
Ratio& Ratio::kuerzen () {
   long teiler = ggT(abs(zaehler), nenner);
   if (teiler > 1) {
      zaehler = zaehler / teiler;
      nenner  = nenner  / teiler;
   }
   return *this;
}

// Konstruktor: Eine rationale Zahl wird immer mit positivem Nenner
//       und in gekuerzter Form gespeichert
```

```
Ratio::Ratio (long z, long n) {
   assert (n != 0); // nenner == 0 fuehrt zur Ausnahme
   if (n > 0) {
      zaehler = z; nenner = n;
   } else {
      zaehler = -z; nenner = -n;
   }
   kuerzen();
   cout << "Ratio:Konstruktor " << *this << endl;
}

// Inkrement- und Dekrementoperatoren
Ratio& Ratio::operator++ () { // ++a
   zaehler += nenner;
   return *this;
}

Ratio& Ratio::operator-- () { // --a
   zaehler -= nenner;
   return *this;
}

Ratio  Ratio::operator++ (int) { // a++
   Ratio tmp = *this;
   zaehler += nenner;
   return tmp;
}

Ratio  Ratio::operator-- (int) { // a--
   Ratio tmp = *this;
   zaehler -= nenner;
   return tmp;
}

// Zuweisungsoperatoren
Ratio& Ratio::operator+= (const Ratio& a) {
   zaehler = zaehler * a.nenner + nenner * a.zaehler;
   nenner  = nenner * a.nenner;
   kuerzen();
   return *this;
}

Ratio& Ratio::operator-= (const Ratio& a) {
   zaehler = zaehler * a.nenner - nenner * a.zaehler;
   nenner  = nenner * a.nenner;
   kuerzen();
   return *this;
}

Ratio& Ratio::operator*= (const Ratio& a) {
   zaehler *= a.zaehler;
   nenner  *= a.nenner;
   kuerzen();
   return *this;
}

Ratio& Ratio::operator/= (const Ratio& a) {
   zaehler *= a.nenner;
   nenner  *= a.zaehler;
   kuerzen();
   return *this;
}

// Unaere Operatoren + und -
Ratio Ratio::operator+ () {
```

```
   return *this;
}

Ratio Ratio::operator- () {
   return Ratio(-zaehler, nenner);
}

// Arithmetische Operatoren
Ratio operator+ (const Ratio& a, const Ratio& b) {
   return Ratio(a.zaehler * b.nenner + a.nenner * b.zaehler,
                a.nenner  * b.nenner);
// alternativ
//   Ratio tmp = a;
//   return tmp += b;
}

Ratio operator- (const Ratio& a, const Ratio& b) {
   return Ratio(a.zaehler * b.nenner - a.nenner * b.zaehler,
                 a.nenner * b.nenner);
}

Ratio operator* (const Ratio& a, const Ratio& b) {
   return Ratio(a.zaehler * b.zaehler, a.nenner * b.nenner);
}

Ratio operator/ (const Ratio& a, const Ratio& b) {
   return Ratio(a.zaehler * b.nenner, a.nenner * b.zaehler);
}

// Vergleichsoperatoren
bool operator== (const Ratio& a, const Ratio& b) {
  return (a.zaehler==b.zaehler && a.nenner==b.nenner);
}

bool operator!= (const Ratio& a, const Ratio& b) {
  return (a.zaehler!=b.zaehler || a.nenner!=b.nenner);
}

bool operator< (const Ratio& a, const Ratio& b) {
   return (a.zaehler * b.nenner - a.nenner * b.zaehler) < 0;
}

bool operator> (const Ratio& a, const Ratio& b) {
   return (a.zaehler * b.nenner - a.nenner * b.zaehler) > 0;
}

bool operator<= (const Ratio& a, const Ratio& b) {
   return (a.zaehler * b.nenner - a.nenner * b.zaehler) <= 0;
}

bool operator>= (const Ratio& a, const Ratio& b) {
   return (a.zaehler * b.nenner - a.nenner * b.zaehler) >= 0;
}

// Ein-/Ausgabeoperatoren
ostream& operator<< (ostream& o, const Ratio& a) {
   if (a.nenner != 1)
      return (o << a.zaehler << "/" << a.nenner);
   else
      return (o << a.zaehler);
}

istream& operator>> (istream& in, Ratio& r) {
  char ch;
```

```
  in >> r.zaehler >> ch >> r.nenner;
  assert (ch == '/' && r.nenner != 0);
  if (r.nenner < 0) {
     r.nenner  = - r.nenner;
     r.zaehler = - r.zaehler;
  }
  r.kuerzen();
  return in;
}

Ratio abs (const Ratio& a) {
   return Ratio (Ratio::abs(a.zaehler), a.nenner);
}
```

13.4 Der Index-Operator [ ]

Der Operator [ ] ist speziell für Container-Klassen beim Zugriff auf einzelne Komponenten sinnvoll, z.B. für Arrays, Strings, Listen, Mengen, Bäume, ...

[ ] ist ein binärer Operator, der als Elementfunktion überladen werden muss.

`a[b]` ↔ `a.operator[](b);`

`b` kann von beliebigem Datentyp sein.

Damit der Rückgabewert von `[ ]` auch auf der linken Seite einer Zuweisung stehen kann, kann es sinnvoll sein, eine Referenz zurückzugeben.

Im folgenden Beispiel wird die im letzten Abschnitt begonnene String-Klasse um den Operator [ ] erweitert. Dabei soll der Operator zum einen das Ausgeben einzelner Zeichen des Strings ermöglichen, als auch das Ändern einzelner Zeichen durch Zuweisung.

Beispiel:

```
// string4.cpp:  Eine einfache String-Klasse
//               erweitert um den Operator []

class String {
public:
   ...
   int   length() const { return len; } // Stringlaenge
   char  operator[](int i) const; // Zugriff auf i-tes Element
   char& operator[](int i);       // Zugriff als L-Wert
   ...
};

// Zugriff auf i-tes Element
char String::operator[](int i) const {
   assert(i >= 0 && i < len);
   return str[i];
}

// Rückgabe des i-ten Zeichens als L-Wert
char& String::operator[](int i) {
   assert(i >= 0 && i < len);
   return str[i];
}

int main() {
   String s1;
   cout << "Bitte String eingeben : ";
```

```
   cin >> s1;
   for (int i = 0; i < s1.length(); ++i)
      cout << s1[i] << ' ';        // operator[](int)
   cout << endl;
   cout << "Bitte ein Zeichen und einen Index eingeben: ";
   char c;
   int i;
   cin >> c >> i;
   s1[i] = c;                      // operator[](int)
   cout << "Ergebnis nach s1[" << i << "]=" << c << ": "
        << s1 << endl;
   cin >> c;
   return 0;
}
```

Beispiel: Eine Klasse zur Implementierung eines Assoziativen Arrays

Aufgabe: Zähle von der Standardeingabe eingelesene Worte und gebe ihre Häufigkeit aus.

Eingabe:

```
aaa bbb ccc bbb↵
aaa aaa↵
^D (End of file) oder ^Z
```

Ausgabe:

```
aaa   : 3
bbb   : 2
ccc   : 1
```

Datenstrukturen:

```
class Element{
private:
   string name;
   int anzahl;
....
};
```

Aufgabe: Aufnahme eines gelesenen Wortes und die Anzahl des Auftretens

```
class Assoc {
public:
...
private:
   Element* tab;
   int maxAnzElemente;            // Groesse der Tabelle
   int anzElemente;               // Anzahl enthaltener Elemente
};
```

Aufgabe: Organisation eines Containers von `Element`-Objekten

Anwendung

```
// assoctst.cpp: Testprogramm fuer die Klasse Assoc

//  Zaehlt die Anzahl der aus der Standardeingabe gelesenen Woerter
int main() {
   const int SIZE = 5;      // Groesse des assoziativen Arrays
   string wort;
   Assoc atab(SIZE);
   try {
      while (cin >> wort)
         atab[wort]++;   // Was passiert hier eigentlich??
      atab.ausgeben();
   }
   catch (const char* s) {  cout << "Ausnahme: " << s << endl;   }
}
```

Klassendefinitionen:

```
// assoc.h: Eine Klasse fuer ein Assoziatives Array

#include <iostream>
#include <string>

// Elementklasse fuer assoziatives Array
class Element {
public:
   Element() : anzahl(0) {}
   Element& operator++(int);     // Anzahl inkrementieren Postfix

   friend class Assoc;
private:
   string name;                  // Name des Eintrags
   int anzahl;                   // Anzahl des Auftretens
};


// Klasse fuer assoziatives Array
class Assoc {
public:
   explicit Assoc(int);
  ~Assoc() { delete [] tab; }

   Element& operator[](const string&);
   bool full() const {  return anzElemente >= maxAnzElemente; }
   void ausgeben();
private:
   Element* tab;
   int maxAnzElemente;         // Groesse der Tabelle
   int anzElemente;            // Anzahl enthaltener Elemente

   int finde (const string& w) const;      // Wort intern finden
   Assoc(const Assoc&);              // Kopie verbieten
   Assoc& operator=(const Assoc&);   // Zuweisung verbieten
};
```

Klassenimplementierung:

```
// assoc.cpp: Implementierung der Klasse Assoc

#include "assoc.h"

// Anzahl inkrementieren Postfix
Element& Element::operator++(int) {
   anzahl++;
   return *this;
}

// Konstruktor
Assoc::Assoc(int s): maxAnzElemente(s), anzElemente(0) {
   assert(s > 0);
   tab = new Element[maxAnzElemente];
}

// Suche Element mit einem bestimmten Wort
// Returnwert:     Index in tab, falls gefunden
//              -1            falls nicht gefunden
int Assoc::finde(const string& w) const {
   for (int i = 0; i < anzElemente ; i++)
      if (tab[i].name == w)
         return i;
   return -1;
```

```
}

// Zugriff auf Element mit bestimmtem Schluessel
Element& Assoc::operator[](const string& w) {
   int i = finde(w);
   if (i >= 0)
      return tab[i];

   if (full())
       throw "Kapazitaet erschoepft";

   ++anzElemente;
   tab[anzElemente-1].name = w;
   tab[anzElemente-1].anzahl = 0;
   return tab[anzElemente-1];
}

void Assoc::ausgeben() {
   for (int i=0; i < anzElemente; i++)
      cout << tab[i].name   << ": "
           << tab[i].anzahl << endl;
}
```

13.5 Der Operator ( )

Der Operator ( ) ist ein binärer Operator, der als Elementfunktion definiert werden muss. Dabei ist der erste Parameter das Objekt selbst, der zweite ist die Parameterliste des Funktionsaufrufes. Der Aufruf der Operatorfunktion ( ) sieht dabei aus, als wäre das Objekt selbst eine Funktion.

Aufruf	*Bedeutung*
x()	x.operator()()
x(a)	x.operator()(a)
x(a,b)	x.operator()(a,b)

Die Parameterliste kann beliebig lang sein, d. h. beliebig viele Parameter beliebigen Typs haben.

Eine erste Anwendung ist die Implementierung einer Substring-Funktion für unsere String-Klasse.

Beispiel:

```
// string5.cpp: Eine einfache String-Klasse
//         erweitert um die Operatoren [] und ()

class String {
public:
   ...
   // Teilstring ab pos anz Zeichen
   String substr    (int pos, int anz) const;
   String operator()(int pos, int anz) const { // gleiche Funktion
      return substr(pos, anz);
   }
   ...
};

// Teilstring von Position pos anz Zeichen
String String::substr(int pos, int anz) const {
   assert ( 0 <= pos && pos < len && anz >= 0);
   if (pos + anz > len)
      anz = len - pos;
```

```
   if (anz == 0)
      return String("");

   char* q = new char[anz+1];
   strncpy(q, str+pos, anz);
   q[anz] = '\0';

   String s(q);
   delete [] q;
   return s;
}

int main() {
   String s1;
   cout << "Bitte String eingeben : ";
   cin >> s1;
   cout << "Ihre Eingabe lautete  : " << s1 << endl;

   int pos, anz;
   String s2;
   do {
      cout << "Substring: pos und anz eingeben : ";
      cin >> pos >> anz;
      s2 = s1(pos,anz);
      cout << "Substring: " << s2 << endl;
   } while  (true);
   return 0;
}
```

Eine zweite sinnvolle Anwendung ist bei der Implementierung einer Matrix-Klasse der Zugriff auf ein einzelnes Matrixelement.

Beispiel:

```
// matrix1.h: rudimentaere Matrix-Klasse

class DoubleMatrix {
public:
   DoubleMatrix();                       // Standardkonstruktor

   double  operator() (int n, int m) const; // Element (n,m)
   double& operator() (int n, int m);      // Element (n,m) als L-Wert

   // Ein- und Ausgabe
   friend istream& operator>> (istream&, DoubleMatrix&);
   friend ostream& operator<< (ostream&, const DoubleMatrix&);
private:
   double tab[3][3];
   static const int dim;        // Dimension der Matrix
};

const int DoubleMatrix::dim=3;
```

```
// matrix1.cpp: rudimentaere Matrix-Klasse Implementierung

#include "matrix1.h"

// Standardkonstruktor
DoubleMatrix::DoubleMatrix() {
   for (int i = 0; i < dim; i++)
   for (int j = 0; j < dim; j++)
     tab[i][j] = 0.0;
```

```
}

// Element(n,m)
double  DoubleMatrix::operator()(int n, int m) const {
   assert (0 < n && n <= dim &&
           0 < m && m <= dim);
   return tab[n-1][m-1];
}

// Element(n,m) als L-Wert
double& DoubleMatrix::operator() (int n, int m) {
   assert (0 < n && n <= dim &&
      0 < m && m <= dim);
   return tab[n-1][m-1];
}

// Ein- und Ausgabe
istream& operator>> (istream& in, DoubleMatrix& M) {
   for (int i = 0; i < DoubleMatrix::dim; i++)
   for (int j = 0; j < DoubleMatrix::dim; j++)
      in >> M.tab[i][j];
   return in;
}

ostream& operator<< (ostream& out, const DoubleMatrix& M) {
   int i, j;
   for (i = 0; i < DoubleMatrix::dim; i++) {
      out << "( ";
      for (j = 0; j < DoubleMatrix::dim; j++)
         out << setw(6) << M.tab[i][j] << ' ';
      out << ")\n";
   }
   return out;
}
```

```
// matrix1t.cpp: Testprogramm fuer Matrix-Klasse

#include "matrix1.h"

int main() {
   DoubleMatrix m1;
   cout << "Matrix eingeben:\n";
   cin >> m1;
   cout << "eingegeben wurde:\n" << m1 << endl;

   int i, j;
   double wert;
   do {
      cout << "Index i und j eingeben: ";
      cin >> i >> j;
      cout << "m1(" << i << ", " << j << ") = " << m1(i,j) << endl;
      cout << "Wert eingeben: ";
      cin >> wert;
      m1(i,j) = wert;
      cout << "m1(" << i << ", " << j << ") = " << m1(i,j) << endl;
   } while (true);
   return 0;
}
```

Eine weitere wichtige Anwendung des Funktionsoperators besteht in der Möglichkeit, sogenannte Funktionsobjekte zu definieren. Funktionsobjekte sind Objekte, die nach außen wie Funktionen aufgerufen werden, aber tatsächlich echte Objekte sind.

13.6 Intelligente Zeiger: Die Operatoren -> und *

Ein großes Problem bei C und C++ entsteht in der Praxis dadurch, dass man z.B. Zeiger auf dynamisch allokierte Datenobjekte hat und "vergisst", den allokierten Speicher wieder explizit zu entfernen.

```
void f() {
   X* xp = new X;     // allokiere X-Objekt
   ...                // irgendwelche Aktionen
   return;            // xp wird freigegeben, das zugehörige
}                     // X-Objekt aber nicht
```

Wünschenswert wäre ein Zeiger, der das Objekt, auf das er verweist mit Ende seiner eigenen Lebensdauer automatisch freigibt. Mit Hilfer der Operatoren -> und * ist es möglich Klassen zu entwickeln, die wie solche *intelligenten Zeiger,* auch *smart pointer* genannt, funktionieren.

Aufruf	*Bedeutung*
`x->elem`	`(x.operator->())->elem`
`*x`	`x.operator*()`

Der Aufruf `x.operator->()()` muss als Funktionswert einen Zeiger auf ein Objekt, das eine Komponente `elem` hat, liefern. Der Aufruf `x.operator*()` muss eine Referenz auf ein Objekt zurückliefern.

Beispiel:

```
// smart1.cpp: Intelligente Zeiger

class PunktZeiger {
public:
   PunktZeiger(double x=0.0, double y=0.0) { ptr = new Punkt(x,y);  }
  ~PunktZeiger()            {   delete ptr;   }
   Punkt* operator->()      {   return ptr;   }
   Punkt& operator*()       {   return *ptr; }
private:
   PunktZeiger(const PunktZeiger&);            // Kopieren und
   PunktZeiger& operator=(const PunktZeiger&); // Zuweisen verbieten
   Punkt* ptr;
};

ostream& operator<< (ostream& o, const Punkt& p) {
   o << "x = " << p.getX() << ", y = " << p.getY();
}

int main() {
   PunktZeiger pp1(1.0,2.0);
   PunktZeiger pp2(1.0,5.0);
   cout << "pp1: " << *pp1 << '\n';    // operator*
   cout << "pp2: " << *pp2 << '\n';
   *pp2 = *pp1;
   pp1->skaliere(2.0);
   cout << "pp1: " << *pp1 << '\n';
   cout << "pp2: " << *pp2 << '\n';
   cout << "Abstand zwischen pp1 und pp2 : "
        << pp1->abstand(*pp2) << endl; // operator->

   return 0;   // Automatisches Entfernen der Punkt-Objekte
}
```

Als Variante könnte man auch dafür sorgen, dass das Objekt erst beim ersten tatsächlichen Zugriff erzeugt wird (*create on demand*). Das ist sinnvoll, wenn es sich um ein aufwendig zu konstruierendes Objekt handelt, das nicht immer benötigt wird. Intelligente Zeiger findet man noch in zahlreichen weiteren Varianten (siehe z.B. Stroustrup, Breymann). Mit der Standardklasse `auto_ptr` gibt es in der Standardklassenbibliothek sogar eine Implementierung.

13.7 Benutzerdefinierte Typumwandlungen

Wir haben bereits verwendet, dass Konstruktoren mit einem Argument nichts anderes als benutzerdefinierte Typumwandlungen darstellen.

```
Ratio a;
a = 1;   // Implizit wird Ratio(1) ausgeführt
```

Was aber, wenn auch die umgekehrte Richtung benötigt wird? Ein Objekt ist in einen elementaren Datentyp zu konvertieren.

```
ratio a(1,3);
double x = a;  // keine Konvertierung vorgesehen
```

Lösen lässt sich dieses Problem mit Hilfe einer speziellen Operatorfunktion.

Allgemeine Form:

```
Quelltyp::operator Zieltyp();  // konvertiere Quelltyp in Zieltyp
```

Im Beispiel:

```
Ratio::operator double() { return double(zaehler) / double(nenner); }
```

Durch eine solche Funktion wird eine Vorschrift definiert, wie ein Ratio-Objekt in einen `double`-Wert zu konvertieren ist.

Typumwandlungsoperatoren sind parameterlose Elementfunktionen und haben keinen expliziten Rückgabetyp. Der Rückgabetyp ist automatisch durch den Zieltyp der Konvertierung vorgegeben.

```
double x = a;  // entspricht x = a.operator double();
```

Ähnlich kann man in der Klasse `Ratio` auch eine Konvertierungsfunktion nach `long` implementieren

```
operator long   () { return Zaehler / Nenner; }
```

Dies mag zwar sehr elegant sein, bringt jedoch auch Probleme durch zusätzliche Mehrdeutigkeiten.

```
Ratio a(1/2), b;
b = a + 1;
```

Der Ausdruck `a + 1` kann nunmehr auf zwei verschiedene Arten interpretiert werden

```
a + Ratio(1)   oder  a.operator long() + 1
```

Dies führt im Allgemeinen zu einem Compilerfehler. Um dieses Problem zu vermeiden, kann man jedoch auch normale Elementfunktionen definieren, um Konvertierungen in bestimmte

Zieltypen durchzuführen. Diese müssen dann jedoch auch explizit aufgerufen werden. In der Klasse `Ratio` sieht dies dann so aus:

```
long   toLong()   { return zaehler / nenner; }
double toDouble() { return double(zaehler) / double(nenner); }
```

Erweiterung im ANSI-Standard:

Um zu verhindern, dass ein Konstruktor automatisch vom Compiler als Typumwandlung verwendet wird, kann das Schlüsselwort `explicit` verwendet werden.

Beispiel:

```
class intArray {
public:
   intArray(int n);  // Array mit n Elementen erzeugen
   ...
};
```

Anwendung:

```
intArray A = 10;  // wird implizit zu A = intArray(10) !!!
```

Damit ist der Konstruktor `intArray` automatisch eine Typumwandlung von `int` zu `intArray`. Um dies zu verhindern, definiert man besser:

```
class intArray {
public:
   explicit intArray(int n);  // Array mit n Elementen erzeugen
   ...
};
```

Dann muss der Konstruktor explizit aufgerufen werden und obiger Ausdruck wird vom Compiler abgelehnt.

```
intArray A = intArray(10);  // expliziter Konstruktoraufruf
```

13.8 Eine String-Klasse

Eine erste sinnvolle Anwendung von Klassen mit überladenen Operatoren als Elementfunktionen ist eine Klasse zur Behandlung von Strings. Wir werden im folgenden eine Klasse vorstellen, die im wesentlichen der Standardklasse `string` aus der ANSI-C++-Klassenbibliothek entspricht.

Welche Eigenschaften benötigt eine String-Klasse ?

Erzeugung:

```
String s1;                 // Leerstring
String s2("Hallo Welt");   // Erzeugung mit char-String
String s3('c');            // Erzeugung mit char
```

Mögliche Elementfunktionen:

```
s1 = s2 + s3;              // Konkatenation von Strings
s1 += s2;
if (s1 == s2)              // Vergleiche == != < > <= >=
s3 = s1;                   // Zuweisung
cin >> s3; cout << s1;     // Ein-/Ausgabe
getline(cin, s1, '\n');
char c = s1[2];            // Zugriff auf einzelne Zeichen mit [ ]
```

```
s3 = s1(2,4);              // Zugriff auf Substrings
```

Listing:

```
// string6.h: Eine String-Klasse mit den wichtigsten Eigenschaften

#include <cstring>
#include <cassert>
#include <iostream>
#include <cctype>

class String {
public:
// Konstruktoren, Destruktor, Zuweisungsoperator
   String (char *s=0);
   String (char);
   String (const String& s);
  ~String ();
   String& operator= (const String&);

// Konvertierung nach char*
   const char* c_str() { return str; }

// String loeschen
   void clear();

// String-Laenge
   int size()   const { return len; }
   int length() const { return len; }

// Zugriff auf i-tes Element
   char  operator[](int i) const;
   char& operator[](int i);       // Rueckgabe als L-Wert

   char  at (int i) const;
   char& at (int i);

// Substring-Zugriffe
   String substr    (int pos, int anz) const; // Teilstring ab pos anz
                                              // Zeichen
   String operator()(int pos, int anz) const { // gleiche Funktion
      return substr(pos, anz);
   }

// Konkatenation
   String& operator+=(const String& );
   String& operator+=(char );
   friend String  operator+ (const String&, const String&);

// Vergleichsoperatoren
   friend bool operator==(const String&, const String&);
   friend bool operator!=(const String&, const String&);
   friend bool operator< (const String&, const String&);
   friend bool operator> (const String&, const String&);
   friend bool operator<=(const String&, const String&);
   friend bool operator>=(const String&, const String&);

   // Ein-/Ausgabe ueber Streams
   friend istream& operator>>(istream&, String&);
   friend ostream& operator<<(ostream&, const String&) ;
   friend istream& getline (istream& in, String& str, char delim='\n');
private:
   char *str;                    // interner char-String
   int len;                      // Laenge des Strings
   static char* str_dup(const char*);  // Duplizieren eines char-Strings
```

```
};
```

```
// string6.cpp: Implementierung der String-Klasse

#include "string6.h"

// Konstruktoren
String::String(char *s) {
   if (s == 0) {
      len = 0;
      str = new char[1];
      *str = '\0';
   } else {
      str = str_dup(s);
      len = strlen(s);
   }
}

String::String(char c) {
   str = new char[2];
   str[0] = c;
   str[1] = '\0';
   len = 1;
}

// Kopierkonstruktor
String::String(const String& s) {
   str = str_dup(s.str);
   len = s.len;
}

// Destruktor
String::~String() {   delete [] str;  }

// Zuweisungsoperator
String& String::operator=(const String& s) {
   if (this == &s)
      return *this;
   if (len != s.len) {
      delete [] str;
      str = new char[s.len + 1];
      len = s.len;
   }
   strcpy(str,s.str);
   return *this;
}

// String loeschen
void String::clear() {
   delete [] str;
   str = new char[1];
   *str = '\0';
   len = 0;
   }

// Zugriff auf i-tes Element
char String::operator[](int i) const {
   assert(i >= 0 && i < len);
   return str[i];
}

char& String::operator[](int i) {
   assert(i >= 0 && i < len);
   return str[i];
```

```
}

char String::at(int i) const {
   assert(i >= 0 && i < len);
   return str[i];
}

char& String::at(int i) {
   assert(i >= 0 && i < len);
   return str[i];
}

// Teilstring von Position pos anz Zeichen
String String::substr(int pos, int anz) const {
   assert ( 0 <= pos && pos < len && anz >= 0);
   if (pos + anz > len)
      anz = len - pos;

   if (anz == 0)
      return String("");

   char* q = new char[anz+1];
   strncpy(q, str+pos, anz);
   q[anz] = '\0';

