**Prof. Dr. Peter Baeumle-Courth**

# C++ kompakt

# Vorwort

Das vorliegende PDF-Dokument ist die digitale Version meines bereits 1995 erschienen Buches “C++ kompakt”. Da dieses nicht mehr neu aufgelegt und daher im Buchhandel nur noch schwer verfügbar sein wird, stelle ich diesen Text hiermit im Intranet der FHDW Bergisch Gladbach zur Verfügung.

Als Dozent der Fachhochschule der Wirtschaft (FHDW) und zuvor am Bildungszentrum für informationsverarbeitende Berufe e.V. (b.i.b.) in Bergisch Gladbach, einer praxisorientierten Erwachsenenbildungseinrichtung im Bereich der Elektronischen Datenverarbeitung, beschäftige ich mich seit vielen Jahren unter anderem mit modernen Programmiersprachen, zu denen natürlich gerade die objektorientierten Sprachen C++ und Smalltalk gehören; aber auch die in Zusammenhang mit der Verbreitung des Internet geborene Sprache Java ist in diesem Rahmen zu erwähnen.

Meine Erfahrungen haben gezeigt, daß es bei weitem nicht genügt, die Syntax einer Programmiersprache wie C++ zu beherrschen, ohne das Prinzip der objektorientierten Programmierung (OOP) gründlich kennengelernt und verinnerlicht zu haben. Auf der anderen Seite habe ich manches konzeptionelle Seminar zur objektorientierten Software-Entwicklung mit dem Gefühl verlassen, daß es ruhig hätte konkret in einer Programmiersprache zur Sache gehen dürfen.

Auf diesem Hintergrund entstand dieses Buch auf der Grundlage meines mehrfach geänderten Unterrichtsskriptums, in dem ich versuche, sowohl objektorientierte Gedankengänge als auch die Syntax und Semantik von C++ zu vermitteln: allgemein sowie an zahlreichen Beispielen. Die in diesem Buch, teilweise nur auszugsweise, abgedruckten Quelltexte können von mir per elektronischer Mail unter der unten aufgeführten Adresse angefordert werden.

Die Programme sind, soweit nicht im Einzelfall anders vermerkt, im ANSI-Standard für C++ implementiert. Lediglich dort, wo die zur Verfügung stehenden Compiler die neuen Features noch nicht umsetzen, wurde auf den bisherigen AT&T-Quasistandard zurückgegriffen.

Ursprünglich wurden die hier vorgestellten Programme für einen Unterricht auf einer Hewlett Packard HP9000 RISC-Maschine unter dem Betriebssystem HP-UX 9.04 mit dem dortigen HP C++ Compiler 3.0 bereitgestellt. Die allermeisten dieser Quellen lassen sich jedoch ohne oder ohne große Änderungen mit anderen Compilern übersetzen. Ich selbst habe hierzu unter DOS und Windows NT die C++ Compiler von Borland und Symantec eingesetzt.

Die Übungsaufgaben habe ich bewußt in gesammelter Form an den Schluß des Buches gestellt, damit Leser, die intensiv üben wollen, diese Anregungen an einer zentralen Stelle vorfinden. Gleichzeitig werden jedoch im laufenden Text Hinweise auf die Übungsaufgaben gegeben, so daß auch zu einem bestimmten Thema eine Übung gefunden werden kann.

Durch ein umfangreiches Stichwortverzeichnis hoffe ich, daß dieses Buch auch zum gelegentlichen Nachschlagen verwendet werden kann. Darüber hinaus möchte ich auf

die Newsgruppen zum Thema C++ (comp.lang.c++, de.comp.lang.c++) sowie die entsprechenden FAQs (frequently asked questions) hinweisen: auf der World Wide Web-Seite http://nz.com/webnz/robert/ sind eine ganze Reihe von Quellen im Internet (WWW, ftp, News) genannt, wo weitere Informationen zu C++ abgerufen werden können. Speziell sei die FAQ-Liste von Marshall Cline (unter der World Wide Web Adresse http://www.cerfnet.com/~mpcline/C++-FAQs-Lite) genannt.

Das vorliegende Buch faßt einige Jahre Unterrichtserfahrung im Bereich C++ zusammen. Gleichwohl ist Unterricht immer lebendigen Prozessen unterworfen und hängt von den konkreten Personen ab. Jede Form konstruktiver Kritik ist daher herzlich willkommen.

Die Abschnitte 1.2 und 1.3 zur Datenabstraktion und den Grundlagen der Objektorientierten Programmierung entstanden in enger Zusammenarbeit mit meinem Kollegen, Herrn Norbert Groß-Eitel, dem ich dafür an dieser Stelle sehr herzlich danke. Gleichwohl zeichne ich für alle inhaltlichen Fehler oder Ungenauigkeiten selbst verantwortlich.

Desweiteren danke ich Herrn Joachim Koch vom b.i.b. in Paderborn und Herrn Dirk Hennies vom Steuer- und Wirtschaftsverlag (S+W Verlag) in Hamburg herzlich für die vertrauensvolle und konstruktive Zusammenarbeit.

Selbstverständlich werden in diesem Buch sämtliche verwendeten Markennamen und Warenzeichen auch ohne explizite Aufzählung berücksichtigt.

Ich wünsche den Leserinnen und Lesern dieses Buches viel Spaß und Erfolg.

Prof. Dr. Peter Baeumle-Courth
Bergisch Gladbach, im Januar 2006

eMail: peter.baeumle-courth@fhdw.de
WWW: http://www.bg.bib.de/~fhdwbap

# Inhaltsverzeichnis


## 1. Einführung 1

1.1. Voraussetzungen: Was Sie immer schon wissen sollten... 1

1.2. Datenabstraktion 2

1.3. Was ist Objektorientierte Programmierung? 7

1.4. Von C zu C++ 12

1.5. Erste Programme 16

1.6. Übersicht: Geschichtliche Entwicklung 18

## 2. Das Klassenkonzept 19

2.1. Klasse und Instanz 19

2.2. Schnittstelle und Implementation 23

2.3. Schutzbereiche 25

2.4. Inline 25

2.5. Zeiger auf Klassenelemente (Operatoren .* und ->*) 31

## 3. Konstruktoren und Destruktoren 35

3.1. Initialisierung von Objekten 35

3.2. Konstruktoren, Destruktoren und das Überladen 37

3.3. Referenzen 40

3.4. Der Kopierkonstruktor 42

3.5. Statische Klassenmitglieder 43

3.6. Konstante Klassenmitglieder 47

## 4. Dynamische Speicherverwaltung 49

4.1. C: malloc und free 49

4.2. C++: new und delete 50

## 5. Überladen von Operatoren 55

5.1. Überladen als Klassenmitgliedsfunktionen 55

5.2. Überladen als Freundfunktion 61

6. Streams I: Ein- und Ausgabe 63
6.1. Standard-Streams des Betriebssystems UNIX 63
6.2. Standard-Ein- und -Ausgabe in ANSI-C 63
6.3. Ein- und Ausgabe-Streams in C++ 64
6.4. Überladen der Schiebeoperatoren 68

7. Freunde 71
7.1. Befreundete Funktionen 71
7.2. Überladene Freundfunktionen 72
7.3. Befreundete Klassen 74

8. Vererbungslehre: Abgeleitete Klassen 79
8.1. Vererbung und die Wiederverwendbarkeit von Code 79
8.2. Vererbungsarten: private, protected und public 80
8.3. Mehrfachvererbung 92
8.4. Kopierkonstruktoren in abgeleiteten Klassen 96

9. Polymorphismus in C++ 101
9.1. Virtual Reality 101
9.2. Frühe und späte Bindung 102

10. Streams II: Dateioperationen 113
10.1. Sequentielle Ein- und Ausgabe 113
10.2. Wahlfreier Zugriff (Random Access) 115

11. Templates 121
11.1. Template-Funktionen 121
11.2. Klassen-Templates 124

12. Exception Handling 129
12.1. Exception Handling in C 129
12.2. Exception Handling in C++ 132

# 13. Ausblick 137

# Übungen 139

Übung 1: Das erste Programm 139
Übung 2: Überladene Funktionen 139
Übung 3: Überladene Funktionen 140
Übung 4: Tauschen 141
Übung 5: Tastatureingabe in C++ 141
Übung 6: Rechnen mit Brüchen 142
Übung 7: Konstruktoren und Destruktoren 145
Übung 8: Statische Klassenmitglieder 145
Übung 9: Operator Overload 145
Übung 10: Operator Overload 146
Übung 11: Referenzen 146
Übung 12: Klassendesign 146
Übung 13: Ein fehlerhaftes Programm 147
Übung 14: Vererbungslehre - Klasse ANGESTELLTER 147
Übung 15: Weitere Operatoren in der Klasse BRUCH 153
Übung 16: Eine String-Klasse 154
Übung 17: Noch einmal eine String-Klasse 154
Übung 18: Funktionstemplates 158
Übung 19: Ein Klassentemplate 158

# A. Anhang I: Weitere Beispielprogramme 161

A.1. Beispielklasse KTime: Zeit und Datum 161
A.2. Beispielklassen KElement und KListe: Lineare Listen 173
A.3. Templateklassen KElement<T> und KListe<T> 178

# B. Anhang II: Ergänzungen 183

B.1. Digraphen 183
B.2. Trigraphen 184
B.3. Schlüsselwörter 185
B.4. Vererbung und Objektorientiertes Design - Einige Tips 186

**Literaturhinweise** 187

**Stichwortverzeichnis** 189

# 1. Einführung

Objektorientierte Programmierung bedeutet für viele Software-Entwicklerinnen und -Entwickler[1] häufig zwangsläufig auch C++, denn diese Sprache ermöglicht einen sanften Umstieg von einer prozeduralen Sprache wie C. Anders als in der zweiten, recht weit verbreiteten objektorientierten Sprache Smalltalk, die eine radikale Umgewöhnung bereits in der Arbeitsweise der Programmerstellung erfordert, läßt es C++ zu, daß ganz langsam und gemütlich, je nach Arbeitstempo des einzelnen C-Programmierers, mehr und mehr objektorientierte Elemente in die Programmierung einfließen.

Und dies genau ist einer der Hauptkritikpunkte an C++. Denn: es gibt viel zu viel sogenannten C++-Code, der noch sehr weit entfernt ist von Objektorientierung! Die Softwarefirmen, die damit leben (müssen), werden dies spätestens beim ersten Update, beim Release- oder gar beim Betriebssystemwechsel deutlich spüren!

Im Rahmen dieses Buches soll daher von Anfang an der Hauptgedanke „objektorientiertes Programmieren” im Vordergrund stehen, C++ dient hierbei als das Werkzeug.

Formale Grundlage ist der sogenannte „ISO/ANSI C++ Draft”, der (vorläufige bzw. geplante) Standard für C++, den Sie im World Wide Web unter der Adresse http://www.cygnus.com/misc/wp finden können[2]. Die hier vorgestellten Programme wurden unter UNIX (HP-UX von Hewlett-Packard) mit dem HP-C++-Compiler 3.0 entwickelt und getestet, zum Teil auch unter MS-DOS mit Borland Turbo C++ 3.0 und 5.0 sowie unter Microsoft Windows NT mit den Symantec C++ Compilern in den Versionen 6.11 und 7.21. Eventuelle compilerspezifische Sachverhalte sind im Einzelfall besonders erwähnt.

## 1.1. Voraussetzungen: Was Sie immer schon wissen sollten...

Der vorliegende Text geht davon aus, daß der Leser bzw. die Leserin über solide Kenntnisse der Programmiersprache C (ANSI-C) verfügt. Auf eine ausführliche Wiederholung von C wird in diesem Rahmen verzichtet.

Insbesondere sind für die Praxis von Bedeutung: Strukturen (structs in C, in Pascal records), das Arbeiten mit Headerfiles (`#include <xy.h>`, `#include "xy.h"` usw.), mit Projekten (`make` o. ä.) und zumindest im Ansatz auch mit den sogenannten

---

[1] Ich bitte alle Leserinnen um Verständnis, daß ich im folgenden die im Deutschen übliche männliche Form verwende, damit der Satzbau nicht noch aufwendiger wird, als er bei meiner Vorliebe für lange Sätze manchmal zu werden droht.

[2] Wenn Sie wollen, können Sie auch dem ANSI-Komitee einen Brief schreiben und um die jeweils aktuelle Fassung des C++-Standards bitten. Die Adresse des Komitees lautet: American National Standards Institute (ANSI), Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005.

*Libraries* (Bibliotheken). Alleine aus Platzgründen werden jedoch die hier vorgestellten Programme in der Regel „am Stück” und nicht auf mehrere Dateien verteilt vorgestellt.

Unter UNIX enden C-Quelltext-Dateinamen in der Regel auf .c (kleingeschriebenes c), C++-Quelltexte auf .C (großgeschriebenes C); unter DOS/Windows wird zwischen Groß- und Kleinschreibung nicht unterschieden, dort haben sich die Endungen .C für C- und .CPP für C++-Quelltexte durchgesetzt.

Da alle relevanten Compiler mit der Endung .CPP zurechtkommen, werden wir im folgenden die Dateinamen von C++-Quelltexten auf .cpp enden lassen. Headerfiles erhalten, wie bereits in C, die Endung .h.

Für die optische Gestaltung der Quelltexte (Einrückung, Leerzeilen, Klammerung usw.), die Kommentierungen usw. sollen hier keine verbindlichen Vorgaben gemacht werden. Jeder sollte sich nur stets bemühen, ein einheitliches und Übersicht gewährendes Konzept durch all seine Programme hindurch beizubehalten.

Die Aufteilung des Quelltextes in Deklarationen, Definitionen und Impementationen sollte bei etwas größeren Projekten wohldurchdacht sein. Im folgenden[3)] werden schon aus Platzgründen bei den kleineren Programmen oftmals alle Komponenten in eine einzige cpp-Datei aufgenommen. Ebenso wird in vielen Beispielen ein Programmquelltext auf die gerade relevanten Punkte reduziert, insbesondere findet dort dann keine umfassende *Alles-mögliche- kann-passieren*-Fehlerbehandlung statt.

## 1.2. Datenabstraktion[4)]

Die grundlegende Idee hinter der Datenabstraktion ist die, daß ein Objekt (z.B. eine Datei, ein Stack[5)], eine Warteschlange, ein Fenster usw.) nur über Operationen verwendet und verändert wird; seine interne Struktur wird jedoch vor all denen verborgen, die dieses Objekt nutzen möchten. Dabei lassen sich zwei Grade der Datenabstraktion unterscheiden:

- die einfache Datenabstraktion
- und das Bilden von abstrakten Datentypen (ADT).

---

3) Der Autor ist sich darüber bewußt, daß die neue deutsche Rechtschreib-Reform (ehemals: Rechtschreibreform) von 1996 einige orthographische Spielräume läßt. Dieser Text entstand ohne einen Duden neuester Auflage auf dem Schreibtisch.

4) Dieser Abschnitt lehnt sich stark an das als Standardwerk zu bezeichnende Buch von Helmut Balzert (s. Literaturverzeichnis) an.

5) Ein *Stack* (Stapel) ist ein „LIFO”-Speicher (*last in, first out*), das heißt ein Speicherbereich, bei dem das zuletzt abgelegte Element als erstes wieder weggenommen wird. So funktioniert in der Kantine ein Tablettstapel.

### 1.2.1. Einfache Datenabstraktion

Eine einfache Datenabstraktion besteht aus logisch zusammengehörenden Zugriffsoperationen, die selbst wiederum funktionale Abstraktionen darstellen; sie ist dadurch gekennzeichnet, daß ihre Zugriffsoperationen auf ein gemeinsames internes Gedächtnis zugreifen, das Daten über das Aufruf-Ende einer Zugriffsoperation hinaus aufbewahrt. Das interne Gedächtnis ist entweder an die Laufzeit des entsprechenden Software-Systems gebunden ist oder es werden dessen Daten in einer Datei gespeichert.

Sie beschreibt ein Objekt nicht mehr durch dessen Struktur, sondern charakterisiert es ausschließlich durch die Definition der darauf ausführbaren Operationen, und sie wird durch ihre Beschreibung kreiert, d.h. mit der Beschreibung ist gleichzeitig die Vereinbarung genau eines Exemplars verbunden.

### Beispiel: Ein abstrakter Stack

Bei einem *Stack* (auch *Kellerspeicher* genannt) kann immer nur auf das oberste Objekt zugegriffen werden; das Objekt, das zuletzt in den Kellerspeicher gebracht wurde, muß auch als erstes wieder entnommen werden. Ein Stack kann mit den folgenden Operationen bearbeitet werden:

- neues Objekt im Keller aufbewahren (*push*),
- oberstes Objekt aus Keller entnehmen (*pop*),
- oberstes Objekt im Keller ansehen (*top* oder *tos - top of stack*).

Damit kann das abstrakte Objekt STACK allein durch die ihn charakterisierenden Zugriffsoperationen definiert werden[6]:

```
data abstraction  INTEGER_STACK;

  procedure PUSH(in: Z: integer; out F: boolean);

      Die Zahl Z wird im Keller abgelegt.
      Bei Überlauf des Kellers wird F := true gesetzt.


  procedure POP(out F: boolean);

      Die oberste Zahl wird aus dem Keller entfernt.

      Bei leerem Keller wird F := true gesetzt.
```

---

[6] Wir verwenden hier eine bewußt an Pascal angelehnte Pseudocodenotation, damit deutlich wird, daß dies nicht für C++ spezifisch ist.

```
  procedure TOP(out Z: integer, out F: boolean);

      Die oberste Zahl im Keller wird gelesen.
      Bei leerem Keller wird F := true gesetzt.

end INTEGER_STACK;
```

Wenn der Benutzer sich über die Wirkung der Operationen im klaren ist, dann reicht diese Beschreibung zur Benutzung der Datenabstraktion aus. Wie der Keller nun im Detail implementiert wird, ist für den Benutzer der Datenabstraktion STACK unwichtig.

Eine Implementierung kann z.B. aussehen wie folgt.

```
implementation INTEGER_STACK;

const  N = 100;  { internes Gedächtnis }

var    S : array[ 1 .. N ] of integer;

   DEPTH : integer;


  procedure PUSH(in: Z: integer; out F: boolean);

  begin

  F := false;

  if DEPTH >= N then

      F := true

  else begin

      DEPTH := DEPTH + 1;

      S[DEPTH] := Z

      end

  end;


  procedure POP(out F: boolean);

  begin
```

```
  F := false;
  if DEPTH >= 1 then
      DEPTH := DEPTH - 1
  else
      F := true
  end;

  procedure TOP(out Z: integer, out F: boolean);
  begin
  F := false;
  if DEPTH >= 1 then
      Z := S[DEPTH]
  else
      F := true
  end;

  procedure init;
  begin
  DEPTH := 0
  end;
end implementation.
```

Die Verwendung der einfachen Datenabstraktion bringt einige Vorteile mit sich:

- Die abstrakten Datenobjekte können einzig und allein durch Anwendung der definierten Zugriffsoperationen benutzt werden, ohne daß irgendwelche Kenntnisse über die konkrete Implementation vorliegen müßten!
- Durch die Datenabstraktion wird das „was” vom „wie” getrennt.
- Die Zugriffsoperationen können von den Anforderungen der Anwendung her festgelegt werden, ohne sich Gedanken über die Struktur des internen Gedächtnisses (z.B. Datei-, Satz- und Datenstrukturaufbau) zu machen.
- Änderungen an einer Zugriffsoperation haben im allgemeinen keine Auswirkungen auf andere Zugriffsoperationen, während ohne Datenabstraktion eine Änderung der Datenstruktur unter Umständen Auswirkungen auf alle zugreifenden Anweisungen haben können.
  Je weniger Daten in der Parameterliste einer Zugriffsoperation übertragen werden, desto änderungsfreundlicher ist eine Datenabstraktion; desto mehr zu implementierende Zugriffsoperationen erhält man dadurch aber auch.

### 1.2.2. Abstrakte Datentypen

Bei den einfachen Datenabstraktionen bewirkt die Beschreibung der Zugriffsoperationen und die Beschreibung der Implementierung die Erzeugung eines Exemplars der beschriebenen Datenabstraktion; d.h. die Vereinbarung einer Datenabstraktion entspricht der Vereinbarung eines Datentyps inklusive der darauf zugelassenen Operationen.

Eine erste Verallgemeinerung der einfachen Datenabstraktion liegt vor, wenn man analog zur Definition von Datentypen die Definition von Datenabstraktionstypen ermöglicht; d.h. Typdefinition und Variablenvereinbarung werden voneinander getrennt. Dadurch können beliebig viele Exemplare eines ADT erzeugt werden. Jedes Exemplar besitzt dann sein eigenes internes Gedächtnis.

Ein *abstrakter Datentyp* (ADT) definiert eine Klasse von abstrakten Objekten, die vollständig durch die auf diesen Objekten verfügbaren Operationen charakterisiert sind. Das bedeutet, daß ein abstrakter Datentyp durch Definition der charakteristischen Operationen durch diesen Typ definiert werden kann.

Die Definition läßt allerdings offen, was die charakteristischen Operationen für einen Typ sind. Ebenso ist unklar, wann ein Typ vollständig charakterisiert ist. Diese Fragen betreffen im wesentlichen die Semantik einer Datenabstraktion.

Abstrakte Datentypen ermöglichen also die Definition von von Datenabstraktionen als Typen. Von diesen Datenabstraktionstypen müssen erst Variable vereinbart werden, damit eine Datenabstraktion existiert. Von ADTs können mehrere Exemplare angelegt werden.

Eine weitere Verallgemeinerung erreicht man durch Parametrisierung von ADTs: abstrakte Datentypen können mit formalen Parametern versehen werden, und solche Parameter können selbst wiederum abstrakte Datentypen sein. Die praktische Anwendung solcher Abstraktionskonzepte hängt dabei weitgehend von der Unterstützung durch geeignete Sprachkonzepte ab.

## Beispiel: Ein generischer STACK-Typ

Eine Definition eines generischen Stack-Typs[7] wird nachstehend vorgestellt. Der hierbei verwendete Begriff *template* bedeutet „Schablone” und wird bei C++ als Schlüsselwort verwendet.

```
type data abstraction STACKTYP;

{ Vereinbarung der Exemplarvariablen / internes Gedächtnis }

const N = 100;

var   S : array[ 1 .. N ] of <TEMPLATE_TYPE>;

  DEPTH : integer;


{ Vereinbarung der Zugriffsprozeduren }

  procedure PUSH(in: Z: <TEMPLATE_TYPE>; out F: boolean);

  procedure POP ( out F: boolean);

  procedure TOP ( out Z: <TEMPLATE_TYPE>, out F: boolean);


{ Implementierung der Zugriffsprozeduren }

  procedure PUSH(in: Z: <TEMPLATE_TYPE>; out F: boolean);

  begin

  F := false;

  if DEPTH >= N then

      F := true

  else begin

      DEPTH := DEPTH + 1;

      S[DEPTH] := Z
```

[7] vgl. hierzu auch das Beispiel 11.2.1, das einen generischen Stack in C++ umsetzt.

```
    end
end;


procedure POP(out F: boolean);
begin
F := false;
if DEPTH >= 1 then
    DEPTH := DEPTH - 1
else
    F := true
end;
```

```
  procedure TOP(out Z: <TEMPLATE_TYPE>, out F: boolean);
  begin
  F := false;
  if DEPTH >= 1 then
      Z := S[DEPTH]
  else
      F := true
  end;

  procedure init;
  begin
  DEPTH := 0
  end;

end STACK;
```

Eine konkrete Variablenvereinbarung (Deklaration):

```
var S1, S2, S3 : STACKTYP;
```

Und das Absenden entsprechender Botschaften an die Objekte:

```
S1.PUSH (34); S2.POP(F);  oder  PUSH (S3, 68); TOP(S1, element, F).
```

## 1.3. Was ist Objektorientierte Programmierung?

Objektorientierte Programmiersprachen wie C++ sind im wesentlichen durch drei Eigenschaften gekennzeichnet: unsichtbare Daten (Objekt/Klasse), eine Hierarchie von Objektdefinitionen / Klassen (Vererbung) und Polymorphie. Im nachfolgenden werden einige Erläuterungen, im wesentlichen unabhängig von der konkreten Programmiersprache C++, gegeben.

### 1.3.1. Klassen und Objekte

Objekte bestehen aus Daten und den zugehörigen Vorschriften, wie diese zu verändern sind. Daten und Vorschriften sind in einer Einheit (untrennbar) verschmolzen.
Objekte reagieren auf an sie gerichtete Botschaften und verändern ihren inneren

Zustand nach Vorschriften in Form von Programmcodes, die Methoden genannt werden.

Objektorientiert Programmieren bedeutet, *Botschaften* (*messages*) an Objekte zu senden. Sie entsprechen Prozedur- bzw. Funktionsaufrufen in anderen Programmiersprachen. Eine Botschaft wird als Aufforderung an ein Objekt aufgefaßt, eine seiner Operationen auszuführen. Diese wird in Eigenverantwortung des Objektes ausgeführt. Datenverarbeitung wird als innere Fähigkeit der Objekte angesehen, welche die gewünschte Aufgabe nach ihnen vorgegebenen Methoden erledigen. Der direkte Zugriff von außen auf Datenelemente eines Objekts ist nicht notwendig - und üblicherweise auch nicht erlaubt. Die Menge der anwendbaren Botschaften eines Objektes heißt seine *Schnittstelle* oder sein *Methoden-Interface*.

Objekte sind also das, was man sonst als abstrakte Datenobjekte bezeichnet. Die Idee hinter der Datenabstraktion ist die, daß ein Objekt nur über Operationen verändert wird, daß seine Internstruktur vor allen Verwendern dieses Objekts aber verborgen wird. Dieses Prinzip wird *Datenkapselung* oder *Geheimhaltungsprinzip* (*Information Hiding*) genannt. Eine Konsequenz dieses Prinzips ist, daß bestimmte Informationen lokal gehalten werden, sich damit nicht über ein ganzes Softwaresystem verteilen und dieses Softwaresystem damit übersichtlich und gut veränderbar bleibt.

Daten und Methoden (Unterprogramme) werden untrennbar voneinander in Objekten gekapselt. Die Methoden greifen auf Datenelemente zu, die den Zustand des Objektes speichern. Da Objekte aber immer als Repräsentanten, als sogenannte Exemplare eines Objekttyps, vorkommen, sprechen wir bei diesen Daten von *Exemplarvariablen* oder*Instanzvariablen*. Ein konkretes Objekt wird auch als *Instanz* bezeichnet.

Gleichartige Objekte werden zu sogenannten *Klassen* zusammengefaßt. Eine Klasse enthält alle Informationen, die notwendig sind, um Objekte dieser Klasse zu beschreiben, zu erzeugen und zu benutzen, ähnlich wie Typ-Definitionen in konventionellen prozeduralen Sprachen. Instanzen einer Klasse hören auf dieselben Botschaften, haben also dieselbe Funktionalität und die gleiche Struktur des Datenspeichers. Sie unterscheiden sich nur im Inhalt der konkreten Exemplarvariablen: die Instanzen kunde1 und kunde2 haben denselben internen Aufbau und kennen dieselben Methoden, die eine Instanz verwaltet jedoch den Kunden Meier, die andere die Kundin Müller.

Zusätzlich zu den Instanzvariablen können Objekte noch Klassenvariable benutzen. Klassenvariable sind globale Variable, die für alle Instanzen einer Klasse zur Verfügung stehen und nur einmal in der Klasse selbst vorhanden sind: beispielsweise kann eine Klassenvariable zaehler mitprotokollieren, wieviele Instanzen von einer bestimmten Klasse jeweils existieren.

Bei den Methoden einer Klasse unterscheiden wir zwischen *Instanzmethoden* und *Klassenmethoden*. Instanzmethoden werden an die Instanzen der Klasse gebunden, Klassenmethoden an die Klasse selbst: so kann die oben erwähnte Klassenvariable zaehler z. B. von zwei Klassenmethoden InkrementZaehler() und DekrementZaehler()

aktualisiert werden. Dagegen ist Ausgabe() i. a. eine Instanzmethode, die die konkreten Daten einer bestimmten Instanz auf den Bildschirm schreibt.

*Abstrakte Klassen* sind Klassen, die nur dazu da sind, ihre Eigenschaften zu vererben. Von ihnen existieren daher keine Instanzen. So kann eine abstrakte Klasse GraphikObjekt gebildet werden, von der später Kreise, Rechtecke und Dodekaeder abgeleitet werden. In der Klasse GraphikObjekt finden sich dann z. B. die für alle Graphikklassen erwünschten Vereinbarungen von Schnittstellen, etwa für eine Methode Print() zur Druckerausgabe. Dabei kann natürlich nicht der Code für diese Methode(n) in der Klasse GraphikObjekt abgelegt werden, sondern i.a. nur die dann für alle vererbten Klassen verbindliche Vorgabe, welche Schnittstelle eine solche Methode Print() besitzen wird.

Die fast schon klassisch zu nennende objektorientierte Programmiersprache *Smalltalk* beinhaltet über einhundert bereits installierte Systemklassen, die jederzeit in Anwendungen eingesetzt werden können. Im Bereich von C++ sind die von den Compilern bereits mitgelieferten Klassen häufig etwas sparsamer ausgestattet. Hier ist es üblich, professionelle Programme unter Einsatz eigens hinzugekaufter *Klassenbibliotheken* zu implementieren[8)].

Um sich einen Überblick über die Klassen zu verschaffen, gibt es bei den modernen OOP-Entwicklungssystemen ein mächtiges Werkzeug: den *Class Hierarchy Browser*[9)]. Wie bereits erläutert, sind Objekte abstrakte Datenobjekte, d.h. Objekte, die ausschließlich über Zugriffsoperationen veränderbar sind. Klassen dienen ausschließlich dazu, solche Objekte zu beschreiben. Eine Klasse ist also eine Schablone (*template*) zur Erzeugung solcher abstrakter Datenobjekte und ferner eine Beschreibung ihrer Veränderung, d.h. eine Klasse ist ein abstrakter Datentyp.

## 1.3.2. Vererbung

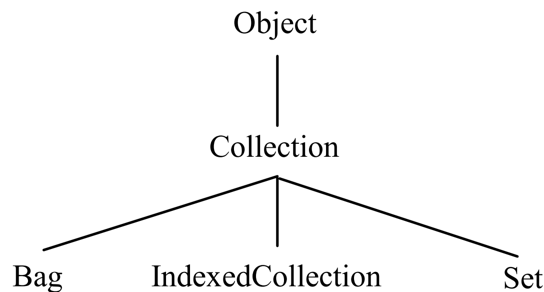
Objektorientierte Systeme kennen eine Vielzahl von vordefinierten Klassen, und nahezu jede Anwendung fügt neue Klassen hinzu. Diese Klassen sind jedoch nicht unabhängig voneinander angeordnet, sondern werden über eine Rangordnung zueinander in Beziehung gesetzt. Dieses Schema, Klassenhierarchie oder Taxonomie genannt, geht vom Allgemeinen zum Speziellen und bringt Ordnung in das Klassengefüge. Klassen, die hierarchisch unter einer Klasse angesiedelt sind, heißen Unterklassen oder Subklassen dieser Klasse. Eine Klasse, die hierarchisch höher liegt, heißtOberklasse oder Superklasse.

Beispiel:

---

[8)] Allerdings bieten moderne C++-Compiler heute ebenfalls weit über den ANSI-Standard hinausgehende Klassenbibliotheken an: Borland C++ die *Object Windows Library (OWL)*, Microsoft Visual C++ oder Symantec C++ die *Microsoft Foundation Classes (MFC*). In der derzeit neuesten Version des Borland Compiler-Systems ist allerdings auch die MFC enthalten.

[9)] Zu Beginn von Kapitel 8 wird beispielhaft ein Bildschirmschnappschuß des Class Editors von Symantec C++ 7.21 gezeigt.




Mit dieser Beziehung ist eine bestimmte Semantik verbunden: Die Unterklasse erbt alle Eigenschaften der jeweiligen Oberklasse, sofern diese Eigenschaften nicht neu definiert werden. Man drückt dies auch manchmal so aus, daß eine Klasse ihre Eigenschaften an die Unterklassen vererbt. Somit besitzen die Objekte der Klasse Set die Eigenschaften von Collection, darüber hinaus aber auch die zusätzlichen in Set definierten Eigenschaften. Man sagt dann auch, die Klasse Set ist eine Spezialisierung von Collection. Da die Eigenschaft, wie Objekte selbst aufgebaut sind, nur klasseninternen Charakter haben[10)], betrifft die Vererbung nur die Schnittstelle einer Klasse, nämlich die anwendbaren Methoden. Somit ist eine Methode der Klasse Collection auch für Objekte der Klasse Set anwendbar, sofern Set diese Methode nicht neu definiert hat, d.h. eine Methode mit dem gleichen Botschaftsmuster definiert wurde. Dies bezeichnet man auch als *Überladen* (*overloading*) einer Methode.

Diese Vererbungsbeziehung ist nun transitiv: die Objekte von Set haben nicht nur die Eigenschaften von Collection, sondern auch von Object. Der Vererbungsbaum hat die Klasse Object als Wurzel, die alle Methoden enthält, die auf alle Objekte, welcher Klasse auch immer, anwendbar sind. Die Klassen auf einem Pfad des Baumes nennt man *Vererbungskette*. Somit sind die auf ein Objekt der Klasse Set anwendbaren Methoden diejenigen der Vererbungskette einschließlich Object mit Ausnahme der Methoden, die neu definiert wurden.

Die Vererbung der Eigenschaften einer Klasse auf ihre Unterklassen hat folgende Konsequenzen:

a) Der gesamte Datenspeicher, d.h. alle *Instanz-* oder *Exemplarvariablen*, werden übernommen. Eine Unterklasse wird vielleicht noch neue Exemplarvariablen hinzufügen, um das Verhalten ihrer Exemplare zu spezialisieren.
   Es ist aber generell nicht möglich, Exemplarvariable wegzulassen.

---

10) Man kann sich dies konkret so vorstellen, daß die Verwaltung von Meßwerten in C++ sowohl ein Array von float-Werten wie eine dynamische Liste von double-Werten verwendet werden kann. Dieses Implementierungsdetail ist aber (meist) insofern unwichtig, da die entsprechende Klasse in beiden Fällen zum Beispiel eine Methode *GetGroesstenMesswert()* bereitstellen kann, die in float-Darstellung den größten gespeicherten Wert zurückliefert.

b) Alle Methoden der Ober- oder Elternklassen können von Objekten einer Unterklasse verwendet werden[11]. Jedes Objekt ist Instanz genau einer bestimmten Klasse, was seine interne Struktur und seine anwendbaren Methoden angeht. Die anwendbaren Methoden sind die der entsprechenden Klasse sowie des entsprechenden Vererbungspfades. Das Objekt der Klasse hat die Exemplarvariablen, die in der Klassendefinition angegeben sind. Dadurch, daß die Methoden des Vererbungspfades anwendbar sind, die wiederum auf Komponenten übergeordneter Klassen wirken, hat dieses Objekt aber implizit auch die Komponenten, die in den übergeordneten Klassendefinitionen enthalten sind. Diese übergeordneten Komponenten sind jedoch nicht zugreifbar, d.h. können somit nicht über eine Zuweisung direkt verändert werden, sondern nur dadurch, daß die Methoden der Klassen des Vererbungspfades aktiviert werden.

Vererbte Methoden existieren nur einmal: Alle Unterklassen greifen auf denselben in der Elternklasse vorhandenen Code zu. Wenn eine Instanz einer Klasse eine Mitteilung nicht verarbeiten kann, wird sie an die entsprechende Oberklasse geschickt. Kann auch diese die Mitteilung nicht verarbeiten, wird sie weiter geschickt. Dies geschieht so lange, bis die Mitteilung verstanden wird oder es in der höchsten Klasse zu einem Fehler kommt.

Einfache Vererbung bedeutet, daß Klassen nur Unterklassen einer einzelnen Klasse sein können. Mehrfache Vererbung (*multiple inheritance*) bedeutet, daß eine Klasse mehrere Oberklassen haben kann. Sie wird vom derzeit aktuellen ANSI-C++-Standard und den entsprechenden Compilern unterstützt.

Vererbung ist ein wichtiges Mittel, um bereits vorhandene Objekte und Methoden zu benutzen und damit Redundanz von Daten und Code zu vermeiden. Der Programmierer kann neue Klassen erzeugen, indem er die Unterschiede zu bestehenden Klassen beschreibt. Er muß nicht jedesmal von Grund auf neu beginnen. Der Implementierungsaufwand wird dadurch stark reduziert. Viele schon vorhande Teile können wiederbenutzt werden (*reusability*). Der Programmcode für eine bestimmte Aufgabe befindet sich an nur einem Platz. Dies erleichtert die Software-Pflege von komplexen Systemen.

### 1.3.3. Polymorphismus

Der Ausdruck *Polymorphismus* (oder *Polymorphie*) kommt aus dem Griechischen und bedeutet soviel wie „*viele Formen (habend)*".

Polymorphismus besagt einfach, daß mehr als eine Methode den gleichen Namen aufweisen kann. Die Methoden müssen zu unterschiedlichen Objekttypen gehören.

Polymorphismus ist ein leistungsstarkes Werkzeug bei der Codestrukturierung. In Verbindung mit der Vererbung ermöglicht die Polymorphie eine signifikante Teilung

---

[11] Bei C++ müssen solche Methoden im sogenannten public- oder protected-Schutzbereich definiert werden.

und Wiederverwendbarkeit des Codes. Sie wird durch einen Mechanismus erreicht, der *späte Bindung* (*late binding*) genannt wird. Späte oder auch *dynamische Bindung* bedeutet, daß ein Objekt zur Laufzeit entscheidet, welche Methode ausgeführt, d.h. ob eine Methode der unmittelbaren Klasse des Objekts oder eine Methode der Oberklasse ausgeführt wird. Es besteht eine dynamische Bindung von der Mitteilung zur Methode.

Dagegen steht die *frühe* oder *statische Bindung*. Hier wird schon zur Übersetzungszeit des Programms festgelegt, welche Methode aufgerufen wird. Zur Laufzeit kann dann keine Veränderung des Aufrufs mehr vorgenommen werden.

### 1.3.4. Charakterisierung der Objektorientiertheit

Objektorientierte Vorgehensweise ist also: das Konzept der *Abstrakten Datentypen* (*ADT*), u.a. Kapselung der Daten und Information Hiding, wird erweitert um Vererbung, dynamische Bindung, Methoden und Mitteilungsverbindungen, die Methoden auslösen.

Bei der Entwicklung von komplexeren Softwaresystemen als strukturierten Sammlungen von Implementierungen derartiger abstrakter Datentypen haben sowohl das Ziel der Wiederbenutzbarkeit als auch der Gedanke der „Software-Bausteine” (Verlagerung von Leistungen in die einzelnen Bauelemente) einen entscheidenden Einfluß.

## 1.4. Von C zu C++

Kritik erhebt sich an C++ häufig von „objektorientierten Dogmatikern”, z.B. Programmierern aus der Smalltalk-Ecke, daran, daß ein C++-Programm zunächst einmal auch ein C-Programm sein kann. Mit anderen Worten: man kann sehr pragmatisch „langsam” in C++ hineinwachsen, während man lange Zeit noch vieles im Stil des guten alten C, also prozedural, programmiert.

Dabei erfordert C++ als objektorientierte Sprache eigentlich ein ganz neues Denken, ein prinzipiell andersartiges Vorgehen als das der konventionellen strukturierten und prozeduralen Programmierweise. Gleichzeitig rekrutieren sich heutzutage die meisten C++-Programmierer aus Kreisen ehemaliger oder noch aktuell in C programmierender Personen, so daß es nur natürlich erscheint, zunächst einmal die sanften Übergänge von C zu C++ darzustellen. In späteren Kapiteln soll dann jedoch auch das objektorientierte Design im Vordergrund stehen.

Zunächst einmal bietet C++ neben den in C üblichen Kommentaren der Form /* ... */ auch die Möglichkeit, Zeilenkommentare zu schreiben; ab einem doppelten Schrägstrich // wird der Rest der betreffenden Zeile als Kommentar verstanden und vom C++-Compiler ignoriert (bzw. durch den Präprozessor eliminiert).

```
int i=1;        // Dies ist ein Kommentar zur Definition der Variablen i.
```

Desweiteren können Deklarationen bzw. Definitionen[12)] von (z.B.) Variablen überall vorkommen, nicht nur zu Beginn eines Blockes. Und selbst innerhalb der for-Schleifenklausel kann eine Variable vereinbart werden.[13)]

---

[12)] Zur Erinnerung: unter einer Deklaration versteht man das namentliche Bekanntmachen eines Bezeichners; die Definition einer Variablen ist der Ort, an dem für diese auch Speicherplatz angelegt wird.

[13)] Die bisherigen C++ Implementationen geben dieser Variablen `i` eine Lebensdauer bis zum Ende des Blockes, in dem die for-Schleife plaziert ist. Nach dem neuen ANSI/ISO-Standard ist die Lebensdauer jedoch auf den Block der for-Schleife selbst reduziert worden.

```
int main()
{
  for (int i=0; i<=100; i++)
  {
    printf("\nDas Quadrat von %3d ist ",i);
    int quadrat=i*i;    // Definition inmitten eines Blockes!
    printf("%5d\n",quadrat);
  } // end for i
  return EXIT_SUCCESS;  // Rückgabe einer Fehlerkennung
} // end main          // (sogenannter Exit-Code)
```

Funktionen können in C++ Default-Parameter haben, also initialisiert sein:

```
int AnyFunc(int a=5);
```

Wird die Funktion `AnyFunc()` mit einem `int`-Parameter aufgerufen, so wird `a` auf den übergebenen Wert gesetzt; wird die Funktion `AnyFunc()` jedoch ohne Parameter aufgerufen, so wird implizit `a` mit dem Wert `5` initialisiert.

Funktionen können überladen werden: die folgenden beiden Prototypdeklarationen sind in C++ legal, nicht jedoch innerhalb eines C-Programms.

```
void Ausgabe(int a=0);
void Ausgabe(float x);
```

Die beiden Aufrufe

```
Ausgabe(123);
Ausgabe(1.23);
```

sind somit beide legal und betreffen die jeweils „passende" Funktion. Zum Überladen von Funktionen sei auch auf die Übung 2 auf Seite 0 und die Übung 3 auf Seite 0 hingewiesen.

Während in C das *Prototypisieren*[14)] optional ist und nur einen guten Software-Stil kennzeichnet, ist es in C++ zwingende Verpflichtung! C++ prüft die Einhaltung (z.B.) von Datentypen wesentlich strenger als C. So ist beispielsweise `'A'` etwas vom Datentyp `char`, während dies in C noch vom Typ `int` ist! Dies sieht man daran, was

---

14) Damit ist das ordnungsgemäße Schreiben von Funktionsprototypen gemeint.

der `sizeof`-Operator in C bzw. C++ dazu sagt: `sizeof('A')` ist in C und C++ nicht dasselbe!

Die beiden Funktionsdeklarationen

```
/* 1 */   int f();

/* 2 */   int f(void);
```

sind in C++ identisch; in ANSI-C bedeutet die erste Variante, daß `f()` irgendeine beliebige Parameterliste beim aktuellen Aufruf erhalten kann, während die zweite Form definitiv sagt, daß `f()` keine Parameter besitzt.

Konstanten, also Daten mit dem Schlüsselwort `const`, sind in C++ per Voreinstellung statisch, also nur für das betreffende Modul zugreifbar, solange nicht explizit extern deklariert wird. Bei C sind `const`-Definitionen per default extern.

Die Identifier (Bezeichner) in C++ dürfen beliebig lang sein. Bei Funktionen, die wie bereits gesagt, überladen werden dürfen, wird ein sogenanntes *Name Mangling* durchgeführt. Eine C++-Funktion `int f(char *,int)` kann so beispielsweise einen compilerinternen Namen der Form `f__FPci_i` erhalten. Dieses Name Mangling ist erforderlich, damit der Linker die o.g. Funktion `f()` von einer weiteren Funktion namens `f()` mit einer abweichenden Parameterliste unterscheiden kann.

### 1.4.1. Übersicht über die reservierten Worte (Schlüsselwörter)

Für das weitere Arbeiten mit C++ werden nachstehend sämtliche Schlüsselwörter aufgeführt; diese Begriffe dürfen nicht als eigene Bezeichner herangezogen werden. Dabei sind gerade mal fünfzehn der 48 Schlüsselwörter gegenüber ANSI-C neu in C++ aufgenommen worden[15].

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| asm | auto | break | case | catch | char | class | const |
| continue | default | delete | do | double | else | enum | extern |
| float | for | friend | goto | if | inline | int | long |
| new | operator | private | protected | public | register | return | short |
| signed | sizeof | static | struct | switch | template | this | throw |
| try | typedef | union | unsigned | virtual | void | volatile | while |

[15] Für eine komplette Liste aller im ANSI/ISO-Draft vorgesehenen Schlüsselwörter wird auf Anhang B.3 verwiesen.

## 1.4.2. Übersichtstabelle: Operatoren und deren Prioritäten

Auch wenn zum momentanen Zeitpunkt im wesentlichen nur die aus C stammenden Operatoren bekannt sind, so soll dennoch schon hier für den weiteren Gebrauch eine Übersichtstabelle über die Operatoren in C++ und deren Prioritäten abgedruckt werden.

| Priorität | Operator | Erläuterung | Syntaxbeispiel |
|---|---|---|---|
| 17 | `::` | Scope Resolution | class_name::member |
| | `::` | global | ::name |
| 16 | `,` | Mitgliedsspezifizierer | object.member |
| | `->` | Mitgliedsspezifizierer | objectpointer->member |
| | `[]` | Indizierung | pointer[expression] |
| | `()` | Funktionsaufruf | expression(expression-list) |
| | `()` | Wertkonstruktion | type(expression-list) |
| | `sizeof` | Größe eines Objektes | sizeof expression |
| | `sizeof` | Größe eines Typs | sizeof(type) |
| 15 | `++ --` | pre/post in/decrement | lvalue++ lvalue-- ++lvalue --lvalue |
| | `~` | binäres Komplement | ~expression |
| | `!` | Negation | !expression |
| | `+ -` | unäres Plus/Minus | +expression -expression |
| | `&` | Adresse von | &lvalue |
| | `*` | Dereferenzierung | *expression |
| | `new` | Allokierung | new type |
| | `delete` | Deallokierung | delete pointer |
| | `delete[]` | Deallokierung (Array) | delete [] pointer |
| | `()` | Cast(ing) | (type) expression |
| 14 | `.*` | Mitgliedszugriff | object.*pointer-to-member |
| | `->*` | Mitgliedszugriff | pointer->*pointer |
| 13 | `* / %` | Multiplikation, Division | expression * expression (etc). |
| 12 | `+ -` | Addition, Subtraktion | expression±expression |
| 11 | `<< >>` | Links/Rechts-Shift | expression>>expression |
| 10 | `< <=`<br>`> >=` | Vergleichsoperatoren | expression > expression |
| 9 | `==`<br>`!=` | gleich<br>ungleich | expression==expression<br>expression!=expression |

<table>
<tr><td>8</td><td>&</td><td>bitweises Und</td><td>expression & expression</td></tr>
<tr><td>7</td><td>^</td><td>bitweises XOR</td><td>expression ^ expression</td></tr>
<tr><td>6</td><td>|</td><td>bitweises Oder</td><td>expression | expression</td></tr>
<tr><td>5</td><td>&&</td><td>logisches Und</td><td>expression && expression</td></tr>
<tr><td>4</td><td>||</td><td>logisches Oder</td><td>expression || expression</td></tr>
<tr><td>3</td><td>? :</td><td>Bedingungsoperator</td><td>expression ? expression : expression</td></tr>
<tr><td>2</td><td>=<br>*= /= %=<br>+= -=<br><<= >>=<br>&= ^= |=</td><td>Zuweisung<br>Kombinierte Zuweisungsoperatoren</td><td>lvalue = expression<br>lvalue *= expression (etc.)</td></tr>
<tr><td>1</td><td>,</td><td>Komma-Operator</td><td>expression, expression</td></tr>
</table>

## 1.5. Erste Programme

Sehen wir uns das nachfolgende kleine Beispielprogramm `simple1.cpp` an. Hier wird von der neuen Form der Zeilenkommentare Gebrauch gemacht; außerdem wird eine erste Realisierung von Überlagerung an einem einfachen Beispiel von zwei Funktionen namens `Ausgabe()` gezeigt , die anhand ihrer Parameterliste identifiziert werden. Beachten Sie bitte (noch einmal), daß es in C innerhalb eines Moduls keine zwei verschiedenen Funktionen desselben Namens geben kann!
Schließlich wird der Operator `<<` vorgestellt, der die Ausgabe auf stdout in C++ realisiert. Dafür wird die Headerdatei `iostream.h` eingebunden. Streams werden in einem späteren Kapitel ausführlicher behandelt.

```
// simple1.cpp ---------------------------------------------------



// Diese Form von Kommentaren ist in C++ möglich: ein sogenannter

// Zeilenkommentar, d.h. ab den //-Zeichen ist alles bis zum

// Zeilenende Kommentar.



// Header-Dateien einbinden .....................................

#include <iostream.h>

#include <stdlib.h>

// Prototypen ...................................................

void Ausgabe(int i);          // Ausgabe für int's
```

```
void Ausgabe(char c);          // Ausgabe für char's


// Implementation ...............................................

void Ausgabe(int i)

{

   cout << "Ausgabe(int): " << i << endl;

} // end Ausgabe(int)
```

```
void Ausgabe(char c)
{
   cout << "Ausgabe(char): " << c << endl;
} // end Ausgabe(char)


int main()
{
   int zahl=1;
   Ausgabe(zahl);


   char zeichen='A';
   Ausgabe(zeichen);


   return EXIT_SUCCESS;
} // end main
// end of file simple1.cpp --------------------------------------
```

Das Ablauflisting dieses Programms ist erwartungsgemäß einfach:

```
Ausgabe(int): 1
Ausgabe(char): A
```

Im folgenden Programm `simple2.cpp` wird mit dem aus ANSI-C bekannten Schlüsselwort `const` gespielt. Das Programm zeigt m. E. sehr gut, daß damit mehr ein Wunsch ausgedrückt wird, einen Speicherplatz read-only verwenden zu wollen, und daß dies kein vollkommen effektiver Schutz vor Veränderung sein kann. In dem Programm wird auch eine *Referenz* deklariert. Zum Trost für alle die, denen das jetzt zu schnell geht: Referenzen sind neu in C++ und werden in Abschnitt 3.3 detaillierter vorgestellt.

```
// simple2.cpp --------------------------------------------------


// Header-Dateien einbinden .....................................
#include <iostream.h>
```

```
#include <stdlib.h>


// Implementation ..............................................

int main()

{

   cout << "Betrachtungen zum Schlüsselwort const" << endl;

   char * const pc1 = "Konstanter Zeiger auf char";

   const char * pc2 = "Zeiger auf konstante chars";

   // char * pc3=pc2;   // unzulaessige Konvertierung von const

                  // char* zu char*

   char * pc4=(char *)pc2;     // über Casting geht es aber doch

   char* & rpc4=pc4;     // rpc4 ist eine Referenz auf pc4, d.h.

                  // ein Aliasname für pc4


   // pc1++;           // pc1 darf nicht geändert werden,

   pc1[0]='k';           // aber das, worauf es zeigt schon


   // pc2[0]='z';      // pc2 zeigt auf konstante chars!

   *(char *)pc2 = 'z';         // "schmutziger" Trick durch Casting


   pc4[11]='c';          // hier wird - durch den Umweg über pc4

                  // bzw. rpc4 - die Zeichenkette, auf

   rpc4[21]='C';         // die pc2 zeigt, verändert


   cout << "pc1: " << pc1 << endl;

   cout << "pc2: " << pc2 << endl;

   return EXIT_SUCCESS;

} // end main
```

Das Ablauflisting dieses kleinen Programms sieht wie folgt aus:

```
Betrachtungen zum Schlüsselwort const

pc1: konstanter Zeiger auf char

pc2: zeiger auf constante Chars
```

## 1.6. Übersicht: Geschichtliche Entwicklung

Aus Stroustrups Buch „The Design and Evolution of C++” (sh. Literaturverzeichnis) soll nachstehend ein Auszug aus der *C++ Timeline* abgedruckt werden.

| | | |
|---|---|---|
| 1979 | Mai | Beginn von Stroustrups Arbeit an „C with Classes” |
| | Oktober | Erste Implementation von „C with Classes” |
| 1982 | Januar | Erste Publikationen zu „C with Classes” |
| 1983 | August | Erste intern bei AT&T verwendete Implementation von C++ |
| 1985 | Februar | Erstes externes C++ Release |
| | Oktober | Cfront Release 1.0 (erste kommerzielle Version) |
| | Oktober | Erstausgabe von „*The C++ Programming Language*” |
| 1986 | November | Erste PC-Version von Glockenspiels Cfront |
| 1987 | Dezember | Erstes GNU C++ Release (1.13) |
| 1988 | Juni | Erstes Zortech C++ Release (PC, UNIX) |
| 1989 | Dezember | Erste Treffen des ANSI-Komitees |
| 1990 | Mai | Erstes Borland C++ Release |
| | Juli | Templates implementiert |
| | November | Exceptions implementiert |
| 1992 | Februar | Erstes DEC C++ Release |
| | März | Erstes Microsoft C++ Release |
| | Mai | Erstes IBM C++ Release (inkl. Exceptions und Templates) |
| 1993 | März | Run-Time-Type-Information (RTTI) definiert |
| 1995 | April | Erste Draft-Version des ANSI-C++-Standards verfügbar |

# 2. Das Klassenkonzept

Die zentralen Begriffe der objektorientierten Programmierung sind Klasse, Vererbung und Polymorphie. Beginnen wir langsam und greifen den ersten dieser Begriffe auf.

## 2.1. Klasse und Instanz

Auf dem Hintergrund der prozeduralen Programmierweise in C oder Pascal ist der Begriff des Datentyps bekannt: ein *Datentyp* ist eine Menge von Werten und eine Menge von Operationen auf der Wertemenge. So ist in C `int` ein solcher einfacher Datentyp, für den bereits fertige Operationen wie die Addition oder die Bildschirmausgabe mit jedem Compiler mitgeliefert werden.

In Erweiterung dieses Begriffs gibt es verschiedenste Erläuterungen dafür, was ein abstrakter Datentyp ist; wir wollen uns hier mit der folgenden begnügen: Ein *abstrakter Datentyp* ist ein Datentyp, dessen Schnittstelle von seiner Implementierung getrennt ist[16)]. Zur Definition des Wortes *Klasse* findet sich bei Gerike (siehe Literaturverzeichnis) der Satz: *Klassen sind in einer Programmiersprache formulierte Beschreibungen abstrakter Datentypen*.

Konkret bedeutet das: eine Klasse in C++ ist eine Zusammenfassung von Datenelementen und Funktionen (sogenannten *Methoden*[17)]). Während eine C-Struktur nur Datenelemente beinhaltet, werden bei C++ im Sinne der objektorientierten Sichtweise auch die inhaltlich zugehörigen Bearbeitungsfunktionen mit in die Klasse aufgenommen.

Als „Dogma" formuliert: auf die Datenelemente eines konkreten Klassenobjektes wird in einer sauber objektorientierten Programmierung ausschließlich über die Methoden, die Schnittstellen zugegriffen. Diese Methoden kontrollieren den Zugriff und stellen beispielsweise sicher, daß in einem Objekt, das einen mathematischen Bruch darstellt, der Nenner nie 0 sein darf und sein wird.

Sehen wir uns dies im folgenden etwas konkreter an.

---

16) Machen wir ein konkretes Beispiel: arbeiten wir mit Brüchen, so ist in C eine geeignete Datenstruktur für die Implementation struct Bruch { int zaehler, nenner; }; die Schnittstellen hierfür sind nun diejenigen Funktionen, die mit diesen Brüchen operieren, z.B. zwei Brüche addieren usw. Für die Außenwelt, diejenigen, die mit solchen Brüchen arbeiten, ist es dabei (in der Regel) vollkommen unerheblich, wie die Brüche „intern" implementiert sind. So könnte beispielsweise die obige Definition irgendwann einmal abgeändert werden zu struct Bruch { long zaehler, nenner; }; solange die Schnittstellen gleich bleiben, „merkt" die Außenwelt nichts von der Änderung der Implementierung.

17) Wir wollen im folgenden im Zusammenhang mit Klassen die Begriffe *Funktion* und *Methode* synonym verwenden. Der letztere Begriff wird jedoch nur für klasseninterne Funktionen verwendet, insoweit ist *Funktion* eine Art Oberbegriff.