   String s(q);
   delete [] q;
   return s;
}

// Konkatenation
String operator+ (const String& s1, const String& s2) {
   String tmp = s1;
   tmp += s2;
   return tmp;
}

String& String::operator+=(const String& s) {
   char *p = new char[len + s.len +1];
   strcpy(p, str);
   strcpy(p + len, s.str);
   len += s.len;
   delete [] str;
   str = p;
   return *this;
}

String& String::operator+=(char c ) {
   char *p = new char[len + 2];
   strcpy(p, str);
   p[len] = c;
   p[len+1] = '\0';
   len += 1;
   delete [] str;
   str = p;
   return *this;
}

// Vergleichsoperatoren
bool operator==(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) == 0; }

bool operator!=(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) != 0; }
```

```
bool operator<(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) < 0; }

bool operator>(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) > 0; }

bool operator<=(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) <= 0; }

bool operator>=(const String& s1, const String& s2)
{ return strcmp(s1.str, s2.str) >= 0; }

// Eingabe-Operator (einfache Loesung)
istream& operator>>(istream& in, String& s) {
   char c;
   while (in.get(c)) {
      if (!isspace(c)) {   // nichtleeres Zeichen gelesen ?
         in.putback(c);    // falls ja zurueck in den Eingabepuffer
         break;
      }
   }
   while (in.get(c)) {
      if (isspace(c)) {   // Leerzeichen gelesen ?
         in.putback(c);   // falls ja zurueck in den Eingabepuffer
         break;
      } else
         s += c;          // gelesenes Zeichen anhaengen
   }
   return in;
}

// Ausgabe-Operator
ostream& operator<<(ostream& out, const String& s) {
   return out << s.str;
}

istream& getline (istream& in, String& s, char delim) {
   s.clear();
   char c;
   while(!in.eof()) {
      in.get(c);
       // Begrenzer erreicht? (wird gelesen)
      if(c == delim)
         break;
      else  s += c;
   }
   return in;
}

// Private Elementfunktionen
// Duplizieren eines char-Strings
char* String::str_dup(const char* p) {
   assert(p!=0);
   char* q = new char[strlen(p)+1];
   strcpy(q,p);
   return q;
}
```

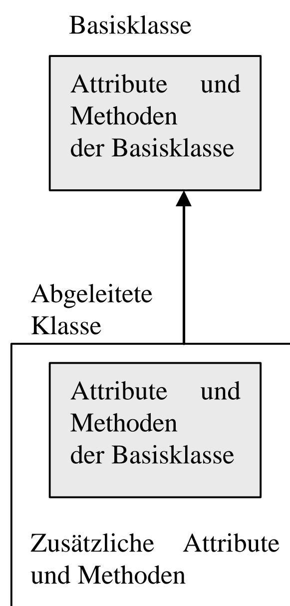
14 Vererbung

14.1 Einführung

Eine wesentliche Grundforderung an die Entwicklung von Software ist die Erfüllung von *Erweiterbarkeit* und *Wiederverwendbarkeit* von Software. Die Objektorientierung bietet hierzu ein ganz besonderes Instrument an, die *Vererbung*.

Definition:

Vererbung ist ein Mechanismus, bei dem eine Klasse als Spezialfall einer allgemeinen Klasse definiert wird. Dabei "erbt" die *abgeleitete Klasse (Unterklasse)* automatisch alle Attribute und Methoden der *Basisklasse (Oberklasse)*. Zusätzlich kann die abgeleitete Klasse weitere Attribute und Methoden hinzufügen und geerbte Methoden redefinieren.



Beispiel:

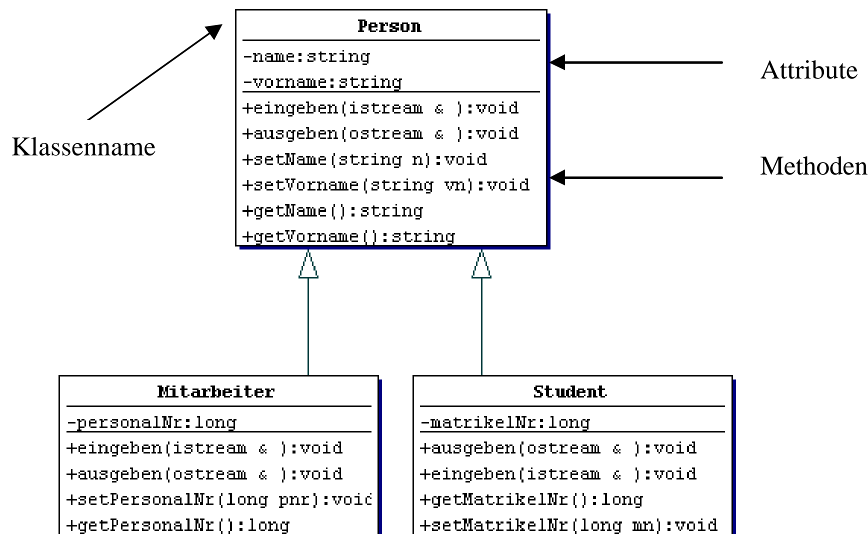
Für ein Hochschulinformationssystem wird eine Klasse für Mitarbeiter und eine Klasse für Studenten benötigt. Die Analyse ergibt die folgenden benötigten Merkmale:

Klasse `Mitarbeiter`:	Klasse `Student`:
Attribute: • Name • Vorname • Personalnummer Methoden: • set- und get-Methoden • Einlesen von einem Stream • Ausgabe auf einen Stream	Attribute: • Name • Vorname • Matrikelnummer Methoden: • set- und get-Methoden • Einlesen von einem Stream • Ausgabe auf einen Stream

Mit den bisher betrachteten Techniken wird man hier zwangsläufig zu einer Code-Vervielfachung kommen. Die Vererbung bietet jedoch eine bessere Lösung an:

Bilde eine Klasse `Person`, die die gemeinsamen Merkmale von `Mitarbeiter` und `Student` besitzt und leite die Klassen `Mitarbeiter` und `Student` von der Klasse `Person` ab.

Man erhält dann die folgende Situation: (Diagramm mit *Together/Enterprise* erstellt)



- Die Klassen `Mitarbeiter` und `Student` erben jeweils alle Attribute und Methoden von `Person`.
- Die Methoden `eingeben` und `ausgeben` werden geerbt, aber jeweils redefiniert.
- Die Klasse `Mitarbeiter` hat ein zusätzliches Attribut `personalNr` und die zugehörige `set`- bzw. `get`-Methode.
- Die Klasse `Student` hat ein zusätzliches Attribut `matrikelNr` und die zugehörige `set`- bzw. `get`-Methode.

Zugehöriger Quelltext:

```
class Person {
public:
   void eingeben   (istream&);
   void ausgeben   (ostream&) const;
   void setName    (string n);
   void setVorname(string vn);
   string getName    () const;
   string getVorname() const;
private:
   string name;
   string vorname;
};
```

```
class Mitarbeiter : public Person {
public:
   void eingeben (istream&);
   void ausgeben (ostream&) const;
   void setPersonalNr(long pnr);
   long getPersonalNr()    const;
private:
   long personalNr;
};
```

```
class Student : public Person {
public:
   void ausgeben (ostream&) const;
   void eingeben (istream&);
   long getMatrikelNr() const;
   void setMatrikelNr(long mn);
private:
```

```
    long matrikelNr;
};
```

Wichtige Eigenschaften:

- die *abgeleitete Klasse* (Unterklasse) erbt von der *Basisklasse* (*Oberklasse*) alle Attribute und Methoden
- zusätzliche Attribute und Methoden können hinzugefügt werden
- geerbte Methoden können redefiniert werden
- geerbte Attribute können nicht redefiniert werden
- die Methoden der abgeleiteten Klasse können auf alle `public-` und `protected`-Merkmale der Basisklasse zugreifen
- die privaten Merkmale der Basisklasse werden zwar vererbt, jedoch ist ein direkter Zugriff in der abgeleiteten Klasse nicht erlaubt
- Klassen können an beliebig viele Klassen weitervererben
- eine Klasse kann von mehreren Basisklassen erben (*Mehrfachvererbung*)
- abgeleitete Klassen können wieder als Basisklassen dienen, so dass regelrechte Klassenhierarchien entstehen können.

Bemerkung:

- Mehrfachvererbung ist möglich bei C++ und EIFFEL
- nur Einfachvererbung ist möglich bei SMALLTALK, OBJEKT-COBOL
- bei JAVA gibt es eine eingeschränkte Form der Mehrfachvererbung

Ist-ein(e)-Beziehung und hat-ein(e)-Beziehung

Zwischen der abgeleiteten Klasse und der Basisklasse besteht eine sogenannte *ist-ein(e)-Beziehung*, d. h. ein Objekt der abgeleiteten Klasse ist ein spezielles Objekt der Basisklasse. In unserem Beispiel heißt das:

- ein `Student`-Objekt ist auch ein `Person`-Objekt
- ein `Mitarbeiter`-Objekt ist auch ein `Person`-Objekt

Eine andere Art der Beziehung zwischen zwei Klassen ist die *hat-ein(e)-Beziehung*. Zwei Klassen stehen in einer *hat-ein(e)-Beziehung*, wenn eine Klasse ein Objekt der anderen als Element besitzt.

Beispiel:

Die Klasse `Strecke` aus Kapitel 3 besitzt zwei `Punkt`-Objekte als Attribute.

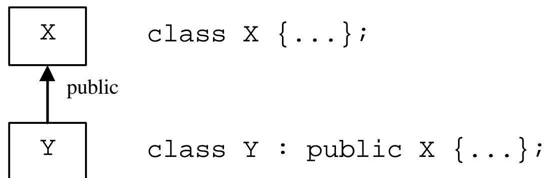
Man bezeichnet das in der objektorientierten Analyse auch als *Assoziation* (allgemeine Beziehung) oder in diesem Falle auch als *Aggregation* (Ganzes-Teil-Beziehung).

Es gibt oft Situationen, wo man für ein Design zwischen hat-ein(e) und ist-ein(e)-Beziehung wählen kann.

14.2 public-, protected- und private-Vererbung

Es gibt drei verschiedene Möglichkeiten, eine Klasse von einer anderen abzuleiten. Diese unterscheiden sich im wesentlichen durch die Zugriffsmöglichkeiten auf die vererbten Merkmale und die Typverträglichkeiten zwischen Basisklasse und abgeleiteter Klasse.

14.2.1 public-Ableitung



- Y erbt alle Merkmale von X
- Alle `public` Merkmale von X sind automatisch in Y auch `public`. Man bezeichnet die public-Ableitung daher auch als *Interface Inheritance*, da die abgeleitete Klasse die komplette Schnittstelle der Basisklasse zur Verfügung stellt.
- Alle `protected`-Merkmale von X sind in Y ebenfalls `protected` und zugrefbar innerhalb von Y.
- Auf `private`-Merkmale von X kann in Y nicht direkt zugegriffen werden, obwohl sie natürlich in Y vorhanden sind.
- Durch die `public`-Ableitung wird eine ist-ein(e)-Beziehung zwischen Y und X hergestellt.

Standardumwandlungen

Betrachte noch einmal das Beispiel:

```
class Person { ... };

class Mitarbeiter : public Person { ... };

class Student : public Person { ... };
```

Definiere:

```
Mitarbeiter m1;
Student s1;
Person p1;
```

1. Situation:

```
p1 = s1; // Ein Student ist eine Person, daher ist die Zuweisung erlaubt
p1 = m1; // Ein Mitarbeiter ist eine Person, daher ist die Zuweisung
         // erlaubt
```

Die Zuweisung erfolgt komponentenweise. Die zusätzlichen Attribute von Student bzw. Mitarbeiter gehen dabei allerdings verloren.

2. Situation:

```
s1 = p1; // Falsch, da eine Person i. a. kein Student ist
m1 = p1; // Falsch, da eine Person i. a. kein Mitarbeiter ist
```

Die ist-ein(e)-Relation geht nur in eine Richtung. In der anderen Richtung wäre auch unklar, wie die zusätzlichen Attribute von `Student` bzw. `Mitarbeiter` zu belegen wären.

3. Situation:
```
s1 = m1; // Falsch, da ein Mitarbeiter i. a. kein Student ist
```

Die beiden Klassen `Mitarbeiter` und `Student` sind zwar "verwandt" und haben sogar eine identische innere Struktur, sind aber nicht typverträglich.

14.2.1.1 Zeiger und Referenzen

- Zeiger auf automatische Objekte
```
Person *pp;           // Zeiger auf Person-Objekt
Student s1;           // Student-Objekt
pp = &s1;             // erlaubt. Es wird aber keine Typumwandlung
                      // durchgeführt. Der Person-Zeiger pp zeigt auf das
                      // Student-Objekt s1
pp->getMatrikelNr(); // nicht möglich, da zur Übersetzungszeit nur der
                      // Typ von pp (Person*) bekannt ist, aber nicht der
                      // von dem Objekt, auf das pp zur Laufzeit zeigt.
```

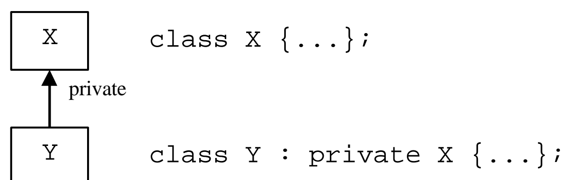
- Zeiger auf dynamische Objekte
```
Person *pp = new Mitarbeiter;
         // pp zeigt auf ein dynamisch erzeugtes Mitarbeiter-Objekt
```

- Referenzen
```
Student s1;
Person &pr = s1;      // Die Person-Referenz referenziert ein Student-
                      // Objekt. Auch hier findet keine Typumwandlung
                      // statt!
```

- Umkehrung
```
Person *pp;
if (funktion == 1)
   pp = new Student;
else
   pp = new Mitarbeiter;

Mitarbeiter *mp = pp;
```

Die Zuweisung ist nicht möglich, obwohl `pp` zur Laufzeit auf ein `Mitarbeiter`-Objekt zeigen könnte, genauso gut könnte `pp` allerdings auch auf ein `Student`-Objekt zeigen. Hier entscheidet alleine der zur Übersetzungszeit feststellbare Typ von `pp` und das ist der Typ `Person*`.

14.2.2 private-Ableitung



- Alle `public`- und `protected`-Merkmale von X werden in Y automatisch `private`.
- `private`-Ableitung ist keine *ist-ein(e)-Beziehung*, sondern eine *ist-implementiert-durch-Beziehung*.
- Eine Typumwandlung von Y-Objekten in X-Objekte ist daher nicht möglich, ebenso gibt es keine Typverträglichkeit zwischen Y-Zeigern und X-Zeigern.

- `private`-Ableitung ist eine reine Implementierungstechnik (*implementation inheritance*) . Die abgeleitete Klasse wird implementiert mit Hilfe der Basisklasse, stellt aber nicht deren Schnittstelle zur Verfügung.
- die `private`-Ableitung ist die Voreinstellung, d. h. `class X : Y {...}` bedeutet automatisch `class X : private Y {...}`

<u>Beispiel</u>: Implementierung eines `StringStack` mit Hilfe einer `StringList`

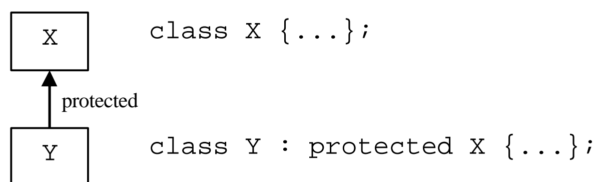
1. `StringList` als Elementobjekt
```
class StringStack {
...
protected:
   StringList l;
...
};
```

2. Durch private-Ableitung
```
class StringStack : private StringList{...};
```

<u>Wesentlicher Unterschied:</u> Bei `private`-Vererbung kann direkt auf die `protected`-Merkmale der Klasse `StringList` zugreifen. Hat man ein `StringList`-Objekt als Elementobjekt, so ist nur der Zugriff auf die `public`-Merkmale von `StringList` möglich.

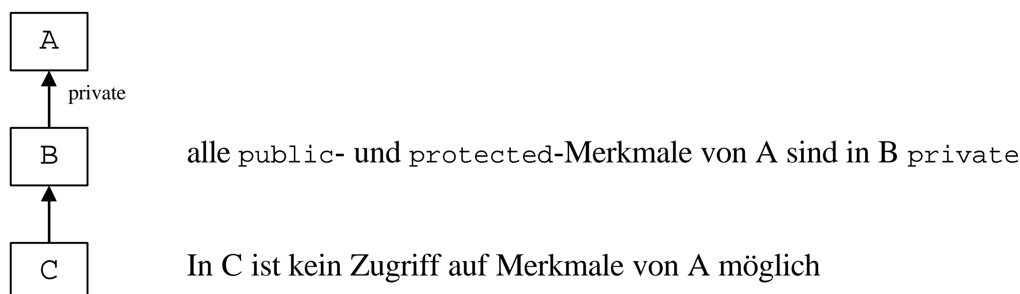
14.2.3 protected-Ableitung

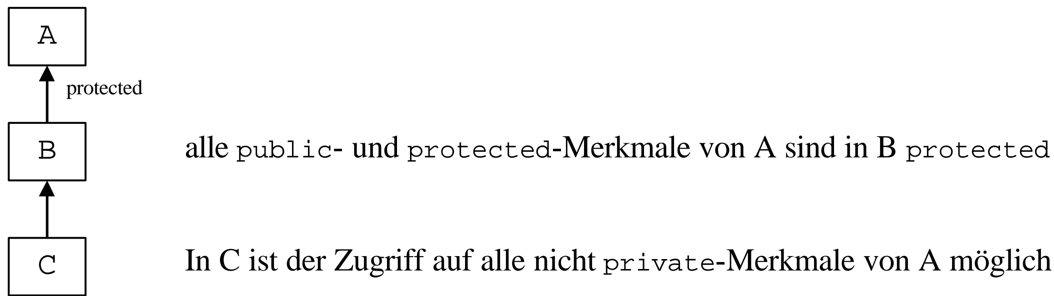
Die `protected`-Ableitung ist im Prinzip nur eine Variante der `private`-Ableitung.



- Alle `public`- und `protected`-Merkmale von X werden in Y automatisch `protected`.
- ansonsten wie bei der `private`-Ableitung.

<u>Gegenüberstellung von private- und protected-Vererbung</u>





Übersicht

Ableitung	Basisklasse	abgeleitete Klasse
public	public →	public
	protected →	protected
	private ⇢	private
protected	public ↘	public
	protected →	protected
	private ⇢	private
private	public ↘	public
	protected ↘	protected
	private ⇢	private

Die punktierten Linien zeigen dabei an, dass die `private`-Merkmale der Basisklasse zwar an die abgeleitete Klasse vererbt werden, dort allerdings nicht direkt zugreifbar sind. Obwohl diese geerbten `private`-Merkmale Bestandteil der Objekte der abgeleiteten Klasse sind, ist der Zugriff auf sie nur über die Schnittstelle der Basisklasse möglich.

Empfehlung
`private`- und `protected`-Ableitung sollten möglichst vermieden werden.

Alle weiteren Ausführungen in diesem Abschnitt beziehen sich, wenn nicht ausdrücklich anders gesagt, immer auf `public`-Vererbung.

14.3 Attribute und Methoden

- Attribute werden von der Basisklasse an die abgeleitete Klasse weitervererbt und können nicht redefiniert werden.
- Folgende Konstruktion ist aber möglich:

```
class X {
    ...
protected:
    int i;
};

class Y : public X {
    ...
private:
    long i;
};
```

Das Attribut `i` in `Y` überdeckt dabei das gleichnamige Attribut in `X`. Der Zugriff auf das geerbte `i` kann jedoch noch mit Hilfe des Bereichsoperators `X::i` erfolgen.

- Methoden werden von der Basisklasse an die abgeleitete Klasse weitervererbt und können redefiniert werden. Die Redefinition muss dabei eine Methode mit der gleichen Signatur sein, d. h. gleicher Name, gleicher Parameteranzahl und Parametertypen und gleicher `const`-Zusatz.

Beispiel:

```
class Person {
public:
   ...
void ausgeben  (ostream&) const;
   ...
};

void Person::ausgeben(ostream& o) const {
    o << name << ", " << vorname << '\n';
}

class Student : public Person {
public:
   ...
  void ausgeben(ostream&) const;
...
};

void Student::ausgeben(ostream& o) const {
    Person::ausgeben(o);
    o << "Matr-Nr: " << matrikelNr << '\n';
}
```

- Die Methode `ausgeben` der Basisklasse `Person` wird in der abgeleiteten Klasse `Student` redefiniert. Jedoch kann mittels `Person::ausgeben(o)` auf die Methode der Basisklasse zugegriffen werden.

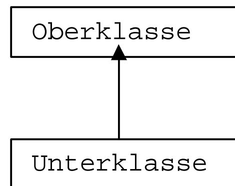
Beispiel:

```
// using1.cpp: Anwendung von namespaces auf Klassen
class A {
public:
    void f(char c)   { cout << "A::f(char)  : " << c << endl; }
    void f(double d) { cout << "A::f(double): " << d << endl; }
};

class B : public A {
public:
    void f(int i)   { cout << "A::f(int)   : " << i << endl; }
};

int main() {
    B b;
    b.f('c');  // ruft B::f(int) auf !
    b.f(3.14); // ruft B::f(int) auf !
    return 0;
}
```

Die in der abgeleiteten Klasse B definierte Methode f() <u>überdeckt</u> die gleichnamigen Methoden `f()` in der Basisklasse A ! Dieses Problem lässt sich nur mit Hilfe der Namespace-Syntax lösen. Eine Klasse ist implizit auch ein Namensraum. Um dem Compiler bekanntzumachen, dass es auch in der Basisklasse A Funktionen namens `f()` gibt, müssen diese wie folgt bekanntgemacht werden:

```
class B : public A {
public:
   using A::f;
   void f(int i)   { cout << "A::f(int)   : " << i << endl; }
};
```

14.4 Konstruktoren und Destruktoren



- Die Konstruktoren und der Destruktor einer Klasse werden nicht vererbt.
- Vor der ersten Anweisung eines Konstruktors einer abgeleiteten Klasse wird zunächst der Konstruktor der Basisklasse durchlaufen.

```
Unterklasse::Unterklasse(...)   { ... }
                              ↑
                             Konstruktor Oberklasse
```

- Beim Destruktor ist es genau umgekehrt: Der Destruktor der Basisklasse wird erst dann durchlaufen, wenn der Destruktor der abgeleiteten Klasse abgearbeitet ist.

```
Unterklasse::~Unterklasse() { ... }
                                   ↑
                                   Destruktor Oberklasse
```

- Benötigt ein Konstruktor der Basisklasse Argumente, so sind diese vom Konstruktor der abgeleiteten Klasse über die Elementinitialisierungsliste mitzugeben. Ist in der Initialisierungsliste kein Konstruktor der Basisklasse explizit angegeben, so wird automatisch der Standardkonstruktor der Basisklasse aufgerufen.