### 2.1.1. Beispiel: Schrittweise von der Struktur zur Klasse

Sollen zu einhundert Dingen (Objekten?) Name und Alter abgespeichert werden, so schreiben wir in C vielleicht folgendes.

```
/* Programmauszug struct1.c */

#define ANZAHL  100

char name[ANZAHL][80];

int  alter[ANZAHL], i;

for (i=0; i<ANZAHL; i++)       /* Einlesen der Daten */

{

  printf("Name: ");

  scanf("%s",name[i]);

  printf("Alter: ");

  scanf("%d",&(alter[i]));

} /* end for i */

for (i=0; i<ANZAHL; i++)       /* Ausgeben der Daten */

{

  printf("%s %d\n",name[i],alter[i]);

} /* end for i */
```

Hier sind also die eigentlich zusammengehörenden Daten `name[i]` und `alter[i]` syntaktisch voneinander getrennt; nur aufgrund des kleinen, überschaubaren Codes verstehen wir, daß `name[i]` und `alter[i]` zusammengehören. Ein zweiter Schritt sähe also vielleicht so aus.

```
/* Programmauszug struct2.c */

#define ANZAHL  100

struct INFO {

  char name[80];

  int  alter;

}; // end struct INFO

struct INFO item[ANZAHL];
```

```
for (i=0; i<ANZAHL; i++)       /* Einlesen der Daten */
{
   printf("Name: ");
   scanf("%s",item[i].name);
   printf("Alter: ");
   scanf("%d",&(item[i].alter));
} /* end for i */
for (i=0; i<ANZAHL; i++)       /* Ausgeben der Daten */
{
   printf("%s %d\n",item[i].name,item[i].alter);
} /* end for i */
```

Hier sind die zusammengehörigen Daten in einer Struktur (`struct INFO`) ordentlich gebündelt; die Aus- und Eingabe wurde jedoch „vor Ort" umgesetzt. Wird diese mehrfach benötigt, so müßten die for-Schleifen mehrfach implementiert werden. Es bietet sich somit eine Modularisierung an.

```
/* Programmauszug struct3.c */
#define ANZAHL  100
struct INFO
{
   char name[80];
   int  alter;
}; // end struct INFO
struct INFO item[ANZAHL];


struct INFO Read(void);        /* Prototyp einer Einlesefunktion */
void Write(struct INFO);/* Prototyp einer Ausgabefunktion */


for (i=0; i<ANZAHL; i++)/* Einlesen der Daten */
   item[i] = Read();
```

```
for (i=0; i<ANZAHL; i++) /* Ausgeben der Daten */
  Write(item[i]);
```

Auf die Implementierung der Funktionen `Read()` und `Write()` soll hier aus Platzgründen verzichtet werden.

Nun sind die Daten in der Struktur gekapselt, die Funktionen ausgelagert. Allerdings gehören die Daten und die Funktionen syntaktisch noch nicht zueinander; dies ist nur inhaltlich zu erschließen. Und genau hier setzt die Klassenbildung ein. Wir schreiben also nun eine erste C++ Variante:

```
/* Programm struct4.cpp */
#include <iostream.h>   // Einbinden für C++-gemäße Ein- und Ausgabe
#include <stdlib.h>    // mit cin und cout (s.u.)
const int ANZAHL =100;  // In C++ kann ein const int für eine Array-
                       // deklaration verwendet werden!

class INFO              // Deklaration einer Klasse namens INFO
{
  private:             // Privater Bereich: an diese Daten und
      char name[80];   // Funktionen kommen nur Klassenmitglieder
      int  alter;      // heran.
  public:          // Public-Bereich: offen für alle...
      void Read(); // Prototyp einer Einlesefunktion
      void Write();    // Prototyp einer Ausgabefunktion
}; // end class INFO
void INFO::Read()       // Implementation der Einlesefunktion
{
  cout << "Name:  ";   // Ausgabe auf den Bildschirm
  cin >> name;         // Einlesen von Tastatur
  cout << "Alter: ";
  cin >> alter;
```

```
} // end INFO::Read()



void INFO::Write()     // Implementation der Ausgabefunktion
{
  cout << name << ", " << alter << endl;
} // end INFO::Write()


int main()          // In C++ hat main() stets den Rückgabetyp
{                   // int, daher das return als letzte Anweisung
  INFO item[ANZAHL];  // Deklaration von ANZAHL vielen Objekten
                    // der Klasse INFO
  for (i=0; i<ANZAHL; i++)// Einlesen der Daten
      item[i].Read();
  for (i=0; i<ANZAHL; i++)// Ausgeben der Daten
      item[i].Write();
  return EXIT_SUCCESS;// Rückgabe eines Fehlercodes an das
} // end main          // Betriebssystem
```

In diesem Programm geschieht ablaufmäßig nach außen hin dasselbe wie zuvor. Aber das Design und das Innenleben haben sich gewaltig geändert.

Da es sich hier um ein Programm mit gegenüber C zahlreichen neuen Details handelt, ist `struct4.cpp` vollständig abgedruckt. Zunächst werden die Headerdateien eingebunden, hier also `iostream.h` für die Ein- und Ausgaberoutinen von C++ und `stdlib.h` für die Rückgabe des vordefinierten Wertes `EXIT_SUCCESS` (an das Betriebssystem bzw. den Aufrufer des Programms).

Während in C kein Array mit einer `const int`-Variablen dimensioniert werden kann, geht dies in C++: daher ist dies dem C-gemäßen `#define` vorzuziehen, denn so kann der Compiler auch hierbei (für `ANZAHL`) eine Typprüfung vornehmen.

Statt der Struktur `struct INFO` folgt dann neu mit `class INFO` die Klassendeklaration. Im Gegensatz zu C liegen in C++ die Strukturen und Unions, die `enum`-Deklarationen und die hier vorgestellten Klassen alle in demselben Namensraum

(*namespace*)[18], ein `typedef` für die Klasse entfällt daher: später kann mit `INFO anyinfo;` ein Objekt der Klasse `INFO` deklariert werden.

Technisch kann eine Klasse als Erweiterung der C-Struktur angesehen werden. D.h. innerhalb der `class`-Deklaration folgen im wesentlichen einfach die Einzeldeklarationen der Komponenten. Bei C++ können dies Daten und Funktionen sein, während es beim C-`struct` bekanntlich nur Datenkomponenten gibt. Weiterhin unterscheidet man innerhalb einer Klasse sogenannte Schutzbereiche: `private` bedeutet, daß alle so deklarierten Datenelemente und Methoden nur den Klassenmitgliedern zur Verfügung steht, nach außen hin nicht sichtbar, nicht zugreifbar ist. `public` bedeutet dagegen erwartungsgemäß, daß die ganze Welt darauf Zugriff hat. Einen dritten Schutzbereich (`protected`) lernen wir später kennen.

Unter dem etwas saloppen Motto[19] „*Daten schützen, Funktionen nützen*" heißt das konkret: werden Daten mit `private` gekapselt, so muß naturgemäß mit `public`-Funktionen der Zugriff auf diese geschützten Daten geregelt werden, sonst kann mit dieser Klasse nicht ernsthaft gearbeitet werden.

Syntaktisch werden die Klassenfunktionen nach der Klasssendeklaration (sofern dafür nicht eine eigene Headerdatei angelegt wird) definiert in der Form

```
Rückgabetyp Klassenname::Funktionsname(Parameterliste)
```

Beim konkreten Aufruf (siehe im Hauptprogramm `main()` in unserem obigen Beispiel `struct4.cpp`) wird dann eine solche Funktion an ein Objekt gebunden ausgeführt; der beispielhafte Aufruf

```
item[1].Read();
```

ruft also die Methode `Read()` für das Objekt `item[1]` auf, so daß damit die privaten Datenelemente `name` und `alter` von `item[1]` von Tastatur eingelesen werden können. Objektorientiert wird auch davon gesprochen, daß das Objekt `item[1]` die Botschaft `Read()` erhält.

Insgesamt ist bereits an diesem kleinen Beispiel zu sehen: die Daten und deren Zugriffsfunktionen sind (syntaktisch) so zusammengefaßt, wie sie logisch auch zusammengehören. Dabei sind die Daten privatisiert (gekapselt, geschützt), können also nur

---

[18] Stroustrup stellt in seinem Buch *The Design and Evolution of C++* das Konzept der (benannten und anonymen) Namensräume vor; die benannten Namensräume werden jedoch von den zur Zeit vorhandenen Compilern noch nicht umgesetzt, obschon einige bereits das Schlüsselwort `namespace` kennen. Aus diesem Grund wird an dieser Stelle darauf auch nicht weiter eingegangen.
Der ANSI/ISO C++ Draft (Diskussionsentwurf) von Februar 1995 sieht die Schlüsselworte `namespace` und `using` für diesen Themenkomplex vor.

[19] Der Autor bittet die Leserinnen und Leser, nicht so genau auf die Grammatik dieses Mottos zu sehen.

mittels der klassenintern festgelegten Zugriffsfunktionen im Schutzbereich `public` verändert oder gelesen werden.[20)]

Im übrigen können - wie in C die Strukturen - natürlich auch die Klassen in C++ ineinander geschachtelt definiert werden. Darauf soll an dieser Stelle nicht weiter eingegangen werden; ein Beispiel finden Sie jedoch in 12.2.2 auf Seite 0.

## 2.2. Schnittstelle und Implementation

Wie bereits erwähnt, wird in der Praxis die Schnittstelle einer Klasse (also die Klassendeklaration) in einer Headerdatei (etwa `struct4c.h` im obigen Beispiel) abgelegt; die Implementation wird vom Hauptprogramm und anderen Modulen ebenfalls getrennt und in einer eigenen cpp-Datei (z.B. `struct4c.cpp`) gespeichert. Die Erstellung des Gesamtprojektes (in der Regel über ein Makefile oder im PC-Bereich auch eine Projektdatei gesteuert) besteht dann darin, die Quelltexte `struct4.cpp` (`main`) und `struct4c.cpp` (Klassenimplementation) zu compilieren (wobei die Quelltexte die `struct4c.h`-Headerdatei einbinden) und anschließend die Objectfiles zusammenzubinden zu einem ausführbaren Gesamtprogramm.

Unter UNIX kann das elementare Kommando zur Erzeugung des ausführbaren Gesamtprogramms `struct4` so aussehen:

```
CC struct4.cpp struct4c.cpp -o struct4
```

Bei einer integrierten Entwicklungsumgebung wie z.B. Borland C++ wird man in der Regel für Programme, die in mehrere Quelltexte aufgeteilt vorliegen, eine Projektdatei oder ein Makefile anlegen. Dort wird dann nicht nur verwaltet, welche Dateien mit zu dem Softwareprojekt gehören, sondern sogar weitergehend, welche Dateien logisch von welchen anderen abhängen, welche im Falle einer Änderung in einem Quelltext also neu compiliert werden müssen.

Durch diese Trennung wird das Fundament gelegt für eine spätere, sehr umfassende Wiederverwendbarkeit (*reusability*) des so geschaffenen Quellcodes. In der Praxis werden viele sogenannte Klassenbibliotheken kommerziell angeboten für die verschiedensten Aufgabenstellungen. Mit dem später noch zu behandelnden Mittel der Vererbung kann ein Programmierer dann mit wenigen Zeilen (durch Ableitung einer eigenen Klasse aus einer fertig vorgelegten Klasse mit zahlreichen, bereits brauchbaren Eigenschaften) recht komplex arbeitende Programme erstellen.

Die Anbieter solcher Klassenbibliotheken liefern neben der Headerdatei noch ein oder mehrere Object- (`.o` bei UNIX, `.obj` bei DOS) oder Library-Files (`.a` oder `.sl` bei HP-UX, `.lib` bei DOS, `.lib` oder `.dll` bei MS-Windows) sowie ein Handbuch; der Programmierer ist damit dann gefordert, sich einen Überblick über den Leistungsumfang der bereits definierten Klassen zu verschaffen, damit er sachkundig weitere Klassen konstruieren kann.

---

[20)] Mögliche Abweichungen von dieser Regel werden später dargestellt.

Diese rigide Trennung zwischen Schnittstelle und Implementation führt dazu, daß den Programmierer im Normalfall nicht zu interessieren braucht, *wie* eine Methode der Klassenbibliothek arbeitet, sondern lediglich, welche konkret beschriebene Teilaufgabe von ihr umgesetzt wird. Ebenso kann der Anbieter der Klassenbibliothek jederzeit Änderungen an seinen klasseninternen Methoden nachreichen: solange er die Gesamtfunktionalität und die Schnittstellen seiner Klassen(methoden) nicht ändert, ist auch das Programm vor Ort nicht (störend) davon betroffen.

## 2.3. Schutzbereiche

C++ kennt drei Schutzbereiche in einer Klassendefinition:

- *private*: Elemente, die in diesem Schutzbereich liegen, können nur von innerhalb dieser Klasse referenziert (gelesen, geändert) werden.
  Dies ist der voreingestellte Wert, d.h. wenn nichts anderes in einer Klassendefinition vereinbart wird, sind alle Elemente (Daten und Funktionen) privat, nicht von außen zugänglich.
- *protected*: Dieser Schutzbereich wirkt nach außen wie private; lediglich für Angehörige abgeleiteter Klassen (siehe Kapitel 8) wird der Zugriff auf Elemente dieses Schutzbereiches gewährt.
- *public*: Dieser Schutzbereich ist öffentlich, d.h. auf die in diesem Bereich deklarierten Elemente darf von überall her zugegriffen werden.

Im Beispiel `struct4.cpp` sind die Datenelemente `name` und `alter` als `private` deklariert, die Zugriffsfunktionen `Read()` und `Write()` dagegen waren öffentlich (`public`) vereinbart worden.

Selbstverständlich dürfen in einer praktisch einzusetzenden Klasse nicht alle Daten und Funktionen als `private` vereinbart werden, denn dadurch wäre keinerlei Zugriff auf diese Klasse möglich! In der Regel wird man Datenelemente als `private`, Zugriffsfunktionen als `public` vereinbaren. Lediglich im Zusammenhang mit Vererbung tritt `protected` häufig an die Stelle von `private`.

## 2.4. Inline

Bei der objektorientierten Programmierung in C++ werden, vor allem innerhalb von Klassendefinitionen, mehr als sonst üblich Funktionen geschrieben für häufig sehr kurze Code-Sequenzen. Dadurch entsteht ein Verwaltungsoverhead, der ein C++-Programm langsamer werden läßt.

Zur Steigerung der Effizienz können Klassenfunktionen *inline* formuliert werden: entweder werden Sie bereits in der Klassendeklaration codiert oder aber es wird ihnen das Schlüsselwort `inline` vorangestellt.

Wichtig ist allerdings folgendes: für inline-Funktionen wird kein Stack-Frame bereitgestellt, weshalb rekursive Aufrufe solcher Funktionen nicht möglich sind. Außerdem müssen, im Gegensatz zu herkömmlichen Funktionen, bei einer Änderung einer inline-Funktion sämtliche Aufrufe dieser Funktion verständlicherweise neu compiliert werden!

## 2.4.1. Beispiel: Implizites und explizites `inline`

Nachstehender Auszug aus einer Klassendeklaration und -definition zeigt die beiden Möglichkeiten einer inline-Methode: die Funktion `onefunc()` wurde implizit inline codiert dadurch, daß bereits in der Klassendeklaration die Methode `onefunc()` definiert (d.h. implementiert) wurde. Die Mitgliedsfunktion `anotherfunc()` dagegen ist außerhalb der Klassendeklaration definiert; dort wurde (explizit) das Schlüsselwort `inline` vorangestellt.

```
class ANYCLASS
{
   private:
       // ...
   public:
      int onefunc()  { return(1); } // implizit inline deklariert
      void anotherfunc();
}; // end ANYCLASS

inline void ANYCLASS::anotherfunc()  // explizit als inline deklariert
{
   // ...
} // end inline void ANYCLASS::anotherfunc()
```

## 2.4.2. Beispiel: Code-Generierung mit und ohne `inline`

Daß der durch inline-Methoden generierte Code auch wirklich effizienter ist (oder zumindest sein kann) als der durch „outline" geschriebene Routinen, sollen die nachfolgenden Listings zeigen.

Das Programm `inline1.cpp` ist ein Demonstrationsprogramm, bei dem in der Klasse `DEMO` der Konstruktor und die Methode `DEMO::ausgabe()` außerhalb der Klassendeklaration definiert werden; `inline1.cod` zeigt den durch den Symantec C++ Compiler generierten Assemblercode (für Intel-Prozessoren). Das Programm `inline2.cpp` ist demgegenüber die `inline`-Variante, `inline2.cod` zeigt den entsprechend kürzeren Assemblercode hierzu. Beachten Sie insbesondere auch, daß (naturgemäß) in der `inline`-Version weniger `calls` stattfinden und die erzeugte Code-Größe deutlich differiert!

Der C++-Quelltext ohne `inline`:

```
// ------------------------------------------------------------------
// inline1.cpp: Realisierung ohne inline-Funktionen
// ------------------------------------------------------------------

#include <iostream.h>

class DEMO
{
  private:
     int id;
  public:
     DEMO();
     void ausgabe();
}; // end class DEMO

DEMO::DEMO()
{
  id=100;
} // end DEMO::DEMO()

void DEMO::ausgabe()
{
  cout << "id=" << id << endl;
} // end void DEMO::ausgabe()

void main()
{
```

```
  DEMO aDemoObject;

  aDemoObject.ausgabe();

} // end main
```

Der hieraus erzeugte Assemblercode[21]:

```
; --- inline1.cod --- Assemblercode zu inline1.cpp ---

includelib SDs.lib

_TEXT  segment word use16 public 'CODE'    ;size is 105

_TEXT  ends

_DATA  segment word use16 public 'DATA'    ;size is 4

_DATA  ends

CONST  segment word use16 public 'CONST'   ;size is 0

CONST  ends

_BSS   segment word use16 public 'BSS'     ;size is 0

_BSS   ends

DGROUP group   _DATA,CONST,_BSS

  extrn   __acrtused

  extrn   ?endl@@YAAAVostream@@AAV1@@Z

  extrn   ?insert@_2Comp@@CAAAVostream@@AAV2@PBXHH@Z

  extrn   ??6ostream@@QACAAV0@PBD@Z

  extrn   ?cout@@3Vostream_withassign@@A

  public  ??0DEMO@@QAC@XZ,?ausgabe@DEMO@@QACXXZ,_main

_TEXT  segment

  assume  CS:_TEXT

??0DEMO@@QAC@XZ:

      push    BP

      mov BP,SP
```

[21] Wer sich mit Assemblercode nicht auskennt oder nicht dafür interessiert, darf diesen Teil überspringen.

```
        mov CX,4[BP]
        mov BX,CX
        mov [BX],064h
        mov AX,BX
        pop BP
        ret 2
?ausgabe@DEMO@@QACXXZ:
        push    BP
        mov BP,SP
        sub SP,2
        push    SI
        push    DI
        mov SI,4[BP]
        mov AX,offset DGROUP:_DATA
        push    AX
        mov AX,offset DGROUP:?cout@@3Vostream_withassign@@A
        push    AX
        call    near ptr ??6ostream@@QACAAV0@PBD@Z
        mov BX,SI
        mov CX,[BX]
        mov -2[BP],CX
        mov CX,1
        push    CX
        inc CX
        push    CX
        lea CX,-2[BP]
        push    CX
        push    AX
```

```
call    near ptr ?insert@_2Comp@@CAAAVostream@@AAV2@PBXHH@Z
add SP,8
mov CX,offset ?endl@@YAAAVostream@@AAV1@@Z
push    AX
call    CX
add SP,2
pop DI
pop SI
mov SP,BP
pop BP
ret 2
```

```
_main:
        push    BP
        mov BP,SP
        sub SP,2
        lea AX,-2[BP]
        push    AX
        call    near ptr ??0DEMO@@QAC@XZ
        lea AX,-2[BP]
        push    AX
        call    near ptr ?ausgabe@DEMO@@QACXXZ
        mov SP,BP
        pop BP
        ret
_TEXT   ends
_DATA   segment
D0 db   069h,064h,03dh,000h
_DATA   ends
CONST   segment
CONST   ends
_BSS    segment
_BSS    ends
  end
; --- end of file inline1.cod --- Assemblercode zu inline1.cpp ---
```

Der C++-Quelltext mit `inline`:

```
// -----------------------------------------------------------------
// inline2.cpp: Realisierung mit inline-Funktionen
// -----------------------------------------------------------------
```

```
#include <iostream.h>


class DEMO
{
  private:
     int id;
  public:
     DEMO()                          { id=100; };
     void ausgabe();
}; // end class DEMO


inline void DEMO::ausgabe()
{
  cout << "id=" << id << endl;
} // end void DEMO::ausgabe()


void main()
{
  DEMO aDemoObject;
  aDemoObject.ausgabe();
} // end main
```

Und auch hier der daraus erzeugte Assemblercode:

```
; --- inline2.cod --- Assemblercode zu inline2.cpp ---
includelib SDs.lib
_TEXT  segment word use16 public 'CODE'    ;size is 72
_TEXT  ends
_DATA  segment word use16 public 'DATA'    ;size is 4
_DATA  ends
```

```
CONST  segment word use16 public 'CONST'   ;size is 0
CONST  ends
_BSS   segment word use16 public 'BSS'     ;size is 0
_BSS   ends
DGROUP group   _DATA,CONST,_BSS
  extrn   __acrtused
  extrn   ?endl@@YAAAVostream@@AAV1@@Z
  extrn   ?insert@_2Comp@@CAAAVostream@@AAV2@PBXHH@Z
  extrn   ??6ostream@@QACAAV0@PBD@Z
  extrn   ?cout@@3Vostream_withassign@@A

  public  _main
_TEXT  segment
  assume  CS:_TEXT
_main:
        push    BP
        mov BP,SP
        sub SP,4
        push    SI
        push    DI
        lea AX,-4[BP]
        mov BX,AX
        mov [BX],064h
        lea AX,-4[BP]
        mov DI,AX
        mov CX,offset DGROUP:_DATA
        push    CX
        mov CX,offset DGROUP:?cout@@3Vostream_withassign@@A
```

```
        push    CX
        call    near ptr ??6ostream@@QACAAV0@PBD@Z
        mov BX,DI
        mov CX,[BX]
        mov -2[BP],CX
        mov CX,1
        push    CX
        inc CX
        push    CX
        lea CX,-2[BP]
        push    CX
        push    AX
        call    near ptr ?insert@_2Comp@@CAAAVostream@@AAV2@PBXHH@Z
        add SP,8
        mov CX,offset ?endl@@YAAAVostream@@AAV1@@Z
        push    AX
        call    CX
        add SP,2
        pop DI
        pop SI
        mov SP,BP
        pop BP
        ret
_TEXT   ends
_DATA   segment
D0 db   069h,064h,03dh,000h
_DATA   ends
CONST   segment
```

```
CONST  ends

_BSS   segment

_BSS   ends

  end

; --- end of file inline2.cod --- Assemblercode zu inline2.cpp ---
```

## 2.5. Zeiger auf Klassenelemente (Operatoren `.*` und `->*`)

In diesem Abschnitt sollen der Vollständigkeit halber sogenannte *Zeiger auf Klassenelemente* behandelt werden. Für ein erstes Kennenlernen von C++ sind diese noch nicht relevant, so daß dieser Abschnitt auch übersprungen werden kann.

Neben den gewohnten Zeigern, die dann auch auf eine Instanz einer Klasse verweisen können, gibt es in C++ auch sogenannte Zeiger auf Klassenelemente. Diese unterscheiden sich von den normalen Pointern sowohl in der Deklaration als auch in der Derefenzierung.

Sehen wir uns die folgenden Deklarationen an:

```
int *pi;
int T::*Tp;                    // T sei irgendeine Klasse
```

Hiermit werden ein Zeiger `pi` auf int sowie ein Zeiger `Tp` auf ein Klassenelement vom Typ int deklariert. Hintergrund dieses zunächst kompliziert wirkenden Vorgehens ist die Tatsache, daß `Tp` nicht nur auf Elemente der Klasse `T` sondern auch auf Elemente daraus abgeleiteter Klassen (vgl. Kapitel 8) zeigen kann.

In dem folgenden kleinen Beispielprogramm soll der Einsatz der Operatoren `.*` und `->*` erläutert werden.

### 2.5.1. Beispiel: Die Operatoren `.*` und `->*`

Das nachfolgende kleine (auf das wesentliche abgespeckte) Programm zeigt den Einsatz von Zeigern auf Klassenelemente. Zur Vermeidung der sonst in der Literatur üblichen komplexen Beispiele werden hier drei `public`-Datenelemente[22] `i`, `j` und `k` in der Klasse `CLASSX` deklariert.

In dem Programm wird mit `pinstcomponent` ein Zeiger auf ein int-Mitglied der Klasse deklariert:

```
int CLASSX::*pinstcomponent;
```

Mit der Zuweisung

```
pinstcomponent = &CLASSX::i;
```

wird dieser Zeiger auf den Offset des Elementes `i` innerhalb der Klasse `CLASSX` gesetzt. Nun kann `pinstcomponent` für alle Instanzen der Klasse CLASSX genutzt werden, wie im Beispielprogramm mit `inst` und `inst2` gezeigt.

Der Operator `->*` ist – wie die struct-Pointerderefenzierung `->` zum gewöhnlichen struct-Komponentenzugriff `.` – die naheliegende Erweiterung, wenn nicht statische, sondern dynamisch erzeugte Instanzen angesprochen werden.

Mit der Deklaration

```
CLASSX * px = new CLASSX(7,8,9);
```

wird eine dynamische `CLASSX`-Instanz angelegt, die über den Zeiger `px` referenziert werden kann. Gilt die Zuweisung

```
pinstcomponent = &CLASSX::i;
```

so wird mit

```
px->*pinstcomponent
```

die Komponente `i` angesprochen.

Hier das Programmlisting:

```
// celemptr.cpp - Zeiger auf Klassenelemente


#include <iostream.h>
```

[22] Hier wird zur Erläuterung der nicht trivialen Operatoren `.*` und `->*` bewußt die Faustregel verletzt, daß Datenelemente `private` (oder `protected`) sein sollten.

```
class CLASSX
{
  public:
     int i, j, k;   // zu Demozwecken public
     CLASSX(int ii=1, int jj=2, int kk=3)  { i=ii; j=jj; k=kk; }
}; // end class CLASSX



void main()
{
  CLASSX inst, inst2(4,5,6);        // Zwei Instanzen werden deklariert
  int CLASSX::*pinstcomponent;// Ein Zeiger auf ein int-
                       // Klassenelement wird vereinbart

  pinstcomponent = &CLASSX::i;      // Hier wird keine absolute Adresse,
                       // sondern der Offset von i innerhalb
                       // der Klasse CLASSX zugewiesen!

                       // Kontrollausgaben
  cout << "inst.*pinstcomponent =  " << inst.*pinstcomponent << endl;
  cout << "inst2.*pinstcomponent = " << inst2.*pinstcomponent << endl;

  pinstcomponent = &CLASSX::j;
  cout << "inst.*pinstcomponent =  " << inst.*pinstcomponent << endl;
  cout << "inst2.*pinstcomponent = " << inst2.*pinstcomponent << endl;

  pinstcomponent = &CLASSX::k;
  cout << "inst.*pinstcomponent =  " << inst.*pinstcomponent << endl;
```

```
  cout << "inst2.*pinstcomponent = " << inst2.*pinstcomponent << endl;


  CLASSX * px = new CLASSX(7,8,9);

  cout << "px->*pinstcomponent =   " << px->*pinstcomponent << endl;


} // end main


// end of file celemptr.cpp
```

Und das erwartungsgemäße Ablauflisting:

```
inst.*pinstcomponent =  1

inst2.*pinstcomponent = 4

inst.*pinstcomponent =  2

inst2.*pinstcomponent = 5

inst.*pinstcomponent =  3

inst2.*pinstcomponent = 6

px->*pinstcomponent  =  9
```

# 3. Konstruktoren und Destruktoren

Fundamentaler Bestandteil von Klassen in C++ sind spezielle Methoden, die automatisch bei Anlegen („Geburt”) eines Objektes (z.B. auf dem Stack, im Datensegment oder auf dem Heap) und dessen Löschen („Tod”) vom System aufgerufen werden: die sogenannten*Konstruktoren* und *Destruktoren*.

## 3.1. Initialisierung von Objekten

Ist eine Klasse `K` deklariert worden, so kann mit `K k;` ein Objekt `k` dieser Klasse deklariert und definiert werden. Dabei haben jedoch die Datenelemente der Klasse noch keine festgelegten Werte erhalten, sind somit nicht initialisiert worden. Während sich bei vielen numerischen Variablen die Zahl 0 für eine Initialisierung anbietet, muß dies natürlich nicht generell eine sinnvolle Wahl sein.

In C++ sorgen spezielle Klassenmethoden für all das, was mit der Geburt eines Objektes zu geschehen hat, so auch ggf. für eine Initialisierung. Diese speziellen Funktionen heißen *Konstruktoren*; sie haben stets denselben Namen wie die Klasse selbst und besitzen keinen Rückgabetyp, auch nicht `void`! Wird vom Programmierer einer Klasse kein Konstruktor implementiert, so erzeugt C++ selbst einen (mit einem leeren Rumpf).

Konstruktoren stellen eine logische Erweiterung der klassischen Initialisierung dar: während in C bei einer Variablendefinition direkt nur Werte zugewiesen werden können, erlaubt C++ bei der Geburt (Definition) eines Objektes einen Funktionsaufruf. Hierbei kann nun beispielsweise eine Wert-Initialisierung stattfinden, es kann aber auch dynamischer Speicher angefordert werden usw.

### 3.1.1. Beispiel: Klasse RATIONAL — Ein Default-Konstruktor

Betrachten wir eine Klasse `RATIONAL`, die die Arbeit mit Brüchen implementieren soll. Auszugsweise kann dies aussehen wie im nachstehenden Programmfragment abgedruckt.

Als Datenelemente wurden `z` und `n` für Zähler und Nenner vereinbart, daneben sind `GGT()` zur Ermittlung des größten gemeinsamen Teilers und `Kuerzen()` zum Kürzen eines Bruches als private Hilfsfunktionen vorgesehen, hier aber nicht wirklich implementiert. Im `public`- Teil schließlich finden sich der Default-Konstruktor `RATIONAL()` und zwei Funktionen `Eingabe()` und `Ausgabe()`.

```
// ----------------------------------------------------------------------
// rational.cpp (Auszug)
// ----------------------------------------------------------------------
#include <iostream.h>
#include <stdlib.h>

// Klassendefinition
class RATIONAL
{
  private:
    int  z, n;           // Zähler, Nenner
                               // Beispiele für private Funktionen:
    int  GGT(int, int); // Intern: GGT bestimmen
    void Kuerzen();    // Intern: Kürzen des Bruchs
  public:
    RATIONAL();          // Default-Konstruktor
    void Eingabe();    // Tastatureingabe und
    void Ausgabe();    // Bildschirmausgabe
    // hier weitere Zugriffsfunktionen ...
}; // end class RATIONAL

// Klassenimplementation RATIONAL
RATIONAL::RATIONAL()    // Default-Konstruktor
{
  z=0;
  n=1;
} // end RATIONAL::RATIONAL()
```

```
void RATIONAL::Eingabe()
{
  do
  {
    cin >> z >> n;      // Einlesen von z und n
    if (n==0)           // Fehlerkorrektur
      cerr << "\n*** Fehler: Der Nenner darf nicht 0 sein!"
           << "\n*** Eingabe bitte wiederholen!\n";
  }
  while (n==0);
  Kuerzen();            // Bruch gekürzt abspeichern
} // end RATIONAL::Eingabe()

void RATIONAL::Ausgabe()
{
  cout << z << "/" << n << " (dezimal: " << (float)z/n << ") " ;
} // end RATIONAL::Ausgabe()

int RATIONAL::GGT(int a, int b)
{
  // Code aus Platzgründen weggelassen (vgl. Übung 6 auf Seite 0)
} // end RATIONAL::GGT(int a, int b)
```

```
void RATIONAL::Kuerzen()

{

  // Code aus Platzgründen weggelassen

} // end RATIONAL::Kuerzen()


// Hauptprogramm ---------------------------------------------------------

int main()

{

  RATIONAL p;

  p.Eingabe();

  cout << "Ausgabe p: ";

  p.Ausgabe();

  return EXIT_SUCCESS;

} // end main

// end of file rational.cpp ----------------------------------------------
```

Bei diesem Programm `rational.cpp` wird eine Klasse `RATIONAL` definiert, deren Default-Konstruktor (namens `RATIONAL()`) dafür sorgt, daß die Datenelemente Zähler `z` auf `0` und Nenner `n` auf `1` gesetzt werden. Damit ist bereits nach der Zeile

```
RATIONAL p;
```

die Instanz `p` der Klasse `RATIONAL` initialisiert, `p.z` ist `0` und `p.n` ist `1`. Dieses Objekt könnte also nun bereits direkt mit dem Aufruf

```
p.Ausgabe();
```

auf den Bildschirm geschrieben werden.

### 3.1.2. Anmerkung

Bereits an dieser Stelle sei angemerkt, daß innerhalb einer Klassenfunktion unter dem Namen `this` ein Zeiger auf das aktuelle Objekt selbst angesprochen werden kann. Dieser Pointer ist erforderlich, wenn die betreffende Mitgliedsfunktion das aktuelle Objekt (ggf. verändert) zurück- oder weitergeben muß.

## 3.2. Konstruktoren, Destruktoren und das Überladen

Halten wir noch einmal fest: Konstruktoren heißen stets so wie die Klasse; sie haben keinen Rückgabetyp (auch nicht `void`) und können nicht explizit aufgerufen werden. Wird ein Klassenobjekt angelegt, so wird von C++ selbst der (bzw. ein) Konstruktor aufgerufen.

Als *Default-Konstruktor* einer Klasse X wird der parameterlose Konstruktor mit dem Prototypen `X::X()` bezeichnet, denn bei der elementaren Objektdeklaration `X x;` wird dieser Konstruktor aufgerufen.

Wie erwähnt, können in C++ Funktionen überladen werden; dies trifft auch für die Konstruktoren zu. Neben dem Default-Konstruktor können also (auch) andere Konstruktoren festgelegt werden. Betrachten wir hierzu erneut die Klasse `RATIONAL` aus dem obigen Beispiel 3.1.1 (auf Seite 0).

```
// ------------------------------------------------------------------------

// rational.cpp (Weiterer Auszug)

// ------------------------------------------------------------------------



// Klassendefinition

class RATIONAL

{

  private:

    int   z, n;         // Zähler, Nenner

    // ...

  public:

    RATIONAL();               // Default-Konstruktor

    RATIONAL(int,int=1);      // Ein weiterer Konstruktor

    // hier weitere Zugriffsfunktionen ...

}; // end class RATIONAL



// ...

RATIONAL::RATIONAL(int zz, int nn)
```

```
{
  z=zz;
  n=nn;                   // Hier ohne Fehlerbehandlung!
  Kuerzen();
} // end RATIONAL::RATIONAL(int zz, int nn)
```

Zu dem bereits vorher deklarierten Default-Konstruktor `RATIONAL::RATIONAL()` ist nun ein weiterer Konstruktor mit dem Prototyp

```
RATIONAL::RATIONAL(int zz, int nn=1);
```

gekommen. Der zweite Parameter ist hierbei vorbelegt (initialisiert), so daß dieser Konstruktor auf die beiden Deklarationsformen

```
RATIONAL einBruch(1);
RATIONAL nochEinBruch(1,3);
```

zutrifft. Das Objekt `einBruch` (hier ist nicht der Straftatbestand gemeint!) erhält (über die Parameter `zz` und `nn`) die Werte `z=1` (`zz=1` wurde beim Anlegen des Objektes mitgegeben) und `n=1` (`nn=1` ist die Vorbelegung in der Konstruktor-Deklaration), stellt also die ganze Zahl `1` dar; `nochEinBruch` erhält dagegen die Werte `z=1` und `n=3`, inhaltlich also den Bruch `1/3`.

Das Gegenstück zu den Konstruktoren sind die Destruktoren. Diese speziellen Funktionen werden zum Ende der Lebensdauer eines Objektes automatisch aufgerufen. Hier findet z. B. die Freigabe von dynamisch angefordertem Speicher statt.

Destruktoren heißen immer so wie die Klasse, erhalten jedoch vorangestellt eine *Tilde* (~). Die Parameterliste ist stets leer, es gibt keinen Rückgabetyp. Die Klasse `X` hat daher (genau) einen Destruktor namens `X::~X()`. Wird vom Programmierer einer Klasse kein Destruktor explizit definiert, so generiert C++ selbst einen solchen (mit einem leeren Funktionsrumpf).

Im Beispiel unserer Bruch-Klasse kann der Destruktor zu Demonstrationszwecken so implementiert werden, daß er eine Diagnose-Meldung auf den Bildschirm ausgibt. Da unsere Klasse `RATIONAL` keine dynamischen Speicherbereiche verwaltet, muß natürlich hier auch kein Heap-Speicher freigegeben werden.

```
// ----------------------------------------------------------------------
// rational.cpp (Wieder nur ein Auszug)
// ----------------------------------------------------------------------
```

```
// Klassendefinition
class RATIONAL
{
  private:
    int   z, n;         // Zähler, Nenner
    // ...
  public:
    ~RATIONAL();        // Der Destruktor
    // ...
}; // end class RATIONAL

// Klassenimplementation RATIONAL
// ...
RATIONAL::~RATIONAL()                // Der Destruktor
{
  cout << "*** Destruktor aufgerufen für ";
  Ausgabe();
  cout << endl;
} // end RATIONAL::~RATIONAL()
```

Zu dem Hauptprogramm

```
// Hauptprogramm ---------------------------------------------------------
int main()
{
  RATIONAL p, q(5), r(14,70);  // Drei Objekte (0, 5 und 14/70)
  cout << "p: ";
  p.Ausgabe();
  cout << endl << "q: ";
```

```
  q.Ausgabe();
  cout << endl << "r: ";
  r.Ausgabe();
  cout << endl;
} // end main ------------------------------------------------------------
```

sieht das Ablaufprotokoll, also die Bildschirmausgabe, dann etwa so aus:

```
p: 0/1 (dezimal: 0)
q: 5/1 (dezimal: 5)
r: 1/5 (dezimal: 0.2)
*** Destruktor aufgerufen für 1/5 (dezimal: 0.2)
*** Destruktor aufgerufen für 5/1 (dezimal: 5)
*** Destruktor aufgerufen für 0/1 (dezimal: 0)
```

Das heißt: mit Verlassen des Lebensdauerbereiches eines Objektes (hier `main()` für die Objekte `p`, `q` und `r`) werden die Destruktoren (in der umgekehrten Reihenfolge der Deklarationen) automatisch aufgerufen. Beachten Sie bitte in diesem Zusammenhang den Unterschied zwischen Gültigkeitsbereich und Lebensdauer eines Objektes!

Für die Arbeit mit Konstruktoren und Destruktoren sei Übung 7 auf Seite 0 genannt.

## 3.3. Referenzen

Während in ANSI-C die Parameterliste einer Funktion ausschließlich *call by value* Parameter enthalten kann und das aus Pascal bekannte *call by reference* durch eine Pointerübergabe emuliert werden muß, gibt es in C++ die sogenannten *Referenzparameter*. Hierzu sei auf Übung 4 auf Seite 0 hingewiesen.
Über Pascal hinausgehend bietet C++ sogar „stand-alone” Referenzen an. Dabei handelt es sich um bereits zur Compilationszeit festgelegte Alias-Bezeichnungen für deklarierte Speicherplätze.

Sehen wir uns nachstehendes Demonstrationsprogramm an.

```
// -----------------------------------------------------------
// referenz.cpp
// -----------------------------------------------------------
#include <iostream.h>
```

```
#include <stdlib.h>


int main()
{
  int i=25;            // int-Speicherplatz i wird deklariert
  int &ref=i;      // und mit ref eine Referenz darauf... -
                       // d.h.: ref ist ein Synonym für i!

  cout << " i=" << i <<  "       ref=" << ref << endl;
  cout << "&i=" << &i << "  &ref=" << &ref << endl;

  return EXIT_SUCCESS;

} // end main
```

Das dazugehörige Ablaufprotokoll:

```
 i=25       ref=25
&i=0x2594  &ref=0x2594
```

Wie man sieht: `i` und `ref` beinhalten nicht nur dieselben Werte, sie liegen sogar an derselben Adresse! Das heißt: anders als bei einem Pointer-Mechanismus wird hier für `ref` keine eigenständige Variable, kein eigener Speicherplatz angelegt; compilerintern dient `ref` nur als Aliasname (wie aus UNIX vielleicht bekannt) für etwas anderes, hier eben den Speicherplatz mit dem Namen `i`.

Dieser Mechanismus kann nun sehr gut für ein call-by-reference in Parameterlisten verwendet werden.

```
// ------------------------------------------------------------
// callbyreference.cpp
// ------------------------------------------------------------
// Demonstration des Referenz-Mechanismus in C++ anhand
```

```
// einer trivialen Vertauschungsfunktion (bzw. -prozedur).
// ----------------------------------------------------------

#include <iostream.h>
#include <stdlib.h>

void Tauschen1(int *, int *);        // Prototypen der unten beschriebenen
void Tauschen2(int &, int &);        // Funktionen

// ANSI-C-Version: Pointerübergabe zur Emulation des Call-By-
// Reference-Mechanismus
void Tauschen1(int *p1, int *p2)
{
  int tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
} // end Tauschen1

// C++-Version: Übergabe eines Referenzparameters
void Tauschen2(int &ref1, int &ref2)
{
  int tmp = ref1;
  ref1 = ref2;
  ref2 = tmp;
} // end Tauschen2

int main()
{
```

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

  Tauschen1(&a,&b);    // Aufruf mit Adressen in ANSI-C

  Tauschen2(a,b);      // Aufruf mit "normalem" Variablennamen

  return EXIT_SUCCESS;

} // end main
```

Wie Sie sehen, ist der Aufruf über Referenzparameter eleganter: sowohl innerhalb der Implementation der Funktion, als auch beim Aufruf sieht der Code lesbarer aus, denn es müssen keine Pointerdereferenzierungen und keine Adreßoperatoren angegeben werden. Darüber hinaus ist der Code in der Funktion strukturell identisch mit dem, der direkt in `main()` für das Vertauschen der beiden Variablen geschrieben werden müßte, was das spätere Auslagern von Codeteilen in eine Funktion natürlich erleichtert.

Betrachten wir noch folgendes Beispiel: eine Funktion `ToUpper()`, die einen Kleinbuchstaben (inklusive der deutschen Umlaute) in einen Großbuchstaben umwandeln soll. (Die Ausarbeitung des Algorithmus wird Ihnen als Übung überlassen.) Der grobe Rahmen einer solchen Funktion kann in C++ wie folgt aussehen.

```
char & ToUpper(char & zeichen)

{

  // ... hier kommt der interessante Code ...

  return zeichen;

} // end ToUpper
```

Neu gegenüber C ist die Möglichkeit, daß auch die Rückgabe einer Funktion eine Referenz sein kann! Hier wird in der Parameterliste ein Referenzparameter `zeichen` vereinbart, d.h. es wird beim Aufruf `ToUpper(z)` direkt auf dem Speicherplatz der übergebenen `char`-Variablen `z` gearbeitet. Und dieser Speicherplatz (i.e. eine Referenz darauf) wird auch zurückgegeben.

Warnung: Bei diesem Mechanismus wird eine Referenz auf etwas zurückgeliefert, was natürlich nicht auf dem für diese Funktion aufgebauten Stack-Teil liegen darf. Sehen wir uns hierzu das folgende abschreckende Beispiel an.

```
char & ToUpper(char zeichen)      // ** Vorsicht: fehlerhafter Code! **

{

  // ... hier kommt der interessante Code ...

  return zeichen;
```

```
} // end ToUpper
```

Wieso ist diese Version von `ToUpper()` falsch?

Durch den Werteparameter `char zeichen` wird bei einem konkreten Aufruf auf dem Stack eine Kopie angelegt; eine Referenz auf diese Kopie wird dann mit `return zeichen` zurückgeliefert. Bei der aufrufenden Instanz ist diese Adresse jedoch nicht mehr gültig, da der Stack, der für die `ToUpper()`-Funktion benötigt wurde, bereits wieder freigegeben worden und damit ungültig ist!

## 3.4. Der Kopierkonstruktor

Eine spezielle Sorte von Konstruktoren sind die sogenannten *Kopierkonstruktoren*. Diese werden stets dann benötigt, wenn ein temporäres Objekt angelegt werden muß. Dies ist dann der Fall, wenn

- ein Objekt durch ein anderes Objekt initialisiert wird,
- ein Objekt als call by value (Wert-)Parameter an eine Funktion übergeben wird,
- ein Objekt – nicht eine Referenz darauf – von einer Funktion zurückgegeben wird.

Sehen wir uns einen solchen Kopierkonstruktor für unser Beispiel der Klasse `RATIONAL` an.

```
RATIONAL::RATIONAL(RATIONAL &r)            // Kopierkonstruktor
{
  z=r.z; // ist stets gekürzt, daher muß Kuerzen() nicht aufgerufen
  n=r.n; // werden.
} // end RATIONAL::RATIONAL(RATIONAL)
```

In den folgenden Situationen wird er (implizit) aufgerufen:

```
RATIONAL bruch1(1,2);
RATIONAL bruch2(bruch1);       // Initialisierung von bruch2 durch bruch1
RATIONAL Kehrwert(RATIONAL);   // Prototyp einer Funktion mit einem
                       // call-by-value-Parameter und dem
                       // Rückgabetyp RATIONAL
```

Auch hier eine Anmerkung: ein Kopierkonstruktor einer Klasse X hat generell die Form `X::X(X&)` bzw. `X::X(const X&)`. Mit dem `const` kann insbesondere gesagt werden, daß dieser Konstruktor das betreffende Objekt nicht verändert: man erhält also die Sicherheit eines call-by-value-Aufrufes zusammen mit der Geschwindigkeit eines Referenzaufrufes in einem!

Achtung: hätte der Kopierkonstruktor nicht einen Referenz-, sondern einen Werteparameter (Bauart `X::X(X)`), dann würde er sich implizit rekursiv immer wieder selbst aufrufen, das Programm hinge fest, denn wie bereits erwähnt: für einen Werteparameter bei einer Funktion wird der Kopierkonstruktor automatisch aufgerufen!

C++ genereriert automatisch zu jeder Klasse einen Kopierkonstruktor, so daß man dies nicht unbedingt selbst erledigen muß. Dabei werden die Datenelemente einfach eins zu eins kopiert. Wird jedoch mit dynamischem Speicherplatz und mit Pointern innerhalb einer Klasse gearbeitet, so reicht dieses Kopieren der Datenelemente natürlich nicht mehr aus, denn verschiedene Pointer würden dann auf dieselben Objekte zeigen, die impliziten Destruktoraufrufe würden damit versuchen, ein und denselben Speicherplatz mehrfach freizugeben!

## 3.5. Statische Klassenmitglieder

Klassenmitglieder (Daten und Funktionen) können das Schlüsselwort `static` tragen. In diesem Kontext bedeutet das dann, daß diese Mitglieder zur Klasse und nicht zur jeweiligen Instanz (dem jeweiligen Objekt) gehören. Sie existieren also nur einmal für die gesamte Klasse und werden bereits angelegt, bevor das erste Objekt der Klasse generiert wird[23]. Sinn macht dies für Informationen oder Routinen, die der gesamten Klasse zugänglich sein müssen, die aber damit trotzdem (im Gegensatz zu globalen Variablen oder Funktionen) im Gültigkeitsbereich der Klasse verbleiben (sollen).

Für statische Mitgliedsdaten gelten die üblichen Zugriffsregeln. Diese Daten müssen allerdings außerhalb der Klassendeklaration initialisiert werden. Da ein solches `static`-Mitglied nicht an ein Objekt gebunden ist, muß es in der Form `classname::membername` angesprochen werden.

Betrachten wir das folgende kleine Beispiel, in dem eine statische Klassenvariable `numberOfObjects` deklariert und im Hauptprogramm mit `0` initialisiert wird. Der Klassenkonstruktor und der Destruktor in- bzw. dekrementieren diese Variable dann entsprechend; zusätzlich findet im Destruktor zu Diagnosezwecken eine Fehlerausgabe statt, wenn der Destruktor öfter aufgerufen werden sollte als Klasseninstanzen vorhanden sind.

Beachten Sie die fettgedruckte Zeile: auf globaler Ebene muß für dieses statische Klassenmitglied Speicherplatz allokiert werden[24]!

```
class GRAPHIC
{
  public:
    static int numberOfObjects;   // Anzahl der Klasseninstanzen
    GRAPHIC();                    // Konstruktor
    ~GRAPHIC();                   // Destruktor
}; // end class GRAPHIC


GRAPHIC::GRAPHIC()
{
  numberOfObjects++;
```

---

[23] Beachten Sie bitte unbedingt, daß diese Bedeutung von static etwas vollkommen anderes meint als das static, das in C für Variablen oder Funktionen benutzt werden kann!

[24] Stroustrup schreibt, daß diese „irgendwo" im Quelltext erfolgen dürfe; die bisherigen Erfahrungen zeigen jedoch, daß manche Compiler diese z.B. nicht lokal in `main()` akzeptieren.

```
} // end Konstruktor GRAPHIC::GRAPHIC()


GRAPHIC::~GRAPHIC()
{
  numberOfObjects--;
  if (numberOfObjects < 0)
  {                              // Fehlerkontrolle / Ausgabe auf stderr
    cerr << "Fehler aufgetreten! Bitte Code kontrollieren!" << endl;
  }
} // end Destruktor GRAPHIC::~GRAPHIC()


int GRAPHIC::numberOfObjects=0;      // hier erst reserviert der Compiler
                       // den Speicherplatz!


// ... weiterer Code ...


int main()
{
  GRAPHIC::numberOfObjects = 0;   // Initialisierung
  // ... weiterer Code ...
} // end main
```

Statische Mitgliedsfunktionen einer Klasse dürfen (verständlicherweise) nur die statischen Mitgliedsdaten verändern: Existiert in einer Klasse X eine nicht-statische Variable `i`, so wird ein solches `i` für jede Klasseninstanz angelegt, der statischen Klassen-Funktion wäre nun nicht klar, welches der eventuell mehreren `is` manipuliert werden soll. Aus dem gleichen Grund haben statische Klassenfunktionen auch keinen `this`-Pointer, auf den sie zurückgreifen könnten: schließlich gehören diese Funktionen nicht einer Instanz, sondern der Klasse selbst! – Vergleichen Sie hierzu bitte die nachstehenden Bildschirmabzüge, in denen der UNIX- bzw. der Symantec-C++-Compiler aufgerufen wurden.

```
// ----------------------------------------------------------
```

```
// staticmemberfunction.cpp
// ---------------------------------------------------------
#include <stdlib.h>
class GRAPHIC
{
  public:
    static int numberOfObjects;
    int i;
    GRAPHIC()  { numberOfObjects++; i=10; }  // Inline formulierte Klassen-
    ~GRAPHIC() { numberOfObjects--; i=20; }  // Funktionen25)
    static void function(void)  { numberOfObjects=i; }
    // ...
}; // end GRAPHIC


int main()
{
  // ...
  return EXIT_SUCCESS;
} // end main
// ---------------------------------------------------------
// UNIX-Bildschirmprotokoll --------------------------------
/baeumle/cplusplus> CC statmfct.cpp
CC: "statmfct.cpp", line 12: error: object or object pointer missing for
GRAPHIC::i (1293)
// ---------------------------------------------------------
// Symantec C++ 6.11 ---------------------------------------
```

[25] Wie hier gezeigt wird können Klassenmethoden auch *inline*, d.h. innerhalb der Klassendefinition selbst, formuliert werden. Dies empfiehlt sich i.d.R. jedoch nur für sehr kurz zu schreibende Funktionen.

```
c:\cpp> sc statmfct.cpp

Symantec Compiler Driver Version 6.11

Copyright (C) Symantec Corporation 1985-1994.  All Rights Reserved.

scppx statmfct.cpp

statmfct.cpp(12)  :  Error:  no  instance  of  class  'GRAPHIC'  for  member
'GRAPHIC::i'

--- errorlevel 1

// -----------------------------------------------------------
```

Nachstehend noch ein weiteres Beispielprogramm, das eine statische (private) Klassenvariable und eine statische Klassenfunktion enthält. Die Definition (Speicherplatzallokierung) der Variablen `ANYCLASS:staticmember` findet in der Tat erst bei der fettgedruckten Zeile statt! Beachten Sie auch die (scheinbare) Merkwürdigkeit, daß der Compiler diese Zeile akzeptiert, obgleich diese Variable als `private` deklariert worden ist!

```
// -----------------------------------------------------------

// staticclassmember.cpp

// -----------------------------------------------------------

#include <iostream.h>

#include <stdlib.h>


class ANYCLASS

{

    private:

        int normalmember;

        static int staticmember;

    public:

        ANYCLASS();

        static void Print();

}; // end ANYCLASS
```

```
ANYCLASS:ANYCLASS()
{
    normalmember=1;
    staticmember++;
}

void ANYCLASS::Print()
{
    cout << "staticmember=" << staticmember << endl;
}

int ANYCLASS::staticmember=100;

int main()
{
    ANYCLASS::Print();
    return EXIT_SUCCESS;
} // end main
```

Zu statischen Klassenmitgliedern verweisen wir auch auf Übung 8 auf Seite 0.

## 3.6. Konstante Klassenmitglieder

Klassenmitglieder können auch konstant (`const`) sein. Dies sollte immer dann verwendet werden, wenn die Objekte einer Klasse schreibgeschützt werden sollen. Das Schlüsselwort `const` kann hier in drei Situationen auftreten: eine Instanzvariable kann konstant sein, ein ganzes Klassenobjekt kann als konstant deklariert werden, und eine Mitgliedsfunktion kann als konstant markiert sein.

```
class X

{

  private:

    const int nummer;

  public:

    // ...

    void Ausgabe(void) const;

}; // end class X

// ...

int main()

{

  X anobject;

  const X aConstObject;

  // ...

}// end main
```

Hier wird `nummer` als konstante Instanzvariable vereinbart, die also nur in den Konstruktoren einmal gesetzt und danach nicht mehr verändert werden darf; das Objekt `aConstObject` ist als konstant deklariert, d.h. nach einer (auch logisch) erforderlichen Initialisierung durch den Konstruktor kann dieses Objekt nicht mehr geändert werden; schließlich ist die Funktion `Ausgabe()` als konstant gekennzeichnet: dadurch wird dem Compiler und dem Leser des Quellcodes deutlich, daß diese Funktion keine Mitgliedsdaten verändern will bzw. kann.

Ein paar Regeln zur Arbeit mit konstanten Klassenmitgliedern seien noch erwähnt.

- `const`-Mitgliedsdaten müssen über den Konstruktor initialisiert werden;
- `const`-Mitgliedsfunktionen dürfen keine Änderungen an den Mitgliedsdaten vornehmen;

- auf `const`-Klassenobjekte dürfen ausschließlich die Mitgliedsfunktionen angewendet werden, die ihrerseits als `const` vereinbart worden sind!

Diese Seite wurde absichtlich leer gelassen.

# 4. Dynamische Speicherverwaltung

Selbstverständlich kennt C++ neben der statischen Variablendeklaration auch dasdynamische Allokieren von Speicherplatz auf dem Heap[26]. Da C++ im wesentlichen eine Obermenge von ANSI-C darstellt, kann wie für den C-Programmierer gewohnt mit `malloc()` Speicherplatz angefordert und mit `free()` wieder freigegeben werden. Dies wird im folgenden Abschnitt ganz kurz wiederholt. Im darauffolgenden Abschnitt wird dann erläutert, wie die in C++ neu eingeführten Operatoren `new` und `delete` arbeiten – und welche Vorzüge diese gegenüber den Bibliotheksfunktionen `malloc()` und `free()` dem (hoffentlich objektorientierten) Programmierer bieten.

Allerdings sei bereits an dieser Stelle eindringlich darauf hingewiesen, daß die beiden Speicherverwaltungskonzepte im allgemeinen nicht synchronisiert und aufeinander abgestimmt sind! Das bedeutet: entscheiden Sie sich dafür, in einem C++-Programm (oder -Projekt) entweder alles mit `malloc()` und `free()` oder aber, und das ist empfehlenswert, durchgehend und ausschließlich mit den Operatoren `new` und `delete` zu implementieren!

## 4.1. C: malloc und free

Im gewohten ANSI-C kann ein Pointer auf einen Speicherplatz vom Typ `TYPE` deklariert werden als `TYPE * ptr;` statisch wird dabei lediglich der Speicherplatz für eine Adresse reserviert. Erst mit der Anweisung

```
ptr = (TYPE *)malloc(sizeof(TYPE));
```

wird Speicherplatz für einen Datensatz vom Typ `TYPE` auf dem Heap bereitgestellt und `ptr` erhält dessen Adresse. Im Fehlerfall, d.h. wenn der angeforderte Speicherplatz nicht allokiert werden konnte, erhält `ptr` den Wert `NULL`.

Mit der entsprechenden Anweisung

```
ptr = (TYPE *)malloc(n*sizeof(TYPE));
```

werden n (aufeinanderfolgende) Speicherplätze des Typs `TYPE` allokiert, auf die über die Adressen `ptr`, `ptr+1`, usw. bis `ptr+n-1` zugegriffen werden kann. Dies ist die sogenannte *Pointer-Arithmetik* von ANSI-C.

Wird ein solcher dynamisch allokierter Speicherplatz auf dem Heap nicht mehr benötigt, so kann er mit `free(ptr);` auch wieder freigegeben werden. Ein Aufruf von `free()` mit einem anderen Pointer(wert) als dem von `malloc()` zurückgelieferten führt zu einem undefinierten Verhalten.

---

[26] Nun gut, so selbstverständlich muß das nicht sein: Programmiersprachen wie Smalltalk oder Java lassen dem Programmierer nicht die Freiheit, eigenmächtig auf dem Heap herumzufuhrwerken. Dort existiert eine sogenannte *Garbage Collection*, die diese Speicherverwaltung automatisiert.

## 4.2. C++: new und delete

In C++ existieren (gegenüber C) zwei neue Operatoren zur dynamischen Speicherverwaltung: `new` und `delete`. Damit kann, auch in Zusammenhang mit Klassenobjekten, die Speicherverwaltung sehr flexibel in die eigene Hand genommen werden; außerdem ist der Aufruf des Operators `new` sehr viel einfacher als ein `malloc()`-Aufruf.

Ist `ptr` der Zeiger auf `TYPE` wie zuvor. Dann wird mit dem Aufruf

```
ptr=new TYPE;
```

ein Speicherplatz des Typs `TYPE` bereitgestellt und dessen Adresse an `ptr` zugewiesen. Später kann dann mit `delete ptr` dieser wieder freigegeben werden.

Wird nicht ein einzelner Speicherplatz benötigt, sondern gleich n Stück, dann lautet der Aufruf mit dem `new`-Operator `ptr = new TYPE[n];` die entsprechende Freigabe der n Speicherplätze wird dann entweder mit `delete [n] ptr;` oder aber – kürzer und besser – mit `delete [] ptr;` durchgeführt.[27)]

Der grundlegende Mechanismus soll im folgenden kleinen Beispielprogramm dargestellt werden.