```
Oberklasse::Oberklasse(Typ1 t1, Typ2 t2) { ... }

Unterklasse::Unterklasse(Typ1 t1, Typ2 t2, Typ3 t3)
            :Oberklasse(t1, t2), ...    { ... }
             ↑
             Konstruktor Oberklasse mit Parametern
```

- Der Destruktor der Basisklasse wird immer automatisch aufgerufen.

Beispiel: Personenhierarchie

```
class Person {
public:
   Person() {}
   Person(string, string);
   void eingeben  (istream&);
   void ausgeben  (ostream&) const;
   void setName   (string n);
   void setVorname(string vn);
   string getName   () const;
   string getVorname() const;
protected:
   string name, vorname;
};
```

```
class Mitarbeiter : public Person {
public:
   Mitarbeiter();
   Mitarbeiter(string, string, long);
   void eingeben(istream&);
   void ausgeben(ostream&) const;
   void setPersonalNr(long pnr);
   long getPersonalNr()    const;
protected:
   long personalNr;
};
```

```
class Student : public Person {
public:
   Student();
   Student(string, string, long);
   void ausgeben(ostream&) const;
   void eingeben(istream&);
   long getMatrikelNr() const;
   void setMatrikelNr(long mn);
protected:
   long matrikelNr;
};
```

```
// Implementierung Person
// Konstruktor
Person::Person(string n, string vn)
   : name(n), vorname(vn) {}

void Person::ausgeben(ostream& o) const
{   o << name << ", " << vorname; }

void Person::eingeben(istream& in)
{   in >> name >> vorname; }

void Person::setName(string n)     {   name = n;   }
void Person::setVorname(string vn) {   vorname = vn;     }
string Person::getName() const     {   return name;      }
```

```
// Implementierung Mitarbeiter
// Standardkonstruktor: implizit Aufruf des Standardkonstruktors
// von Person
Mitarbeiter::Mitarbeiter() : personalNr(0) {}

// Konstruktor: Aufruf des Konstruktors von Person über die
//              Initialisierungsliste
Mitarbeiter::Mitarbeiter(string n, string vn, long pnr)
   : Person(n, vn), personalNr(pnr) {}

void Mitarbeiter::ausgeben(ostream& o) const {
   Person::ausgeben(o);
   o << "\tPers-Nr: " << personalNr;
}

void Mitarbeiter::eingeben(istream& in) {
   Person::eingeben(in);
   in >> personalNr;
}

void Mitarbeiter::setPersonalNr(long pnr) {   personalNr = pnr;  }
long Mitarbeiter::getPersonalNr() const   {   return personalNr; }
```

```
// Implementierung Student
// Standardkonstruktor: implizit Aufruf des Standardkonstruktors
// von Person
Student::Student(): matrikelNr(0) {}

// Konstruktor: Aufruf des Konstruktors von Person über die
//              Initialisierungsliste
Student::Student(string n, string vn, long mnr)
   : Person(n, vn), matrikelNr(mnr) {}

void Student::ausgeben(ostream& o) const {
   Person::ausgeben(o);
   o << "\tMatr-Nr: " << matrikelNr;
}

void Student::eingeben(istream& in) {
   Person::eingeben(in);
   in >> matrikelNr;
}

void Student::setMatrikelNr(long mnr) {   matrikelNr = mnr;   }
long Student::getMatrikelNr() const   {   return matrikelNr;   }
```

14.5 Statisches und dynamisches Binden

Beispiel:

```
Person* pp = new Student("Meier", "Sepp", 1234567);
pp->ausgeben();
```

Welche Methode `ausgeben()` wird aufgerufen, die der Klasse `Student` oder die der Klasse `Person`?

Antwort: Es gibt beide Möglichkeiten!

Testprogramm zu den Klassen `Person`, `Student`, `Mitarbeiter`:

```
// persont2.cpp: 2. Testprogramm fuer die Klassen Person, Mitarbeiter
//               und Student

#include "mitarbeiter.h"
#include "student.h"

int main() {
   Person * persTab[5];
   persTab[0] = new Person("Schmitt", "Hans");
   persTab[1] = new Student("Meier", "Fritz", 1111111);
   persTab[2] = new Student("Hoffmann", "Petra", 2222222);
   persTab[3] = new Mitarbeiter("Adam", "Albert", 4711);
   persTab[4] = new Mitarbeiter("Beyer", "Gerda", 4712);

   for (int i = 0; i < 5; ++i) {
      persTab[i]->ausgeben(cout);
      cout << endl;
   }
   return 0;
}
```

Startet man das Programm, so stellt man fest, dass immer nur die Methode `Person::ausgeben()` aufgerufen wird, obwohl der Zeiger `persTab[i]` entweder auf ein `Person`-, ein `Student-` oder ein `Mitarbeiter`-Objekt zeigt.

<u>Grund</u>: Standardmäßig wird in C++ *statisch gebunden*, d. h. der sogenannte *statische Typ* eines Objektes entscheidet darüber, welche Methode aufzurufen ist

<u>Statischer Typ</u>*:* der zur Übersetzungszeit feststellbare Typ eines Objektes, im Beispiel ist der Zeiger `persTab[i]` vom Typ `Person*`, egal auf welches Objekt er zeigt. Der Compiler entscheidet also bereits, welche Methode an dieser Programmstelle aufgerufen wird.

<u>Dynamischer Typ</u>*:* der zur Laufzeit feststellbare Typ eines Objektes, im Beispiel kann der Zeiger vom Typ `Person*` zur Laufzeit auf Objekte vom Typ `Person`, `Student` oder `Mitarbeiter` zeigen.

Erklären wir im obigen Beispiel die Funktion `Person::ausgeben()` als `virtual`, d. h. als *virtuelle Funktion*, und testen den Ablauf der `for`-Schleife noch einmal:

```
class Person {
public:
   virtual void ausgeben  (ostream&) const;
...
};
```

Ablauf:

```
for (int i = 0; i < 5; ++i) {
   persTab[i]->ausgeben(cout);
   cout << endl;
}
```

In der `for`-Schleife wird nun je nachdem, welcher Eintrag bearbeitet wird, entweder `Person::ausgeben()`, `Student::ausgeben()` oder `Mitarbeiter::ausgeben()` aufgerufen.

<u>Grund</u>: Die Angabe `virtual` zeigt an, dass von der Funktion `ausgeben()` bei abgeleiteten Klassen unterschiedliche Versionen existieren können und dass erst zur Laufzeit anhand des *dynamischen Typs* entschieden wird, welche davon tatsächlich aufgerufen werden soll. Man spricht in diesem Zusammenhang auch von *dynamischem Binden*.

<u>Beispiele:</u>

`*(persTab[1])`	statischer Typ	`Person`
	dynamischer Typ:	`Student`
`*(persTab[3])`	statischer Typ	`Person`
	dynamischer Typ:	`Mitarbeiter`

d. h. in der Schleife

```
for (int i = 0; i < 5; ++i) {
      persTab[i]->ausgeben(cout);
      cout << endl;
   }
```

wird bei `i==1` die Methode `Student::ausgeben()` und bei `i==3` die Methode `Mitarbeiter::ausgeben()` aufgerufen.

Funktionen, die dynamisch gebunden werden, nennt man in C++ *virtuelle Funktionen*. Das dynamische Binden ist eines der Hauptkriterien für Objektorientierung, meist als *Polymorphismus* (=Vielgestaltigkeit) bezeichnet.

Bemerkung:

In C++ gibt es statisches und dynamisches Binden, in Smalltalk, Eiffel und Java gibt es nur dynamisches Binden.

Beispiel:

```
// virtual1.cpp: Virtuelle Funktionen

class A {
public:
   virtual void zeige() { cout << "A" << endl; }
};
class B : public A {
public:
   void zeige() { cout << "B" << endl; }
};
class C : public B {
public:
   void zeige() { cout << "C" << endl; }
};
class D : public C { };

int main() {
   A* za[4] = { new A, new B, new C, new D };
   for (int i = 0; i < 4; i++)
      za[i]->zeige();
   return 0;
}
```

```
A     virtual zeige()     za[i]->zeige();
###                       statischer  Typ: A
B     zeige()             dynamischer Typ: A, B, C oder D
###                       je nachdem Aufruf von
C     zeige()                   A::zeige
###                             B::zeige
D                               C::zeige
```

Bemerkung:

Man kann auch spezielle Varianten einer virtuellen Funktion aufrufen, indem man sie explizit qualifiziert:

```
za[i]->A::zeige();   // funktioniert!
za[i]->B::zeige();   // geht nicht, da za[i] von statischen Typ A ist!
```

Regeln:

- `virtual` darf nur in der Klassendefinition stehen. Bei der Implementierung ist die Angabe `virtual` nicht erlaubt.
- Virtuelle Funktionen müssen Elementfunktionen sein (es gibt keine virtuellen `friends`)

- Eine virtuelle Funktion muss für diejenige Klasse, in der sie das erste Mal deklariert wurde, auch definiert werden (Ausnahme: *rein virtuelle* Funktionen)
- Virtuelle Funktionen können auch `friend`-Funktionen von anderen Klassen sein
- `static`-Elementfunktionen können nicht virtuell sein
- Wird in der Basisklasse eine Funktion `virtual` deklariert, so ist die Redefinition dieser Funktion in der abgeleiteten Klasse automatisch ebenfalls `virtual`.
- Die Redefinition in der abgeleiteten Klasse muss die selbe Signatur und den selben Ergebnistyp wie die virtuelle Funktion der Basisklasse besitzen.

```
A     virtual void zeige();
###
B     void zeige() const; // keine Redefinition, aber Ueberdeckung!
```

- Empfehlung: Auch in der abgeleiteten Klasse sollte man `virtual` vor die Funktionsdeklaration schreiben.

Veränderter Ergebnistyp bei der Redefinition

Von der Regel, dass die Redefinition einer virtuellen Funktion den gleichen Ergebnistyp haben muss wie die redefinierte Funktion gibt es eine Ausnahme.

Ist der Rückgabewert der virtuellen Funktion der Basisklasse ein Zeiger oder eine Referenz auf eine Klasse A, so darf die Redefinition auch einen Zeiger bzw. eine Referenz auf eine von A abgeleitete Klasse zurückgeben.

Beispiel:

```
// virtual2.cpp: Virtuelle Funktionen: anderer Ergebnistyp

class Oberklasse {
public:
   virtual Oberklasse& f() {
      cout << "Oberklasse::f()" << endl;
      return *this;
   }
};

class Unterklasse : public Oberklasse {
public:
   Unterklasse& f() { // Redefinition von Oberklasse::f()
      cout << "Unterklasse::f()" << endl;
      return *this;
   }
};

int main() {
   Oberklasse* op = new Unterklasse;
   op->f(); // Unterklasse::f()
   return 0;
}
```

Frage: Wann sollen Funktion `virtual` definiert werden?

Antwort: Beim Aufbau von Klassenhierarchien sollten die Elementfunktionen, die in abgeleiteten Klassen überladen werden können, immer virtuell definiert werden. Im Extremfall können dies alle Elementfunktionen sein.

Interne Realisierung von virtuellen Funktionen

Intern werden die Aufrufe von virtuellen Methoden mit Hilfe von Funktionszeigern realisiert.

Beispiel:

```
class Oberklasse {
public:
   Oberklasse (int ii): i(ii) {}
   virtual void f() {  cout << "Oberklasse::f()" << endl;   }
   virtual void g() {  cout << "Oberklasse::g()" << endl;   }
protected:
   int i;
};

class Unterklasse : public Oberklasse {
public:
   Unterklasse(int ii, int jj): Oberklasse(ii), j(jj) {}
   virtual void f() {  cout << "Unterklasse::f()" << endl;  }
protected:
   int j;
};
```

Sobald eine Klasse mindestens eine virtuelle Funktion enthält, wird vom Compiler eine sogenannte *virtuelle Methodentabelle (VMT*, engl. *virtual table*) angelegt, das ist eine Tabelle von Funktionszeigern auf die virtuellen Funktionen der Klasse.

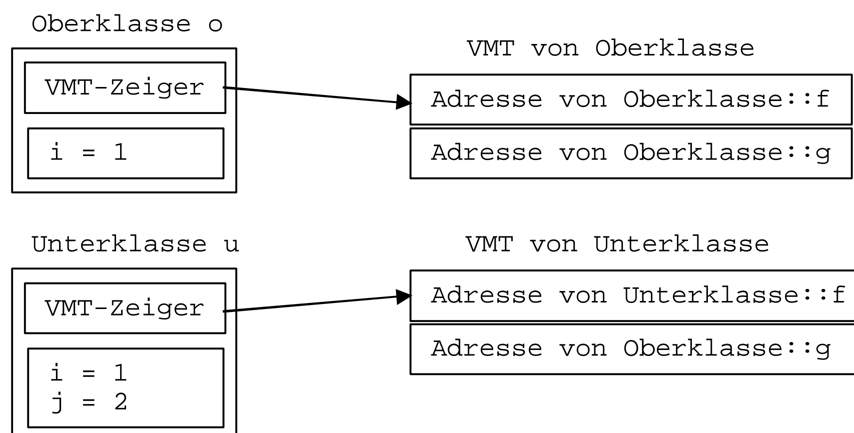
Die Adresse jeder virtuelle Funktion der Klasse ist dabei in der VMT eingetragen. Dabei besitzen die verschiedenen Versionen einer virtuellen Funktion in den virtuellen Methodentabellen einer Klassenhierarchie jeweils immer den gleichen Index.

Jedes Objekt der Klasse besitzt als zusätzliches (nicht sichtbares) Attribut einen VMT-Zeiger) (*virtual table pointer*), einen Zeiger auf diese Tabelle.

Die Objekte

```
Oberklasse  o(1);
Unterklasse u(1, 2);
```

sind intern in etwa so dargestellt:



Die Ausdrücke

```
Oberklasse* op = &u;
op->f();
```

```
op->g();
```

werden wie folgt ausgewertet:

- Zunächst wird der VMT-Zeiger des referenzierten Objektes ausgewertet. `op` zeigt auf das Objekt `u` also der VMT-Zeiger auf die VMT von `Oberklasse`.
- Dann wird der VMT die Adresse der zugehörigen virtuellen Funktion entnommen. Diese wird schließlich aufgerufen. Im Falle `op->f()` ist dies die Funktion `Unterklasse::f()`, im Falle `op->g()` die Funktion `Oberklasse::g()`.

14.6 Virtuelle Destruktoren

Problem:

```
A        A* ap=new B(..);
↑        ...
B        delete ap;  // A-Objekt destruieren, B-Anteil bleibt
```

Lösung:

```
class A {
   virtual ~A(){ ... };
};
```

dann entscheidet bei `delete ap` der dynamische Typ.

Wirkungsweise des virtuellen Destruktors:

Ähnlich wie bei virtuellen Funktionen entscheidet der dynamische Typ des Destruktoraufrufes darüber, welcher Destruktor tatsächlich zur Laufzeit aufzurufen ist.

Unterschied:

Nach dem Aufruf des Destruktors der untersten abgeleiteten Klasse werden auch die Destruktoren aller Basisklassen aufwärts der Vererbungshierarchie entlang aufgerufen. Letztendlich werden die Destruktoren aller Klassen aufgerufen, von dem das zu entfernende Objekt erbt.

Wann sind Destruktoren virtuell zu definieren ?

- Wenn eine Klasse als Basisklasse verwendet werden soll
- auf jeden Fall, wenn die Klasse mindestens eine virtuelle Funktion besitzt

14.7 Vererbung bei Kopierkonstruktor und Zuweisungsoperator

Für den Kopierkonstruktor und den Zuweisungsoperator gelten bei der Vererbung besondere Regeln. Sie werden beide nicht automatisch vererbt. Es gelten hier folgende Regeln:

- Wird in der Unterklasse der Kopierkonstruktor nicht explizit definiert, so erzeugt der Compiler einen Standard-Kopierkonstruktor, der den Kopierkonstruktor der Oberklasse automatisch aufruft.
- Wird in Unterklasse der Zuweisungsoperator nicht explizit überladen, so erzeugt der Compiler einen Standard-Zuweisungsoperator, der automatisch den Zuweisungsoperator der Oberklasse aufruft, um den gererbten Anteil korrekt zuzuweisen.
- Wird der Kopierkonstruktor in der Unterklasse explizit implementiert, so muss man selbst dafür Sorge tragen, dass die geerbten Attribute aus der Oberklasse richtig versorgt werden. Normalerweise geschieht dies durch den Aufruf des Kopierkonstruktors der Oberklasse in der Elementinitialisierungsliste.

- Auch beim expliziten Überladen des Zuweisungsoperators hat man dafür zu sorgen, dass die geerbten Attribute richtig übertragen werden. Am einfachsten geschieht dies mit Hilfe des Aufrufs des Zuweisungsoperators der Oberklasse

Beispiel: Standard-Kopierkonstruktor der Unterklasse

```
// kopierkonst1.cpp: Vererbung beim Kopierkonstruktor

class A {
public:
    A(int j): i(j) {}
    A(const A& a) : i(a.i) {
       cout << "A::Kopierkonstruktor " << i << '\n';
    }
    virtual void print() {
       cout << "A: i = " << i;
    }
private:
    int i;
};

class B : public A {
public:
    B(int k): A(k),j(k*k) {}
    void print() {
        A::print();
        cout << "; B: j = " << j;
    }
private:
   int j;
};

int main() {
    B b1(2);
    B b2(b1);
    cout << "b1: "; b1.print(); cout << endl;
    cout << "b2: "; b2.print(); cout << endl;
}
/* Ausgabe:
A::Kopierkonstruktor 2
b1: A: i = 2; B: j = 4
b2: A: i = 2; B: j = 4     */
```

Beispiel: Explizit implementierter Kopierkonstruktor in der Unterklasse

```
// kopierkonst2.cpp: Vererbung beim Kopierkonstruktor

...

class B : public A {
public:
    B(int k): A(k),j(k*k) {}
    B(const B& b) : A(b), j(b.j) {
       cout << "B::Kopierkonstruktor " << j << '\n';
    }
    void print() {
        A::print();
        cout << "; B: j = " << j;
    }
private:
   int j;
};
```

```
int main() {
    B b1(2);
    B b2(b1);
    cout << "b1: "; b1.print(); cout << endl;
    cout << "b2: "; b2.print(); cout << endl;
}
/* Ausgabe:
A::Kopierkonstruktor 2
B::Kopierkonstruktor 4
b1: A: i = 2; B: j = 4
b2: A: i = 2; B: j = 4   */
```

Beispiel: Standard-Zuweisungsoperator in der Unterklasse

```
// zuwop1.cpp: Vererbung beim Zuweisungsoperator
class A {
public:
   A(int j): i(j) {}
   A& operator=(const A& a) {
      cout << " A::operator=\n";
      i = a.i;
      return *this;
   }
   virtual void print()
   { cout << " A: i = " << i; }
private:
   int i;
};

class B : public A {
public:
   B(int k): A(k),j(k*k) {}
   void print() {
      A::print();
      cout << "; B: j = " << j;
   }
private:
   int j;
};

int main() {
   B b1(2), b2(3);
   cout << "b1: "; b1.print(); cout << endl;
   cout << "b2: "; b2.print(); cout << endl;
   cout << "b1 = b2:"; b1=b2; // automat. Aufruf von A::operator=
   cout << "b1: "; b1.print(); cout << endl;
   return 0;
}
// Ausgabe:
// b1: A: i = 2; B: j = 4
// b2: A: i = 3; B: j = 9
// b1 = b2: A::operator=
// b1: A: i = 3; B: j = 9
```

- In B wird ein Standard-Zuweisungsoperator generiert
- Die Zuweisung erfolgt komponentenweise
- Für die geerbten Merkmale wird automatisch `A::operator=` aufgerufen

Beispiel: Explizites Überladen des Zuweisungsoperators

```
class B : public A {
public:
    B(int k): A(k),j(k*k) {}
    B& operator=(const B& b) {
       A::operator=(b); // muss explizit aufgerufen werden !
       cout << "; B::operator=";
       j = b.j;
       return *this;
    }
    ...
};
// Ausgabe des Testprogramms:
// b1: A: i = 2; B: j = 4
// b2: A: i = 3; B: j = 9
// b1 = b2: A::operator=; B::operator=
// b1: A: i = 3; B: j = 9
```

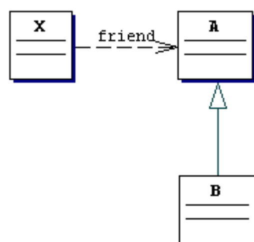
Überladen des Zuweisungsoperators in B: Der Zuweisungsoperator von A muss explizit aufgerufen werden, damit der A-Anteil des B-Objektes korrekt zugewiesen wird.

Bemerkung: Der Zuweisungsoperator kann auch als virtuelle Funktion definiert werden um polymorphe Zuweisungen zu ermöglichen (siehe dazu auch Prinz oder Breymann).

14.8 Vererbung und friends

Wie die folgenden Beispiele zeigen, wird der `friend`-Mechanismus nicht vererbt.

1. Situation



Zugriff von Elementfunktionen von A auf `private`-Merkmale von X möglich

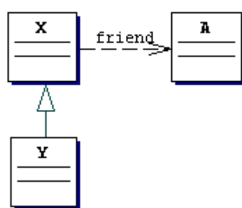
B kein `friend` von X

Beispiel:

```
// friend2.cpp: Vererbung von friends ?

class X {
    friend class A;
public:
    X (int i = 0):xi(i) {}
private:
    int xi;
};

class A {
public:
   int f(const X*xp){return xp->xi;} // moeglich, da A friend von X
};
class B : public A {
public:
    void zweiX (X* xp) { xp->xi *=2; }
                     // nicht moeglich, da B nicht friend von X
```

```
};
```

2. Situation



Zugriff von Elementfunktionen von A auf `private`-Merkmale von X möglich

A kein `friend` von Y

Beispiel:

```
// friend3.cpp: Vererbung von friends ?

class X {
    friend class A;
public:
    X (int i = 0):xi(i) {}
private:
    int xi;
};

class Y : public X {
public:
    Y (int i):X(i), yi(i*i) {}
private:
    int yi;
};

class A {
public:
    int f(const X* xp) { return xp->xi; }
    // moeglich, da A friend von X
    int g(const Y* yp) { return yp->yi; }
    // nicht moeglich, da A nicht friend von Y ist !
};
```

Virtuelle `friend`-Funktionen?

Es wurde bereits gesagt, dass `friend`-Funktionen nicht virtuell sein können. Oft wird aber gerade bei überladenen Operatoren das dynamische Binden benötigt. Angenommen in unserem früheren Beispiel mit der Personenhierarchie hätten wir die Ausgabe wie folgt implementiert:

```
// Ausgabe mit operator<<
   for (int i = 0; i < 5; ++i) {
      cout << *persTab[i] << endl;
   }
```

Wie kann man die Ausgabe polymorph implementieren, so dass die jeweiligen Objekte, auf die `persTab[i]` zeigt, korrekt ausgegeben werden?

Lösung: Implementiere zu der Klasse Person die Operatoren >> und << mit Hilfe der virtuellen Elementfunktionen `ausgeben()` und `eingeben()`:

```
// Ein-/Ausgabe mit << und >>
ostream& operator<< (ostream& o, const Person& p) {
   p.ausgeben(o);  // Aufruf der Version des jeweiligen Objekttyps
```

```
   return o;
}
istream& operator>> (istream& in, Person& p) {
   p.eingeben(in); // Aufruf der Version des jeweiligen Objekttyps
   return in;
}
```

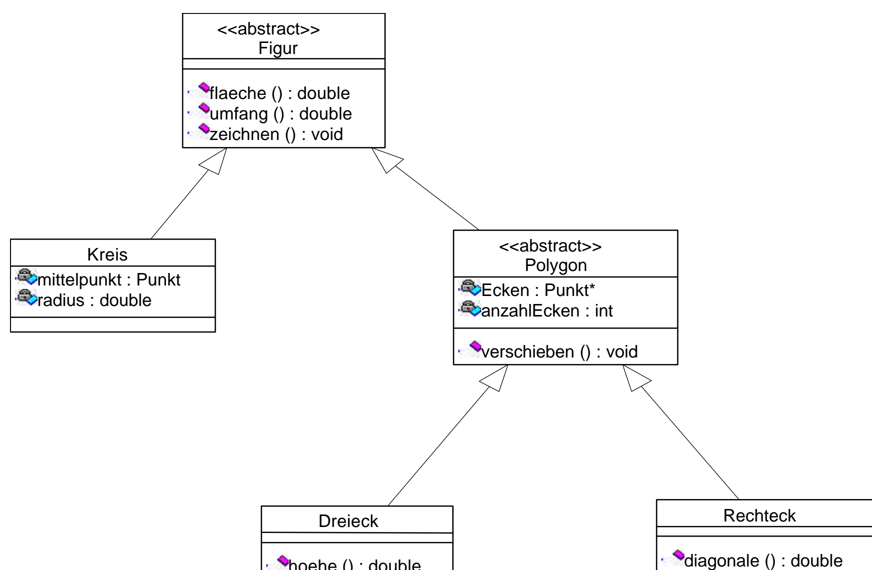
Betrachte nun das folgende Testprogramm:

```
// persont4.cpp: 4. Testprogramm fuer die Klassen Person, Mitarbeiter
//              und Student: dynamisches Binden und friends

int main() {
   Person * persTab[3];
   persTab[0] = new Person("Schmitt", "Hans");
   persTab[1] = new Student("Meier", "Fritz", 1111111);
   persTab[2] = new Mitarbeiter("Adam", "Albert", 4711);

// "virtuelle" Ausgabe mit operator<<
   for (int i = 0; i < 3; ++i) {
      cout << *persTab[i] << endl;
   }

// "virtuelle" Eingabe mit operator>>
   for (int i = 0; i < 3; ++i) {
      cout << "\npersTab[" << i << "]:\n";
      cin >> *persTab[i];
   }

   for (int i = 0; i < 3; ++i) {
      cout << *persTab[i] << endl;
   }
   return 0;
}
```

14.9 Abstrakte Klassen

Oft besteht die Notwendigkeit, Methoden einer Klasse zuzuordnen, ohne dass man dafür eine Implementierung angeben kann.

Beispiel: Geometrische Figuren (Diagramm mit *Rational Rose* erstellt)



Mögliche Methoden: `flaeche()`, `umfang()`, `zeichnen()`
Problem: `flaeche()` ist in der Klasse `Figur` nicht implementierbar !

Lösung: `flaeche` in Figur als *rein-virtuelle Funktion* deklarieren (*pure virtual function*), d. h. als virtuelle Funktion ohne Implementierung.
Anderer Begriff: *abstrakte Methode*

Beispiel:

```
// geom1.h: Geometrische Figuren (rudimentaere Klassenhierarchie)

class Figur {
public:
   virtual double Flaeche() = 0;  // rein virtuelle Funktionen
   virtual double Umfang()  = 0;
   virtual void Zeichnen()  = 0;
   virtual ~Figur() {}
};

// Polygonklasse, abstrakt, da Flaeche nicht implementiert
class Polygon : public Figur {
public:
// ...
   virtual void Zeichnen()
   { /* ... */     }
   virtual double Umfang()
   { /* ... */     }
   virtual void Verschieben()
   { /* ... */     }
private:
   Punkt* Ecken;
   int Anzahl_Ecken;
};

class Rechteck : Polygon {
public:
// ...
   virtual double Flaeche()
   { /* ... */     }
};

class Dreieck : Polygon {
public:
// ...
   virtual double Flaeche()
   { /* ... */     }
};

class Kreis : public Figur {
public:
// ...
   virtual void Zeichnen()
   { /* ...  */    }
   virtual double Umfang()
   { return 2 * pi * radius;   }
   virtual void Verschieben()
   { /* ... */     }
   virtual double Flaeche()
   { return pi * radius * radius;  }
private:
   Punkt mittelpunkt;
   double radius;
```

```
};
```

Eine Klasse, die eine rein-virtuelle Funktion enthält, ist eine *abstrakte Klasse* (*aufgeschobene Klasse*), das heißt es können keine Objekte dieser Klasse erzeugt werden.

- Referenzen und Zeiger auf abstrakte Klassen sind zulässig
- Eine abstrakte Klasse kann ausschließlich abgeleiteten Klassen als Basisklasse dienen.
- Man kann trotzdem eine Implementierung für eine rein-virtuelle Funktion definieren. Diese Implementierung wird als Standardimplementierung für abgeleitete Klassen verwendet.
- Wenn eine rein-virtuelle Funktion deklariert wird, muss die Unterklasse diese Funktion explizit implementieren oder sie unverändert übernehmen. Dann ist die abgeleitete Klasse allerdings ebenfalls abstrakt.

Rein virtuelle Destruktoren:

```
virtual ~X()=0;
```

Rein virtuelle Destruktoren sind möglich, aber es muss trotzdem eine konkrete Implementierung für einen rein virtuellen Destruktor angegeben werden.