```
// ------------------------------------------------------------

// dynamic.cpp

// ------------------------------------------------------------

// Demonstration der Operatoren new und delete

// ------------------------------------------------------------

#include <iostream.h>

#include <stdlib.h>

#include <string.h>


int main()

{

   // Einlesen eines Textes in einen statischen Buffer.

   char buf[256];
```

---

27) Die Schreibweise `delete [n] ptr` wird von Stroustrup in *The Design and Evolution of C++* lediglich als *intermediate* (vorübergehend) bezeichnet; aktueller Stand ist also der Aufruf `delete [] ptr` ohne Angabe der Elemente-Anzahl.

```
  cout << "Eingabe: ";
  cin >> buf;


  // Anlegen eines zum eingegebenen Text passenden
  // dynamischen Buffers.
  char *text;
  text=new char[strlen(buf)+1];
  strcpy(text,buf);
  cout << text << endl;
  delete[] text;      // Freigabe der Speicherplätze
  return EXIT_SUCCESS;
} // end of main
```

Zu beachten ist dabei (wie auch schon in ANSI-C): der so allokierte dynamische Speicherplatz hat globale Lebensdauer und Gültigkeit; lediglich der Pointer unterliegt den gewohnten Einschränkungen an Lebensdauer und Gültigkeitsbereich! D.h.: ist die Adresse eines mit `new` angelegten Speicherplatzes bekannt, so kann darauf von überall, auch aus anderen Modulen, zugegriffen werden!

### 4.2.1. Beispiel: Die Klasse TEXTZEILE

Im folgenden Beispiel soll eine einfache Textzeile als eigenständige Klasse implementiert werden. Wir wollen an dieser Stelle, wie in der Praxis üblich, dieses kleine Projekt auf mehrere Quelltextdateien verteilen. Zum einen das Hauptprogramm in der Datei `main.cpp`, sodann die Deklaration der Klasse `TEXTZEILE` in der Headerdatei `textzeile.h`, und schließlich gibt es die Implementierung der Klasse `TEXTZEILE` in der Datei `textzeile.cpp`.

Das Hauptprogramm:

```
// main.cpp
// Demonstration der new und delete-Operatoren


#include <iostream.h>
#include <stdlib.h>
#include "textzeile.h"
```

```
int main()
{
   TEXTZEILE zeile1("Die Heuschrecke lebt am Stadtrand");
   zeile1.Ausgabe(cout); // Spaeter wird der operator<<() ueberladen,
                         // damit in der Form cout << zeile1
                         // wie in C++ ueblich ausgegeben werden kann.

   TEXTZEILE zeile2('*',10);
   zeile2.Ausgabe(cout);

   return EXIT_SUCCESS;

} // end main
// end of file newdelete.cpp
```

Die Headerdatei für die Klasse TEXTZEILE:

```
// textzeile.h
// Headerfile fuer die Klasse TEXTZEILE
#ifndef _TEXTZEILE__
#define _TEXTZEILE__

#include <string.h>
#include <iostream.h>

class TEXTZEILE
{
   private:
```

```
      char * zeile;
      int    laenge;

      // Interne Methoden (Hilfsroutinen)
      char * stringcopy(char *); // Einen String kopieren

   public:
      TEXTZEILE(char * = ""); // U.a. Default-Konstruktor
      TEXTZEILE(char, int);   // Spezieller Konstruktor, der das
                              // angegebene Zeichen mehrfach
                              // wiederholt in die Zeile schreibt
      virtual ~TEXTZEILE();   // Der Destruktor ist virtuell, warum,
                              // das wird spaeter in Kap. 9 behandelt!
      void Ausgabe(ostream&); // Einfache Ausgabemethode.

}; // end class TEXTZEILE

#endif
// end of file textzeile.h
```

Die Implementierung der Klasse `TEXTZEILE`:

```
// textzeile.cpp
// Eine Andeutung einer sehr elementaren Implementation
// einer Textzeile z.B. fuer einen Editor.

#include "textzeile.h"        // Headerdatei zur Klasse TEXTZEILE
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
```

```
// ----------------------------------------------------------------
// Public Methoden der Klasse TEXTZEILE
// ----------------------------------------------------------------

TEXTZEILE::TEXTZEILE(char * _zeile)
{
   stringcopy(_zeile);
} // end TEXTZEILE::TEXTZEILE(char * _zeile)

TEXTZEILE::TEXTZEILE(char zeichen, int wieoft)
{
   laenge = wieoft;
   zeile = new char[laenge+1];
   for (int i=0; i<laenge; i++)
      zeile[i]=zeichen;
   zeile[laenge]='\0';
} // end TEXTZEILE::TEXTZEILE(char zeichen, int wieoft)
TEXTZEILE::~TEXTZEILE()
{
   delete zeile;
   laenge = 0;
} // end TEXTZEILE::~TEXTZEILE()

void TEXTZEILE::Ausgabe(ostream& out)
{
   // Zur Demonstration wird auch die Laenge mit ausgegeben.
   out << zeile
```

```
   << " (Laenge="
   << laenge << ")"
   << endl;
} // end void TEXTZEILE::Ausgabe(ostream& out)


// -------------------------------------------------------------
// Private Methoden der Klasse TEXTZEILE
// -------------------------------------------------------------


char * TEXTZEILE::stringcopy(char * string)
{
   laenge = strlen(string);
   zeile = new char[laenge+1];
   if (zeile == NULL)      // Der Einfachheit halber hier nur ein
   {                       // simpler Programmausstieg.
      cerr << "Fehler: Speicherallokierung gescheitert!" << endl;
      exit(EXIT_FAILURE);
   }
   strcpy(zeile,string);
   return zeile;
} // end char * TEXTZEILE::stringcopy(char * string)
// end of file textzeile.cpp
```

Und schließlich noch das kurze Ablauflisting zu diesem Beispiel:

```
Die Heuschrecke lebt am Stadtrand (Laenge=33)
********** (Laenge=10)
```

# 5. Überladen von Operatoren

In C++ ist das Überladen (*overloading*) von fast allen Operatoren möglich, auch von `new` und `delete`, die im vorherigen Kapitel vorgestellt wurden. Lediglich die Operatoren `.` (Punkt), `.*`, `::` und `?:` können nicht überladen werden. Unter Überladen von Operatoren versteht man dabei die Redefinition eines vordefinierten Operators von C++: während der Additionsoperator für int oder float bereits vordefiniert ist, ist C++ natürlich nicht bekannt, wie die selbstdefinierten Brüche der Klasse `RATIONAL` (vgl. Beispiel 5.1.1) addiert werden sollen. Wir können aber den Operator + für diese Klasse überladen und selbst definieren, was z.B. die Summe zweier Brüche (`RATIONAL`-Objekte) sein soll[28)].

## 5.1. Überladen als Klassenmitgliedsfunktionen

Die allgemeine Syntax des Operator Overloading als Klassenmethode sieht so aus:

```
Rückgabetyp CLASSNAME::operator*(Parameterliste);
```

Hierbei steht der `*` beispielhaft für irgendeinen überladbaren Operator.

### 5.1.1. Beispiel: Addition in der Klasse RATIONAL

Erinnern wir uns an die Klasse `RATIONAL` aus Beispiel 3.1.1.: dort wurde eine Klasse für Brüche (ansatzweise) implementiert. In dieser Klasse könnte ein Überladen des Operators + für die Addition wie folgt aussehen[29)].

```
// Addition in der Klasse RATIONAL

RATIONAL RATIONAL::operator+(RATIONAL &q)

{

  RATIONAL tmp;

  tmp.z=z*q.n+n*q.z;

  tmp.n=q.n*n;

  tmp.Kuerzen();

  return tmp;

} // end RATIONAL RATIONAL::operator+(RATIONAL &q)
```

---

28) Ganz formal betrachtet wird nicht der Operator +, den es für int oder float gibt, überschrieben oder neu definiert, sondern es wird lediglich dasselbe Operatorsymbol für einen neuen Zusammenhang (Kontext) implementiert.

29) Ob sich die Leser noch an die Schulmathematik erinnern? Zwei Brüche werden addiert, indem man sie auf den gemeinsamen Hauptnenner bringt... Ja, so war das...

Sind p und q Objekte der Klasse RATIONAL, so ist hiermit p.operator+(q) als Ergebniswert ebenfalls von der Klasse RATIONAL: inhaltlich die Summe der beiden Brüche. Und da man gewohnt ist, die Addition einfacher als p+q zu schreiben, definiert C++ genau dies: p+q ist die Kurzschreibweise für p.operator+(q), wobei dies natürlich für jeden zweiwertigen Operator ∇, der überladen wird, gilt: p∇q steht für p.operator∇(q).

Anmerkung: der Parameter bei dieser Operatorfunktion ist als Referenz gekennzeichnet, denn dadurch muß keine Kopie auf dem Stack abgelegt werden; zur Betonung, daß dieser Operator diesem Parameter „nichts tut", d.h. diesen nicht verändert, könnte hier noch das Schlüsselwort const ergänzt werden.

### 5.1.2. Beispiel: Zuweisungsoperator der Klasse RATIONAL

Ein weiteres Beispiel: häufig bietet es sich ebenfalls an, den Zuweisungsoperator = zu überladen und ihn für eine Klasse zur Verfügung zu stellen. Auch dies sei nachstehend am Beispiel der beliebten Klasse RATIONAL illustriert.

Wie in Abschnitt 3.1 bereits angemerkt: das Schlüsselwort this bezeichnet einen Zeiger auf das aktuelle Klassenobjekt. Im folgenden Beispiel wird er benötigt für die Rückgabe des überladenen Zuweisungsoperators.

```
// Zuweisungsoperator in der Klasse RATIONAL

RATIONAL& RATIONAL::operator=(RATIONAL& a)

{

  z=a.z;

  n=a.n;

  return *this; // aktuelles Objekt weiterreichen/zurückliefern

} // end RATIONAL& operator=(RATIONAL&)
```

Sind p und q Objekte der Klasse RATIONAL, so ist hiermit p.operator=(q) als Ergebniswert ebenfalls von der Klasse RATIONAL: inhaltlich wird q an p zugewiesen, der gesamte Ausdruck erhält das Objekt p wiederum als Rückgabewert. So sind folgende Aufrufe möglich:

```
RATIONAL p, q, r(1);   // r wird gleich mit 1/1 initialisiert

p=q;                   // entspricht der Anweisung p.operator=(q);

// auch das geht:
```

```
p=q=r;          // entspricht der langatmigeren Darstellung
                // p.operator=(q.operator=(r));
// und mit dem Operator + von vorher geht auch dies:
p=q+r;          // entspricht p.operator=(q.operator+(r));
                // (Damit sollte auch klar sein, wieso der
                // operator+() einen Rückgabewert besitzt!)
```

### 5.1.3. Beispiel: Zuweisungsoperator in der Klasse TEXTZEILE

Wir wollen an dieser Stelle kurz das Beispiel 4.2.1. (s. Seite 0) mit der Klasse `TEXTZEILE` fortsetzen, indem wir dort einen `operator=()` implementieren. Dieses Beispiel bietet sich deshalb an, weil in dieser Klasse ein dynamisches Datenelement (`char * zeile`) verwaltet wird.

Das Hauptprogramm verwendet diesen Operator wie folgt[30]:

```
// main.cpp


// ...

int main()

{

   TEXTZEILE zeile1("Die Heuschrecke lebt am Stadtrand");

   zeile1.Ausgabe(cout);


   TEXTZEILE zeile2('*',10);

   zeile2.Ausgabe(cout);


   // Neu hinzugekommen gegenüber Beispiel 4.2.1.:

   cout << "Und nun ist zeile2=";

   zeile2 = zeile1;

   zeile2.Ausgabe(cout);


   return EXIT_SUCCESS;


} // end main

// end of file newdelete.cpp
```

---

[30] Hier werden aus Platzgründen nur die Änderungen gegenüber den Quelltexten aus Beispiel 4.2.1. aufgeführt.

Die Headerdatei für die Klasse `TEXTZEILE` hat sich wie folgt geändert:

```
// textzeile.h - Headerfile fuer die Klasse TEXTZEILE (gekürzt)

#ifndef _TEXTZEILE__

#define _TEXTZEILE__

// ...

class TEXTZEILE

{

   private:

      // ...

   public:

      TEXTZEILE& operator=(const TEXTZEILE&);  // Zuweisungsoperator

}; // end class TEXTZEILE

#endif

// end of file textzeile.h
```

Und die Implementierung der Klasse `TEXTZEILE` wurde um diesen Operator erweitert:

```
// textzeile.cpp


#include "textzeile.h"        // Headerdatei zur Klasse TEXTZEILE

#include <stdlib.h>

#include <string.h>

#include <iostream.h>


// ------------------------------------------------------------

// Public Methoden der Klasse TEXTZEILE

// ------------------------------------------------------------

// ...


TEXTZEILE& TEXTZEILE::operator=(const TEXTZEILE& other)

{

   if (this == &other)     // Für den Fall, daß jemand A=A; schreibt!

      return *this;        // (Dies kann auch indirekt via Referenzen

                           // geschehen!)

   delete zeile;

   laenge = other.laenge;

   stringcopy(other.zeile);

   return *this;

} // end TEXTZEILE& TEXTZEILE::operator=(const TEXTZEILE& other)

// end of file textzeile.cpp
```

Und schließlich noch das kurze Ablauflisting zu diesem Beispiel:

```
Die Heuschrecke lebt am Stadtrand (Laenge=33)

********** (Laenge=10)
```

```
Und nun ist zeile2=Die Heuschrecke lebt am Stadtrand (Laenge=33)
```

### 5.1.4. Beispiel: Überladener Operator `[]` und Referenzen

Das nachfolgende kleine Programm zeigt das Überladen des Operators []; gleichzeitig wird eine weitere Möglichkeit illustriert, die Referenzen bieten. Daneben wird ein sinnvolles Beispiel für den Einsatz statischer Klassenmitglieder gegeben.

In der hier gezeigten Klasse `DEMO` wird im Konstruktor dynamisch Speicherplatz allokiert, standardmäßig für drei `int`-Werte. Durch das Überladen des Array-Zugriffsoperators `[]` kann nun mit `demo[1]` auf die eigentlich gemeinte Komponente `demo.*data[3]` zugegriffen werden, d.h. nach außen erscheint ein Objekt demo der Klasse `DEMO` wie ein Feld von `int`-Werten. Die Klasse `DEMO` bietet jedoch den Vorteil, daß die Zugriffe auf die Komponenten geprüft erfolgen, eine Index-Überschreitung (oder -Unterschreitung) abgefangen werden kann.

```
// ----------------------------------------------------------------------
// returnref.cpp
// ----------------------------------------------------------------------
// Der überladene Operator [] gibt eine Referenz auf ein Array-Element
// zurück. Daher ist eine Schreibweise wie "demo[i]++" auch für ein Objekt
// demo der Klasse DEMO möglich.
// ----------------------------------------------------------------------

#include <iostream.h>
#include <stdlib.h>

class DEMO
{
    private:
        static const int MAX;
     int * data;
     int max;
   public:
     DEMO(int=3);
     ~DEMO();
     int& operator[](int i);
     void Print();
     int GetMax();
}; // end class DEMO

const int DEMO::MAX=10;       // Initialisierung des statischen Klassen-
                              // mitglieds
```

```
DEMO::DEMO(int anzahl)
{
    if (anzahl<=0 || anzahl>=MAX)
     anzahl=1; // Korrektur
    max=anzahl;
    cout << "data allokiert: " << max << " int-Plaetze" << endl;
    data = new int[max];
    for (int i=0; i<max; i++)
        data[i]=10*i;
} // end DEMO::DEMO(int)

DEMO::~DEMO()
{
    cout << "data deallokiert (max=" << max << ")" << endl;
    delete data;
} // end DEMO::~DEMO()
```

```
int& DEMO::operator[](int i)
{
    if (0<=i && i<max)
        return data[i];
    return data[0];      // im Fehlerfalle: willkürlich die Komponente 0
} // end int& DEMO::operator[](int)

void DEMO::Print()
{
    for (int j=0; j<max; j++)
        cout << "data[" << j << "]=" << data[j] << endl;
} // end void DEMO::Print(void)

int DEMO::GetMax()
{
    return max;
} // end int DEMO::GetMax(void)

int main()
{
    DEMO demo(8);      // Ein erstes Objekt wird angelegt;
    demo[3]++;         // das Datenelement Nr. 3 wird
    demo.Print();      // inkrementiert (um 1 erhöht).

    DEMO demo2;
    demo2[demo2.GetMax()]=999;
    demo2.Print();
    return EXIT_SUCCESS;
```

```
} // end main
// ---------------------------------------------------------------------------
// end of file returnref.cpp
// ---------------------------------------------------------------------------
```

Das Ablauflisting dieses kleinen Programms soll hier noch gezeigt werden:

```
data allokiert: 8 int-Plaetze
data[0]=0
data[1]=10
data[2]=20
data[3]=31              // ... wie oben geschildert: um 1 erhöht.
data[4]=40
data[5]=50
data[6]=60
data[7]=70
data allokiert: 3 int-Plaetze
data[0]=999
data[1]=10
data[2]=20
data deallokiert (max=3)
data deallokiert (max=8)
```

### 5.1.5. Überladen der Präfix- und Postfix-Inkrementoperatoren `operator++()` und `operator--()`

Während die frühen C++ Implementationen beim Überladen der Inkrementoperatoren `++` und `--` nur eine Realisierung zuließen, die dann sowohl die Präfix- als auch die Postfix-Notation umfaßte, werden vom modernen C++ Standard diese beiden Varianten unterschieden. Dabei akzeptieren die Compiler jedoch auch die alte Realisierung, geben hier jedoch in der Regel eine Warnung (*Overloaded prefix 'operator ++' used as a postfix operator*) aus.

Der Funktionskopf `void operator++()` kennzeichnet die Präfix-Operatorfunktion `++`; anstelle von vielleicht zu erwartenden Schlüsselworten `prefix` und `postfix`

wird demgegenüber die Postfix-Operatorfunktion `++` durch ein Dummy-Argument vom Typ int realisiert: `void operator++(int)`.

In Anhang A.1 wird die Beispielklasse `KTime` zur Zeit- und Datumsverwaltung vorgestellt; dort sind diese beiden Operatortypen implementiert. Nachstehend hiervon nur die relevanten Prototypen. Die Implementation findet sich auf Seite 0.

```
class KTime // Auszug

{

   public:

      KTime operator++();          // Präfix-Operator ++

      KTime operator--();          // Präfix-Operator --

      KTime operator++(int);       // Postfix-Operator ++

      KTime operator--(int);       // Postfix-Operator --

}; // end class KTime (Auszug)
```

## 5.2. Überladen als Freundfunktion

Das Klassenkonzept von C++ sieht Schutzmechanismen über die Schlüsselworte `private`, `protected` und `public` vor. Wie bereits erwähnt: `protected` wird erst beim Themengebiet Vererbung in Kapitel 8 eine Rolle spielen. Eine Ausnahme stellen die Freunde (*friends*) dar: das können einzelne befreundete Funktionen sein oder ganze Freund-Klassen. Hierauf soll in Kapitel 7 näher eingegangen werden, da interessante und in der Praxis übliche Anwendungen die in Kapitel 6 vorgestellten Streams nutzen.

Hier nur als Kostprobe ein kleines Beispiel, wo sich eine Freund-Funktion bei unserer Klasse `RATIONAL` sinnvoll einsetzen läßt.

```
class RATIONAL          // nur auszugsweise, vgl. Bsp. 3.1.1.
{
  private:
    int   z, n;         // Zähler, Nenner
    // ...
  public:
    RATIONAL();                  // Default-Konstruktor
    friend float operator=(float&,RATIONAL&); // Zuweisung an float
    // hier weitere Zugriffsfunktionen ...
}; // end class RATIONAL


// Implementation der Freund-Operatorfunktion float operator=():
float operator=(float& f, RATIONAL& r)
{
  f = float(r.z)/float(r.n);
  return f;
} // end float operator=(float& f, RATIONAL& r)
// ...
```

Beachten Sie: die Operatorfunktion =, die es ermöglicht, ein `RATIONAL`-Objekt, d.h. einen Bruch, an eine float-Variable zu übergeben, hat zwei Parameter: da es keine Klassenmethode ist, gibt es kein (`this`-)Objekt, an das die Funktion gebunden werden könnte, daher muß als erster Parameter eine Referenz auf float definiert werden. Der zweite Parameter ist dann eine Referenz auf ein Objekt der Klasse `RATIONAL`. Mit dieser Funktion kann nachstehendes Programmstück geschrieben werden, bei dem die float-Variable `f` den Wert `7` erhält.

```
float f;
RATIONAL r(77,11);
f=r;                // entspricht:  operator=(f,r);
```

Aber es sind (mit den Operatorfunktionen aus Abschnitt 5.1) auch die folgenden Anweisungen möglich.

```
float f;

RATIONAL p(1), q(3,5), r(7,8), s;

s=q+r;    // s.operator=(q.operator+(r));      s <- 59/40

f=p+s;    // operator=(f,p.operator+(s));      f <- 99/40
```

# 6. Streams I: Ein- und Ausgabe

In diesem Kapitel soll die Ein- und Ausgabe über die Standard-Streams `cin`, `cout`, `cerr` und `clog` behandelt werden. In Kapitel 10 wird das Themengebiet Streams dann erweitert um die allgemeine Arbeit mit Dateien (Datei-Streams). Hierzu sei auch auf die Übung 5 auf Seite 0 verwiesen.

## 6.1. Standard-Streams des Betriebssystems UNIX

Der Begriff *Stream* ist am besten mit Verbindung (oder UNIX-gemäß *Kanal*) zu beschreiben. Das Betriebssystem UNIX beispielsweise verwaltet Kanäle für die Standardein- und ausgabedateien (respektive -geräte) *stdin* (Kanal 0, i.d.R. verbunden mit der Tastatur), *stdout* (Kanal 1, i.d.R. verbunden mit dem Bildschirm) und *stderr* (Kanal 2, i.d.R. ebenfalls mit dem Bildschirm verbunden). Durch dieses Konzept wird eine logische Schicht über die physikalische Ebene gelegt: so kann ein Programm in der Form aufgerufen werden, daß die Fehler(kanal)meldungen zwar auf den Bildschirm ausgegeben werden, die reguläre (Standard-)Ausgabe jedoch in eine Datei umgelenkt wird. Unter UNIX sieht das etwa so aus:

```
$ program > program.ausgabe
```

Hiermit wird die (Standard-)Ausgabe des Programs `program` (über den Kanal 1) umgelenkt in die Datei `program.ausgabe`. Eventuelle Fehlermeldungen (auf Kanal 2) bleiben von dieser Umlenkung jedoch unberührt.

Mit dem entsprechenden Aufruf

```
$ program > program.ausgabe 2> program.errors < program.eingabe
```

holt sich das Programm seine Eingabedaten aus der Datei `program.eingabe`, seine normale Ausgabe wird in die bereits erwähnte Datei `program.ausgabe` umgelenkt, und die Fehlermeldungen purzeln in die Datei `program.errors`.

## 6.2. Standard-Ein- und -Ausgabe in ANSI-C

In ANSI-C werden für die Ein- und Ausgabe Funktionen aus `stdio.h` verwendet. Zum Beispiel dienen die Funktionen `scanf()` und `printf()` zum formatierten Einlesen bzw. Ausgeben.

Mit dem Eingabekanal 0 ist (vgl. die Headerdatei `stdio.h`) die Datei (bzw. der Datei-Pointer) *stdin* verbunden, mit den Ausgabekanälen 1 und 2 *stdout* bzw. *stderr*. (Daneben gibt es bei PC/DOS-basierten Compilern Datei-Pointer *stdprn* und *stdaux* für die erste parallele bzw. serielle Schnittstelle.)

Das folgende Beispiel zeigt einen Programmausschnitt in C, bei dem Variablen verschiedener Datentypen interaktiv eingelesen und anschließend auf den Bildschirm (präziser: auf stdout) ausgegeben werden.

```
#include <stdio.h>

double dblval;

int    intval;

char   charval;

char   str[80];

scanf("%f %i %c %s",&dblval,&intval,&charval,str);

printf("%f %i %c %s\n",dblval,intval,charval,str);
```

Zur Erinnerung: beachten Sie dabei, daß dort, wo eine Variablenbezeichnung nicht für eine Adresse (oder einen Pointer) steht, beim Aufruf der `scanf()`-Funktion die Adresse der Variablen angegeben werden muß!

## 6.3. Ein- und Ausgabe-Streams in C++

In C++ wird mit den Stream-Klassen gearbeitet. Dabei handelt es sich um eine Klassenhierarchie (vgl. Kapitel 8): ausgehend von einer Basisklasse `ios` werden verschiedene andere Klassen abgeleitet; an dieser Stelle von Interesse sind Objekte der Klassen `istream` (Eingabe), `ostream` (Ausgabe) und `iostream` (Ein-/Ausgabe). So ist das Objekt `cin` ein Objekt der Klasse `istream`, `cout` ein Objekt der Klasse `ostream`[31].

---

[31] Um ganz präzise zu sein: von den Klassen `ostream` bzw. `istream` werden von den Compilern die Klassen `ostream_with_assign` bzw. `istream_with_assign` abgeleitet, die über Zuweisungsoperatoren verfügen. Nachstehend die Deklarationen aus iostream.h von zwei Compilern.

```
// Symantec C++ 6.11 (DOS, Windows, OS/2, UNIX, XENIX, Macintosh)

extern istream_withassign cin;

extern ostream_withassign cout;

extern ostream_withassign cerr;

#if M_UNIX || M_XENIX

   extern ostream_withassign   clog;

#endif
```

In diesen Klassen sind die Schiebeoperatoren `>>` und `<<` überladen. Nachstehend einige typische Deklarationen (aus `iostream.h`): für alle vordefinierten Datentypen wird ein entsprechender Ausgabe-Operator festgelegt, dadurch muß bei Verwendung der Streams `cin` und `cout` auch keine Information zum jeweiligen Datentyp mitgegeben werden, wie das bei `printf()` der Fall ist.

```
// Auszug aus iostream.h des Symantec C++ 6.11 Compilers

    istream &operator>>(char *);

    istream &operator>>(signed char *s);

    istream &operator>>(unsigned char *s);

    istream &operator>>(char &);

    istream &operator>>(signed char &c);

    istream &operator>>(unsigned char &c);

    istream &operator>>(short &v);

    istream &operator>>(int &v);

    istream &operator>>(long &v);

    istream &operator>>(unsigned short &v);

    istream &operator>>(unsigned int &v);

    istream &operator>>(unsigned long &v);

    istream &operator>>(float &);

    istream &operator>>(double &);

#if macintosh

    istream &operator>>(long double &);

#endif
```

---

```
// HP-UX 9.04 (UNIX)

extern istream_withassign cin;

extern ostream_withassign cout;

extern ostream_withassign cerr;

extern ostream_withassign clog;
```

Allerdings ist die Verwendung der C++-Streams nicht zwingend, die C-gemäßen Ein- und Ausgabe-Funktionen stehen selbstverständlich auch in C++ zur Verfügung, und es gibt viele von C kommende C++-Programmierer, die die Verwendung eines zünftigen `printf()` dem „neumodischen" `cout`-Geschiebe vorziehen. Die Vorzüge der Streams werden vermutlich erst so richtig im Zusammenhang mit Vererbung (Kapitel 8) deutlich werden.

Das im vorherigen Abschnitt gezeigte Programmstück kann in C++ mit Streams formuliert so aussehen.

```
#include <iostream.h>    // Headerdatei für die I/O-Streams[32]

double dblval;

int   intval;

char  charval;

char  str[80];

cin >> dblval >> intval >> charval >> str;

cout << dblval << intval << charval << str << endl;
```

Hier wird automatisch der Schiebeoperator in der jeweiligen überladenen Version für die verschiedenen Datentypen aufgerufen. Auf den Aspekt der möglichen Formatierungen soll an dieser Stelle lediglich in Form des folgenden Beispiels eingegangen werden. Dies geschieht in C++ mittels sogenannter *Manipulatoren*, die über die Header-Datei `iomanip.h` eingebunden werden können. Hiermit stehen dann die von `printf()` bekannten Formatierungsmöglichkeiten auch bei den C++-Streams zur Verfügung. Das hier gezeigte `endl` ist ein solcher (bereits in `iostream.h` deklarierter) Manipulator, der für die Ausgabe eines Zeilenvorschubs (`'\n'`) sorgt.

Hierzu ein kleines Beispielprogramm zur Illustration.

```
// iomanipdemo.cpp - Demonstration der C++ IO-Manipulatoren
// (siehe Headerdatei iostream.h)

#include <iostream.h>
#include <iomanip.h>             // wegen setiosflags()
#include <stdlib.h>

int main()
{
   cout << "iomanipdemo.cpp - Demonstration der Manipulatoren" << endl;

   // Ausgabe in verschiedenen Zahlformaten:
   cout << endl;
   cout << "Ausgabe einer int-Zahl in verschiedenen Formaten:" << endl;
```

---

[32] Bei manchen Compilern ist dies die Headerdatei iostream.hpp!

```
int wert1 = 35, wert2=26;
cout << " dezimal:              "
   << wert1 << " " << wert2 << endl
   << " hexadezimal:          "
   << hex << wert1 << " " << wert2 << endl
   << setiosflags(ios::uppercase)
   << " hexadezimal uppercase: "
   << hex << wert1 << " " << wert2 << endl
   << " oktal:                "
   << oct << wert1 << " " << wert2
   << endl;

// setiosflags: Formatierung der Ausgabe (exemplarisch)
const float PI = 3.1415926;
cout << endl
   << "PI (scientific, showpos): "
   << setiosflags(ios::scientific | ios::showpos)
   << PI << endl
   << "PI (fixed default):       "
   << setiosflags(ios::fixed)
   << PI << endl
   << "PI (fixed precision 3):   "
   << setprecision(3)
   << setiosflags(ios::fixed)
   << PI << endl;
cout << "PI (fixed width 5):       ";
cout << setw(5);
cout << setiosflags(ios::fixed)
   << PI << endl;
cout << "PI (fixed width 8):       ";
cout << setw(8);
cout << PI << endl;
cout << "PI (fixed width 12):      ";
cout << setw(12);
cout << setiosflags(ios::fixed)
   << PI << endl;
cout << "PI (fixed setfill '*'):   ";
cout << setw(12) << setfill('*');
cout << setiosflags(ios::fixed)
   << PI << endl << endl;
cout << "Nun wird resetioflags(ios::showpos) aufgerufen..." << endl;
cout << "Die Anzeige erfolgt nun wieder ohne Vorzeichen:" << endl;
cout << "PI (fixed setfill '*'):   "
      << setw(12) << setfill('*')
      << resetiosflags(ios::showpos | ios::dec);
cout << setiosflags(ios::fixed) << PI << endl;

// Nun wird auf "dezimal" zurueckgeschaltet und mit den Fuellzeichen,
// der Ausgabeweite und den Attributen "links" und "rechts"
// gearbeitet.
```

```
    cout << dec << setfill('#') << endl;
    cout << setiosflags(ios::left)  << setw(22) << " abc123 "
       << setw(12) << 123
       << " " << 1200.45 << " " << 2345 << endl;
    cout << setiosflags(ios::right) << setw(22) << " abc123 "
       << setw(12) << setfill('.') << 123
       << " " << 1200.45 << " " << 2345 << endl;
    return EXIT_SUCCESS;
} // end main
// end of file iomanipdemo.cpp
```

Und das zugehörige Ablauflisting:

```
iomanipdemo.cpp - Demonstration der Manipulatoren

Ausgabe einer int-Zahl in verschiedenen Formaten:
 dezimal:               35 26
 hexadezimal:           23 1a
 hexadezimal uppercase: 23 1A
 oktal:                 43 32

PI (scientific, showpos): +3.141593E+000
PI (fixed default):       +3.14159
PI (fixed precision 3):   +3.14
PI (fixed width 5):       +3.14
PI (fixed width 8):          +3.14
PI (fixed width 12):             +3.14
PI (fixed setfill '*'):   *******+3.14

Nun wird resetioflags(ios::showpos) aufgerufen...
Die Anzeige erfolgt nun wieder ohne Vorzeichen:
PI (fixed setfill '*'):   ********3.14

 abc123 ##############123######### 1.2E+003 2345
 abc123 ##############123......... 1.2E+003 2345
```

Eine Anmerkung zu `cout`, `cerr` und `clog`: Wie bereits erwähnt ist `cout` mit dem Standardausgabekanal 1 (UNIX) verbunden, führt also i.d.R. zu einer Ausgabe auf den Bildschirm. Ebenso ist `cerr` (der Fehlerkanal 2) üblicherweise mit dem Bildschirm verbunden. Der dritte vordefinierte Ausgabekanal, `clog`, ist ebenfalls an den Fehlerkanal gekoppelt, puffert die Ausgaben jedoch und schreibt erst dann auf das Gerät (bzw. in die Datei), wenn der Puffer voll ist oder explizit geleert wird. Hierzu gibt es in ANSI-C kein Analogon.

## 6.4. Überladen der Schiebeoperatoren

Die Schiebeoperatoren `<<` und `>>` eignen sich hervorragend für das Overloading: arbeitet man mit einer eigenen Klasse, so sollten Objekte dieser Klasse üblicherweise auch mit dem `cin`/`cout`-Mechanismus eingelesen bzw. ausgegeben werden können.

## 6.4.1. Beispiel: Klasse RATIONAL: Schiebeoperatoren zur Ein- undAusgabe

Betrachten wir unsere treue Begleiterin, die Klasse `RATIONAL`. Dann kann der Operator `>>` für die Eingabe über ein `istream`-Objekt (z.B. `cin`) so aussehen.

```
// Prototyp innerhalb der Klassendeklaration als Freund-Funktionen

  friend istream& operator>>(istream&, RATIONAL&);



// Implementation (Skizze)

istream& operator>>(istream& in, RATIONAL & r)

{

  return (in >> r.z >> r.n);

  // Spartanische Eingabe eines Bruches als zwei Ganzzahlen.

  // Aus Platzgründen wird hier keine Sicherheitsüberprüfung

  // und kein Kürzen durchgeführt.

} // end operator>>
```

Mit dem Programmausschnitt

```
RATIONAL p;

cin >>  p;
```

wird nun erwartungsgemäß der Bruch `p` (zwei `int`-Werte) von Tastatur eingelesen.

Entsprechend funktioniert die Ausgabe mit einem überladenen Freund-Operator `<<`.

```
// Prototyp innerhalb der Klassendeklaration als Freund-Funktionen

  friend ostream& operator<<(ostream&, const RATIONAL&);



// Implementation

ostream& operator<<(ostream& out, const RATIONAL& r)

{

  return (out << r.z << "/" << r.n);

  // Ausgabe eines Bruches in der Form  z/n, z.B. 1/2 .
```

```
} // end operator<<
```

Mit dem Programmausschnitt

```
RATIONAL p(2,5);

cout << p;
```

wird nun erwartungsgemäß der Wert `2/5` auf dem Bildschirm ausgegeben.

Diese Seite wurde absichtlich leer gelassen.

# 7. Freunde

Freunden vertraut man. In C++ vertraut eine Klasse ihren Freunden den Zugriff auf private Mitgliedsdaten (oder -funktionen) an. Freunde können einzelne Funktionen, auch Operatorfunktionen, oder ganze Klassen sein[33]. Eine Klasse muß in ihrer Deklaration explizit erklären, wer ihre Freunde (*friends*) sein sollen, es kann sich niemand außerhalb einer Klasse zum Freund dieser Klasse erklären. Das wäre auch noch schöner...

## 7.1. Befreundete Funktionen

Es sei eine Klasse `X` definiert wie folgt.

```
// Deklaration

class X

{

   private:

      int intern;

   public:

      void Read(void);

      void Write(void);

      friend void FromTheOuterWorld(X &);

}; // end class X


// Implementation

void X::Read(void)       // Mitgliedsfunktion der Klasse X

{

   cin >> intern;

} // end X::Read


void X::Write(void)      // Mitgliedsfunktion der Klasse X
```

[33] Eine Stilrichtlinie (guideline) von C++ besagt jedoch, daß man nur in Notfällen ganze Klassen zu Freunden erklären sollte!

```
{
   cout << "intern=" << intern << endl;
} // end X::Write


void FromTheOuterWorld(X & x)        // Freundfunktion
{
   cout << "Exklusiv für Sie: der Zugriff auf ein privates"
      << " Klassenobjekt! " << endl
      << "Wie wollen Sie die private Variable"
      << " intern setzen? " << endl << "-> ";
   cin >> x.intern;            // erlaubter Zugriff!!!
   cout << "Danke." << endl;
} // end FromTheOuterWorld
```

Das Ablauflisting zu dem kleinen Hauptprogramm wird nachfolgend ebenfalls abgedruckt.

```
int main()          // Hauptprogramm zu den obigen Definitionen
{
  X x;

  FromTheOuterWorld(x);
  x.Write();

  return 0;
} // end main
```

Das Ablauflisting:

```
Exklusiv für Sie: der Zugriff auf ein privates Klassenobjekt!
Wie wollen Sie die private Variable intern setzen?
-> 77
Danke.
intern=77
```

## 7.2. Überladene Freundfunktionen

In Abschnitt 5.2. wurde bereits ein erstes Beispiel einer mit einer Klasse befreundeten überladenen Operatorfunktion vorgestellt, nämlich ein Zuweisungsoperator eines `RATIONAL`-Objektes an eine `float`-Variable. In Abschnitt 6.4. wurde gezeigt, wie die Schiebeoperatoren sinnvoll überladen werden können, damit ein Objekt einer neu definierten Klasse ebenfalls via `cin` eingegeben und mittels `cout` ausgegeben werden kann; auch hier war es erforderlich, wegen des Zugriffs auf private Klassen- bzw. Objektdaten diese Operatorfunktionen als Freunde zu deklarieren.

Wichtiger Hinweis: Bei einer Freundfunktion muß mindestens ein Parameter der betreffenden Klasse vorhanden sein, die diese Funktion zu ihrem Freund erklärt!

### 7.2.1. Beispiel: Klasse RATIONAL — Rechenoperationen

Sinnvoll ist es sicherlich, in unserer Klasse `RATIONAL` die gewohnten Rechenoperatoren zu implementieren. Um den geneigten Leser bei der Stange zu halten, sollen hier nur sehr einfache Rechenoperationen umgesetzt werden[34].

```
// Innerhalb der Klassendeklaration

   RATIONAL  operator+(RATIONAL &);      // Additionsoperatoren in

   RATIONAL  operator+(int);                      // verschiedenen überladenen

   friend RATIONAL operator+(int, RATIONAL &);// Versionen
```

[34] Der mathematisch ambitionierte Leser findet im einschlägigen Fachbuchhandel zahlreiche Werke der Art "Mathematische Problemlösungen mit C und C++", "Numerische Lösungen in C++" oder "Solving Algorithmic, Numerical, or Mathematical Problems with or without C++".

```
// Implementationen
RATIONAL RATIONAL::operator+(RATIONAL & r)
{
   RATIONAL tmp;
   tmp.z = z*q.n+n*q.z;
   tmp.n = q.n*n;
   tmp.Kuerzen();
   return tmp;
} // end RATIONAL RATIONAL::operator+(RATIONAL & r)

RATIONAL RATIONAL::operator+(int i)
{
   RATIONAL tmp;
   tmp.z = z+n*i;
   tmp.n = n;
   tmp.Kuerzen();
   return tmp;
} // end RATIONAL RATIONAL::operator+(int i)

RATIONAL operator+(int i, RATIONAL & r)
{
   RATIONAL tmp;
   tmp.z = r.n*i+r.z;
   tmp.n = r.n;
   tmp.Kuerzen();
   return tmp;
} // end Freund-Operatorfunktion RATIONAL operator+(int i, RATIONAL &)
```

Nachstehend dazu noch ein kleines Hauptprogramm und das zugehörige Ablauflisting.

```
int main()
{
  RATIONAL p(3,4), q(12,10);
  int i=5;
  cout << "p= " << p << endl;
  cout << "q= " << q << endl;
  cout << "i= " << i << endl;
  cout << "p+q= " << p + q << endl;
  cout << "p+i= " << p + i << endl;
  cout << "i+p= " << i + p << endl;
  return EXIT_SUCCESS;
} // end main
```

Das Ablauflisting:

```
p= 3/4
q= 6/5
i= 5
p+q= 39/20 // verwendet wird: RATIONAL RATIONAL::operator+(RATIONAL & r)
p+i= 23/4  // verwendet wird: RATIONAL RATIONAL::operator+(int i)
i+p= 23/4  // verwendet wird: RATIONAL operator+(int i, RATIONAL & r)
```

## 7.3. Befreundete Klassen

Klassen können nicht nur einzelne Funktionen zu ihren Freunden erklären, sie können sich auch mit ganzen Klassen anfreunden. Dies wird beispielsweise in den I/O-Stream-Klassen gemacht. Es ist aufgrund der Hierarchiebildung der jeweiligen Compiler nicht vollkommen trivial zu durchschauen, aber der interessierte Leser kann sich unter UNIX in `/usr/include/CC` oder unter DOS/Windows im include-Verzeichnis des jeweiligen C++-Compilers gelegentlich einmal einige Headerdateien ansehen.

Hier soll zunächst ein einfaches Beispiel den Grundsachverhalt erläutern.

### 7.3.1. Beispiel: Eine Freund-Klasse

Betrachten wir die beiden im folgenden Programm deklarierten Klassen `FRIEND` und `PERSON`. Die Klasse `PERSON` erklärt die gesamte Klasse `FRIEND` zu ihrem Freund; damit können sämtliche Methoden der Klasse `FRIEND` auch auf die als `private` (und ggf. als `protected`) deklarierten Daten (und evtl. Funktionen) der Klasse `PERSON` zugreifen.

```
// afriendclass.cpp

#include <iostream.h>

#include <stdlib.h>

#include <string.h>



// Deklarationen

class FRIEND;      // compilerabhängig: Microsoft Visual C++ erwartet

              // hier zwingend eine solche Forward-Deklaration.

class PERSON

{

   friend FRIEND;      // Die Klasse FRIEND wird zum Freund erklärt.

              // Das kann überall in der Klassendeklaration

              // stehen, der guten Übersichtlichkeit halber

              // ist es sinnvoll, dies ganz zu Beginn zu

              // formulieren.

   private:

      char  name[80];

      float gehalt;

      long  sozialversicherungsnummer;

   public:

      PERSON(char [80], float = 0.0F, long = 0);

      // ...

}; // end class PERSON
```

```
// Implementationen

PERSON::PERSON(char s[80], float x, long i)
{
   strcpy(name,s);
   gehalt=x;
   sozialversicherungsnummer=i;
} // end Konstruktor PERSON::PERSON(char [80], float, long)


class FRIEND
{
   public:
      void VerrateMir(PERSON&);
}; // end class FRIEND


void FRIEND::VerrateMir(PERSON& person)
{
   cout << "Das Gehalt von " << person.name <<
      " beträgt DM " << person.gehalt << "!" << endl;
} // end void FRIEND::VerrateMir(PERSON& person)


int main()
{
   PERSON FrancoisMitterand("François Mitterand",200000);
   FRIEND JacquesChirac;
   JacquesChirac.VerrateMir(FrancoisMitterand);
   return EXIT_SUCCESS;
} // end main
```

Das Programm teilt uns mit, was in der als `private` geschützten Instanzvariable `gehalt` des Objektes `FrancoisMitterand` steht:

```
Das Gehalt von François Mitterand beträgt DM 200000!
```

Ob und wann es sinnvoll ist, über die Freund-Deklaration die Wirkung der Schutzbereiche `private` und `protected` außer Kraft zu setzen, muß jeder für sich entscheiden. Wo möglich, ist das Verwenden einer Mitgliedsfunktion sicherlich sinnvoller als das Konstruieren einer Freund-Funktion.

Im Zusammenhang mit Vererbung (im nächsten Kapitel) wird auch ein weiterer Mechanismus vorgestellt werden, wie eine Klasse Daten (und Funktionen) einer anderen Klasse nutzen kann.

### 7.3.2. Einige Spielregeln

Bjarne Stroustrup hat einige Spielregeln aufgestellt, wann Freundfunktionen bzw. Klassenfunktionen verwendet werden können oder sollen.

Zwangsläufig müssen Klassenmitgliedfunktionen sein: die Konstruktoren, die Destruktoren sowie die später noch vorzustellenden virtuellen Funktionen.

Operationen, die den inneren Zustand eines Objektes verändern, sollten nach Möglichkeit Elementfunktionen der Klasse sein.

Ebenso sollten Operatorfunktionen, die bei elementaren Datentypen *lvalues*[35] erfordern (beispielsweise die Inkrementierung `operator++()` oder die Zuweisung `operator=()`), als Mitgliedfunktionen implementiert werden.

Operatoren, bei denen ein implizites Casting (automatische Typumwandlung) erwünscht ist, sollten als globale Freundfunktion mit `const&`-Parametern (oder Argumenten ohne Referenz) geschrieben werden. Ein Beispiel hierfür ist der Additionsoperator `operator+`.

Und noch ein (triviales) Muß: Operatoren müssen globale friend-Funktionen sein, wenn sie nicht als Elementfunktionen implementiert werden können. Der Schiebeoperator `>>` zum Beispiel kann nicht (als Elementfunktion der Klasse `istream`) geändert werden, da i.a. der Source Code dieser Klasse nicht verfügbar ist.

### 7.3.3. Beispiel: Die Klasse `complex`

Im bisherigen Quasi-Standard AT&T C++ ist eine Klasse `complex` für die Arbeit mit komplexen Zahlen vorgesehen. Nachstehend sei nur auszugsweise gezeigt, wie es dort von `friend`-Deklarationen wimmelt.

```
class complex          // auszugsweise, hier aus dem Include-File
{                      // des Symantec C++ 6.11 Compilers
   friend double real ( const complex& );  // Type Casting Operatoren
   friend double imag ( const complex& );
   friend complex cos ( const complex& );
   friend complex sin ( const complex& );
   friend complex tan ( const complex& );
   friend complex log ( const complex& );
   // ... und noch viele weitere ...
```

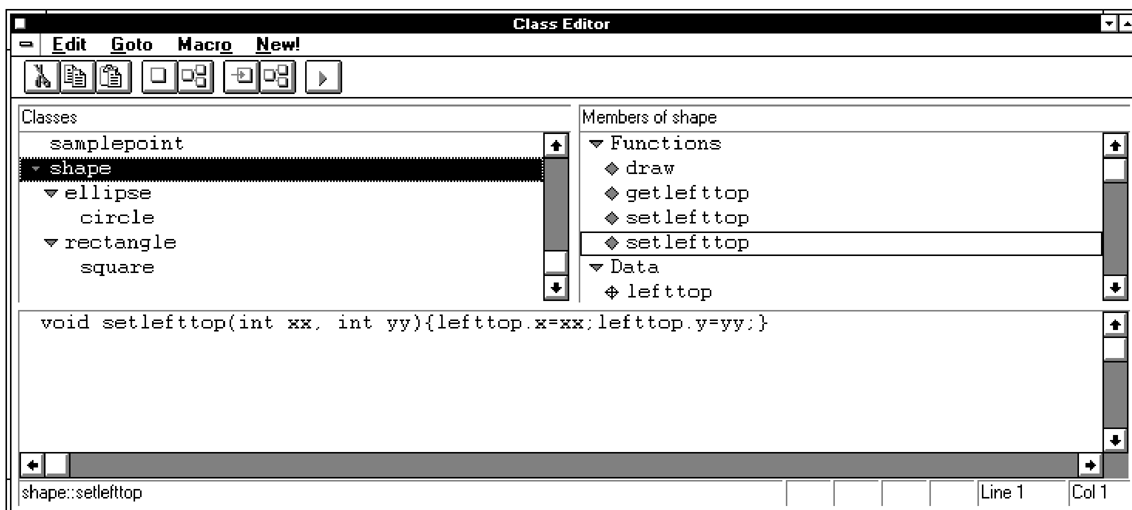
[35] *lvalues* sind Ausdrücke, die auf der linken Seite (*left side values*) einer Zuweisung stehen können.

```
  friend complex sqrt ( const complex& );
  friend double abs ( const complex& );
  friend complex conj ( const complex& );
  friend double norm ( const complex& );
  // Operatoren
  friend complex operator + ( const complex&, const complex& );
  friend complex operator + ( double, const complex& );
  friend complex operator + ( const complex&, double );
  friend complex operator - ( const complex&, const complex& );
  friend complex operator - ( double, const complex& );
  friend complex operator - ( const complex&, double );
  friend complex operator * ( const complex&, const complex& );
  friend complex operator * ( double, const complex& );
  friend complex operator * ( const complex&, double );
  friend complex operator / ( const complex&, const complex& );
  friend complex operator / ( double, const complex& );
  friend complex operator / ( const complex&, double );
  friend int operator != ( const complex&, const complex& );
  friend int operator != ( double, const complex& );
  friend int operator != ( const complex&, double );
  friend int operator == ( const complex&, const complex& );
  friend int operator == ( double, const complex& );
  friend int operator == ( const complex&, double );
  friend ostream& operator << ( ostream& s, const complex& x);
  friend istream& operator >> ( istream& s, complex& x);

private:
  double re;
```

```
    double im;

  public:
    complex ( );
    // ...
};
```

# 8. Vererbungslehre: Abgeleitete Klassen

Vererbung bedeutet, daß eine Klasse von einer (oder später auch von mehreren) Klasse(n) abgeleitet wird; dies drückt eine Spezialisierung der Eigenschaften aus. Zunächst einmal werden (mit noch zu besprechenden Ausnahmen) die Datenelemente und die Methoden der sogenannten Superklasse oder Basisklasse (Elternklasse, *base class*) an die abgeleitete Klasse (Subklasse, Kindklasse, *derived class*) weitergegeben. Daneben können nun in der Subklasse beliebig weitere Elemente (Daten und Funktionen) ergänzt werden; es können aber auch vorhandene Methoden überschrieben werden.




Obiges Bild zeigt einen *Class Hierarchy Browser*, den *Symantec C++ 7.2 Class Editor*. Hier wird gerade angezeigt, daß in der Klasse `shape` eine Methode (Mitgliedfunktion) `setlefttop()` existiert, deren Code im unteren Fenster dargestellt wird.

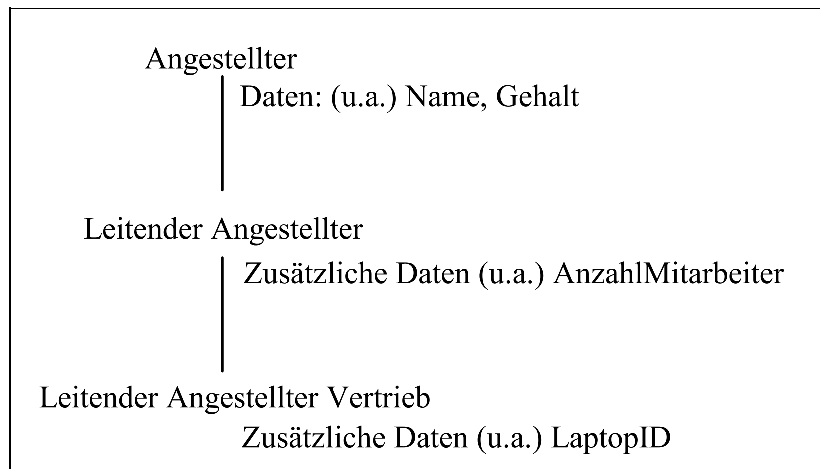
## 8.1. Vererbung und die Wiederverwendbarkeit von Code

Sinn und Zweck der Vererbung ist im wesentlichen, im Software-Entwicklungsprozeß zunehmend wiederverwendbare Komponenten (*reusable components*) einzusetzen.

Wurden bereits Datenelemente und Routinen zu einem Objekt *Angestellter* implementiert, so muß für ein Objekt *LeitenderAngestellter* nur doch das ergänzt (oder modifiziert) werden, was für leitende Angestellte erforderlich ist.
Und ein Objekt einer dritten Klasse *LeitenderAngestellterVertrieb* schließlich hat vielleicht nur noch sehr wenige neu hinzukommende Komponenten, z.B. Daten zu seinem dienstlichen Laptop oder seinem (für die Abteilung Vertrieb spezifischen) Dienstwagen...

Im nachstehenden Diagramm hätte ein Angestellter beispielsweise die Datenelemente Name und Gehalt, ein leitender Angestellter die Datenelemente Name, Gehalt und

AnzahlMitarbeiter, und ein leitender Angestellter im Vertrieb würde über die Datenelemente Name, Gehalt, AnzahlMitarbeiter und LaptopID verfügen.




Sind in der Klasse *Angestellter* bereits Routinen implementiert, wie z.B. Name und Gehalt von Tastatur einzulesen sind, dann vererbt sich dies auf die abgeleiteten Klassen, so daß für einen leitenden Angestellten nur noch gesagt werden muß, was das Datenelement AnzahlMitarbeiter sein soll.

## 8.2. Vererbungsarten: private, protected und public

In C++ werden verschiedene Vererbungsarten unterschieden und angeboten. (Vgl. hierzu die allgemeinen Erläuterungen in Abschnitt 1.3.2.) Hierbei wird nach den Schutzbereichen differenziert: `private` und `public` sind bereits bekannt.
Der Schutzbereich `private` klassifiziert Daten und ggf. auch Methoden, die nur von Objekten der eigenen Klasse angesprochen werden können. Infolgedessen werden die Elemente aus dem `private`-Bereich für die „Nachkommen" gar nicht sichtbar, existieren aber, wie man durch eine einfache sizeof-Abfrage verifizieren kann. Der „Schutz"bereich `public` gestattet dagegen der ganzen Welt freien Zugriff auf die Funktionen und ggf. Daten. Neu im Kontext der Vererbung ist `protected`: dieser Schutzbereich verhält sich im wesentlichen wie `private`, lediglich Objekte von abgeleiteten Klassen dürfen hierauf ebenfalls zugreifen.

Der im allgemeinen übliche Fall ist die sogenannte `public`-Vererbung, d.h. der `protected`-Bereich der Superklasse wird in den `protected`-Bereich der abgeleiteten Klasse, der `public`-Bereich in den entsprechenden `public`-Bereich der Subklasse vererbt.

Dies wollen wir uns konkret an einem kleinen Demonstrationsbeispiel ansehen.

### 8.2.1. Beispiel: Elementare public-Vererbung

Betrachten wir im folgenden Beispiel zwei Klassen `X` (Superklasse) und `Y` (Subklasse).