```
X::~X() {}
```

Grund: Beim Destruktoraufruf einer Nachkommenklasse wird auf jeden Fall der Destruktor auch einer abstrakten Vorfahrenklasse aufgerufen, folglich muss eine Implementierung für diesen vorhanden sein.

Bemerkung: Rein virtuelle Destruktoren werden höchstens dazu benötigt, um eine Klasse, die ansonsten keine rein virtuellen Methoden hat, zur abstrakten Klasse zu machen.

Beispiel Anwendung von abstrakten Klassen als Vorgabe von Schnittstellen

```
// Filterklasse (vgl Stroustrup §25.8)

#include <iostream>

class Filter {
public:
   virtual void start() {}    // Verarbeitung initialisieren
   virtual int  read() = 0;   // Daten lesen
   virtual void write() {};   // Daten schreiben
   virtual void compute() {}  // Daten verarbeiten
   virtual int  result() = 0; // Verarbeitung beenden
                              // Ergebnis zurueckgeben
   virtual ~Filter() {}
};

int main_loop(Filter* p) {
   p->start();
   while (p->read()) {
      p->compute();
      p->write();
   }
   return p->result();
}
```

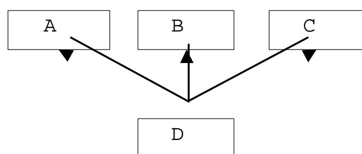
Die Klasse Filter gibt eine universelle Schnittstelle für verschiedene Arten von Programmen vor, die eine Datenquelle sequentiell zu durchlaufen haben und irgendwelche Filteroperationen dort vornehmen. Eine konkrete Implementierung kann dann z. B. so aussehen:

```
// Filterklasse (vgl Stroustrup §25.8)
```

```
#include "filter.cpp"
#include <iostream>

class MyFilter1 : public Filter {
public:
   MyFilter1(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) {}
   int  read();
   void compute() { nchar++; }
   int  result();
private:
   istream &is;
   ostream& os;
   long nchar; // Anzahl gelesene Zeichen
};

int MyFilter1::read() {
   char c;
   is.get(c);
   return is.good();
}

int MyFilter1::result() {
   os << nchar << " Zeichen gelesen\n";
   return 0;
}

int main() {
   Filter *fp = new MyFilter1(cin, cout);
   return main_loop(fp);
}
```

14.10 Mehrfachvererbung

Eine Klasse kann von mehreren Basisklassen direkt erben.




```
class A { ... };
class B { ... };
class C { ... };
class D : public A, public B, public C { ...};
```

D erbt alle Merkmale von A, B, C. Die Reihenfolge in der die Basisklassen angegeben sind, beeinflusst die Reihenfolge, in der die Konstruktoren und Destruktoren aufgerufen werden.

hier: `D::D(…):A(…), B(…), C(…) { ... }`

Bemerkung:

- Mehrfachvererbung ist umstritten.
- Es gibt objektorientierte Programmiersprachen mit
  - Einfachvererbung: Smalltalk, Objective C, Object Pascal, Object Cobol
  - Mehrfachvererbung: C++(seit C++ 2.0), Eiffel, CLOS
  - Einfachvererbung + Mehrfachvererbung von Interfaces: Java

Beispiel:

```
// merf1.cpp: Abstraktes Beispiel fuer Mehrfachvererbung
class A {
public:
   A(int i): a(i) {}
   virtual void print() {
      cout << "A::print() a = " << a << endl;
   }
protected:
   int a;
};

class B {
public:
   B(float x): b(x) {}
   virtual void print() {
      cout << "B::print() b = " << b << endl;
   }
protected:
   float b;
};

class AB : public A, public B {
public:
   AB(int i, float x): A(i), B(x) {}
};

int main() {
   AB ab(1,3.14);
   ab.print();   //Problem: Aufruf ist nicht eindeutig
   return 0;
}
```

Möglich ist nur:

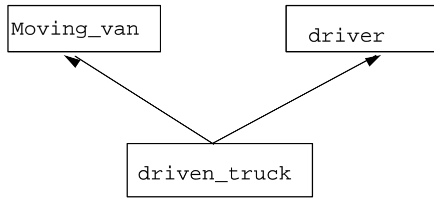
```
ab.A::print();
ab.B::print();       // sind beide eindeutig
```

Lösung: eigene `print`-Methode in AB definieren

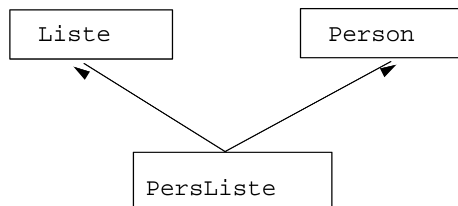
```
// merf1.cpp korrigiert:

class AB : public A, public B {
public:
   AB(int i, float x): A(i), B(x) {}
   virtual void print() {
      cout << "AB::print()\n";
      A::print();
      B::print();
   }
};

int main() {
   AB ab(1,3.14);
   ab.print(); // <-- Aufruf eindeutig !
   return 0;
}
```

Sinnvolle und weniger sinnvolle Beispiele für Mehrfachvererbung

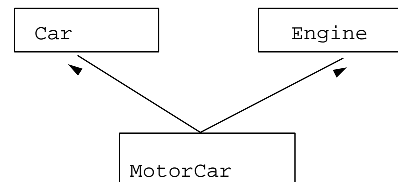
Internet Kurs C++:



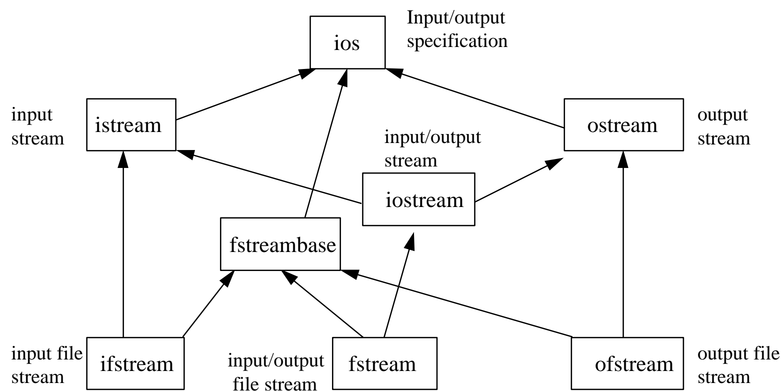
Fern-Uni Hagen:



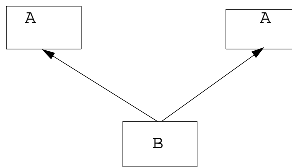
C++ Annotations:



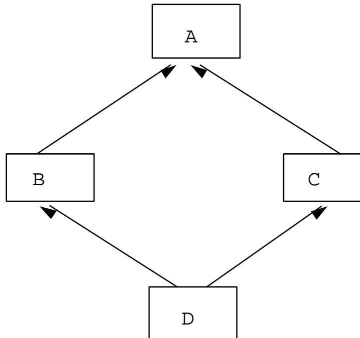
iostreams:



Wiederholtes Erben

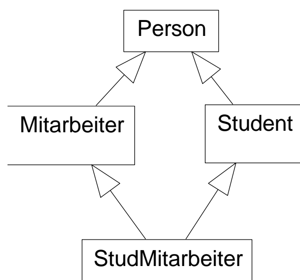


Direktes wiederholtes Erben ist in C++ nicht erlaubt



Indirektes wiederholtes Erben ist in C++ unbeschränkt erlaubt. Normalerweise wird A tatsächlich komplett zweimal geerbt. Jedes Merkmal von A ist in D doppelt vorhanden.

Beispiel: Erweiterung der Personenhierarchie



Beispiel:

```
// studmitarb.h: Klasse fuer studentische Mitarbeiter

#include "student.h"
#include "mitarbeiter.h"

class StudMitarb : public Student, public Mitarbeiter {
public:
   StudMitarb();
   StudMitarb(string n, string vn, long mnr, long pnr);
   void eingeben(istream&);
   void ausgeben(ostream&) const;
};

StudMitarb::StudMitarb() {}

StudMitarb::StudMitarb(string n, string vn, long mnr, long pnr)
   : Student(n, vn, mnr), Mitarbeiter(n, vn, pnr) {}

void StudMitarb::ausgeben(ostream& o) const {
   Student::ausgeben(o);
   Mitarbeiter::ausgeben(o);
}

void StudMitarb::eingeben(istream& in) {
   Student::eingeben(in);
   in >> personalNr;
}
```

Testprogramm:

```
// persont5.cpp: 5. Testprogramm wiederholtes Erben

#include "studmitarb.h"

int main() {
   StudMitarb s1 ("Meier", "Hans", 1234567, 4711);
   StudMitarb s2;
   s2.eingeben(cin); // Meier Sabine
                     // 123456 56789

   cout << "s1:\n"; s1.ausgeben(cout); cout << endl;
   cout << "s2:\n"; s2.ausgeben(cout); cout << endl;
}

/* Ausgabe:
s1:
Meier, Hans     Matr-Nr: 1234567
Meier, Hans     Pers-Nr: 4711
s2:
Meier, Sabine   Matr-Nr: 123456
,       Pers-Nr: 56789          */
```

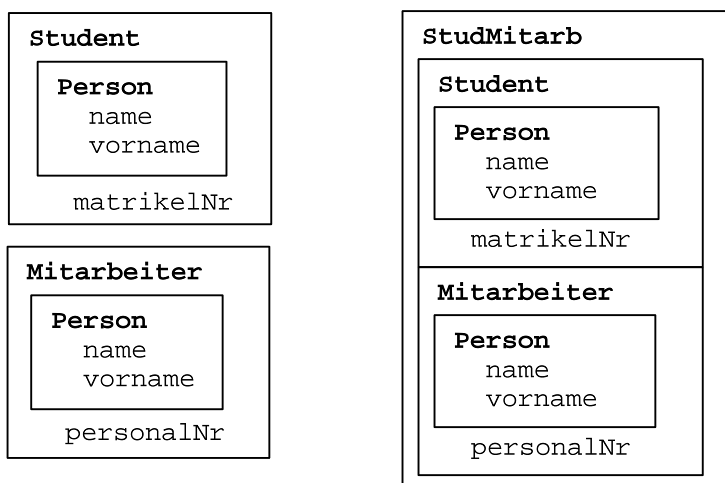
Wie ist diese Ausgabe zu erklären?

In C++ sind die Attribute der Basisklasse so oft vorhanden, wie diese geerbt werden. In unserem Beispiel heißt das, dass die Attribute `name` und `vorname` in der Klasse `StudMitarb` jeweils zweimal vorhanden sind. Auch der Aufruf der von Person geerbten Elementfunktionen `setName()`, `setVorname()`, ... ist mehrdeutig. Möglich ist aber z. B.

```
      s2.Mitarbeiter::setName("Hubert"); // eindeutig!
      s2.Mitarbeiter::setVorname("Hans");
      cout << "s2:\n"; s2.ausgeben(cout); cout << endl;

   /* Ausgabe:
   s2:
   Meier, Sabine   Matr-Nr: 1234567
   Hubert, Hans    Pers-Nr: 56789   */
```

Unsere studentische Mitarbeiterin hat nun also auf einmal zwei Namen und Vornamen!

Interne Struktur der Objekte



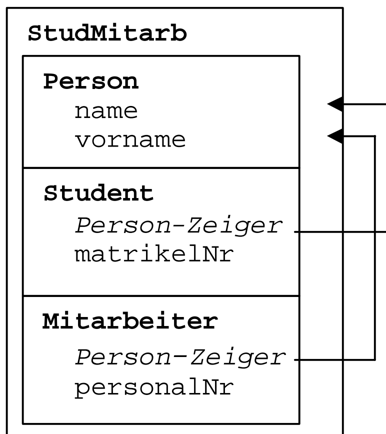
Lösung: Virtuelle Basisklassen.

Beispiel:

```
class Mitarbeiter : public virtual Person { ... };
class Student     : public virtual Person { ... };
```

Wenn im Zuge einer Ableitung eine Basisklasse durch Angabe des Schlüsselwortes `virtual` als virtuell erklärt wird, wird in den folgenden Ableitungen nur jeweils ein Exemplar dieser Basisklasse übernommen.

Interne Struktur bei virtueller Basisklasse



Testprogramm:

```
// persont6.cpp: 6. Testprogramm wiederholtes Erben
//               mit virtueller Basisklasse

#include "studmitarb.h"

int main() {
   StudMitarb s1 ("Meier", "Hans", 1234567, 4711);
   StudMitarb s2;
   cin >> s2; // Eingabe mit >> funktioniert jetzt

   cout << "s1:\n" << s1 << endl; // ebenso die Ausgabe mit << !
   cout << "s2:\n" << s2 << endl;

   s2.setName("Hubert");   // eindeutig aufrufbar!
   s2.setVorname("Hans");
   cout << "s2:\n" << s2 << endl;
}
```

Regeln:

- Virtuelle Basisklassen werden vor allen anderen Klassen konstruiert, in der Reihenfolge, in der sie in der Deklaration der abgeleiteten Klasse auftreten.

  Im Beispiel heißt das, dass bei der Konstruktion eines `StudMitarb`-Objekts vom Konstruktor von `StudMitarb` zuerst der Konstruktor von `Person` aufgerufen wird und dann erst die Konstruktoren von `Student` und `Mitarbeiter`.

- Der explizite Konstruktoraufruf für die virtuelle Basisklasse kann entfallen, wenn diese einen Standardkonstruktor hat.

- Der Aufruf der Destruktoren erfolgt in genau umgekehrter Reihenfolge wie der Aufruf der Konstruktoren, das heißt die virtuelle Basisklasse wird immer zuletzt destruiert.
- Alle Teilobjekte einer virtuellen Basisklasse werden vom Konstruktor der am weitesten abgeleiteten Klasse initialisiert.
- Initialisierungen für virtuelle Basisklassen, die im Konstruktor einer Klasse spezifiziert sind, die nicht die am weitesten abgeleitete Klasse ist, werden ignoriert.

  Beispiel:

  ```
  Student::Student(string n, string vn, long mnr)
     : Person(n, vn), matrikelNr(mnr) {}

  Mitarbeiter::Mitarbeiter(string n, string vn, long pnr)
     : Person(n, vn), personalNr(pnr) {}

  StudMitarb::StudMitarb(string n, string vn, long mnr, long pnr)
     : Person(n, vn), Student(n, vn, mnr), Mitarbeiter(n, vn, pnr) {}
  ```

  Bei der Konstruktion von `StudMitarb` wird der Aufruf des `Person`-Konstruktors durch den Konstruktor von `Student` und `Mitarbeiter` ignoriert.

Empfehlung:

Mehrfachvererbung und vor allem wiederholtes Erben sollte möglichst vermieden werden.

15 Parametrisierte Funktionen und Klassen (Templates)

Ein weiteres sprachliches Hilfsmittel, um die Wiederverwendbarkeit von Software zu erhöhen, sind bei C++ die sogenannten *Templates*. Mit Templates können Schablonen definiert werden, mit deren Hilfe ganze Familien von ähnlichen Funktionen und Klassen generiert werden können.

15.1 Parametrisierte Funktionen (Funktions-Templates)

Funktionsmakros in C:

```
#define MIN (a,b) ((a)<(b)?(a):(b))
```

Nachteile:

- nicht typsicher
- Probleme beim Debuggen, da `MIN` nicht in der Symboltabelle
- Wie wird der Ausdruck `MIN(a++,b)` ausgewertet ?

Bisherige C++-Lösung:

```
inline int Min(int a, int b) {
   return (a < b ? a : b);
}
```

Vorteile:

- typsicher
- ++-Problem beseitigt

Nachteile:

- nur für `int` definiert, was ist z. B. mit `float`, `double`, `char*` oder gar benutzerdefinierten Typen ?

Eine bessere Lösung stellen Funktions-Templates bzw. parametrisierte Funktionen dar.

Beispiel

```
// min01.cpp: Beispiel für ein Funktions-Template
//
template <class T>
T Min (T a, T b) {
   return a < b ? a : b;
}

double Min (double a, long b) {
   return a < b ? a : b;
}

int main() {
   cout << Min(10,20)      << endl; // erzeuge Min (int,int)
   cout << Min(10.5,20)    << endl; // Aufruf  Min (double, long)
   cout << Min(10.5,20.40)<< endl; // erzeuge Min (double, double)
   return 0;
}
```

Der Compiler erzeugt aus dem Funktions-Template, also der Schablone, konkrete Template-Funktionen, indem er den formalen Parameter T durch konkrete Typparameter ersetzt. Dieser Vorgang wird auch *Instantiierung* genannt. Die erzeugten Template-Funktionen werden anschließend wie ganz normale Funktionen übersetzt.

Formale Definition bzw. Deklaration von Funktions-Templates

```
template <formale Parameter>
Deklaration oder Definition der Funktion...
```

Regeln:

- *formale Parameter*: Liste von Parametern durch Komma voneinander getrennt (Typ-Parameter). Jeder Typ-Parameter wird mit dem Schlüsselwort `class` eingeleitet. Alternativ sieht der ANSI-Standard hierfür das Schlüsselwort `typename` vor.

  ```
  template <class S, class T>
  ```

  oder auch

  ```
  template <typename S, typename T>
  ```

- Die formalen Parameter stehen jeweils für beliebige Datentypen, z. B.: `int`, `double`, `char`, `int*`, `char*` oder auch benutzerdefinierte Datentypen

- Die formalen Parameter sollen in der Argumentliste der Funktionsdefinition vorkommen.

  ```
  template <class T>
  void f (int a){...}     // falsch

  template <class T>
  void f (T a)            // korrekt
  ```

  Die formalen Parameter können innerhalb des Funktions-Template als beliebige Typangaben verwendet werden.

  ```
  template <class T>
  void f (T a){
     T t;
     ...
  }
  ```

- Der Geltungsbereich der formalen Parameter ist die Funktionsdefinition.

Beispiel:

```
// min03.cpp: Funktions-Template zur Minimumsbestimmung

// Min: bestimme den Minimalwert des Arrays tab
template <class Typ>
Typ Min (Typ* tab, int groesse) {
   Typ minimum = tab[0];
   for (int i = 1; i < groesse; i++)
      if (tab[i] < minimum)
         minimum = tab[i];

   return minimum;
}

int main() {
   int a[] = { 12, 23, 4, 15, 60 };
   cout << "Minimum von a: " << Min (a, 5) << endl;
```

```
   double b[] = { 3.12, 2.3, 0.4, 2.15, 6.0, 0.123 };
   cout << "Minimum von b: " << Min (b, 6) << endl;

   string c[4];
   for (int i = 0; i < 4; i++) {
      cout << "c[" << i << "]=";
      cin >> c[i];
   }
   cout << "Minimum von c: " << Min (c, 4) << endl;
   return 0;
}
```

Der Compiler erzeugt beim Übersetzen drei Instanzen der Funktion `Min`:

- `int Min(int*, int)`
- `double Min(double*, int)`
- `String Min(String*, int)`

Bei der *Instantiierung* durch den Compiler wird zunächst nicht überprüft, ob die verwendeten Operationen für den einzusetzenden Datentyp alle existieren. Erst bei der expliziten Compilierung der generierten Funktionsinstanz kann dies überprüft werden.

Vorgehensweise bei der Instantiierung

Die Typen der Funktionsargumente beim Aufruf einer Funktion müssen exakt zur `template`-Definition passen.

Beispiel:

```
template <class Typ>
Typ Min (Typ*, int);
   …
template <class Typ>
Typ Min (Typ, Typ);
   …
// Aufruf im Hauptprogramm
   Min(10.0, 20);
// Compiler lehnt dies ab, da fuer diesen Funktionsaufruf keine
// passende Funktion existiert
```

Ist allerdings bereits eine Funktionsinstanz erzeugt worden, so gelten hierfür die normalen Regeln für das Auffinden der passenden Funktion.

```
   Min(10.0, 20.5);  // erzeuge Instanz double Min(double, double);
   Min(10.0, 20);    // Aufruf Min(double, double) !
```

Explizite Instantiierung von Funktions-Templates

Normalerweise werden spezielle Instanzen erst erzeugt, wenn konkrete Funktionsaufrufe vorliegen. Man kann aber durch eine explizite Deklaration die Instantiierung von speziellen Template-Funktionen erzwingen:

```
double Min(double, double);        // normaler Funktionsprototyp
                                   // veraltete Form
template int Min<int> (int, int); // neuere Syntax, eindeutiger
                                   // Bezug auf das Funktions-Template
```

Explizite Spezifizierung von Template-Funktionen

Beim Funktionsaufruf kann explizit die gewünschte Instanz des Funktions-Templates angegeben werden:

```
template <class Typ>
```

```
Typ Min (Typ, Typ);
    ...
// Expliziter Aufruf von Min (int,int):
Min<int>(i, j);   // Compiler erzeugt die <int>-Instanz
Min<double>(i, x) // Compiler erzeugt die <double>-Instanz
```

Dies erlaubt auch die Definition von Funktions-Templates, bei denen der Typparameter nicht als Argument vorkommt.

```
// create.cpp: Funktions-Template ohne Argument

template <class T>
T* create() {
  return new T;
}

int main() {
   int* ip = create<int>();            // allokiere ein int
   double *dp = create<double>();      // allokiere ein double
   return 0;
}
```

Bei der expliziten Spezifizierung ist es auch möglich, nur einen Teil der Typparameter zu spezifizieren.

```
// convert.cpp: Konvertieren von einem Typ in einen anderen

template <class TYP2,class TYP1>
TYP2 convert(const TYP1& t) {
   return t;   // implizite Konvertierung zu TYP2
}

int main() {
   int i = 5;
   double x = convert<double>(i); // TYP2 ist double, TYP1 ist int
   return 0;
}
```

Spezialisierung einer parametrisierten Funktion

Situation:

```
Min("abc ", "abc") // was passiert?
```

Die ursprüngliche `Min`-Funktion vergleicht nur die Adressen der `char`-Strings !

Lösung:

```
template <class T>
T Min (T a, T b) {
   return a < b ? a : b;
}

const char* Min(const char* s1, const char* s2) {
   return (strcmp(s1,s2) < 0) ? s1 : s2;
}
```

Bei allen Aufrufen von Min mit zwei Argumenten vom Typ `const char*` wird die spezialisierte Instanz aufgerufen.

Explizite Spezialisierung einer parametrisierten Funktion

Im ANSI-Standard ist eine Syntax für die explizite Definition einer Spezialisierung einer parametrisierten Funktion vorgesehen. Das Beispiel von oben sieht damit wie folgt aus:

```
template<>
const char* Min<const char*>(const char* s1, const char* s2) {
   return (strcmp(s1,s2) < 0) ? s1 : s2;
}
```

Ist eine explizite Spezialisierung vorhanden, so ist es dem Compiler unmöglich, eine zusätzliche eigene Instanz zu diesem Typ zu erzeugen. Bei der obigen impliziten Spezialisierung ist dies unter bestimmten Konstellationen durchaus noch denkbar.

Überladen von Funktions-Templates

Ein Funktions-Template kann überladen werden, sofern die Identifikation aller Instanzen anhand des Typs oder der Anzahl der Parameter eindeutig möglich ist.

Vorgehensweise des Compilers bei einer aufzurufenden Funktion im Falle überladener Funktionen und Funktions-Templates:

- Untersuche alle konkreten Funktionen, vergleiche Typen und Anzahl der Parameter auf exakte Übereinstimmung.
- Untersuche die Funktions-Templates, ob daraus eine exakt passende Instanz erzeugt werden kann.
- Üblicher Mechanismus für überladene Funktionen. Hierzu werden auch die bereits erzeugten Template-Funktionen herangezogen.

Beispiel:

```
//quicksort.h: Parametrisierte Version von Quicksort

template <class Typ>
void quicksort (Typ *a, int links, int rechts)
{
   if (links < rechts) {
      int i = links;
      int j = rechts;
      Typ v = a[(links + rechts)/2];
      do {
         while (a[i] <  v  ) ++i;
         while ( v   < a[j]) --j;
         if (i <= j) {
            tausche (a, i, j);
            ++i; --j;
         }
      } while ( i <= j);

      quicksort (a, links, j);
      quicksort (a, i, rechts);
     }
}

// Zwei Elemente eines Arrays vertauschen
template <class Typ>
void tausche (Typ *a, int i, int j)
{
   Typ t = a[i];
   a[i]  = a[j];
   a[j]  = t;
}
```

```
// Array im Format < 1 2 3 4 5 > ausgeben
template <class Typ>
void ausgeben (Typ *a, int groesse) {
   cout << "< ";
   for (int i=0; i < groesse; i++)
      cout << a[i] << " ";
   cout << ">" << endl;
}

int main() {
   int ia[] = { 234, 867, 2, 34, 12, 23, 4, 15, 60, 34, 47 };

   double da[] = { 37.7, 59.6, 48.7, 2.5, 3.12, 7.6, 4.8,
                   2.3, 0.4, 2.15, 6.0, 0.123, 3.1, 3.2  };

   int groesse = sizeof(ia) / sizeof(int);
   cout << "Quicksort des int-Arrays (Groesse == "
        << groesse << ")" << endl;
   quicksort (ia, 0, groesse - 1);
   ausgeben  (ia, groesse);

   groesse = sizeof(da) / sizeof(double);
   cout << "Quicksort des double-Arrays (Groesse == "
        << groesse << ")" << endl;
   quicksort (da, 0, groesse - 1);
   ausgeben  (da, groesse);

   groesse = 6;
   String sa[6];
   for (int i = 0; i < groesse; i++) {
      cout << "sa[" << i << "]=";
      cin >> sa[i];
   }
   cout << "Quicksort des String-Arrays (Groesse == "
        << groesse << ")" << endl;
   quicksort (sa, 0, groesse - 1);
   ausgeben  (sa, groesse);
   return 0;
}
```

Problem bei Funktions-Templates

Zum Instantiieren wird immer die komplette Quelle des Funktions-Templates benötigt. Besteht aber die Anwendung aus mehreren Dateien und wird in verschiedenen Quelldateien ein und dasselbe Funktions-Template benutzt, so kann es vorkommen, dass die selbe Instanz mehrfach erzeugt wird. Allerdings können die Linker der heute verfügbaren Compiler dieses Problem mittlerweile lösen.

15.2 Parametrisierte Klassen (Klassen-Templates)

Ähnlich wie für Funktionen können auch für Klassen sogenannte Klassen-Templates, also Schablonen zum Generieren von Klassen definiert werden.

15.2.1 Einführung

Eine parametrisierte Klasse (Klassen-Template, Klassenschablone, generische Klasse) definiert eine Schablone für die Erzeugung von Klassen, die sich jeweils durch konkrete Datentypen unterscheiden, die für formalen Typparametern eingesetzt werden.

Beispiel: Klasse `IntMenge` aus Abschnitt 12

Problem: Wie kann man nun eine `double`-Menge oder eine `string`-Menge implementieren?
Lösung: Parametrisierte Klasse `Menge` mit einem Typparameter `Typ` für die Elemente der Menge.

```
// Menge1: Eine parametrisierte Klasse fuer Mengen

template <class Typ>
class Menge {
public:
   Menge (int m);                        // Menge von maximal m Elementen
   Menge (const Menge&);                 // Kopierkonstruktor
   ~Menge();                             // Menge destruieren
   Menge& operator= (const Menge&);      // Zuweisungsoperator

   bool istElement(const Typ& t) const;// Ist t ein Element der Menge ?
   void einfuege  (const Typ& t);       // t einfuegen in die Menge
   void loesche   (const Typ& t);       // t loeschen aus der Menge
   bool leer      () const;             // Ist die Menge leer ?
   bool voll      () const;             // Ist die Menge voll ?
   int  anzahl    () const;             // Anzahl Elemente

   // Iterator zur Menge
   typedef const Typ* iterator;   // Adresse auf einen Eintrag
   iterator begin() const {  return tab;           }
   iterator end  () const {  return tab + aktAnzahl; }
private:
   Typ *tab;                 // Implementierung als Typ-Array
   int aktAnzahl, maxAnzahl;      // Aktuelle Anzahl, maximale Anzahl
};

template <class Typ>
inline bool Menge<Typ>::leer() const
{   return aktAnzahl == 0; }

template <class Typ>
inline bool Menge<Typ>::voll() const
{   return aktAnzahl >= maxAnzahl; }

template <class Typ>
inline int Menge<Typ>::anzahl() const
{   return aktAnzahl; }
```

```
// Konstruktor: Legt Array mit m Elementen an
// Vorbedingung: M >= 1
template <class Typ>
Menge<Typ>::Menge (int m ) {
```

```
   assert(m >=1);
   aktAnzahl = 0;
   maxAnzahl = m;
   tab = new Typ[maxAnzahl];
}

// Destruktor: Entfernen des int-Arrays
template <class Typ>
Menge<Typ>::~Menge() {
   delete [] tab;
}

// Kopierkonstruktor
template <class Typ>
Menge<Typ>::Menge (const Menge<Typ>& m)
   : aktAnzahl(m.aktAnzahl),
     maxAnzahl(m.maxAnzahl)
{
   tab = new Typ[maxAnzahl];
   for (int i=0; i < maxAnzahl; i++)
      tab[i] = m.tab[i];
}
// Zuweisungsoperator
template <class Typ>
Menge<Typ>& Menge<Typ>::operator=(const Menge<Typ>& m){
   if (this == &m)   // Zuweisung an sich selbst ?
      return *this;
   if (maxAnzahl != m.maxAnzahl){
      delete [] tab;
      maxAnzahl = m.maxAnzahl;
      tab = new Typ[maxAnzahl];
   }
   aktAnzahl = m.aktAnzahl;
   for (int i=0; i < maxAnzahl; i++)
      tab[i] = m.tab[i];
   return *this;
}

// Einfuegen eines Elements in die Menge
// Vorbedingung: Menge ist nicht voll
// Keine Sortierung da Typ beliebig sein kann
template <class Typ>
void Menge<Typ>::einfuege(const Typ& t) {
   assert(!voll());
   if (istElement(t)) return;
   int i = aktAnzahl++;
   Typ tmp;
   tab[i] = t;
}

// Loeschen eines Elements aus der Menge
template <class Typ>
void Menge<Typ>::loesche(const Typ& t) {
   int i;
   for (i = 0; i < aktAnzahl && t != tab[i]; ++i);
   if (i < aktAnzahl) {
      for (int j = i+1; j < aktAnzahl; ++j)
         tab[j-1] = tab[j];
      --aktAnzahl;
   }
}

// Ist t Element der Menge ?
template <class Typ>
```

```
bool Menge<Typ>::istElement(const Typ& t) const {
   int i;
   for (i = 0; i < aktAnzahl && t != tab[i]; ++i);
   return (i < aktAnzahl);
}
```

Anwendung:

```
Menge <int> M(1000);      // int-Menge mit 1000 Elementen
Menge <double> DM(100);   // double-Menge mit 100 Elementen
Menge <string> SM(50);    // String-Menge mit 50 Elementen

// Vereinfachung:
typedef Menge <int> IntMenge;
IntMenge M2(100);
```

Testprogramm für die parametrisierte Klasse Menge:

```
// menge1t: Testprogramm zur Klasse Menge

// Menge ausgeben
template <class Typ>
void ausgabe(Menge<Typ>& m) {
   Menge<Typ>::iterator i;
   for (i = m.begin(); i != m.end(); ++i)
      cout << *i << "  ";
   cout << endl;
}

enum FunktionsTyp { beenden, einfuege, istElement, loesche, ausgeben,
                    kopie, zuweis};

void menue() {
   cout << einfuege   << ": einfuegen / "
        << istElement << ": ist Element / "
        << loesche    << ": loeschen / "
        << ausgeben   << ": ausgeben / \n"
        << kopie      << ": Kopierkonstruktor / "
        << zuweis     << ": Zuweisungsoperator/ \n"
        << beenden    << ": beenden ";
}

int main() {
   const int MAX=1000;
   int anzahl;
   cout << "Maximale Anzahl ? "; cin >> anzahl;

   Menge<double> m(anzahl);
   while (!m.voll()) {
      m.einfuege(randint(MAX)); // Zufallszahl einfuegen
   }
   cout << "Ausgabe der Menge: \n"; ausgabe(m);

// Interaktiver Testrahmen
   int funktion;
   Menge<double>* mp = new Menge<double>(m);
   double t;
   do {
      menue(); cin >> funktion;
      switch (funktion) {
      case kopie     :  mp = new Menge<double>(m);
                        break;
      case zuweis    :  *mp = m;
                        break;
      case einfuege  :  cout << "Wert: "; cin >> t;
                        m.einfuege(t);
```

```
                         break;
      case istElement:  cout << "Wert: "; cin >> t;
                         if (m.istElement(t))
                            cout << t << " enthalten !\n";
                         else
                            cout << t << " nicht enthalten !\n";
                         break;
      case loesche   :  cout << "Wert: "; cin >> t;
                         m.loesche(t);
                         break;
      case ausgeben  :  cout << "m : "; ausgabe(m);
                         cout << "mp: "; ausgabe(*mp);
                         break;
      case beenden   :  break;
      default        :  cout << "Falsche Funktion!\n";
      }
   } while (funktion != beenden);
   return 0;
}
```

Formale Definition bzw. Deklaration

```
template <formale Parameterliste>
class Klassenname [: Basisklasse ...]
{ ... };
```

Die *formale Parameterliste* kann enthalten:

- Typparameter der Form:
  `class TYP bzw. typename TYP`
- Parameter der Form

  `int groesse, char c, char* p, double& d, ...`

  erlaubt sind dabei ganzzahlige Typen, sowie Zeiger und Referenzen

Deklaration konkreter Objekte

`Klassenname <`*`aktuelle Parameter`*`> Objektname`

aktuelle Parameter:

- Konkrete Typangabe, z. B. `double`
- Tatsächlicher Wert, z.B. `512`

Beispiel:

```
// puffer.cpp: Klasse Puffer

template <class T, int groesse>
class Puffer {
public:
   Puffer();
  ~Puffer() { delete [] tab; }
   void einfuegen(T);
   void leeren();
   bool voll() const;
   friend ostream& operator<< (ostream&, const Puffer&);
private:
   Puffer( const Puffer<T,groesse>& );
   Puffer& operator=(const Puffer&);
   T* tab;
   int anzahl;
};
```

```
template <class T, int groesse>
Puffer<T,groesse>::Puffer () {
   anzahl = 0;
   tab = new T[groesse];
}

template <class T, int groesse>
void Puffer<T,groesse>::einfuegen (T t) {
   if (voll()) leeren();
   tab[anzahl++] = t;
}

template <class T, int groesse>
void Puffer<T,groesse>::leeren() {
   if (anzahl > 0){
      for (int i=0; i < anzahl; i++)
         cout << tab[i];
      cout << endl;
      anzahl = 0;
   }
}

template <class T, int groesse>
bool Puffer<T,groesse>::voll(void) const {
   return anzahl == groesse;
}

template <class T, int groesse>
ostream& operator<< (ostream& o, const Puffer<T,groesse>& p) {
   for (int i=0; i < p.anzahl; i++)
      o << p.tab[i] << ' ';
   return o;
}

typedef Puffer<char,1024> CharPuffer;
typedef Puffer<int,10> IntPuffer;

int main() {
   // char-Puffer testen
   Puffer<char,10> cp;
   for (char c = 'a'; c <= 'z'; c++)
      cp.einfuegen(c); // Puffer wird automatisch entleert
   cout << cp << endl;
   cp.leeren();

   // int-Puffer testen
   IntPuffer ip;
   for (int i = 0; i < 100; i++)
      ip.einfuegen(i);
   cout << ip << endl;
   ip.leeren();
   return 0;
}
```

Regeln:

- Alle Elementfunktionen einer parametrisierten Klasse sind automatisch parametrisierte Funktionen mit den selben `template`-Parametern, wie sie die Klasse besitzt.
- Innerhalb der Klassendefinition ist die `template`-Parameterliste bereits implizit Bestandteil des Klassennamens und muss nicht explizit angegeben werden.

```
template <class T, int groesse>
class Puffer {
```

```
...
   Puffer& operator=(const Puffer&);
};
```

- Wird der Name einer parametrisierten Klasse außerhalb der Klassendefinition benutzt, so sind die `template`-Argumente mit anzugeben.
- Bei der Erzeugung einer konkreten Klasse aus einer parametrisierten Klasse wird jeweils ein vollständiger Satz an Elemenfunktionen erzeugt.
- Wie die Implementierung von `operator<<` zeigt, sind friend-Funktionen von Klassen-Templates normalerweise selbst Funktions-Templates und zu jeder instantiierten Template-Klasse gehört genau eine instantiierte Template-friend-Funktion

<u>Typverträglichkeit:</u>

Jede instantiierte Template-Klasse stellt einen eigenen Datentyp dar.

```
Puffer <int, 1024> p1; // Erzeuge Klasse char-Puffer, ersetze groesse
                       // überall durch den Wert 1024
Puffer <int, 512> p2;  // Puffer der Größe 512, nicht typverträglich
                       // zu Puffer <int, 1024>
```

<u>Template-Standardargumente:</u>

Bei der Definition von Klassen-Templates können sogenannte `template`-Standardargumente vergeben werden.

```
template <class T=char, int groesse=1024>
class Puffer { ... };

Puffer <int> ip;            // T=int, groesse=1024
Puffer <> cp;               // T=char, groesse=1024
Puffer <double, 100> dp;    // T=double, groesse=100
Puffer <double, 10.0> dp;   // geht nicht, da 10.0 int sein muss
```

15.2.2 Klassenattribute und Klassenmethoden in parametrisierten Klassen

Genauso wie bei normalen Klassen können auch bei parametrisierten Klassen Klassenattribute und Klassenmethoden definiert werden.

<u>Beispiel:</u>

```
template <class T>
class A {
public:
   A(T t) {b=t;}
   static void f();
private:
   static int a;
   T b;
};
```

Jede konkrete Instanz dieser parametrisierten Klasse hat ihre eigenen Exemplare des Klassenattributs `a` und der Klassenmethode `f()`.

Initialisierung von Klassenattributen:

```
template <class T> int A <T>::a=2000;
```

Initialisieren spezieller Versionen:

```
int A <double>::a=1000;
```

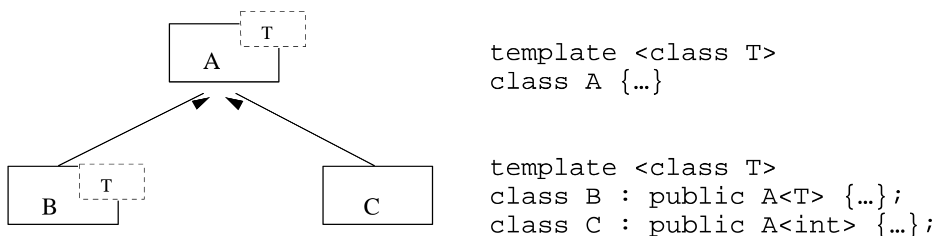
Definition einer Klassenmethode:

```
template <class T>
void A <T>::f() {…}
void A <int>::f() {…}
```

15.2.3 Vererbung bei parametrisierten Klassen

Parametrisierte Klassen können sowohl von parametrisierten als auch von nicht parametrisierten Klassen abgeleitet werden. Umgekehrt können von parametrisierten Klassen sowohl parametrisierte als auch nicht parametrisierte Klassen abgeleitet werden.

Beispiel:




```
template <class T>
class A {…}

template <class T>
class B : public A<T> {…};
class C : public A<int> {…};
```

Beispiel: Eine Stack-Implementierung

```
// stack: Abstraktes Klassen-Template fuer einen Stack

template <class T>
class Stack {
public:
   Stack() : nb_elements(0) {};
   virtual ~Stack() { }

   bool         empty() const  { return (nb_elements == 0); }
   virtual bool full () const  { return false; }
   virtual void push (T x)    = 0;   // Wert in Stack ablegen
   virtual void pop  ()       = 0;   // Wert aus Stack herausholen
   virtual T    top  () const = 0;   // Zuletzt abgelegter Wert
   int          size () const  { return nb_elements; }

protected:
   int nb_elements;   // Anzahl Elemente des Stacks
};
```

```
// Stack-Implementierung mit Hilfe eines Arrays
template <class T>
class FixedStack : public Stack<T> {
public:
   FixedStack (int=FixedStack::stacksize);
   FixedStack (const FixedStack&);
   FixedStack& operator= (const FixedStack&);
  ~FixedStack()         { delete [] tab; }

   bool full  () const  { return (max_size <= nb_elements); }
   void push  (T t);       // Wert in Stack ablegen
   void pop   ();          // Wert aus Stack herausholen
   T top () const;         // Zuletzt abgelegter Wert
   friend ostream& operator<< (ostream&, const FixedStack&);

private:
```

```
   static const int stacksize;
   T *tab;               // Implementierungs-Array
   int max_size;         // Größe des Arrays
   T* tab_copy(T*, int); // Kopieren des internen Arrays
};

template <class T>
const int FixedStack<T>::stacksize=100;
```

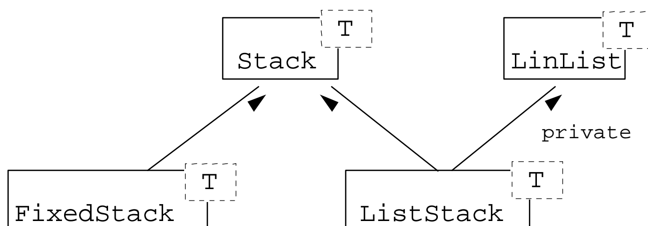
```
template <class T>
FixedStack<T>::FixedStack(int n) {
   assert(n > 0);
   tab = new T[n];
   max_size = n;
}

// Kopieren des T-Arrays
template <class T>
T* FixedStack<T>::tab_copy(T* t, int new_size) {
   T *dp = tab;
   if (max_size != new_size) // nur neu allokieren bei anderer Groesse
      dp = new T[new_size];
   max_size = new_size;
   for (int i = 0; i < max_size; i++)
      dp[i] = t[i];
   return dp;
}

// Kopierkonstruktor
template <class T>
FixedStack<T>::FixedStack(const FixedStack& s) : max_size(0) {
   tab = tab_copy (s.tab, s.max_size);
   nb_elements = s.nb_elements;
}

// Zuweisungsoperator
template <class T>
FixedStack<T>& FixedStack<T>::operator= (const FixedStack<T>& s) {
   if (this == &s) // Ist dies eine Zuweisung an mich selbst ?
      return *this;
   if (max_size != s.max_size)// nur neu allokieren bei anderer Groesse
      delete [] tab;
   tab = tab_copy (s.tab, s.max_size);
   nb_elements = s.nb_elements;
   return *this;
}

template <class T>
void FixedStack<T>::push (T t) {
   assert(!full());
   tab[nb_elements++] = t;
}

template <class T>
void FixedStack<T>::pop() {
   assert(!empty());
   nb_elements--;
}

template <class T>
T FixedStack<T>::top() const {
   assert(!empty());
   return tab[nb_elements - 1];
}
```

```
template <class T>
ostream& operator<< (ostream& o, const FixedStack<T>& s) {
   for (int i = s.nb_elements - 1; i >= 0; --i)
      o << s.tab[i] << ' ';
   return o;
}
```

```
int main() {
   FixedStack<int> s1(10);
   s1.push(5);
   s1.push(6);
   s1.push(7);
   cout<<"\ns1.top: "<< s1.top() << "\ns1: " << s1 << endl;

   FixedStack<int> s2 = s1;
   FixedStack<int> s3(15);
   s3 = s1;
   s1.pop();
   s3.push(21);
   cout << "s1: " << s1 << endl;
   cout << "s2: " << s2 << endl;
   cout << "s3: " << s3 << endl;

   return 0;
}
```

Die abstrakte Klasse `Stack` gibt nur das Protokoll vor, die eigentliche Implementierung kann in Unterklassen erfolgen.

<u>Beispiel:</u>



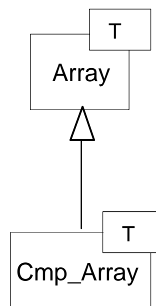
`FixedStack`: Implementierung mittels Array
`ListStack`: Implementierung mittels einer linearen Liste

<u>Explizite Instantiierung von parametrisierten Klassen</u>

```
template class FixedStack <int>;
```

Anweisung an den Compiler, aus der Klassenschablone `FixedStack` eine konkrete Instanz mit `T=int` zu generieren.
<u>Vorteil</u>: Bei Verwendung dieser Instanz muss nur die Deklaration der Template-Klasse inkludiert werden, und nicht der Implementierungsteil.

15.2.4 Beispiel: Array



allgemeines Array für nicht größenvergleichbare Datentypen, d. h. mit dem Operator ==, aber nicht unbedingt mit > oder <

Array für größenvergleichbare Datentypen: <>
Zusätzliche Elementfunktionen: `min, max, sort`

```
// Array: Parametrisierte Klasse Array
// Basisklasse Array fuer nicht vergleichbare Elemente

template<class T>
class Array {
public:
// Konstruktoren, Destruktor, Zuweisungsoperator
   Array (int = 100);
   Array (T* a, int dim);
   Array (const Array&);
   virtual ~Array ()    { delete [] tab; }
   Array& operator= (const Array&);

// Schnittstelle
   virtual T& operator[] (int i);        // [] als L-Wert
   virtual T  operator[] (int i) const; // [] als const-Funktion
   virtual int  find (const T&);        // liefert den Index des
                                        // gefundenen Elements oder -1

   bool operator== (const Array<T>&) const; // Gleichheit pruefen
   bool operator!= (const Array<T>&) const; // Ungleichheit pruefen
   int size () const;              // Groesse des Arrays
   void resize (int m);            // Groesse veraendern

protected:
   int max_size;               // Groesse des Arrays
   T *tab;                     // Daten
};

// Comparable Array: Array von Elementen mit ==, < und > Operatoren
template<class T>
class Cmp_Array : public Array<T> {
public:
   Cmp_Array (int n=100): Array<T>(n) {}
   Cmp_Array (T* a, int dim): Array<T>(a,dim) {}

// Schnittstelle
   T min ();                   // Minimaler Wert
   T max();                    // Maximaler Wert
   void sort();                      // Array sortieren

protected:
   void swap (int, int);
   void quicksort(int, int);
};

typedef Cmp_Array<int> IntArray;
typedef Cmp_Array<double> DoubleArray;

template <class T>
```

```
Array<T>::Array ( int n ) {
    assert (n > 0);
    max_size = n;
    tab = new T[max_size];
}

template <class T>
Array<T>::Array ( T* a, int dim ) {
    assert (dim > 0);
    max_size = dim;
    tab = new T[max_size];
    for (int i = 0; i < max_size; i++)
        tab[i] = a[i];
}

template <class T>
Array<T>::Array (Array<T> const &other) {
    max_size = other.max_size;
    tab = new T [max_size];
    for (int i = 0; i < max_size; i++)
       tab [i] = other.tab [i];
}

template <class T>
Array<T>& Array<T>::operator= (Array<T> const &other) {
    if (this == &other)
        return *this;
    if (max_size != other.max_size) {
        delete [] tab;
        max_size = other.max_size;
        tab = new T [max_size];
    }
    for (int i = 0; i < max_size; i++)
         tab [i] = other.tab [i];

    return *this;
}

template <class T>
T& Array<T>::operator[] (int index) {
    assert (index >= 0 && index < max_size);
    return (tab [index]);
}

template <class T>
T  Array<T>::operator[] (int index) const {
    assert (index >= 0 && index < max_size);
    return (tab [index]);
}

template <class T>
int Array<T>::size () const
{ return max_size; }

template <class T>
void Array<T>::resize (int new_size) {
    if (max_size == new_size) return;
    assert (new_size > 0);
    T* tab_bak = tab;
    tab = new T[new_size];
    for (int i = 0; i < max_size && i < new_size; i++)
        tab[i] = tab_bak[i];
    max_size = new_size;
    delete [] tab_bak;
```

```
}

template <class T>
int  Array<T>::find (const T& t)
{
    for (int i = 0; i < max_size; ++i)
        if (t == tab[i])
            return i;
    return -1;
}

// Zwei Arrays auf Gleichheit pruefen
template <class T>
bool Array<T>::operator== (const Array<T>& A) const {
    if (max_size != A.max_size)
        return false;
    for (int i = 0; i < max_size; i++)
        if (tab[i] != A.tab[i])
            return false;
    return true;
}

// Zwei Arrays auf Ungleichheit pruefen
template <class T>
bool Array<T>::operator!= (const Array<T>& A) const
{   return !(*this == A); }

template <class T>
T Cmp_Array<T>::min () {
    T min = tab[0];
    for (int i = 1; i < max_size; i++)
        if (min > tab[i])
            min = tab[i];
    return min;
}

template <class T>
T Cmp_Array<T>::max() {
    T max = tab[0];
    for (int i = 1; i < max_size; i++)
        if (max < tab[i])
            max = tab[i];
    return max;
}

template <class T>
void Cmp_Array<T>::sort()
{   quicksort(0, max_size - 1); }

template <class T>
void Cmp_Array<T>::quicksort (int links, int rechts) {
   if (links < rechts) {
      int i = links;
      int j = rechts;
      T v = tab[(links + rechts)/2];
      do {
         while (tab[i] <  v    ) ++i;
         while ( v     < tab[j]) --j;
         if (i <= j) {
            swap (i, j);
            ++i; --j;
         }
      } while ( i <= j);
```

```
        quicksort (links, j);
        quicksort (i, rechts);
    }
}

template <class T>
void Cmp_Array<T>::swap (int i, int j) {
    T t    = tab[i];
    tab[i] = tab[j];
    tab[j] = t;
}
```

15.2.5 Beispiele aus der ANSI-Klassenbibliothek

15.2.5.1 Die String-Klasse

Die in Abschnitt 2 eingeführte Klasse `string` ist in Wirklichkeit ein `typedef` auf eine Instantiierung des Klassentemplate `basic_string`. Dieses Klassen-Template implementiert die Funktionalität von Zeichenketten, die aus beliebig definierten Zeichen bestehen können. So ist die Instanttiierung string für den Umgang mit normalen 8-Bit-Zeichen (char) gedacht, während die Instantiierung wstring für den Umgang mit 16-Bit-Zeichen (wchar_t) gedacht ist.