```
// -----------------------------------------------------------------------
// inheritance1.cpp - Kleines Demonstrationsprogramm zur public-Vererbung
// -----------------------------------------------------------------------
#include <iostream.h>
#include <stdlib.h>
// Deklaration und Implementation der Basisklasse X ---------------------
class X
{
  private:
     int ipriv;          // privates Datenelement ipriv
  protected:
     int iprot;          // protected-Datenelement iprot
  public:
     int ipubl;          // öffentliches Datenelement ipubl
     X();                // Default-Konstruktor
     ~X();               // Destruktor
     void AnyFunc();     // Irgendeine weitere Funktion
}; // end class X

X::X()
{
  ipriv=101;
  iprot=202;
  ipubl=303;
  cout << "X-Konstruktor aufgerufen." << endl;
```

```
} // end X::X()
X::~X()
{
  cout << "X-Destruktor aufgerufen." << endl;
} // end X::~X()
void X::AnyFunc()
{
  cout << "ich bin irgendeine Funktion aus X" << endl;
} // end X::AnyFunc(void)

// Deklaration und Implementation der Subklasse Y -----------------------
class Y : public X          // Die Klasse X wird public vererbt, d.h.
{                           // die public-Elemente von X landen in der
                            // public-Sektion von Y, entsprechend werden
                            // die protected-Daten und -Funktionen in die
                            // protected-Sektion von Y vererbt.
  public:
     Y();                   // Default-Konstruktor
     ~Y();                  // Destruktor
     void Show();
}; // end class Y
// Ein Objekt der Klasse Y verfügt also über die Mitgliedsdaten iprot
// (protected) und ipubl (public) sowie den Konstruktor, den Destruktor
// und die Elementfunktionen AnyFunc() [von X geerbt] und Show().

Y::Y()
{
  cout << "Y-Konstruktor aufgerufen." << endl;
```

```
} // end Y::Y()

Y::~Y()
{
  cout << "Y-Destruktor aufgerufen." << endl;
} // end Y::~Y()

void Y::Show()
{
  // cout << "ipriv=" << ipriv << endl; // ipriv ist in Y nicht verfügbar!
  cout << "iprot=" << iprot << endl;
  cout << "ipubl=" << ipubl << endl;
} // end Y::Show(void)

int main()
{
  X x;
  Y y;
  y.AnyFunc();            // beispielhafte Funktionsaufrufe
  y.Show();
  return EXIT_SUCCESS;
} // end main
```

Nachfolgend noch das Ablauflisting dieses Programms.

`X-Konstruktor aufgerufen.` - ausgelöst durch die Deklaration X x;

`X-Konstruktor aufgerufen.` - ausgelöst durch die Deklaration Y y;

`Y-Konstruktor aufgerufen.` - ebenfalls ausgelöst durch die Deklaration Y y;

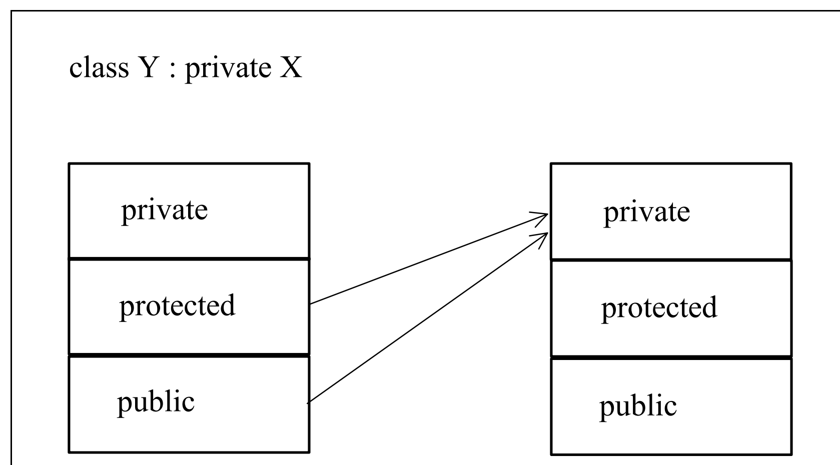
`ich bin irgendeine Funktion aus X` - y.AnyFunc()

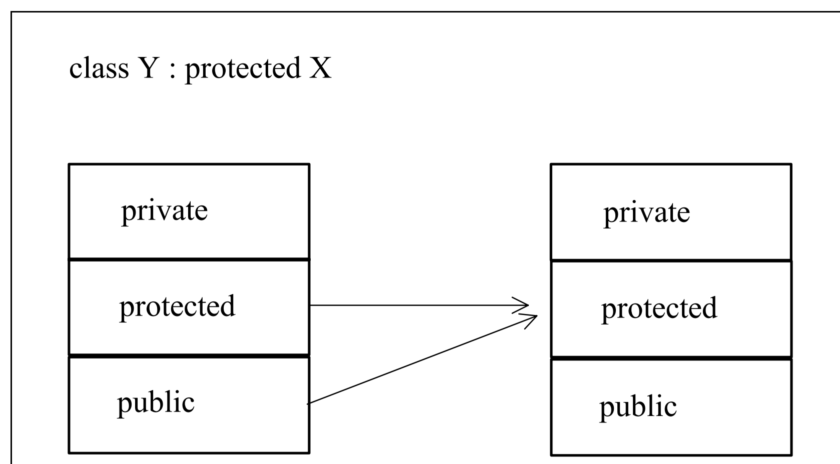
| | |
|---|---|
| `iprot=202` | - y.Show() |
| `ipubl=303` | - y.Show() |
| `Y-Destruktor aufgerufen.` | - Ende der Lebensdauer von y |
| `X-Destruktor aufgerufen.` | - Ende der Lebensdauer von y |
| `X-Destruktor aufgerufen.` | - Ende der Lebensdauer von x |

Vererbung ist ein sehr praktisches Hilfsmittel auf dem Weg, Software-Bausteine zu entwickeln, die ähnlich wie Fertigteile in der klassischen industriellen Produktion sehr leicht aneinandergefügt, hinzugesetzt und wieder weggenommen werden können. Wird in der obigen Klasse `X` ein weiteres Datenelement `identifikationsnummer` aufgenommen, so ist dieses automatisch auch in allen Objekten der Klasse `Y` vorhanden. Wird die Variable `identifikationsnummer` im `X`-Konstruktor gesetzt, so muß dies auch nicht mehr in `Y` wiederholt werden. Der softwaretechnologische Aufwand einer solchen Änderung (hier Erweiterung) fällt somit relativ bescheiden aus.

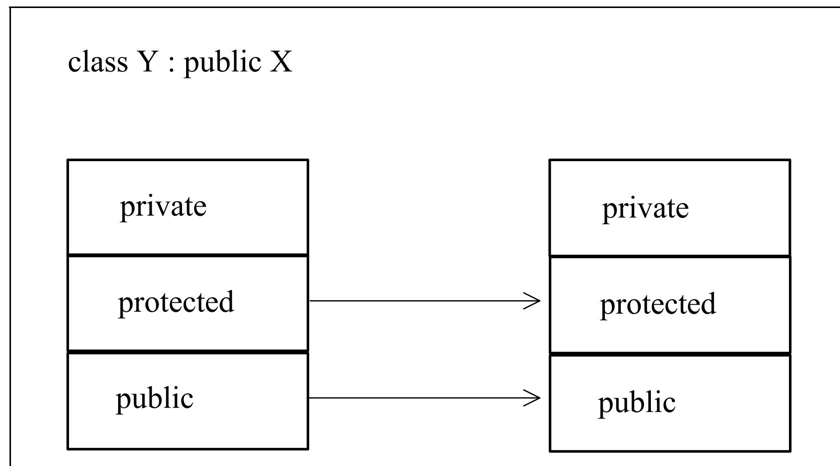
## 8.2.2. Übersicht über die Vererbungsmöglichkeiten in C++

Die nachstehenden Diagramme erläutern die Vererbungsmöglichkeiten mit den drei Schlüsselworten (Schutzbereichspezifizierern) `private`, `protected` und `public`. In der Praxis werden Sie jedoch fast immer die `public`-Vererbung vorfinden[36].




[36] Daß bei diesen Diagrammen vom private-Block der Klasse X kein Pfeil ausgeht soll nur besagen, daß die dort deklarierten Elemente in der abgeleiteten Klasse Y nicht (direkt) zugreifbar sind; physisch vorhanden sind sie gleichwohl!




Diese Diagramme sind so zu lesen, daß z.B. bei einer `private`-Vererbung die Datenelemente der Basisklasse aus den Bereichen `protected` und `public` in den `private`-Bereich der Subklasse übernommen werden.

## 8.2.3. Anmerkung: Konstruktor- und Destruktoraufrufe in Subklassen

Abgeleitete Klassen erben bis auf die Konstruktoren und die Destruktoren alle Elementfunktionen[37]. Demgegenüber rufen die jeweiligen Konstruktoren und Destruktoren der abgeleiteten Klassen (per Voreinstellung) die (Default-)Konstruktoren und -Destruktoren der Basisklassen auf, wie im Listing des obigen Beispiels 8.2.1. bereits zu sehen war.

Dies ist deswegen sinnvoll, weil in Konstruktoren in der Regel Variablen initialisiert werden und eventuell Speicherplatz allokiert wird. So werden zuerst die Datenelemente der Superklasse angelegt bzw. mit sinnvollen Ausgangswerten initialisiert, bevor dann durch den subklasseneigenen Konstruktor die in der abgeleiteten Klasse neu hinzugekommenen Daten ihre Startwerte oder ihren Speicherplatz zugewiesen bekommen.

Die Voreinstellung, daß implizit die Standard-Konstruktoren und -Destruktoren der Superklasse aufgerufen werden, kann durch Angabe des gewünschten Konstruktors in der Liste der sogenannten *Element-Initialisierer* im Funktionskopf geändert werden.

### 8.2.3.1. Beispiel

Ergänzen wir in Beispiel 8.2.1. die Klasse `X` um einen zweiten Konstruktor `X::X(int i);` so kann die Konstruktordeklaration in `Y` geändert werden wie folgt.

---

[37] Lediglich die Operatorfunktion `operator=()` wird ebenfalls nicht weitervererbt: da eine abgeleitete Klasse in der Regel über weitere Datenelemente gegenüber der Superklasse verfügt, wäre ein solcher vererbter Zuweisungsoperator nur eine partielle Zuweisung!

```
Y::Y() : X(4711)            // sog. Element-Initialisierer

{

  cout << "Y-Konstruktor aufgerufen." << endl;

} // end Y::Y()
```

Nun wird statt des Default-Konstruktors `X::X()` der Konstruktor `X::X(int)` mit dem aktuellen Parameter `i=4711` angesprochen.

#### 8.2.3.2. Beispiel: Vererbungsprinzipien anhand von Graphikobjekten

Nachstehend sehen Sie das etwas umfangreichere Beispielprogramm graphobj.cpp, das auf dem PC mit dem DOS-basierten Borland Turbo C++ Compiler implementiert wurde[38)].

Es illustriert das Vererbungsprinzip anhand der recht trivialen Graphikobjekte Kreis (`CIRCLE`) und Rechteck (`RECTANGLE`) und verwendet, der Einfachheit halber, die Borland Turbo C/C++ Graphikroutinen. Selbstverständlich ließe sich dasselbe Beispiel auch unter dem X Window System implementieren.

```
// ------------------------------------------------------------------------
// Projekt:      Kleines Beispiel zur Vererbung (Inheritance)
//               am Beispiel einfacher Graphikobjekte
// ------------------------------------------------------------------------
// Dateiname:    graphobj.cpp
// Bemerkungen:  Hier wird der Einfachheit halber nur ein 640x480-
//               Display im Graphikmodus bzw. ein Textbildschirm
//               implementiert.
// Compiler:     Die Graphikausgaben sind Borland-spezifisch implemen-
//               tiert; die textorientierten Ausgaben sind allgemein
//               gehalten.
// Besonderes:   Hinzulinken der Graphikbibliothek erforderlich.
//               Im aktuellen Verzeichnis muß der BGI-Treiber stehen.
//               (BGI=Borland Graphics Interface)
// ------------------------------------------------------------------------

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
```

---

38) Wer einen anderen als den Borland Compiler einsetzt, muß einige der im folgenden auftretenden spezifischen Aufrufe, die durch `#ifdef __TCPLUSPLUS__` erkenntlich sind, durch andere ersetzen; hierbei müßte ein Blick in das jeweilige Compilerhandbuch genügen.

```
#ifdef __TCPLUSPLUS__  // falls Borland Turbo C++
   #include <graphics.h>
#endif

// --- Fundamentale Konstanten/enum-Typen39) ----------------------------
enum { XLIMIT=640, YLIMIT=480 };


// --- Deklaration der Klasse COORDINATE ----------------------------
class COORDINATE
{
   private:
      int x, y;
   public:
      COORDINATE(int=0, int=0);
      void Set(int,int);
      void SetX(int);
      void SetY(int);
      int GetX();
      int GetY();
      COORDINATE Get();
}; // end class COORDINATE

// --- Implementation der Klasse COORDINATE --------------------------
COORDINATE::COORDINATE(int _x, int _y)
{
   Set(_x,_y);
}

void COORDINATE::Set(int _x, int _y)
{
   SetX(_x);
   SetY(_y);
}

void COORDINATE::SetX(int _x)
{
   x=0;
   if (0 < _x && _x < XLIMIT)
      x=_x;
}

void COORDINATE::SetY(int _y)
{
   y=0;
   if (0 < _y && _y < YLIMIT)
```

[39] Der Einfachheit halber wird hier mit einem klassischen VGA-Bildschirm und der Auflösung 640x480 Pixel gearbeitet.

```
      y=_y;
}



int COORDINATE::GetX()
{
   return(x);
}

int COORDINATE::GetY()
{
   return(y);
}

COORDINATE COORDINATE::Get()
{
   return(*this);
}


// --- Deklaration der Klasse GRAPHOBJECT -----------------------------
class GRAPHOBJECT
{
   private:
      static int isGraphicDisplay; // Erinnern Sie sich: ein statisches
                                   // Klassenelement!
   public:
      virtual void Draw();
      static void SetGraphicDisplay(int);
      static int IsGraphicDisplay();
      static void CloseGraphicDisplay();
}; // end class GRAPHOBJECT

// --- Implementation der Klasse GRAPHOBJECT --------------------------
void GRAPHOBJECT::SetGraphicDisplay(int value)
{
   isGraphicDisplay=value;
   if (value)
   {
      // Borland-spezifischer Code
#ifdef __TCPLUSPLUS__  // falls Borland Turbo C++
      static int gdriver = DETECT, gmode;
      initgraph(&gdriver, &gmode, "");
#endif
      // Initialisierung des Graphikdisplays ohne Fehlerbehandlung!
   }
}

int GRAPHOBJECT::IsGraphicDisplay()
```

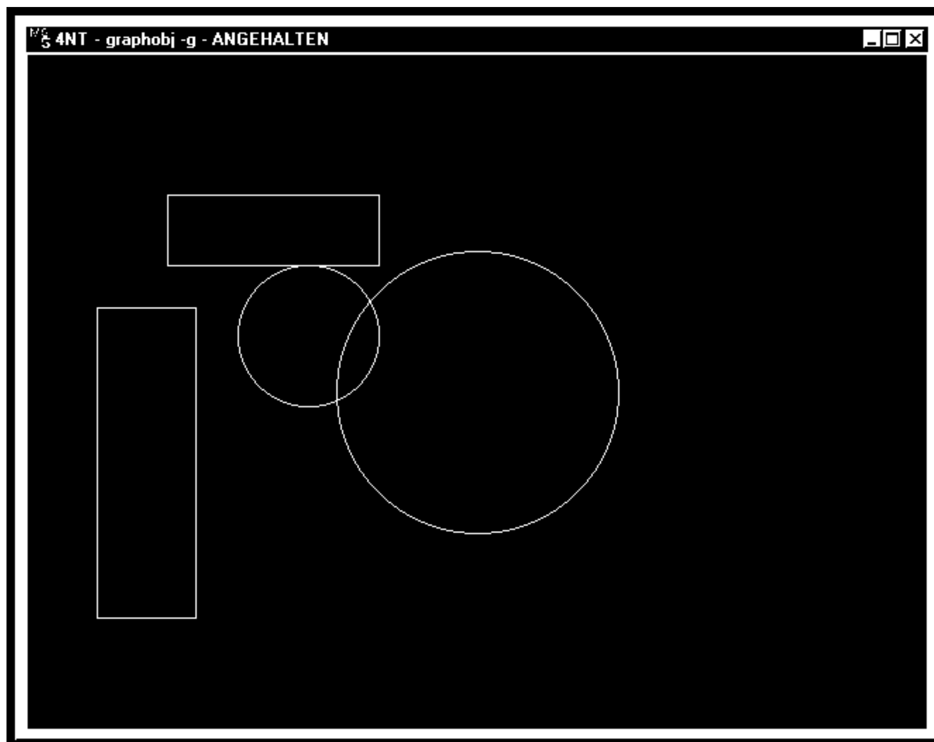
```
{
   return(isGraphicDisplay);
}


int GRAPHOBJECT::isGraphicDisplay; // Text- oder Graphikbildschirm?

void GRAPHOBJECT::CloseGraphicDisplay()
{
   if (isGraphicDisplay)
   {  // Borland-spezifischer Code
#ifdef __TCPLUSPLUS__  // falls Borland Turbo C++
      closegraph();
#endif
   }
}

void GRAPHOBJECT::Draw()
{
   // Dummy-Routine
}


// --- Deklaration der Klasse RECTANGLE ------------------------------
class RECTANGLE : public GRAPHOBJECT
{
   private:
      COORDINATE ecke1, ecke2;
   public:
      RECTANGLE(int=0,int=0,int=0,int=0);
      void Draw();
}; // end class RECTANGLE

// --- Implementation der Klasse RECTANGLE ---------------------------
RECTANGLE::RECTANGLE(int _ecke1x, int _ecke1y, int _ecke2x, int _ecke2y)
{
   ecke1.Set(_ecke1x,_ecke1y);
   ecke2.Set(_ecke2x,_ecke2y);
}

void RECTANGLE::Draw()
{
   if (GRAPHOBJECT::IsGraphicDisplay())
   {
#ifdef __TCPLUSPLUS__  // falls Borland Turbo C++
      line(ecke1.GetX(),ecke1.GetY(),ecke2.GetX(),ecke1.GetY());
      line(ecke2.GetX(),ecke1.GetY(),ecke2.GetX(),ecke2.GetY());
      line(ecke2.GetX(),ecke2.GetY(),ecke1.GetX(),ecke2.GetY());
      line(ecke1.GetX(),ecke2.GetY(),ecke1.GetX(),ecke1.GetY());
#endif
```

```
   }
   else  // Textmodus
      cout << "RECTANGLE: (" << ecke1.GetX() << ";" << ecke1.GetY()
         << ") (" << ecke2.GetX() << ";" << ecke2.GetY() << ")" << endl;
}
// --- Deklaration der Klasse CIRCLE ------------------------------------
class CIRCLE : public GRAPHOBJECT
{
   private:
      COORDINATE center;
      int        radius;
   public:
      CIRCLE(int=0,int=0,int=0);
      void Draw();
}; // end class CIRCLE

// --- Implementation der Klasse CIRCLE --------------------------------
CIRCLE::CIRCLE(int _centerx, int _centery, int _radius)
{
   center.Set(_centerx,_centery);
   radius=0;
   if (_radius > 0)
      radius=_radius;
}

void CIRCLE::Draw()
{
   if (GRAPHOBJECT::IsGraphicDisplay())
   {
#ifdef __TCPLUSPLUS__  // falls Borland Turbo C++
      circle(center.GetX(),center.GetY(),radius);
#endif
   }
   else  // Textmodus
      cout << "CIRCLE:    (" << center.GetX() << ";" << center.GetY()
         << ") mit Radius " << radius << endl;
}
```

```
// === Hauptprogramm ===================================================
int main(int argc, char** argv)
{  // Hier zu Demonstrationszwecken statisch codiert: wollen wir die
   // graphische Ausgabe benutzen (==1) oder nicht (==0)?
   // Programmaufruf: graphobj -t ... für Textmodus (Default)
   //                 graphobj -g ... für Graphikmodus
   //                 graphobj -? ... für kurze Erläuterung
   int useGraphics=0;                     // Defaultwert
   if (argc > 2 || (argc==2 && strcmp(argv[1],"-?")==0))
   {
      cerr << "Aufruf:     graphobj [ -g | -t ] " << endl;
      cerr << "Parameter:  -g ... Ausgabe auf VGA-Bildschirm" << endl;
      cerr << "            -t ... Ausgabe auf Textbildschirm" << endl;
      return(EXIT_FAILURE);
   }
   else if (argc==2 && strcmp(argv[1],"-g")==0)
      useGraphics=1;

   // (Gegebenenfalls) Setzen des Graphikdisplays; dies kann in der
   // Praxis natürlich auch für unterschiedliche Compiler und unter-
   // schiedliche Betriebssysteme und Oberflächen innerhalb dieser
   // einen Methode SetGraphicDisplay() geschehen!
   GRAPHOBJECT::SetGraphicDisplay(useGraphics);

   // Wir definieren einige Graphikobjekte und geben Sie auf den
   // Bildschirm aus: graphisch oder textuell (siehe oben)
   RECTANGLE rect1(0,0,639,479);
   rect1.Draw();

   CIRCLE circle1(XLIMIT/2,YLIMIT/2,100);
   circle1.Draw();

   RECTANGLE rect2(100,100,250,150);
   rect2.Draw();

   // Und nun dynamisch: daß pgo jeweils das "richtige" Draw() aufruft,
   // liegt an dem Attribut virtual bei GRAPHOBJ::Draw() !
   GRAPHOBJECT *pgo;
   pgo = new CIRCLE(200,200,50);
   pgo->Draw();
   delete pgo;

   pgo = new RECTANGLE(120,400,50,180);
   pgo->Draw();
   delete pgo;

   // Mit Return (oder einer sonstigen Eingabe) wird das Programm
   // dann beendet.
   char dummy[10];
   cin.getline(dummy,10);
```

```
   // Gegebenenfalls müssen "Aufräumarbeiten" erledigt werden,
   // beispielsweise das Zurücksetzen des Graphikmodus.
   GRAPHOBJECT::CloseGraphicDisplay();
   return EXIT_SUCCESS;
} // end main
// -------------------------------------------------------------------------
// end of file graphobj.cpp
// -------------------------------------------------------------------------
```

Nachstehend Bildschirmabzüge von diesem Programm, einmal im Graphikmodus, ein zweites Mal im Textmodus[40].




[40] Diese Bildschirmabzüge wurden in DOS-Boxen von Windows NT aufgenommen.

```
4NT

[g:\works\lng\cpp>] graphobj -?
Aufruf:      graphobj [ -g | -t ]
Parameter:   -g ... Ausgabe auf VGA-Bildschirm
             -t ... Ausgabe auf Textbildschirm

[g:\works\lng\cpp>] graphobj -t
RECTANGLE: (0;0) (639;479)
CIRCLE:    (320;240) mit Radius 100
RECTANGLE: (100;100) (250;150)
CIRCLE:    (200;200) mit Radius 50
RECTANGLE: (120;400) (50;180)


[g:\works\lng\cpp>]
```

## 8.3. Mehrfachvererbung

In C++ ist es auch möglich, eine Klasse nicht nur von einer Basisklasse, sondern von mehreren Superklassen abzuleiten. Sprachen wie Smalltalk bieten diesen Mechanismus nicht an, und die Hardcore-Smalltalk-Anhänger behaupten, Mehrfachvererbung (*multiple inheritance*) sei nicht wirklich erforderlich. Die Praxis zeigt jedenfalls, daß Mehrfachvererbung recht selten vorkommt.

### 8.3.1. Beispiel

Sehen wir uns das nachfolgende Einstiegsbeispiel an, in dem eine Klasse `C` von einer Klasse `A` und einer Klasse `B` (jeweils `public`) abgeleitet wird.

```
// -----------------------------------------------------------------------

// multiple_inheritance.cpp - Demonstrationsprogramm zur Mehrfachvererbung

// -----------------------------------------------------------------------

#include <iostream.h>

#include <stdlib.h>



class A

{

  protected:
```

```
    int a;
  public:
    A(int=0);
}; // end class A

A::A(int i)
{
  a=i;
  cout << "A-Konstruktor i=" << i << endl;
}

class B
{
  protected:
    int a;
  public:
    B();
}; // end class B

B::B()
{
  cout << "B-Konstruktor" << endl;
}
```

```
class C : public A, public B  // Die Klasse C wird von A und B abgeleitet
{
  public:
     C(int =1);
}; // end class C

C::C(int k) : A(k)
{
  cout << "C-Konstruktor mit k=" << k << endl;
  A::a=k;                    // Ohne die Scope-Angaben A:: bzw. B::
  B::a=2*k;                  // wäre der Name a zweideutig!
} // end C::C(int)

int main()
{
  C c1, c2(5);               // Es werden nur zwei Objekte kreiert.
  return EXIT_SUCCESS;
} // end main
// ---------------------------------------------------------------------
// end of file multiple_inheritance.cpp
// ---------------------------------------------------------------------
```

Und das Ablauflisting zu diesem Programm:

```
A-Konstruktor i=1            // Das Objekt c1 wird angelegt
B-Konstruktor
C-Konstruktor mit k=1
A-Konstruktor i=5            // Das Objekt c2 wird angelegt
B-Konstruktor
C-Konstruktor mit k=5
```

Dieses einfache Beispiel zeigt, daß im Fall einer Mehrfachvererbung implizit der Reihe nach sämtliche Konstruktoren aller Basisklassen[41] aufgerufen werden, bevor dann der klasseneigene Konstruktor an die Arbeit geht.

In der Klasse `C` haben wir folgende Datenelemente: einen `int`-Speicherplatz `a` aus der Superklasse `A`, einen ebensolchen Speicherplatz `a` aus der Superklasse `B` sowie den deklarierten Konstruktor `C::C(int)`. Zur Erinnerung: die Konstruktoren der Basisklassen werden nicht vererbt, also nicht separat für Objekte der Klasse `C` angelegt, sondern nur automatisch aufgerufen.

### 8.3.2. Beispiel: Die Stream-Klassen istream, ostream und iostream

Die in Kapitel 6 vorgestellten C++-Stream-Klassen besitzen ebenfalls eine Klassenhierarchie, in der Mehrfachvererbung vorkommt. Als Beispiel sei hier aus der Datei `iostream.h`[42] die Deklaration der Klasse `iostream` abgedruckt, die von den Klassen `istream` und `ostream` abgeleitet wird.

```
class iostream : public istream, public ostream

{

  // A stream that supports both insertion and extraction.

  public:

      iostream(streambuf *buf);

      virtual ~iostream();

  protected:

      iostream();

};
```

[41] Die Reihenfolge, in der die Konstruktoren der zwei oder mehr Superklassen aufgerufen werden, ist nicht im Standard festgelegt, somit compilerabhängig. Man sollte bei der Implementation darauf achten, daß es auf diese Reihenfolge nicht ankommt!

[42] Der hier gezeigte Auszug aus iostream.h stammt von dem Symantec C++ Compiler 6.11.

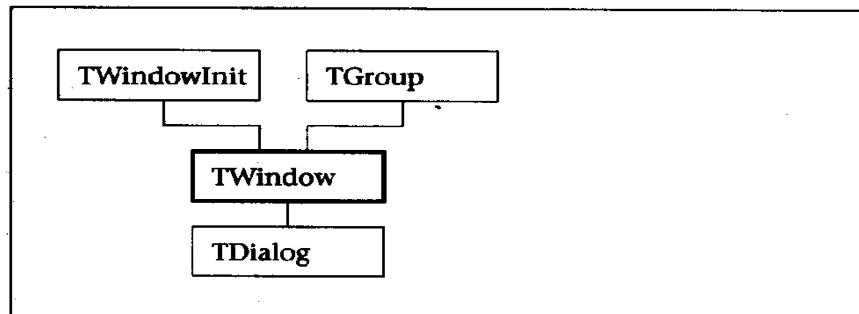
### 8.3.3. Beispiel: Borlands *Object Windows Library (OWL)*

Nachstehend wird als schematisches Beispiel zur Mehrfachvererbung eine Erläuterung aus dem Reference Guide der *Object Windows Library* von Borlands C++ Compiler abgedruckt.

Die dort verwendeten Klassen `TWindowInit`, `TGroup`, `TWindow` und `TDialog`, auf deren Funktion hier aus Platzgründen nicht eingegangen werden soll, stehen in der nachstehend abgebildeten Hierarchie. Ein Objekt der Klasse `TWindow` ist also insbesondere auch von den Klassen `TWindowInit` und `TGroup`, erbt also von dort verschiedene Datenelemente und Methoden.

Borland C++ Object Windows Library

**TWindow** **views.h**




***TWindow*** **ist ein spezialisierter Nachkomme von *TGroup* und damit selbst eine Gruppe, zu der üblicherweise ein *TFrame*-Objekt, ein *TScroller*-Objekt und darüber hinaus auch ein oder zwei *TScrollBar*-Objekte gehören. Diese Objekte machen ein *TWindow*-Objekt auf dem Bildschirm sichtbar; *TFrame* versieht das Fenster mit einen Rahmen, der Platz bietet für einen Titel und verschiedene Symbole enthält, die es ermöglichen, mittels der Maus die Position des Fensters und seine Größe zu ändern. Zu den Fähigkeiten eines *TWindow*-Objekts zählt auch die Integration von Bildlaufleisten. Hat das Datenelement *number* einen Wert zwischen 1 und 9, kann das Fenster über die Tastenkombination [Alt][Ziffer] selektiert werden.**

***TWindow*** **hat zwei unmittelbare Vorfahren, *TGroup* und die virtuelle Basisklasse *TWindowInit*. Die Klasse *TWindowInit* bringt einen Konstruktor und die Elementfunktion *createFrame* zum Erzeugen und Einfügen eines *Frame*-Objekts ein. Eine ähnliche Anwendung der Mehrfachvererbung kommt auch bei *THistoryWindow* und *THistInit* zum Einsatz.**

## 8.4. Kopierkonstruktoren in abgeleiteten Klassen

Abschließend für dieses Kapitel soll noch kurz auf die Kopierkonstruktoren bei abgeleiteten Klassen eingegangen werden.

Der Kopierkonstruktor einer abgeleiteten Klasse hat offensichtlich auch die Aufgabe, die von der Superklasse vererbten Elemente zu kopieren. Hierzu benötigt der Kopierkonstruktor in der Regel den Kopierkonstruktor der Basisklasse. Der schematische Aufbau für den Fall einer Superklasse `A` und einer hiervon abgeleiteten Klasse `B` sieht dann so aus.