```
typedef basic_string<char>    string
typedef basic_string<wchar_t> wstring;
```

15.2.5.2 Das Klassen-Template complex

Auch die Klasse für die Realisierung von komplexen Zahlen ist in Wirklichkeit ein Klassen-Template:

```
template <class FLOAT>
class complex {
public:
  complex (FLOAT r = 0, FLOAT i = 0): re (r), im (i) { }
  complex& operator += (const complex&);
  complex& operator -= (const complex&);
  complex& operator *= (const complex&);
  complex& operator /= (const complex&);
  FLOAT real () const { return re; }
  FLOAT imag () const { return im; }
  // viele weitere Methoden und friend-Funktionen ...
private:
  FLOAT re, im;
};
```

Angewendet werden die konkreten Instantiierungen

```
complex<float>, complex<double>, complex<long double>
```

je nach erforderlicher Genauigkeit.

15.2.5.3 Die Iostream-Bibliothek

Auch die Klassen der Iostream-Bibliothek sind Klassen-Templates. So gibt es die Klassen-Templates `basic_istream` bzw. `basic_ostream` für die Eingabe bzw. Ausgabe von beliebigen Zeichenströmen. Die von uns bisher verwendeten Klassen `istream` bzw. `ostream` sind in Wirklichkeit nicht anderes als typedefs.

```
typedef basic_ostream<char> ostream;
typedef basic_istream<char> istream;
```

Analoge Typdefinitionen gibt es auch für 16-Bit-Datenströme.

15.2.5.4 Der C++-Standardtyp vector

Ein Vektor im Sinne der C++-Standardbibliothek ist im Prinzip ein dynamisches Array, allerdings mit zusätzlichen Funktionalitäten.

Beispiel:

```
// vector1.cpp: Anwendungen des vector-Klassentemplates

#include <iostream>
#include <vector>

using namespace std;

int main() {
   vector<int> tab(10); // Vektor mit 10 Elementen

   // Vektor fuellen
   for (int i = 0; i < tab.size(); ++i)
      tab[i] = i*i;

   // Vektor ausgeben
   for (int i = 0; i < tab.size(); ++i)
       cout << i << ": " << tab[i] << endl;

   // Vektor dynamisch erweitern
   for (int i = 10; i < 20; ++i)
      tab.push_back(i*i);

   // Vektor ausgeben
   for (int i = 0; i < tab.size(); ++i)
      cout << i << ": " << tab[i] << endl;
}
```

Wie das nächste Beispiel zeigt, kann ein Vektor vollkommen dynamisch zur Laufzeit aufgebaut werden:

```
// vector2.cpp: Vector dynamisch aufbauen

#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main() {
   vector<string> textTab;      // leerer Vektor
   cout  << "Bitte Text eingeben (leer = Ende der Eingabe)\n:";
```

```
    string text;
    do {
       getline (cin, text, '\n');
       if (text.size() > 0)
           textTab.push_back(text);
     } while (text.size() > 0);

     cout << "Folgende Texte wurden eingegeben:\n";
     for(int i = 0; i < textTab.size(); ++i)
        cout << i << ": " << textTab[i] << endl;
}
```

Öffentliche Typen in der Klasse `vector<T>`:

Datentyp	***Bedeutung***
`vector::value_type`	T
`vector::reference`	Referenz auf ein Vektorelement
`vector::const_reference`	Referenz auf ein Element eines konstanten Vektors
`vector::iterator`	Iteratortyp
`vector::const_iterator`	Iteratortyp für konstanten Vektor
`vector::size_type`	ganzzahliger Typ ohne Vorzeichen für Größenangaben

Methoden der Klasse `vector<T>`:

Rückgabetyp Methode	***Bedeutung***
`vector()`	Standardkonstruktor, erzeugt leeren Vektor
`vector(n)`	erzeugt Vektor der Größe n
`vector(n, t)`	erzeugt Vektor der Größe n jeweils mit t initialisiert
`vector(const vector&)`	Kopierkonstruktor
`~vector()`	Destruktor, ruft die Destruktoren für alle Vektorelemente auf
`iterator begin()`	Anfang des Vektors
`iterator end()`	Position nach dem letzten Element
`void push_back(t)`	fügt t am Ende an
`void pop_back()`	löscht das letzte Element
`reference operator[](n)`	gibt eine Referenz auf das n-te Element zurück, evt. keine Arraygrenzenüberprüfung
`reference at(n)`	gibt eine Referenz auf das n-te Element zurück, mit Arraygrenzenüberprüfung
`void reserve(n)`	Speicherplatz auf n Elemente erhöhen
`size_type size()`	aktuelle Größe des Vektors
`bool empty()`	`size() == 0` bzw. `begin() == end()`
`void clear()`	Vektor löschen, d. h. Elemente freigeben
`vector& operator=(const vector&)`	Zuweisungsoperator

Das folgende Beispiel zeigt de Anwendung von Iteratoren (vgl Abschnitt 3) und dem Standardalgorithmus `for_each`.

```
// vector3.cpp: Anwendung von Vektor-Iteratoren und Algorithmen

#include <iostream>
#include <vector>
#include <algorith>

using namespace std;

void drucken (int& x) {
   cout.width(4);
   cout << x;
}

int main() {
   vector<int> intTab(10);
   for (int i = 0; i < intTab.size(); i++)
       intTab[i] = i;

   // Fuer alle Vektorelemente die Funktion drucken aufrufen
   for_each(intTab.begin(), intTab.end(), drucken);

   // Iterator anwenden
   vector<int>::iterator I;
   for (I = intTab.begin(); I != intTab.end(); ++I)
      cout << *I << endl;
}
```

Weitere Container aus der STL sind (unter anderen)

Name	*Bedeutung*
`vector<T>`	Dynamische Array
`deque<T>`	Double ended Queue, ein beidseitig erweiterbarer Container
`list<T>`	Doppelt verkettete lineae Liste
`map<Key,Value>`	Assoziatives Array
`set<T>`	Menge

Aus Platz- und Zeitgründen können diese nicht in diesem Rahmen ausführlich besprochen werden.

16 Ausnahmebehandlung

16.1 Einführung

Grundforderungen:

- Korrektheit: "programming by contract" (Zusicherungen)
- Robustheit: vernünftiges Verhalten, auch in nicht spezifizierten oder unvorhergesehenen Situationen.

Was passiert, wenn
- eine Zusicherung verletzt wird: z.B.: pop auf leeren Stack ?
  ### `assert` ### `terminate()` ### `abort()`
- durch 0 dividiert wird ### Absturz mit core-Dump
- eine Bereichsüberschreitung bei einem Array auftritt ?
- ein Netzwerkzugriff schiefgeht, z. B. mitten in einer komplexen Datenbanktransaktion?
- auf eine ungültige Adresse zgegriffen wird?
- ein `new int[1000000]` schiefgeht ?
  `malloc` : liefert Null-Pointer zurück
  `new` : früher : Null-Pointer wird zurückgeliefert
  ANSI-C++: Auslösen einer Ausnahme

Klassische Fehlerbehandlung

- Funktion behandelt Fehler selbst nach jeder Situation, die Fehler verursachen kann.
  - Programmcode wird unnötig aufgebläht und kompliziert
  - Wartung wird aufweniger
- Rückgabe spezieller Returncodes, damit die aufrufende Funktion den Fehler behandeln kann.
  - Ein Returncode sagt zu wenig aus über den Fehler.
  - Jeder Returnwert müsste explizit auf Fehler abgefragt werden.
- Aufruf einer speziellen Fehlerbehandlungsfunktion.
  - Was passiert, wenn man wieder zurückkommt?
- Unix Systemprogrammierung: Setzen einer globalen Fehlervariablen `errno`.
  - Wer fragt die eigentlich ab?
  - Globale Variablen widersprechen schon elementaren Forderungen der strukurierten Programmierung.
- Primitivreaktion: Etwas auf die Standardausgabe schreiben und Programm weiterlaufen lassen.
  - Aber wo geht die Standardausgabe bei einem Druckprogramm hin, das im Hintergrund läuft? Oder bei einem Windows-Programm?
  - Weiterlaufen nach einem Fehler kann unvorherzusehende Konsequenzen haben.
- Maximalreaktion: Meldung ausgeben und Programm abbrechen.
  - Das darf bei einer sicherheitsrelevanten Software aber nicht vorkommen!

Konzept der Ausnahmebehandlung

- Eine Ausnahme (*exception*) ist das Eintreten einer Situation, die einer gesonderten Behandlung bedarf. Das kann eine Fehlersituation sein oder auch ein Situation, in der ein Sonderfall auftritt.
- Ausnahmen können automatisch durch das Laufzeitsystem ausgeworfen werden, z.B. bei Arithmetiküberlauf, oder bewusst durch das Programm selbst.
- Ausnahmen können an zentralen Stellen durch spezielle Ausnahmebehandlungsroutinen behandelt werden.
- In einer Ausnahmebehandlungsroutine kann über eine angemessene Reaktion auf die aufgefangene Ausnahme entschieden werden, z.B. Bereinigung, Fehlerprotokoll, Benutzerentscheid über Abbruch, Abbruch des Programms mit klaren Angaben über den Grund und den Ort des Abbruchs, usw.

Programmiersprachen, in denen es eine echte Ausnahmebehandlung gibt, sind z.B. ADA, Eiffel, C++ und JAVA.

16.2 Ausnahmebehandlung in C++

In C++ gibt es dazu drei Sprachelemente:

`throw` `[`*`Ausdruck`*`]`

Auswerfen einer Ausnahme. Die Programmsteuerung wird an die nächste zuständige Ausnahmebehandlung übergeben, sofern eine vorhanden ist. Ist keine Ausnahmebehandlung vorhanden, so wird automatisch `terminate()`[1] aufgerufen.

`try` `{`*`zusammengesetzte Anweisung`*`}` *`Handler Folge`*

Ein `try`-Block fasst Anweisungen zusammen, für die eine Ausnahmebehandlung durchgeführt werden soll.

`catch` `(`*`Ausnahmedeklaration`*`) {` *`zusammengesetzte Anweisung`* `}`

Ausnahmebehandlungsroutine. Fängt Ausnahmen eines bestimmten Typs auf.

Konkret:

```
try {
   ...
   throw Ausdruck;                    Try-Block
   ...
}
catch (Parameter1) { … }
catch (Parameter2) { … }
…
catch ( ... ) {…}                     Ausnahmebehandlungsroutinen
      Ellipse###, fängt alle übriggebliebenen Ausnahmen auf
```

Ablauf:

Wenn im `try`-Block eine Ausnahme ausgelöst wird, z.B. durch `throw` oder automatisch, so wird an das Ende des Blockes verzweigt und je nach Datentyp der Ausnahme eine passende `catch`-Routine ausgeführt.

[1] `terminate()` ist eine Systemfunktion des Laufzeitsystems zum regulären Beenden des Prozesses.

Nach Abarbeitung der `catch`-Routine wird mit der Anweisung fortgesetzt, die auf die Handler-Liste folgt. Die Ausnahme gilt dann als abgehandelt. Es erfolgt kein Rücksprung zur Programmstelle, wo die Ausnahme ausgelöst wurde.

Beispiel:

```
// ausnahme1.cpp: 1. Test fuer Ausnahmebehandlung

void f() {
   cout << "Beginn: f()\n";
   throw "f()-Ausnahme";   // Auswerfen einer char*-Ausnahme
   cout << "Ende  : f()\n";
}

void g() {
   cout << "Beginn: g()\n";
   throw 1;                // Auswerfen einer int-Ausnahme
   cout << "Ende  : g()\n";
}

int main() {
   cout << "Beginn: main()\n";
   for (int i = 1; i <= 2; i++) {
      cout << i << ". Durchlauf \n";
      try {
         if (i == 1)
            f();
         else
            g();
      }
      catch (char* s)      // char*-Ausnahme behandeln
      {   cout << "catch (char*): " << s << endl; }

      catch (int i)  // int-Ausnahme behandeln
      {   cout << "catch (int): " << i << endl; }
      cout << i << ". Durchlauf Ende\n";
   }
   cout << "Ende  : main()\n";
   return 0;
}

// Beginn: main()
// 1. Durchlauf
// Beginn: f()
// catch (char*): f()-Ausnahme
// 1. Durchlauf Ende
// 2. Durchlauf
// Beginn: g()
// catch (int): 1
// 2. Durchlauf Ende
// Ende  : main()
```

Die Unterscheidung von Ausnahmen erfolgt durch den Typ des ausgeworfenen Objektes. Je nach Typ geht die Kontrolle zum passenden Ausnahme-Handler (`catch`-Routine) über. Diesem wird wie bei einer Funktion das ausgeworfene Objekt übergeben.

Durch `throw` wird ein temporäres Objekt erzeugt. Dieses wird mit Hilfe des Kopierkonstruktors kopiert und an die passende `catch`-Routine weitergereicht.

Gibt es keinen Handler mit einem passenden Typ, so geht die Kontrolle zum nächsten darüberliegenden `try`-Block über und gegebenenfalls dort zu einem passenden Handler.

Ausnahmeklassen

Normalerweise werden für jeden möglichen Ausnahmetyp, eigene Ausnahmeklassen definiert. Beim Auswerfen einer Ausnahme wird dann ein temporäres Objekt dieser Ausnahmeklasse erzeugt. In diesem Ausnahmeobjekt können nun Informationen über den aufgetretenen Ausnahmefall hinterlegt werden, die für die korrekte Behandlung notwendig sind. Oft werden einfach nur leere Klassen definiert, weil alleine der Klassentyp schon eine Aussage über die aufgetretene Ausnahme darstellt.

Beispiel:

```
// ausnahme.h: Definition einer einfachen Ausnahmeklasse

class Ausnahme {
public:
   Ausnahme(char* s) : text(s)
   {   cout << "Ausnahme: Konstruktor\n";          }
   Ausnahme(const Ausnahme& a) : text(a.text)
   {   cout << "Ausnahme: Kopierkonstruktor\n";    }
   ~Ausnahme()
   {   cout << "Ausnahme: Destruktor\n";           }
   virtual void info()
   {   cout << "Ausnahme: " << text << endl;       }
protected:
    char* text;
};
```

Beispiel:

```
// ausnahme2.cpp: 2. Test fuer Ausnahmebehandlung
// Durchlaufen von Destruktoren bei ausgeloester Ausnahme

#include <iostream>
#include "ausnahme.h"

class X {
public:
   X(int i):n(i) {   cout << "X: Konstruktor n = " << n << endl; }
   ~X()          {   cout << "X: Destruktor  n = " << n << endl; }
private:
   int n;
};

void f() {
   cout << "Beginn: f()\n";
   X  x2(2);
   throw Ausnahme("f():"); // Ausnahme vom Typ 'Ausnahme' ausloesen
   cout << "Ende  : f()\n";
}

void g() {
   cout << "Beginn: g()\n";
   X x1(1);
   f();
   cout << "Ende  : g()\n";
}

int main() {
   cout << "Beginn: main()\n";
   try {  g();  }
   catch (Ausnahme& A)
   {   A.info();    }
```

```
   cout << "Ende  : main()\n";
   return 0;
}

/*
Beginn: main()
Beginn: g()
X: Konstruktor n = 1
Beginn: f()
X: Konstruktor n = 2
Ausnahme: Konstruktor
Ausnahme: Kopierkonstruktor
Ausnahme: Destruktor
X: Destruktor  n = 2
X: Destruktor  n = 1
Ausnahme: f():
Ausnahme: Destruktor
Ende  : main()
*/
```

Erläuterung zum obigen Beispiel:

In `f()` und `g()` wird jeweils ein X-Objekt erzeugt, bevor die Ausnahme ausgelöst wird. Nach Auslösen der Ausnahme werden diese Objekte korrekt destruiert. Ebenso wird durch `throw` der Rücksprung-Stack korrekt zurückgesetzt (*stack unwinding*). Wird in einem Konstruktor eine Ausnahme ausgelöst, so werden zumindest die Destruktoren der bis dahin konstruierten Teilobjekte aufgerufen. Allerdings werden dabei mit `new` erzeugte Objekte nicht automatisch entfernt!
In dem Ausdruck

```
throw Ausnahme("f()");
```

wird zunächst ein temporäres Ausnahme-Objekt erzeugt. Dieses wird danach per Kopierkonstruktor kopiert. Die Kopie geht auf die Reise zum Ausnahme-Handler, während das temporäre Ausnahme-Objekt wieder destruiert wird.

Weiterleiten von Ausnahmen

Innerhalb einer `catch`-Routine kann eine aufgefangene Ausnahme erneut ausgeworfen und damit an die nächste zuständige Ausnahmebehandlung weitergegeben werden.

Beispiel:

```
// ausnahme3.cpp: 3. Test fuer Ausnahmebehandlung
// aufgefangene Ausnahmen weiterleiten

#include <iostream>
#include "ausnahme.h"

class X {
public:
   X(int i):n(i) { cout << "X: Konstruktor n = " << n << endl;  }
   ~X()          { cout << "X: Destruktor  n = " << n << endl;  }
private:
   int n;
};

void f() {
   cout << "Beginn: f()\n";
   X  x2(2);
   throw Ausnahme("f():"); // Auswerfen Ausnahme
   cout << "Ende  : f()\n";
```

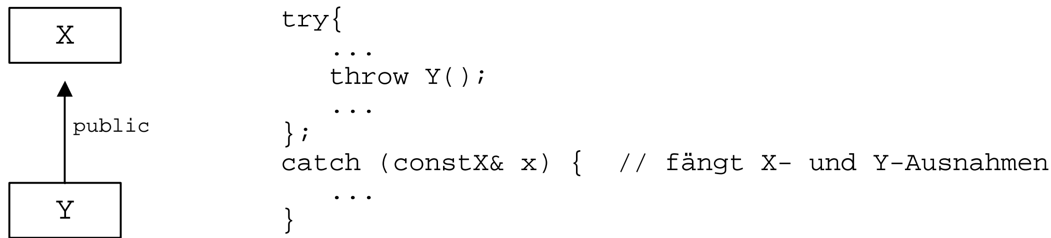
```
}

void g() {
   cout << "Beginn: g()\n";
   X x1(1);
   try { f(); }
   catch (Ausnahme a) {
      cout << "g()   : Ausnahme aufgefangen -> " << endl;
      a.info();
      throw;   // Ausnahme weiterreichen
   }
   cout << "Ende  : g()\n";
}

int main() {
   cout << "Beginn: main()\n";
   try {  g();  }
   catch (Ausnahme& a) {
      cout << "main(): Ausnahme aufgefangen -> " << endl;
      a.info();
   }
   cout << "Ende  : main()\n";
   char c; cin >> c;
   return 0;
}
/*
Beginn: main()
Beginn: g()
X: Konstruktor n = 1
Beginn: f()
X: Konstruktor n = 2
Ausnahme: Konstruktor                    <-- 1. Ausnahme-Objekt
Ausnahme: Kopierkonstruktor              <-- 2. Ausnahme-Objekt
Ausnahme: Destruktor                     <-- 1. Ausnahme-Objekt
destruieren
X: Destruktor  n = 2
Ausnahme: Kopierkonstruktor              <-- 3. Ausnahme-Objekt
g()   : Ausnahme aufgefangen ->
Ausnahme: f():
Ausnahme: Kopierkonstruktor              <-- 4. Ausnahme-Objekt
Ausnahme: Destruktor                     <-- 3. Ausnahme-Objekt
destruieren
Ausnahme: Destruktor                     <-- 2. Ausnahme-Objekt
destruieren
X: Destruktor  n = 1
main(): Ausnahme aufgefangen ->
Ausnahme: f():
Ausnahme: Destruktor                     <-- 4. Ausnahme-Objekt
destruieren
Ende  : main()
*/
```

Auffinden des passenden Ausnahme-Handlers

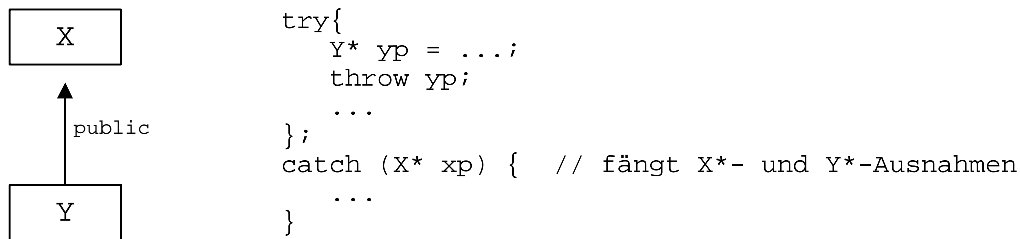
Es gibt drei Regeln zum Auffinden des passenden Ausnahme-Handlers.

Zu einem `throw`-Ausdruck vom Typ A passt ein Handler mit dem Typ `T`, `const T`, `T&` oder `const T&`, wenn gilt:

1. A und T stimmen exakt überein
2. A ist eine von T `public` abgeleitete Klasse



3. A und T sind Zeigertypen und A lässt sich mittels Standardkonvertierungen in T umwandeln.



Die Handler eines `try`-Blocks werden nacheinander auf Übereinstimmung untersucht. Der erste auf die Ausnahme passende Handler wird durchlaufen

Regel: Die allgemeinsten Handler ans Ende der Handler-Liste stellen

Gibt es für eine Ausnahme keinen passenden Handler, so wird `terminate()` aufgerufen.

Ausnahme-Deklarationen

Im Funktionskopf einer Funktion können die innerhalb der Funktion möglichen Ausnahmen deklariert werden.

Beispiel:

```
void f() // beliebige Ausnahmen möglich
{
   try { g(); }
   catch (const char* message)   {…}
   catch (const List<String>& message) {…}
}

void g() throw (List<String>, const char*)
         // Nur Ausnahmen vom Typ List<String> und char* möglich
{
   List<String> ErrorReport;
   ...
   throw ErrorReport;
   ...
   throw "Help!";
   ...
}

void h() throw ()    // keine Ausnahme in h()möglich
{...};
```

Was passiert, wenn in `g()` eine nicht deklarierte Ausnahme oder in `h()` irgendeine Ausnahme ausgelöst wird?

Antwort: Aufruf der Funktion: `unexpected()`### `terminate()`### `abort()`

Beispiel:

```
class X {};
class Y {};
class Z : public X {};
class W {};

void f(int n) throw (X, Y)
{
   if ( n > 0) throw X;    // OK
   if ( n < 0) throw Z;    // OK, da Z von X abgeleitet
   throw W;                // Aufruf von unexpected()
};
```

Beispiel:

```
// unexpected.cpp: Anwendung von unexpected

#include <exception>

typedef void (*func)();  // Pointer auf void-Funktion

void f() throw(int) {    // f behauptet, nur int-Ausnahmen auszuloesen
   cout << "Beginn: f()\n";
   throw "Leider doch eine Ausnahme !"; // char*-Ausnahme
   cout << "Ende  : f()\n";
}

void aetsch() {
   cout << "aetsch : anstatt unexpected aufgerufen\n";
   throw 1;       // int-Ausnahme ausloesen
                  // char*-Ausnahme wuerde terminate ausloesen !
}

int main() {
   cout << "Beginn: main()\n";
   func orig = set_unexpected(aetsch);  // aetsch anstelle von
                                        // unexpected aufrufen
   try {  f();  }
   catch (int) {
      cout << "main: int-Ausnahme aufgefangen!\n";
   }
   catch (...) {
     cout << "main: Irgendeine Ausnahme aufgefangen !\n";
   }
   set_unexpected (orig);        // Status quo herstellen
   cout << "Ende  : main()\n";
   char c; cin >> c;
   return 0;
}

/*
Beginn: main()
Beginn: f()
aetsch : anstatt unexpected aufgerufen
main: int-Ausnahme aufgefangen!
Ende  : main()
*/
```

In der Funktion `unexpected()` gibt es einen Funktionspointer:

```
void (*func) (); // normalerweise 0
```

wenn dieser nicht gleich 0 ist, so wird die zugehörige Funktion aufgerufen, anstatt der Funktion `terminate()`.

Am Anfang des Abschnitts wurde gefragt, was man tun kann, wenn ein `new` schiefgeht. Eine Möglichkeit ist die des `new_handlers`, der analog wie der unexpected-Handler zu installieren ist.

Beispiel:

```
// new1.cpp: Test des new_handlers

#include <new.h>         // set_new_handler()
#include <iostream.h>
#include <stdlib>       // dort ist abort() deklariert

void KeinSpeicher() {
    cerr << "Kein Speicher mehr verfuegbar!\n";
    cin.get();
    abort();         // Abbruch
}

typedef void (*PNHF)(); // Zeiger auf new_handler-Funktion

void f() {
   const long int groesse = 1<<20; // 2 hoch 20
   PNHF pnhf_alt;
   int j;
   pnhf_alt = set_new_handler (KeinSpeicher);
      // new_handler auf KeinSpeicher setzen
      // alten new_handler merken
   char *strtab[5000];
   for (int i = 0; i < 5000; i++) {
      cout << i << endl;
      strtab[i] = new char[groesse]; // jeweils 1 MB !
      for (j = 0; j < groesse; j++)
         strtab[i][j] = ' ';
   }

   cout << "f() nach new !\n";
   set_new_handler(pnhf_alt);
      // alten new_handler restaurieren
}

int main() {
   f();
   return 0;
}
```

Lösung mit Hilfe der Ausnahmebehandlung:

```
// new2.cpp: Ausnahmebehandlung bei new

#include <new>             // dort ist bad_alloc definiert
#include <iostream.h>
#include <stdlib>          // dort ist abort() deklariert

void f() {
   const long int groesse = 1<<20; // 2 hoch 20
   int j;
   char *strtab[5000];
```

```
   for (int i = 0; i < 5000; i++) {
      cout << i << endl;
      strtab[i] = new char[groesse]; // jeweils 1 MB !
      for (j = 0; j < groesse; j++)
         strtab[i][j] = ' ';
   }
   cout << "f() nach new !\n";
}

int main() {
   try {
      f();
   }
   catch (bad_alloc) {
      cerr << "bad_alloc: Kein Speicher mehr verfuegbar!\n";
      cin.get();
      abort();
   }
   return 0;
}
```

Zusicherungen und Ausnahmen

In vielen Beispielen hatten wir Zusicherungen verwendet und diese mit Hilfe des `assert`-Makros realisiert. Das `assert`-Makro ist bereits im ANSI-C-Sprachstandard vorgesehen und sieht etwa wie folgt aus:

```
#ifdef NDEBUG
   #define assert(p)  ((void)0)
#else
   #define assert(p)  ((p) ? (void)0 : _assert(#p, __FILE__, __LINE__))
#endif
```

Nachteil: Im Falle, dass die Bedingung p nicht erfüllt ist, wird das Programm abgebrochen.

Mit Hilfe der Ausnahmebehandlung und Funktions-Templates ist nun aber eine flexiblere Lösung möglich:

```
// assert1.cpp: Assert-Template zum Auswerfen von Ausnahmen

template <class A, class E>
inline void Assert(A assertion, E except) {
   if(!assertion) throw except;
}

class BadArg {
public:
   BadArg(int i):ii(i) {}
   void info() const {
      cout << "BadArg: Falsches Argument: " << ii << endl;
   }
private:
   int ii;
};

void f(int i) {
   Assert (i > 0, BadArg(i));
   cout << "f(" << i << ")\n";
}

int main() {
   int i;
   for (i = 3; i > -3; i--) {
```

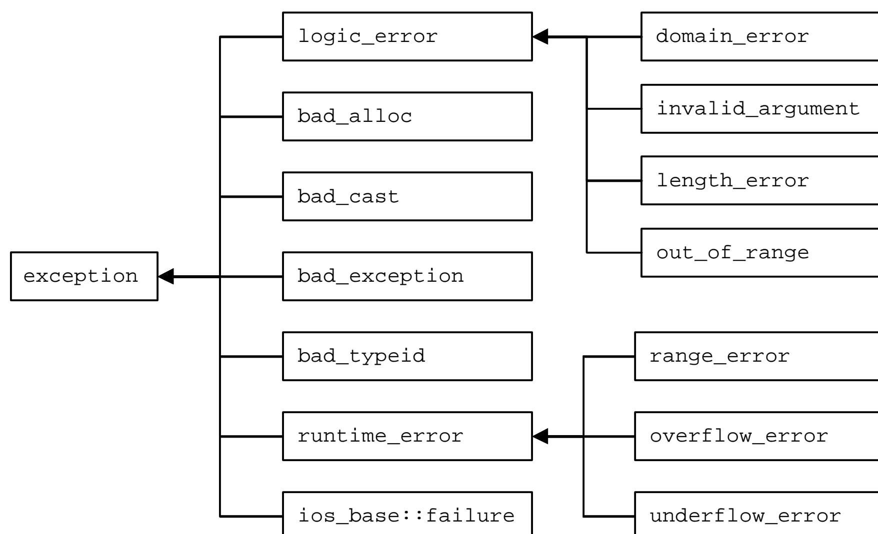
```
      try {  f(i);  }
      catch (BadArg b)    {  b.info();  }
   }
}
```

16.3 Standardausnahmen

Im ANSI-Standard werden verschiedene Standardausnahmetypen definiert. Diese werden im wesentlichen von Funktionen der Standard-Bibliothek ausgeworfen, können aber auch zur Definition von eigenen Ausnehmeklassen verwendet werden.

```
class exeption {
public:
   exception() throw();
   exception& exception (const exception&) throw();
   exception& operator= (const exception&) throw();
   virtual ###exception() throw();
   virtual const char* what() const throw();
private:
   // ...
};
```

Standardausnahmehierarchie:



Übersicht über Standardausnahmen

Klasse	*Bedeutung*	*Header*
`exception`	Standardausnahmeklasse	`<exception>`
`ios_base::failure`	Durch I/O-Streams auslösbare Ausnahme	`<ios>`
`logic_error`	Logische Programmfehler	`<stdexcept>`
`length_error`	Fehler in Funktionen der C++-Standardbibliothek bei Größenüberschreitungen	`<stdexcept>`
`out_of_range`	Bereichsüberschreitungen in Funktionen der C++-Standardbibliothek	`<stdexcept>`
`domain_error`	weitere Fehler der C++-Standardbibliothek	`<stdexcept>`
`invalid_argument`	ungültige Argumente	`<stdexcept>`
`runtime_error`	Zur Laufzeit feststellbare Fehler	`<stdexcept>`
`range_error`	Bereichsüberschreitung zur Laufzeit	`<stdexcept>`

`overflow_error`	arithmetischer Überlauf	`<stdexcept>`
`underflow_error`	arithmetischer Unterlauf	`<stdexcept>`
`bad_alloc`	Fehler beim Speicherallokieren,durch `new` auslösbar	`<new>`
`bad_cast`	Typumwandlungsfehler, durch `dynamic_cast` auslösbar	`<typeinfo>`
`bad_typeid`	Falsche Typid, durch `typeid` auslösbar	`<typeinfo>`
`bad_exception`	Unvorhergesehene Ausnahme, durch `unexpected` auslösbar	`<exception>`

Beispiel: Eigene Ausnahmen ableiten

```
class Ausnahme : public exception {
public:
   Ausnahme():text("Ausnahme ") {}
   virtual const char* what() const throw() {
      return text.c_str();
   }
protected:
   string text;
};
```

Beispiel:

```
// ausnahme4.cpp: Ausnahmebehandlung
//
#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

// Klasse fuer einen Anwendungsrahmen
class Anwendung {
public:
   // anwendungsspezifische Ausnahmetypen
   class Ausnahme;
   class ALeicht;
   class ASchwer;
   class AFunktion;

   // Konstruktor, Destruktor
   Anwendung():funktion(-1) { cout << "Anwendung: Konstruktor\n"; }
  ~Anwendung()              { cout << "Anwendung: Destruktor\n";  }

   // Anwendungsfunktionen
   void steuerung();
   void funktion1(string&) throw (Ausnahme);
   void funktion2(string&) throw (Ausnahme);
   void readFunktion()     throw (Ausnahme);
   int  getFunktion() const { return funktion; }
private:
   int funktion;
   string eingabe;
};

// Implementierung der Ausnahmetypen
class Anwendung::Ausnahme : public exception {
public:
   Ausnahme():text("Ausnahme ") {}
   virtual const char* what() const throw() {
      return text.c_str();
```

```
   }
protected:
   string text;
};

class Anwendung::ALeicht : public Anwendung::Ausnahme {
public:
   ALeicht() { text += "leicht "; }
};

class Anwendung::ASchwer : public Anwendung::Ausnahme {
public:
   ASchwer() { text += "schwer "; }
};

class Anwendung::AFunktion : public Anwendung::Ausnahme {
public:
   AFunktion() { text += "Funktion"; }
};

// Implementierung der Anwendungsfunktionen
void Anwendung::funktion1(string& s) throw (Anwendung::Ausnahme) {
   cout << "Beginn funktion1(" << s <<")\n";
   if (s.size() == 0) throw ALeicht();
   cout << "Ende   funktion1(" << s << ")\n";
}

void Anwendung::funktion2(string& s) throw (Anwendung::Ausnahme) {
   cout << "Beginn funktion2(" << s << ")\n";
   if (s.size() == 0) throw ASchwer();
   cout << "Ende   funktion2(" << s << ")\n";
}

void Anwendung::readFunktion() throw (Anwendung::Ausnahme) {
   cout << "Funktion und Eingabe: ";
   cin >> funktion;
   if (!cin.good()) {
      cin.clear();
      cin.get();
      throw AFunktion();
   }
   getline(cin, eingabe, '\n');
}

// Hauptsteuerung der Anwendung
void Anwendung::steuerung() {
   cout << "Beginn steuerung()\n";
   do {
      try {
         readFunktion();
         switch(funktion) {
            case 1:  funktion1(eingabe);
                     break;
            case 2:  funktion2(eingabe);
                     break;
            case 0:  break;
            default: throw AFunktion();
                     break;
         }
      }
      catch (ASchwer&  x) {
         cout << "steuerung:\t" << x.what() << endl; throw;
      }
      catch (ALeicht&  x) {
```

```
            cout << "steuerung:\t" << x.what() << endl;
         }
         catch (AFunktion x) {
            cout << "steuerung:\t" << x.what() << endl;
         }
      } while (funktion != 0);
      cout << "Ende steuerung()\n";
}

int main() {
   cout << "Beginn main()\n";
   Anwendung a;
   do {
      try { a.steuerung(); }
      catch (Anwendung::Ausnahme& x) {
         cout << "main:\t" <<  x.what() << endl;
      }
      catch (...){
         cout << "main: ...-Ausnahme\n";
      }
   } while (a.getFunktion() != 0);
   cout << "Ende main()\n";
   return 0;
}
/* Ausgabe bei den Eingaben: 1 hallo; 1; 2; 3; x; 0;
Beginn main()
Anwendung: Konstruktor
Beginn steuerung()
Funktion und Eingabe: 1 hallo
Beginn funktion1( hallo)
Ende   funktion1( hallo)
Funktion und Eingabe: 1
Beginn funktion1()
steuerung:      Ausnahme leicht
Funktion und Eingabe: 2
Beginn funktion2()
steuerung:      Ausnahme schwer
main:   Ausnahme schwer
Beginn steuerung()
Funktion und Eingabe: 3
steuerung:      Ausnahme Funktion
Funktion und Eingabe: x
steuerung:      Ausnahme Funktion
Funktion und Eingabe: 0
Ende steuerung()
Ende main()
*/
```

17 Laufzeit-Typinformationen und Typkonvertierungen

17.1 Einführung

Problem: Man hat in einer Programmsituation einen Zeiger auf ein Objekt und benötigt Information über den tatsächlichen dynamischen Typ dieses Objekts.

A ← B ← C (C erbt von B, B erbt von A)	Code

A
↑
B
↑
C
	`C* cp = new C;     // neues C-Objekt`
	`B* bp = cp;        // der Zeiger wird automatisch konvertiert`
	`                   // das Objekt nicht`
	`A* ap = bp;        // ap zeigt nun auf ein C-Objekt`
	`bp = ap;           // geht nicht automatisch, kann zur`
	`                   // Uebersetzungszeit nicht entschieden werden`
	`bp = (B* )ap;      // geht zwar, aber was ist, wenn ap auf ein`
	`                   // A-Objekt zeigt?`

Der sogenannte RTTI-Mechanismus (*Runtime Type Information*) besteht aus drei Teilen:

- einem Operator `typeid` zur Identifikation des exakten dynamischen Typs eines Objektes
- einer Klasse `type_info`, in der Typinformationen abgelegt werden
- einem Operator `dynamic_cast` zum Konvertieren zwischen verwandten Datentypen

17.2 Typen identifizieren mit `typeid` und `type_info`

Der Operator `typeid` (Headerdatei `<typeinfo>`) kann auf zwei verschiedene Arten angewendet werden:

```
const type_info& typeid (ausdruck);
const type_info& typeid (typname) throw (bad_typeid);
```

In beiden Fällen liefert `typeid` eine Referenz zurück auf ein konstantes Objekt vom Typ `type_info`, das den Typ des Ausdrucks bzw. des Typnamens beschreibt.

```
// typeinfo: Laufzeit-Typinformationen
class type_info {
public:
   virtual ~type_info();
   int operator==(const type_info&) const;
   int operator!=(const type_info&) const;
   int before(const type_info&) const;  // Ordnen von Typen möglich
   const char* name() const;            // liefert den Typnamen zurueck
// ...
private:
   type_info(const type_info &);
   type_info& operator=(const type_info&);
// ...
};
```

Für jede Klasse mit mindestens einer virtuellen Funktion, man nennt dies auch eine *polymorphe Klasse*, wird vom Compiler ein `type_info`-Objekt mit Typinformationen angelegt. Für alle anderen Datentypen, elementare Typen oder nicht-polymorphe Klasse werden Typinformationen nur bei Bedarf angelegt.

Beispiel: Elementare Datentypen

```
// typen1.cpp: Beispiele fuer typeid

#include <iostream.h>
#include <typeinfo.h>

int main() {
   cout << "typeid: Standardtypen"     << endl; // typeid: Standardtypen
   cout << typeid(char)       .name()  << endl; // char
   cout << typeid(short)      .name()  << endl; // short
   cout << typeid(int)        .name()  << endl; // int
   cout << typeid(long)       .name()  << endl; // long
   cout << typeid(float)      .name()  << endl; // float
   cout << typeid(double)     .name()  << endl; // double
   cout << typeid(long double).name()  << endl; // long double

   cout << "\ntypeid: Zeigertypen"     << endl; // typeid: Zeigertypen
   cout << typeid(char*)       .name() << endl; // char *
   cout << typeid(short*)      .name() << endl; // short *
   cout << typeid(int*)        .name() << endl; // int *
   cout << typeid(long*)       .name() << endl; // long *
   cout << typeid(float*)      .name() << endl; // float *
   cout << typeid(double*)     .name() << endl; // double *
   cout << typeid(long double*).name() << endl; // long double *
   cout << typeid(void*)       .name() << endl; // void *
}
```

`typeid` bestimmt den dynamischen Typ eines Ausdrucks nur bei polymorphen Klassenobjekten. In allen anderen Fällen wird nur der statische Typ eines Ausdrucks ermittelt.

Beispiel: Klassentypen

```
// typen2.cpp: Beispiele fuer typeid: polymorphe Klasse

class A {
public:
   virtual void zeige() { cout << "A" << endl; }
};

class B : public A {
public:
   void zeige() { cout << "B" << endl; }
};

class C : public B {
public:
   void zeige() { cout << "C" << endl; }
};

class D : public C { };

int main() {
   A* za[4] = { new A, new B, new C, new D };
   for (int i = 0; i < 4; i++) {
      cout << "typeid(za[" << i << "]): "    << typeid(za[i]).name()
           << "\ttypeid(*za[" << i << "]): " << typeid(*za[i]).name()
           << endl;
   }
}
```

Ausgabe des Programms bei nicht-polymorphen Klassen, d. h. die Funktion `info()` ist nicht `virtual` definiert:

```
typeid(za[0]): A *      typeid(*za[0]): A
typeid(za[1]): A *      typeid(*za[1]): A
typeid(za[2]): A *      typeid(*za[2]): A
typeid(za[3]): A *      typeid(*za[3]): A
```

Ausgabe des Programms bei polymorphen Klassen, d. h. `info()` ist `virtual` definiert:

```
typeid(za[0]): A *      typeid(*za[0]): A
typeid(za[1]): A *      typeid(*za[1]): B
typeid(za[2]): A *      typeid(*za[2]): C
typeid(za[3]): A *      typeid(*za[3]): D
```

Dass das Bestimmen des dynamischen Typs auch mit Referenzen funktioniert, zeigt das folgende Beispiel:

Beispiel:

```
// typen4.cpp: Beispiele fuer typeid: polymorphe Klassen

// Klassenhierarchie wie oben ...

void typtest(const A& a) {
   cout << "typeid(a): " << typeid(a).name() << endl;
}

int main() {
   A a; B b; C c; D d;
   cout << "A: "; typtest(a);
   cout << "B: "; typtest(b);
   cout << "C: "; typtest(c);
   cout << "D: "; typtest(d);
}

/* Ausgabe des Programms
A: typeid(a): A
B: typeid(a): B
C: typeid(a): C
D: typeid(a): D
*/
```

Die normale Anwendung des Operators `typeid` sieht allerdings eher so aus.

```
A* ap;
....
// ap zeigt nun auf ein A-, B-, C- oder D-Objekt

if (typeid(*ap) == typeid(B)) {
   B* bp = (B*)ap;
   ...
}
else if (typeid(*ap) == typeid(C)){
   C* cp = (C*)ap;
   ...
}
else if (typeid(*ap) == typeid(D)){
   D* dp = (D*)ap;
   ...
}
```

Empfehlung:

`typeid` sollte nur aus wirklich triftigen Gründen eingesetzt werden (wozu hat man schließlich virtuelle Funktionen?). Typkonvertierungen mit Hilfe von C-Casts sollten bei Konvertierungen von Klassentypen generell überhaupt nicht benutzt werden.

17.3 Dynamische Typumwandlungen mit `dynamic_cast`

Mit Hilfe des Operators `dynamic_cast` können Typkonvertierungen von Zeigern und Referenzen innerhalb von polymorphen Klassenhierarchien vorgenommen werden.

Syntax:

```
dynamic_cast<T> (A)
```

A Ausdruck, dessen Auswertung einen Zeiger oder eine Referenz auf ein Klassenobjekt ergibt.

T Zeiger oder Referenz auf einen Klassentyp oder auf `void*, const void*`

Wirkungsweise:

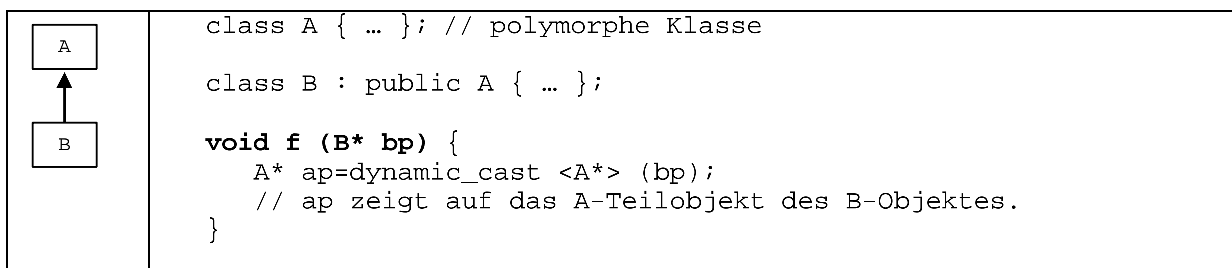
Der Ausdruck A wird in den Typ T konvertiert

Regeln:

- Falls der Typ T ein Zeigertyp ist, muss der Ausdruck A ebenfalls einen Zeigertyp haben.
- Falls T ein Referenztyp ist, muss der Ausdruck A ein L-Wert sein.
- Ein `const` im Typ des Ausdrucks A darf nicht wegkonvertiert werden.

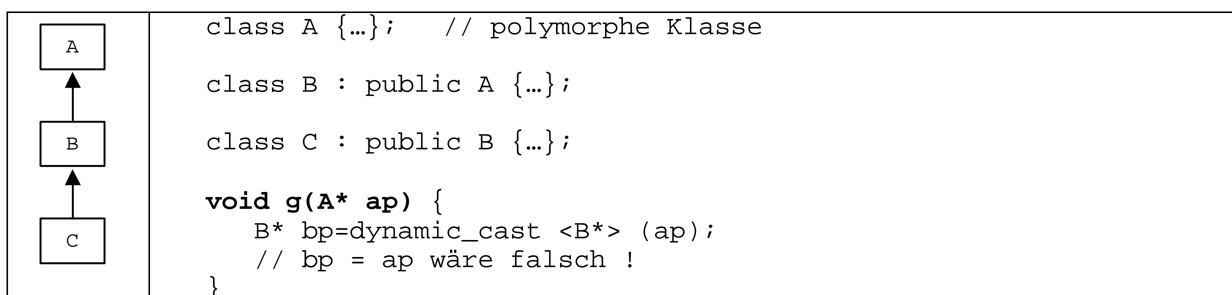
Im Fehlerfall wird bei Zeigerkonvertierungen der Zeigerwert 0 zurückgegeben, bei Referenzen wird eine Ausnahme von Typ `bad_cast` ausgeworfen.

Beispiel 1:




```
class A { … }; // polymorphe Klasse

class B : public A { … };

void f (B* bp) {
   A* ap=dynamic_cast <A*> (bp);
   // ap zeigt auf das A-Teilobjekt des B-Objektes.
}
```

Beispiel 2:



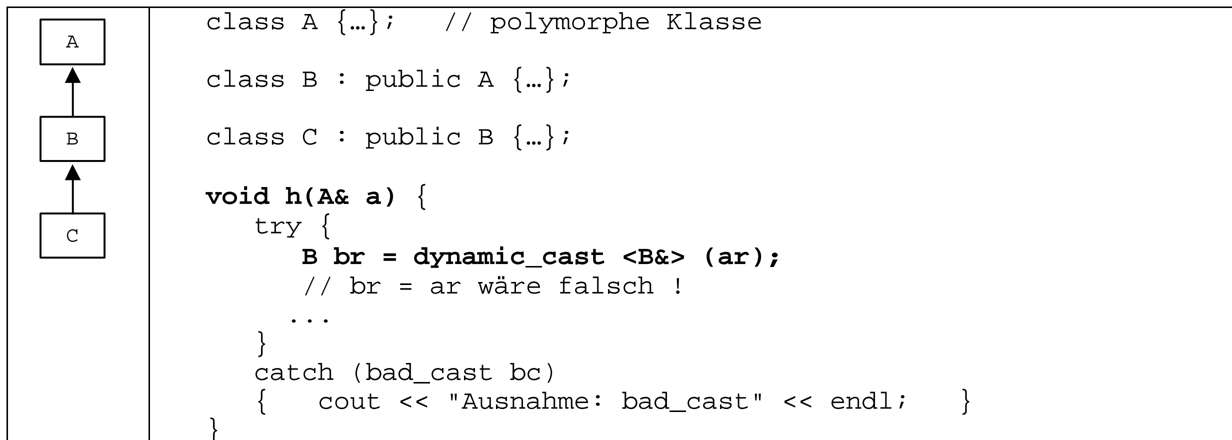

```
class A {…};   // polymorphe Klasse

class B : public A {…};

class C : public B {…};

void g(A* ap) {
   B* bp=dynamic_cast <B*> (ap);
   // bp = ap wäre falsch !
}
```

Was passiert ?

- zeigt `ap` auf ein B-Objekt, so wird `ap` in den Typ `B*` konvertiert
- zeigt `ap` auf ein C-Objekt, so wird `ap` zunächst zu `C*` und dann zu `B*` konvertiert.

- zeigt `ap` auf ein A-Objekt, so ist das Ergebnis der Konvertierung ein Nullzeiger.

Beispiel 3:

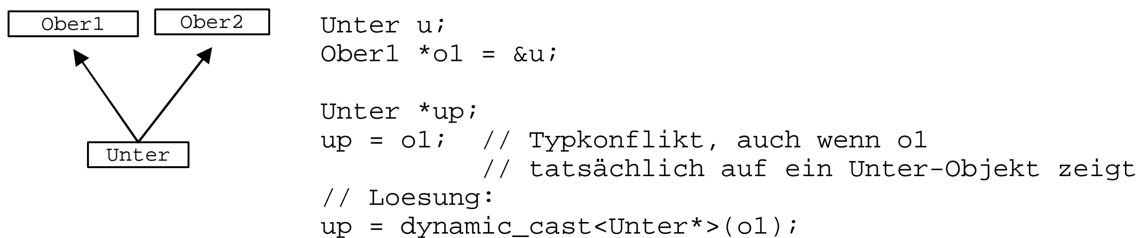



```
class A {…};   // polymorphe Klasse

class B : public A {…};

class C : public B {…};

void h(A& a) {
   try {
      B br = dynamic_cast <B&> (ar);
      // br = ar wäre falsch !
     ...
   }
   catch (bad_cast bc)
   {   cout << "Ausnahme: bad_cast" << endl;   }
}
```

Was passiert ?

- referenziert `a` ein B-Objekt, so wird `a` in den Typ `B&` konvertiert
- referenziert `a` ein C-Objekt, so wird `a` zunächst zu `C&` und dann zu `B&` konvertiert.
- referenziert `a` ein A-Objekt, so wird die Ausnahme `bad_cast` ausgelöst !

Beispiel 4:




```
Unter u;
Ober1 *o1 = &u;

Unter *up;
up = o1;  // Typkonflikt, auch wenn o1
          // tatsächlich auf ein Unter-Objekt zeigt
// Loesung:
up = dynamic_cast<Unter*>(o1);
```

```
// dynamic4.cpp: Beispiel fuer ein Cross-Cast mit dynamic_cast

#include <iostream>
#include <typeinfo>

class Ober1 {  virtual void f() {} };

class Ober2 {  virtual void g() {} };

class Unter : public Ober1, public Ober2 { };

int main() {
   try {
      Ober1 *o1 = new Unter;
      Ober2 *o2;

      if ((o2 = dynamic_cast<Ober2 *>(o1)) != 0) {
         cout << "typeid(o2) : " << typeid(o2) .name() << endl;
         cout << "typeid(*o2): " << typeid(*o2).name() << endl;
         }
      else throw bad_cast();

      Ober1 &o1r = *o1;
      Ober2 &o2r = dynamic_cast<Ober2 &>(o1r);
```

```
      cout << "typeid(o2r) : " << typeid(o2r).name() << endl;

   }
   catch (bad_cast) {
      cout << "Ausnahme: bad_cast" << endl;
   }
   char c; cin >> c;
   return 0;
}
/* Ausgabe
typeid(o2) : Ober2 *
typeid(*o2): Unter
typeid(o2r) : Unter
*/
```

17.4 Der Operator `static_cast`

Mit Hilfe des Operators `static_cast` können Typkonvertierungen für nichtpolymorphe Klassen und für elemenare Datentypen durchgeführt werden. Im Unterschied zum dynamic_cast findet keine Typüberprüfung zur Laufzeit statt.

Syntax:

```
static_cast<T> (a)
```

Wirkungsweise:

Der Ausdruck a wird in den Typ T konvertiert.

Regeln:

Annahme: a habe den Typ A.
A kann zu T konvertiert werden, wenn folgendes gilt:

- A kann implizit in T umgewandelt werden

```
long l = 123456789L;
...
int i    = static_cast<int> (l);
double d = static_cast<double>(l);
```

- T kann implizit zu A umgewandelt werden. Dann ist `static_cast` die Umkehrfunktion.

```
int i = 1;
void *ptr = &i;
int  *ip  = static_cast<int*>(ptr);
```

- T ist ein Aufzählungstyp und a ein ganzzahliger Wert, der im Wertebereich von T liegt.

```
enum Tag { Mo, Di, Mi, Do, Fr, Sa, So };
Tag naechsterTag (Tag t)
{ return(t == So)?  Mo : static_cast<Tag>(t+1); }
```

- A und T sind Zeigertypen und zeigen auf Objekte von Klassen, die voneinander abgeleitet sind.

```
class A {};
class B : public A {};
void f (A* ap) {
   B* bp = static_cast<B*>(ap);
}
```

Vorsicht: `static_cast` wird bereits zur Übersetzungszeit vom Compiler generiert und kann nicht feststellen, ob `ap` zur Laufzeit tatsächlich auf ein B-Objekt zeigt!

- `static_cast` darf `const` nicht wegkonvertieren.

```
const int max = 100;
int *ip = &max;                  // Typkonflikt, vom Compiler abgelehnt
ip = static_cast<int*>(&max);  // vom Compiler abgelehnt
ip = (int*)&max;                 // erlaubt
(*ip)++;                         // erhöht max !
```

Beispiel:

```
class X {};
class Y {};

   X* xp = new X;
   Y* yp = new Y;
   xp = yp;                      // nicht erlaubt
   xp = (X*)yp;                  // erlaubt !
   xp = static_cast<X*>(yp);     // vom Compiler abgelehnt!
```

Regel:

`static_cast` ist generell den normalen C-Casts vorzuziehen bei statischen Typkonvertierungen.

17.5 Der Operator `reinterpret_cast`

Der Operator `reinterpret_cast` wird dazu verwendet, den Inhalt von Speicherbereichen neu zu interpretieren. Man kann damit Zeigertypen in beliebige andere Zeigertypen oder sogar ganzzahlige Typen konvertieren. Das Resultat einer solchen Konvertierung ist jedoch implementierungsabhängig!

Syntax:

```
reinterpret_cast<T>(a)
```

Wirkungsweise:

Konvertiere den Typ A des Ausdrucks a in den Typ T unter Umgehung der C++ Typ-Prüfung.

Regeln:

- A, T beliebige Zeigertypen. Dann ist das Resultat die Reinterpretation des Speicherbereiches, auf den a zeigt.

```
class X {…};
class Y {…};

   Y *yp = reinterpret_cast<Y*>(xp);
   // Resultat implementierungsanhaengig
   int *ip = new int;
   char *cp = reinterpret_cast<char*>(ip);
```

Jeder Zeigertyp kann in jeden Zeigertyp konvertiert werden (gefährlich !).

- A Zeigertyp, T ganzzahliger Typ, der groß genug ist, um Adressen aufzunehmen.

```
X *xp = new X;
long l = reinterpret_cast<long>(xp);
l = 500;
xp = reinterpret_cast<X*>(l);
```

- • T Zeigertyp, A ganzzahliger Typ

```
long l = 1500;
```

```
char *cp = reinterpret_cast<char*>(1);
// sehr gefährlich !
```

- `reinterpret_cast` ist nicht geeignet für Konvertierungen innerhalb von Klassenhierarchien, da dieser Operator nur Low-level-Interpretationen des Speicherbereiches durchführt.
- `const` kann nicht wegkonvertiert werden.

17.6 Der operator `const_cast`

Der Operator `const_cast` kann ausschließlich dazu verwendet werden, um die `const`- oder `volatile`-Eigenschaft einem Typ hinzuzugeben oder wegzunehmen.

Syntax:

```
const_cast<T>(a)
```

Wirkungsweise:

Konvertiere den Ausdruck a vom Typ A in einen stärker oder schwächer `const` spezifizierten Typ T.

Regeln:

- A und T Zeigertypen, die sich lediglich durch ein oder mehrere `const`- oder `volatile`-Qualifizierer unterscheiden. Dann hat `const_cast<T>(a)` denselben Wert wie a, aber den Typ T.
- A ist ein L-Wert eines beliebigen Typs A und T stimmt bis auf ein oder mehrere `const`-Qualifizierer überein. Dann ist `const_cast<T>(a)` ein L-Wert des Typs T, der auf a verweist.
- Das Wegkonvertieren von `const` bei einem ursprünglich `const` definierten Objekt ist nicht erlaubt, wird aber von manchen Compilern noch zugelassen.

Beispiel 1:

```
const int *zci = new int(12345); // Zeiger auf const-int zeigt auf
                                 // nicht-const-int
delete zci; // geht nicht, da konstant
delete const_cast<int*>(zci);    // ist möglich
```

Beispiel 2:

```
int i = 10;
const int *ip = &i;
++(*ip);   // Fehler, da ip Zeiger auf int-Konstante
++*const_cast<int*>(ip); // funktioniert korrekt

const int max = 100;
++*const_cast<int*>(maxp); // nicht erlaubt, BC 5 tut's aber doch
```

Beispiel 3:

```
// const1.cpp: Beispiel fuer const_cast
class A {
public:
   A(int i):inhalt(i) {}
   void setInhalt(int i)  { inhalt = i; }
   int  getInhalt() const { return inhalt; }
private:
   int inhalt;
```

```
};

void f(const A& a) {
   const_cast<A&>(a).setInhalt(10);
   cout << "a.getInhalt(): " << a.getInhalt() << endl;
}

int main() {
   const A a1(5);
   f(a1);      // duerfte nicht gehen, BC 5 laesst dies aber zu
   A a2(6);
   f(a2);      // erlaubt
}
```

Bemerkung

Wenn man const_cast verwendet, sollte man genau wissen, was man tut.

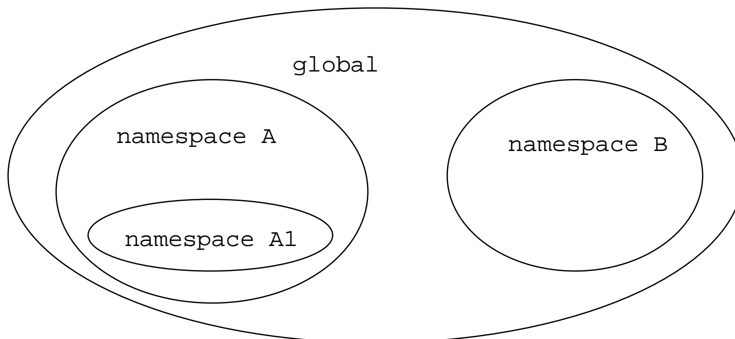
Fazit: Klassische Casts sollten generell nicht mehr verwendet werden. Entweder sollte man sich auf automatische Typumwandlungen verlassen oder sichere Casts anwenden. Da Casts immer potientielle Fehlerquellen darstellen, sind diese zumindest leichter zu lokalisieren !

18 Namensräume

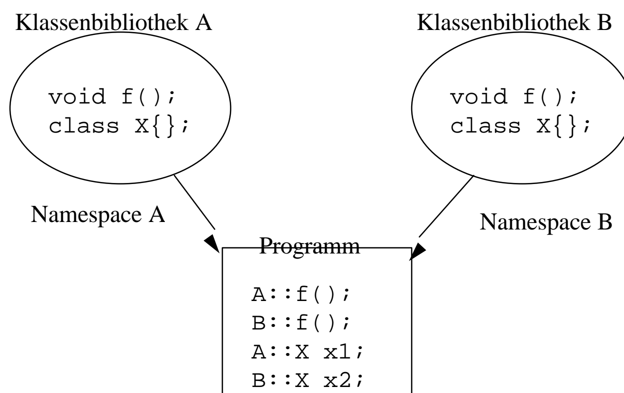
Alle Namen, die außerhalb von Blöcken oder Klassen deklariert werden, sind global. Sie liegen damit im globalen Namensraum.

Problem: Beim Einsatz verschiedener Klassenbibliotheken können Namenskonflikte auftreten, z.B. durch gleichnamige Klassen, Funktionen, Konstanten.

Lösung: *namespaces* (Namensbereiche, Namensräume))



Konkreter:



Allgemeine Definition:

```
namespace [Bezeichner] { Deklarationsfolge }
```

Beispiel:

```
namespace A {
  void f(int);
  void f(char);
  class String {…};
}
```

```
namespace B {
  void f(int);
  void f(char);
  class String {…};
}
```

Anwendung:

```
A::String s1="Hallo";
A::f(1);
B::String s2="Welt"; // B:: ist eine eindeutige Qualifikation
// Vorsicht! s1=s2 ist falsch, da unterschiedliche Typen vorliegen!
```

Regeln:

- Ein Namensraum kann innerhalb einer Programmdatei in mehrere zusammengehörende Teile zerlegt werden.

```
namespace A {
      class X {…};
}
…
namespace A {
      const int max=1000;
}
```

- Namensräume können beliebig geschachtelt werden.

```
namespace X {
   namespace Y {
      void f();
   }
}
X::Y::f(); // Vollständig qualifizierter Zugriff auf f
```

- Namen, die außerhalb von Namensräumen deklariert sind, gehören zum globalen Namensraum und können mit `::` zugegriffen werden.

```
int i;
namespace A {
   int i = 0;
   ::i = 1;
}
```

- Klassen sind implizit auch Namensräume

```
class A { … void f(); }
void A::f() {…}
```

- Der Standardnamensraum der ANSI-Klassenbibliotheken heißt `std`.

Definition der Elemente eines Namensraumes

Innerhalb eines Namensraumes kann man Variablen, Funktionen und Klassen sowohl deklarieren, als auch definieren. Es ist auch möglich, Größen, die innerhalb des Namensraumes deklariert wurden, außerhalb des Namensraumes zu definieren.

Beispiel:

```
namespace A {
   void f();
}

void A::f() {  cout << "f()\n"; }
```

Beispiel:

```
// namespace1.h: Beispiel fuer Unterteilung eines Namensraumes

namespace A {
   class X {
   public:
      X(int = 0);
      int info() const;
   private:
      int x;
   };
} // Ende namespace A
```

```
// namespace1.cpp: Implementierungsdatei

#include "namespace1.h"
#include <iostream>

namespace A {
   X::X(int n) : x(n) {}
   int X::info() const {
      cout << "X: " << x << "\n";
   }
} // Ende namespace A
```

```
// namespace1t.cpp: Testprogramm

#include "namespace1.h"
#include <iostream>

int main() {
   A::X x1(4711);
   x1.info();
}
```

Aliasnamen

Kurze Namen für Namensräume bergen wiederum die Gefahr von Namenskonflikten deshalb wird man im Allgemeinen lange sprechende Namen verwenden. Beim Zugriff im Programm sind diese langen Namen allerdings lästig, daher definiert man dafür kurze Aliasnamen.

Aliasdefinition:

```
namespace Bezeichner = NamespaceName;
```

Beispiel:

```
namespace generic {
   …     // generic++ Klassenbibliothek
}
namespace G=generic;
G::G_String S="Hallo";
```

Beispiel:

```
namespace X { namespace Y {void f(); }}
X::Y::f();           // so wuerde man normal f aufrufen
namespace XY=X::Y;   // Aliasdefinition
XY::f();             // so ruft man jetzt f auf
```

using-Deklaration

In einer using-Deklaration wird ein Name aus einem Namensraum in den Geltungsbereich eingeführt, in dem die using-Deklaration erfolgt.

Syntax:

```
using NamespaceName::Bezeichner;
```

Beispiel 1:

```
using A::String;
using B::f;
   …
String s2="Welt";    // A::String
f(3);                // B::f(int)
```

Beispiel 2:

```
namespace A {
   int i;
   int h(int);
   int h(double);
}

void g() {
   int i;
   using A::i;  //Fehler, da ein lokales i vorhanden ist
   using A::h;
   h(3);          //A::h(int)
   h(3.14);       //A::h(double)
}
```

using-Direktive

Durch eine using-Direktive können sämtliche Namen eines angegebenen Namensraums zugreifbar gemacht werden.

Syntax:

```
using namespace NamespaceName;
```

Beispiel

```
namespace A {
   class X {…};
   void f();
}

void g() {
   using namespace A;
   X x1;       // A::X
   f();        // A::f()
};
```

Voreinstellung für den Standard-Namensraum:

```
using namespace std;
```

Unbenannte Namensräume

```
namespace {
   int i;
   double x;
   void f();
}
```

Die definierten Namen sind nur innerhalb der aktuellen Source-Datei bekannt. In C würde man anstelle des obigen Beispiels `static int i;` schreiben. Deshalb kann man unbenannte Namensräume als Ersatz für globale `static`-Größen verwenden. Generell sollte man daher auf klassische `static`-Variablen und -Funktionen verzichten.

Klassen als Namensraum

Jede Klasse stellt automatisch einen Namensraum dar. Eine abgeleitete Klasse ist ein geschachtelter Namensraum innerhalb des Namensraumes der Basisklasse.

Beispiel: (vgl. Abschnitt 3)

```
// using1.cpp: Anwendung von namespaces auf Klassen

#include <iostream.h>

class A {
public:
   void f(char c)   { cout << "A::f(char): "   << c << endl; }
   void f(double d) { cout << "A::f(double): " << d << endl; }
};

class B : public A {
public:
   void f(int i) { cout << "B::f(int): " << i << endl; }
};

int main() {
    B b;
    b.f('c');  // ruft B::f(int) auf !
    b.f(3.14); // ruft B::f(int) auf !
    return 0;
}
```

Die Funktion `B::f` überdeckt alle geerbten Funktionen `A::f` !

Lösung:

```
class B : public A {
public:
    void f(int i) { cout << "B::f(int): " << i << endl; }
    using A::f;
};
```

Die f´s von A ueberladen nun das f von B.

Anhang

A.1. Standardheaderdateien von ANSI-C:

ANSI-C	*ANSI-C++*	*Bedeutung*
`<assert.h>`	`<cassert>`	Zusicherung
`<ctype.h>`	`<cctype>`	Typprüfungs- und Konvertierfunktionen für `char`
`<errno.h>`	`<cerrno>`	ANSI-C Fehlerbehandlung
`<float.h>`	`<cfloat>`	Implementationsspezifische Makros für die Floating-Point-Arithmetik
`<limits.h>`	`<climits>`	typ-spezifische Grenzen
`<locale.h>`	`<clocale>`	länderspezifische Anpassungen
`<math.h>`	`<cmath>`	mathematische Funktionen
`<setjmp.h>`	`<csetjmp>`	Deklaration von setjmp und longjmp
`<signal.h>`	`<csignal>`	Elementare Signalbehandlung
`<stdarg.h>`	`<cstdarg>`	Verarbeitung beliebig langer Argumentlisten
`<stddef.h>`	`cstddef>`	Standardtypdefinitionen
`<stdio.h>`	`<cstdio>`	Standard-Ein-/Ausgabe ANSI-C
`<stdlib.h>`	`<cstdlib>`	Standard-Makros und Funktionen
`<string.h>`	`<cstring>`	`char`-String-Funktionen
`<time.h>`	`<ctime>`	Zeitfunktionen
`<wchar.h>`	`<cwchar>`	Definitionen zu `wchar_t`, den 16-Bit-Zeichen
`<wctype.h>`	`<cwctype>`	Analogon zu `<cctype>` für `wchar_t`

A.2. Reservierte Wörter

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

A.3. Übersicht über C++-Operatoren

Rang	*Operator*		*Anwendung*
0	::	Bereichsoperator Klasse	*Klasse::komponente*
	::	Bereichsoperator Namensraum	*namespace::komponente*
	::	Zugriff auf globales Element	*:: name*
1	.	Elementzugriff	*objekt.komponente*
	->	Elementzugriff	*pointer->komponente*
	[ ]	Subskribierung	*pointer[ausdruck]*
	( )	Funktionsaufruf	*ausdruck(ausdr_liste)*
	( )	Typkonvertierung	*typ(ausdr_liste)*
	++	Postfix-Inkrementierung	*lvalue++*
	--	Postfix-Dekrementierung	*lvalue--*
	typeid	Typidentifikation	*typeid(typ)*
	typeid	Laufzeit-Typidentifikation	*typeid(ausdruck)*
	dynamic_cast	geprüfte Laufzeit-Typkonvertierung	*dynamic_cast<typ>(ausdruck)*
	static_cast	geprüfte Compile-Zeit-Typkonvertierung	*static_cast<typ>(ausdruck)*
	reinterpret_cast	ungeprüfte Typkonvertierung	*reinterpret_cast<typ>(ausdruck)*
	const_cast	Const-Typkonvertierung	*const_cast<typ>(ausdruck)*
2	sizeof	Größe eines Objekts	*sizeof ausdruck*
	sizeof	Größe eines Typs	*sizeof(typ)*
	++	Präfix-Inkrementierung	*++lvalue*
	--	Präfix-Dekrementierung	*--lvalue*
	~	Komplement	*~ausdruck*
	!	NOT	*!ausdruck*
	-	unäres Minus	*-ausdruck*
	+	unäres Plus	*+ausdruck*
	&	Adresse von	*&L-Wert*
	*	Dereferenzierung	**ausdruck*
	new	Allokieren	*new typ* *new typ(ausdrucks-liste)*
	new[]	Array allokieren	*new [ ] typ*
	delete	Deallokieren	*delete pointer*
	delete[]	Array deallokieren	*delete[] pointer*
	()	cast (Typkonvertierung)	*(typ)ausdruck*
3	.*	Elementzugriff	*objekt.pointer_auf_komponente*
	->*	Elementzugriff	*pointer->pointer_auf_komp*
4	*	Multiplikation	*ausdruck * ausdruck*
	/	Division	*ausdruck / ausdruck*
	%	Modulo	*ausdruck % ausdruck*
5	+	Addition	*ausdruck + ausdruck*
	-	Subtraktion	*ausdruck - ausdruck*
6	<<	Links-Shift	*ausdruck << ausdruck*
	>>	Rechts-Shift	*ausdruck << ausdruck*

7	`<`
	`<=`
	`>`
	`>=`
		Kleiner
		Kleiner oder gleich
		Größer
		Größer oder gleich
			ausdruck < ausdruck
			ausdruck <= ausdruck
			ausdruck > ausdruck
			ausdruck >= ausdruck

8	`==` `!=`	gleich ungleich	*ausdruck == ausdruck* *ausdruck != ausdruck*
9	`&`	Bitweises AND	*ausdruck & ausdruck*
10	`^`	Bitweises exklusives OR	*ausdruck ^ ausdruck*
11	`|`	Bitweises OR	*ausdruck | ausdruck*
12	`&&`	logisches AND	*ausdruck && ausdruck*
13	`||`	logisches OR	*ausdruck || ausdruck*

14	`=`
	`*=`
	`/=`
	`%=`
	`+=`
	`-=`
	`<<=`
	`>>=`
	`&=`
	`|=`
	`^=`
		Zuweisung
		Multiplikation und Zuweisung
		Division und Zuweisung
		Modulo und Zuweisung
		Addition und Zuweisung
		Subtraktion und Zuweisung
		Links-Shift und Zuweisung
		Rechts-Shift und Zuweisung
		And und Zuweisung
		Or und Zuweisung
		exklusives OR und Zuweisung
			l-wert = ausdruck
			*l-wert *= ausdruck*
			l-wert /= ausdruck
			l-wert %= ausdruck
			l-wert += ausdruck
			l-wert -= ausdruck
			l-wert <<= ausdruck
			l-wert >>= ausdruck
			l-wert &= ausdruck
			l-wert |= ausdruck
			l-wert ^= ausdruck

15	`? :`	Bedingter Ausdruck	*ausdruck ? ausdruck : ausdruck*
16	`throw`	Ausnahme auswerfen	*throw ausdruck*
17	`,`	Folge von Ausdrücken	*ausdruck,ausdruck*

A.3.1. Die Operatoren `.*` und `->*`

Die Operatoren `.*` und `->*` werden verwendet bei Zeigertypen, die auf Komponenten von Klassen bzw. Strukturen verweisen können.

```
struct X {
      int i, j, k;
      int g (int);
      int h (int);
};

X x1, x2;
int *pi=&x1.i; //Adresse der Komponente i von x1
```

Alternative:

```
int X::*pix;       // lies: pix ist Zeiger auf eine X-Komponente vom Typ int
pix = &X::i;       // relative Adresse von i innerhalb eines X-Objektes

x1.*pix = 0;       // implizit x1.i=0;
x2.*pix = 0;       // implizit x2.i=0;

pix = &X::k;       // relative Adresse von k innerhalb eines X-Objektes
x1.*pix = 0;       // x1.k = 0;

// Zugriff über Pointer
X *px = &x1;
px->*pix = 1;      // px->i = 1;
// alternativ: (*px).*pix=1;
```

Anwendung auf Funktionspointer

```
int(X::*fp)(int);       // fp ist ein Zeiger auf X-Komponente vom Typ
                        // int-Funktion mit einem int-Argument

fp = &X::g();           //Relative Adresse von g innerhalb eines X-Objektes
int ii = (x1.*fp)(12); // entspricht x1.g(12)
// alternativ: int jj = (px->*fp)(12); //entspricht px->g(12)
```

A.4. Das Einbinden von C-Funktionen

Wegen des Überladens von Funktionsnamen, werden bei C++ bei der Übersetzung die Funktionsnamen anders umgesetzt als bei C. Daher können mir C-Compilern übersetzte Funktionen nur hinzugebunden werden, wenn sie im C++-Programm explizit als C-Code deklariert werden.

```
extern "C" void printf(char *, ... );
extern "C"
{ irgendwelche Deklarationen... }
```

Sollen Headerfiles gleichzeitig durch C- und C++-Compiler verwendet werden, so kann dies durch die Verwendung eines speziellen Makros möglich gemacht werden:

```
#ifdef_cplusplus
      extern "C" ...
#endif
```

Das Symbol `_cplusplus` wird vom C++-Compiler vordefiniert.

A.5. Logische Einteilung von Elementfunktionen

- Verwaltungsfunktionen
- Initialisierungen (Konstruktoren)
- Zuweisungen (=, +=, ...)
- Speicherreservierungen (eigenes `new`, `delete`)
- Datentypumwandlungen (Meistens implizit aufgerufen.)

Implementierungsfunktionen

- mögliche Operationen der Klasse (Schnittstelle nach außen)

Hilfsfunktionen

- intern benötigte Funktionen (meist `private` oder `protected`)

Zugriffsfunktionen

- Zugriffe (lesend oder schreibend) auf Attribute (z. B.: `getX`)
- Lade- und Speicherfunktionen (zum Speichern und Laden von Objekten in oder aus Dateien)

A.6. Vordefinierte Elementfunktionen bei Klassen

Beispiel: `class Leer {};`

Welche Funktionen sind bereits vordefiniert ?

- *Standardkonstruktor* `Leer();`

Deklarieren von Objekten, keine Initialisierung.

- *Standardkopierkonstruktor* `Leer(const Leer&);`

Elementweises Kopieren aller nicht-statischen Datenelemente, dabei ggf. Aufruf des Kopierkonstruktors von Elementobjekten.

- *Destruktor* `~Leer();`

Wird tatsächlich nur generiert, wenn die Klasse von einer Klasse mit Destruktor abgeleitet wird, hat standardmäßig nur die Funktion, ggf. den Destruktor der Basisklasse aufzurufen beim Destruieren eines Objektes.

- *Zuweisungsoperator* `Leer& operator = (const Leer&);`

Elementweises Zuweisen aller nicht-statischen Datenelemente, dabei ggf. Aufruf des Zuweisungsoperators von Elementobjekten.

- *Adress-Operator* `Leer* operator& ();`
  `const Leer* operator& () const;`

Liefert die Adresse des aktuellen Objektes zurück, also den Wert des `this`-Zeigers.

A.7. Container-Operationen

Als Ergänzung zu dem Abschnitt über Klassen-Templates hier noch eine Übersicht über standardmäßig vorhandene Funktionen und ihrer Wirkungsweise bei den wichtigsten Container-Klassen der Standard Template Library.

`Sei a` ein Containerobjekt vom Typ `vector`, `list` oder `deque`
`n int`, `t` Element, `i,j` Iteratoren

Ausdruck	*Returntyp*	*Semantik*	*Container*
`a.insert(i,t)`	`iterator`	Kopie von `t` vor `i` einfügen.	`vector, list, deque`
`a.erase(i)`	`iterator`	löscht das Element auf das `i` zeigt.	`vector, list, deque`
`a.erase(i,j)`	`iterator`	löscht die Elemente im Bereich `[i,j)`	`vector, list, deque`
`a.clear()`	`void`	entspricht `erase(begin(), end())`	`vector, list, deque`
`a.front()`	`T&; const T&`	erstes Element	`vector, list, deque`
`a.back()`	`T&; const T&`	letztes Element	`vector, list, deque`
`a.push_front(x)`	`void`	am Anfang einfügen	`list, deque`
`a.push_back(x)`	`void`	am Ende einfügen	`vector, list, deque`
`a.pop_front()`	`void`	am Anfang entfernen	`list, deque`
`a.pop_back()`	`void`	am Ende entfernen	`vector, list, deque`
`a[n]`	`T&; const T&`	n-tes Element	`vector, deque`
`a.at(n)`	`T&; const T&`	n-tes Element	`vector, deque`