```
B::B(B & einBobjekt) : A(einBobjekt)
{
   // hier die B-spezifischen Kopieraktionen
} // end Kopierkonstruktor B::B(B&)
```

Der Kopierkonstruktor der Klasse `B` erhält als Argument ein Objekt seiner Klasse per Referenz. Dieses wird – ebenfalls per Referenz – an den Kopierkonstruktor der Basisklasse durchgereicht, wobei der Compiler für die notwendige Typanpassung, das Casting, zu sorgen hat. Referenzen und Zeiger auf Basisklassen dürfen in C++ auch mit Referenzen bzw. Zeigern auf abgeleitete Klassen vorbelegt werden. In unserem obigen Beispiel ist ein `B`-Objekt naturgemäß auch ein (spezielles) `A`-Objekt.

### 8.4.1. Beispiel: Personen und Angestellte

Nachstehend wird ein etwas längeres Programmlisting gezeigt, in dem eine Klasse `PERSON` deklariert wird, der Einfachheit halber nur mit den Datenelementen `name` und `vorname`. Von dieser Klasse wird die Klasse `ANGESTELLTER` abgeleitet, die als neues Datenelement eine Angestelltennummer (`angnr`) mitbekommt.

Die Funktionen `Ausgabe1()` und `Ausgabe2()` sollen noch einmal den Unterschied zwischen einem normalen *call by value* und einer Referenzübergabe in C++ verdeutlichen: bei dem üblichen call by value sehen wir, daß die Kopierkonstruktoren aufgerufen werden für das Klassenobjekt in der Parameterliste von `Aufgabe1()`, denn es wird auf dem Stack eine Kopie der Hauptprogramm-Instanz `a` angelegt.

```
// ----------------------------------------------------------------------
// inherkks.cpp[43] - Kopierkonstruktoren in abgeleiteten Klassen
// ----------------------------------------------------------------------
#include <iostream.h>
```

[43] Wundern Sie sich nicht: dieser exotische Name steht für „inheritance mit Kopierkonstruktoren” - ist doch naheliegend, oder?

```
#include <stdlib.h>
#include <string.h>

const int BUFLEN = 80;
```

```
// Klassendeklarationen
class PERSON     // hier nur flüchtig skizziert
{
  protected:
     char name[BUFLEN];
     char vorname[BUFLEN];
  public:
     friend ostream & operator<<(ostream&,PERSON&);

     PERSON();
     PERSON(char*,char*);
     PERSON(PERSON&);   // Kopierkonstruktor
}; // end class PERSON

PERSON::PERSON()
{
  strcpy(name,"(unbekannt)");
  strcpy(vorname,"(unbekannt)");
} // end PERSON::PERSON()

PERSON::PERSON(char *_name, char *_vorname) // Beachten Sie die Fußnote 44)
{
  strcpy(name,_name);
  strcpy(vorname,_vorname);
} // end PERSON::PERSON()
```

[44] Einige C++ Entwickler geben den Übergabeparametern in diesem Kontext denselben Namen wie den zu setzenden Datenelementen, erweitert lediglich um einen Unterstrich. Diese oder ähnliche Konventionen sind sehr sinnvoll, damit der Code leserlich bleibt! Vergleichen Sie hierzu beispielsweise auch das Übungsprogramm auf Seite0.

```
PERSON::PERSON(PERSON& p)
{
  cout << "Kopierkonstruktor der Klasse PERSON: Kopiere Name="
     << p.name << endl;
  strcpy(name,p.name);
  strcpy(vorname,p.vorname);
} // end PERSON::PERSON(PERSON&)

ostream & operator<<(ostream& out,PERSON& p)
{
  out << "Name: " << p.name << ", Vorname: " << p.vorname << endl;
  return out;
} // end friend ostream & operator<<(ostream& out,PERSON& p)
```

```
class ANGESTELLTER : public PERSON
{
  private:
     char angnr[8];
  public:
     ANGESTELLTER();
     ANGESTELLTER(char*,char*,char*);
     ANGESTELLTER(ANGESTELLTER&);  // Kopierkonstruktor
     friend ostream & operator<<(ostream&,ANGESTELLTER&);
}; // end class ANGESTELLTER
ANGESTELLTER::ANGESTELLTER()
{
  strcpy(angnr,"000.000");
} // end ANGESTELLTER::ANGESTELLTER

ANGESTELLTER::ANGESTELLTER(char * _name, char * _vorname, char* _angnr)
{
  strcpy(name,_name);
  strcpy(vorname,_vorname);
 strcpy(angnr,_angnr);
} // end ANGESTELLTER::ANGESTELLTER(char*,char*,char*)

ANGESTELLTER::ANGESTELLTER(ANGESTELLTER& a) : PERSON(a)
{
  cout << "Kopierkonstruktor der Klasse ANGESTELLTER: "
     << "Kopiere Name=" << a.name << endl;
  strcpy(angnr,a.angnr);
} // end ANGESTELLTER::ANGESTELLTER(ANGESTELLTER&)
```

```
ostream & operator<<(ostream& out,ANGESTELLTER& a)
{
  out << "Name: " << a.name << ", Ang-Nr.: " << a.angnr << endl;
  return out;
} // end friend ostream & operator<<(ostream& out,ANGESTELLTER& a)


// Prototypen
void Ausgabe1(ANGESTELLTER a);
void Ausgabe2(ANGESTELLTER& a);
void DrawLine(void);

// Implementation
void DrawLine(void) {
  cout << "------------------------------"
       << "------------------------------" << endl;
}
```

```
void Ausgabe1(ANGESTELLTER a)
{
  cout << "Ausgabe1(): " << endl << a;
} // end Ausgabe1

void Ausgabe2(ANGESTELLTER& a)
{
  cout << "Ausgabe2(): " << endl << a;
} // end Ausgabe2

// Hauptprogramm
int main()
{
  DrawLine();

  // Eine Person wird deklariert...
  PERSON p("Meier","Thomas");
  cout << "Person:         " << p;
  DrawLine();

  // Eine Angestellte kommt hinzu...
  ANGESTELLTER a("Weber","Gerlinde","123.456");
  cout << "Angestellte(r): " << a;
  DrawLine();

  // Zwei Funktionsaufrufe
  Ausgabe1(a);
  DrawLine();
```

```
  Ausgabe2(a);
  DrawLine();


  return EXIT_SUCCESS;
} // end main
```

Und zu diesem Programm wird auch wieder das Programmlisting gezeigt.

```
Person:         Name: Meier, Vorname: Thomas
--------------------------------------------------------------
Angestellte(r): Name: Weber, Ang-Nr.: 123.456
--------------------------------------------------------------
Kopierkonstruktor der Klasse PERSON: Kopiere Name=Weber
Kopierkonstruktor der Klasse ANGESTELLTER: Kopiere Name=Weber
Ausgabe1():
Name: Weber, Ang-Nr.: 123.456
--------------------------------------------------------------
Ausgabe2():
Name: Weber, Ang-Nr.: 123.456
--------------------------------------------------------------
```

# 9. Polymorphismus in C++

In 1.3.3. wurde bereits der für die Objektorientierung zentrale Begriff Polymorphismus vorgestellt. Soll beispielsweise eine (dynamische) lineare Liste verwaltet werden, deren Mitglieder von verschiedenen Typen (Klassen) sein können, so muß eine mögliche Bearbeitungsfunktion zur Laufzeit wissen, mit was für einem Objekt sie es jeweils zu tun hat. Stehen in der Liste ein `int`-Wert, ein Objekt einer Klasse `PUNKT`, eine Zeichenkette und abschließend ein `float`-Wert, so muß etwa eine Bildschirmausgabe damit korrekt umgehen können: der `int`-Wert muß als ganze Zahl dargestellt, der Punkt vielleicht gezeichnet werden usw.

## 9.1. Virtual Reality

Betrachten wir eine ganz einfache Klassenhierarchie: die Klasse `B` sei aus einer Superklasse `A` abgeleitet. In beiden Klassen gebe es eine Methode `TueWas()`. Dann kann folgendes geschrieben werden.

```
#include <iostream.h>
#include <stdlib.h>
class A               // nur skizziert
{
  public:
     void TueWas();
}; // end class A


void A::TueWas()
{
  cout << "Ich bin A::TueWas()..." << endl;
} // end A::TueWas()


class B : public A
{
  public:
     void TueWas();
```

```
}; // end class B
void B::TueWas()
{
  cout << "Ich bin B::TueWas()..." << endl;
} // end B::TueWas()
```

```
int main()

{

  B  bobjekt;

  A* pointer;

  pointer=&bobjekt;

  pointer->TueWas();

  return EXIT_SUCCESS;

} // end main
```

Ein Objekt der Klasse `B` wird angelegt, ebenso ein Zeiger `pointer` auf ein `A`-Objekt. Die Zuweisung `pointer=&bobjekt` ist korrekt, denn ein `B`-Objekt ist insbesondere auch ein Objekt der Klasse `A`. Mit der Anweisung `pointer->TueWas()` wird allerdings die Methode `TueWas()` aus der Klasse `A` aufgerufen, denn `pointer` ist eben als ein Zeiger auf ein Objekt der Klasse `A` deklariert! Wird das oben gezeigte Programm gestartet, so erscheint erwartungsgemäß folgende Ausgabe auf dem Bildschirm:

```
Ich bin A::TueWas()...
```

Damit hier Polymorphie zum Tragen kommt, d.h. das Laufzeitsystem registriert, daß im Moment `pointer` auf ein Objekt der Klasse `B` zeigt und demzufolge auch die Instanz `TueWas()` der Klasse `B` aufgerufen werden muß, ist das Schlüsselwort `virtual` vor die Funktion `TueWas()` in der Klasse `A` zu schreiben.

```
class A   // mit virtueller TueWas()-Funktion

{

   public:

      virtual void TueWas(void);    // mit Schlüsselwort virtual !

}; // end class A
```

Wird `A::TueWas()` auf diese Weise zu einer virtuellen Funktion, so registriert der Compiler, daß im Falle eines Pointers (oder einer Referenz) auf A erst zur Laufzeit (und nicht schon während der Compilation) entschieden werden kann, welche Methode konkret aufzurufen ist. Das entsprechende Programmlisting sieht dann aus wie folgt:

```
Ich bin B::TueWas()...
```

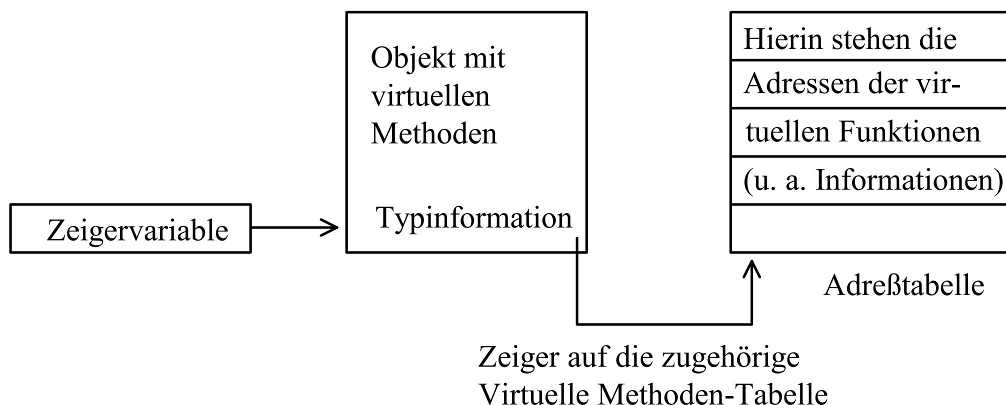
Das heißt, nochmals gesagt: erst zur Laufzeit wird ermittelt, auf was für eine Art Objekt der Pointer (oder ggf. die Referenz) verweist.

## 9.2. Frühe und späte Bindung

Als *frühe Bindung* (*early binding*) wird die bereits gewohnte Vorgehensweise bezeichnet: der Compiler bestimmt bereits zur Compilationszeit, ausgehend von der Deklaration eines Objektes, welche Methode dazu „paßt”. Dies findet zum Beispiel bei überladenen Funktionen oder Operatoren statt.

*Späte Bindung* (*late binding*) ist demgegenüber eine Technik, die Laufzeit-Information über die jeweiligen Datentypen verlangt. Wie im vorherigen Abschnitt demonstriert, wird hierbei von einem Zeiger (oder einer Referenz) einer Basisklasse ausgegangen. Diesem Zeiger (oder dieser Referenz) kann dann ein Objekt[45)] einer beliebigen abgeleiteten Klasse zugewiesen werden.

Wie wird nun diese späte Bindung umgesetzt? Die oben erwähnten Laufzeit-Informationen über die Datentypen[46)] werden in einer *Virtuellen Methoden-Tabelle* (*VMT*) abgelegt. Das bedeutet: Wird ein Objekt einer Klasse mit mindestens einer virtuellen Methode deklariert, so werden zu diesem Objekt neben den üblichen Informationen auch noch Angaben zum aktuellen Datentyp gespeichert. Zur Laufzeit wird dann in dieser VMT nachgesehen, auf welchen aktuellen Datentyp (bzw. ein Objekt dieses Typs) der betreffende Zeiger gerade verweist.




### 9.2.1. Beispielprogramm zur Polymorphie

Im nachfolgenden, etwas umfangreicheren Beispiel werden ausgehend von einer abstrakten Basisklasse `OBJECT` zwei Klassen `INPUTWINDOW` und `OUTPUTWINDOW` abgeleitet; von diesen beiden Klassen wiederum wird die Klasse `IOWINDOW`

---

45) streng genommen natürlich die Adresse eines Objektes

46) Bei Borland werden diese entsprechend auch *Run Time Type Information* (*RTTI*) genannt.

abgeleitet. Auch wenn in diesem Rahmen diese Klassen nur vage angedeutet werden, so können Sie sich dennoch wahrscheinlich vorstellen, daß in einer praktischen Programmierung Objekte benötigt werden, die als Eingabefeld (`INPUTWINDOW`), als Ausgabefenster (`OUTPUTWINDOW`) oder eben als Ein- und Ausgabefenster (`IOWINDOW`) dienen.

Im Hauptprogramm wird u.a. gezeigt, wie ein Zeiger auf `OBJECT` dynamisch auf Objekte dieser drei abgeleiteten Klassen verweisen kann. Außerdem wird als Beispiel für späte Bindung (*late binding*) am Ablauflisting demonstriert, wie das C++-Laufzeitsystem die jeweils zum Pointer-Objekt aktuell passende Methode aufruft, d.h. wie der Polymorphismus praktisch funktioniert.

In der nachfolgend gezeigten Klasse `OBJECT` gibt es eine Methode

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

Hierbei handelt es sich um eine sogenannte *pur virtuelle Methode*. Durch das Suffix „= 0" wird festgelegt, daß in der Klasse `OBJECT` diese Methode nicht implementiert wird, der Compiler also keinen Funktionsrumpf erwartet. Das wiederum bedeutet, daß (nur) die Schnittstelle weitervererbt wird, jede von `OBJECT` abgeleitete Klasse also eine solche Methode implementieren **muß**[47)].

Eine Klasse mit pur virtuellen Methoden heißt *abstrakte Basisklasse* (*abstract base class*, kurz: *ABC*); von ihr können keine Objekte angelegt werden.

```
// polymorphismus.cpp - Ein elementares Beispiel zum Polymorphismus

#include <iostream.h>

#include <stdlib.h>

#include <string.h>


class OBJECT

{

  protected:

     char field[128];
```

---

47) Stroustrup erläutert in seinem Buch *The Design and Evolution of C++* sehr malerisch, wie es zur Notation mit der `=0` Syntax kam: zum Zeitpunkt der Einführung der Umsetzung der abstrakten Basisklassen im AT&T-Release 2.0 von C++ wollte Stroustrup nicht die Auslieferung dieser C++-Version durch die Einführung neuer Schlüsselworte wie `pure` oder `abstract` verzögern, was seiner Einschätzung nach die Gefahr gewesen wäre, wenn er im entsprechenden Standardisierungskomitee solche Vorschläge unterbreitet hätte!

```
  public:
     OBJECT();
     OBJECT(OBJECT&);
     virtual void WhoAmI() = 0;  // pur virtuelle Funktion
}; // end class OBJECT

OBJECT::OBJECT()
{
  strcpy(field,"(no entry)");
}

OBJECT::OBJECT(OBJECT& obj)
{
  strcpy(field,obj.field);
}
```

```
class INPUTWINDOW : virtual public OBJECT
{
  protected:
     char prompt[128];
  public:
     INPUTWINDOW(char* ="Eingabe: ")
     void WhoAmI();
}; // end class INPUTWINDOW
INPUTWINDOW::INPUTWINDOW(char* _prompt)
{
  strcpy(prompt,_prompt);
}

void INPUTWINDOW::WhoAmI()
{
  cout << "WhoAmI: INPUTWINDOW" << endl;
}

class OUTPUTWINDOW : virtual public OBJECT
{
  public:
     OUTPUTWINDOW(char * ="???");
     void WhoAmI();
}; // end class OUTPUTWINDOW

OUTPUTWINDOW::OUTPUTWINDOW(char* _field)
{
  strcpy(field,_field);
```

```
}

void OUTPUTWINDOW::WhoAmI()
{
  cout << "WhoAmI: OUTPUTWINDOW" << endl;
}


class IOWINDOW : public INPUTWINDOW, public OUTPUTWINDOW
{
  public:
     IOWINDOW(char * ="??");
     void GetField();
     void WhoAmI();
     friend ostream& operator<<(ostream&,IOWINDOW&);
}; // end class IOWINDOW

IOWINDOW::IOWINDOW(char* msg) : INPUTWINDOW(msg)
{
}


void IOWINDOW::GetField()
{
  cout << prompt;
  // cin >> field; - liest nur bis zum ersten WhiteSpace, z.B. Leerzeichen
  cin.get(field,128-1,'\n'); // Lesen von maximal 128-1 Zeichen bzw. bis
                             // zum Newline-Zeichen (ausschließlich)
```

```
  cin.ignore(1,'\n');         // Überlesen des Newline-Zeichens
} // end IOWINDOW::GetField()


void IOWINDOW::WhoAmI()
{
  cout << "WhoAmI: IOWINDOW" << endl;
}


ostream& operator<<(ostream& out,IOWINDOW& iow)
{
  out << "IOWINDOW: field=" << iow.field << endl;
  return out;
} // end ostream& operator<<(ostream&,IOWINDOW&)
int main()
{
  IOWINDOW iow("Bitte geben Sie Ihren Namen ein: ");
  iow.GetField();
  cout << "Kontrollausgabe: " << endl << iow;
  OBJECT *pointer;
  // pointer = new OBJECT; // Fehler! OBJECT ist eine abstrakte Klasse!


  // Polymorphismus:
  // Es wird jeweils die korrekte WhoAmI()-Version aufgerufen.
  pointer = new IOWINDOW;
  pointer->WhoAmI();
  delete pointer;


  pointer = new INPUTWINDOW;
```

```
  pointer->WhoAmI();
  delete pointer;

  pointer = new OUTPUTWINDOW;
  pointer->WhoAmI();
  delete pointer;

  return EXIT_SUCCESS;
} // end main
```

Nachstehend das Ablauflisting dieses Programms. Die Benutzereingaben sind kursiv gesetzt.

```
Bitte geben Sie Ihren Namen ein: Meier, Karl

Kontrollausgabe:

IOWINDOW: field=Meier, Karl

WhoAmI: IOWINDOW

WhoAmI: INPUTWINDOW

WhoAmI: OUTPUTWINDOW
```

Eine Bemerkung zu dem Programm: bei der Deklaration der Klassen `INPUTWINDOW` und `OUTPUTWINDOW` wurde die Superklasse `OBJECT` als virtuell gekennzeichnet:
`class INPUTWINDOW : virtual public OBJECT` bzw.
`class OUTPUTWINDOW : virtual public OBJECT`
- dies bewirkt, daß bei einer weiteren Ableitung, wie hier mit `IOWINDOW` geschehen, nur jeweils eine Instanz der Basisklasse `OBJECT` weitervererbt wird. Würde das Schlüsselwort `virtual` fehlen, so gäbe es in einem Objekt der Klasse `IOWINDOW` gleich zweimal ein Datenelement `field`!

Wir wollen uns noch ein weiteres Beispiel zur Mehrfachvererbung ansehen.

## 9.2.2. Beispiel: Mehrfachvererbung mit und ohne virtual

Nachstehend sehen Sie zwei Programme zur Mehrfachvererbung. In multiple1.cpp werden von einer Klasse `TEIL` die Klassen `TOPF` und `DECKEL` virtual abgeleitet, in multiple2.cpp geschieht diese Vererbung ohne das Schlüsselwort virtual. Die Auswirkung ist dann in der mehrfachvererbten Klasse `TOPFMITDECKEL` zu beobachten.

```
1  // -----------------------------------------------------------------
2  // Dateiname:   multiple1.cpp
3  // -----------------------------------------------------------------
4
5  #include <iostream.h>
6  #include <stdlib.h>
7
8
9  class TEIL
10 {
11     protected:
12         int teilnr;
13     public:
14         TEIL(int=0);
15         void SetzeTeilNr(int);
```

```
16 };
17
18 TEIL::TEIL(int _teilnr)
19 {
20     SetzeTeilNr(_teilnr);
21 }
22 void TEIL::SetzeTeilNr(int _teilnr)
23 {
24     teilnr = _teilnr > 0 ? _teilnr : 0;
25 }
26
27 class TOPF : virtual public TEIL
28 {
29     protected:
30         static int MAXHOEHE;
31         static int MAXBREITE;
32         int hoehe;
33         int breite;
34     public:
35         TOPF(int=0, int=0);
36 };
37
38 int TOPF::MAXHOEHE = 60;
39 int TOPF::MAXBREITE= 45;
40
41 TOPF::TOPF(int _hoehe, int _breite)
42 {
43     hoehe = breite = 0; // technischer Defaultwert
44     if (0 < _hoehe && _hoehe < MAXHOEHE)
45         hoehe = _hoehe;
46     if (0 < _breite && _breite < MAXBREITE)
47         breite = _breite;
48 }
49
50 class DECKEL : virtual public TEIL
51 {
52     protected:
53         static int MAXGRIFFARTEN;
54         int breite;
55         int griffart; // verschiedene Sorten stehen zur Auswahl, hier
56                       // einfach nur durchnumeriert
57     public:
58         DECKEL(int=0, int=0);
59 };
60
61 int DECKEL::MAXGRIFFARTEN = 5;
62
63 DECKEL::DECKEL(int _breite, int _griffart)
64 {
65     breite = _breite > 0 ? _breite : 0;
```

```
66     griffart = 0<_griffart && _griffart<MAXGRIFFARTEN ? _griffart : 0;
67 }
```

```
68 class TOPFMITDECKEL : public TOPF, public DECKEL
69 {
70     public:
71         TOPFMITDECKEL(int=0, int=0, int=0);
72         void Show();
73 };
74
75 TOPFMITDECKEL::TOPFMITDECKEL(int _hoehe, int _breite, int _griffart) :
76     TOPF(_hoehe,_breite), DECKEL(_breite,_griffart)
77 {
78 }
79
80 void TOPFMITDECKEL::Show()
81 {
82     cout << "TOPFMITDECKEL::Show(): " << endl
83         << "hoehe = " << hoehe << endl
84         << "TOPF::breite = " << TOPF::breite << endl
85         << "DECKEL::breite = " << DECKEL::breite << endl
86         << "griffart = " << griffart << endl
87         << "TOPF::teilnr = " << TOPF::teilnr << endl
88         << "DECKEL::teilnr = " << DECKEL::teilnr << endl
89         << endl;
90 }
91
92 int main()
93 {
94     TOPFMITDECKEL topfMitDeckel(50,40,2);
95     topfMitDeckel.SetzeTeilNr(123);
96     topfMitDeckel.Show();
97     return EXIT_SUCCESS;
98 } // end main
99 // end of file multiple1.cpp
```

Ablauflisting multiple1.cpp:

```
TOPFMITDECKEL::Show():
hoehe = 50
TOPF::breite = 40
DECKEL::breite = 40
griffart = 2
TOPF::teilnr = 123
DECKEL::teilnr = 123
```

```
108 // -------------------------------------------------------------------
109 // Dateiname:   multiple2.cpp
110 // Anmerkung:   Nachstehend werden nur die Änderungen gegenüber
111 //              multiple1.cpp (S. 0) abgedruckt.
112 // -------------------------------------------------------------------
113
114 class TEIL
115 {
116     // wie zuvor
117 };
118
119 class TOPF : public TEIL                    // ohne virtual-Markierung
120 {
121     // wie zuvor
122 };
123
124 class DECKEL : public TEIL                  // ohne virtual-Markierung
125 {
126     // wie zuvor
127 };
128
129 class TOPFMITDECKEL : public TOPF, public DECKEL
130 {
131     // wie zuvor
132 };
133
134 int main()
135 {
136     TOPFMITDECKEL topfMitDeckel(50,40,2);
137     topfMitDeckel.TOPF::SetzeTeilNr(123);
138     topfMitDeckel.DECKEL::SetzeTeilNr(145);
139     topfMitDeckel.Show();
140     return EXIT_SUCCESS;
141 } // end main
142 // end of file multiple2.cpp
```

Ablauflisting multiple2.cpp:

```
TOPFMITDECKEL::Show():
hoehe = 50
TOPF::breite = 40
DECKEL::breite = 40
griffart = 2
TOPF::teilnr = 123
DECKEL::teilnr = 145
```

Wir stellen fest, daß ohne die virtual-Markierung bei der Vererbung das Datenelement `TEIL::teilnr` in der Klasse `TOPFMITDECKEL` zweifach vorkommt! Wird dagegen die Vererbung virtual gemacht (vgl.. Zeilen 30 und 55), so taucht das Datenelement `teilnr` der obersten Klasse `TEIL` nur einmal auf in einem `TOPFMITDECKEL`-Objekt.

Dabei kann nicht generell bestimmt werden, was von beiden Alternativen „richtig" wäre: in dem hier vorliegenden kleinen Beispiel kann es so sein, daß ein „Topf mit Deckel" in einem Haushaltsgeschäft als „ein Teil" geführt wird, also auch nur eine Inventarnummer erhält; es kann aber genausogut sein, daß auf die physische Existenz von zwei Einzelteilen geachtet werden muß, dann sind zwei (verschiedene) Teilenummern zu verwalten.

# 10. Streams II: Dateioperationen

In der Include-Datei `FSTREAM.H` werden u.a. die Klassen `ofstream`, `ifstream` und `fstream` als indirekt von der Klasse `ios` abgeleitet deklariert, die die Ausgabe auf einen Datei-Stream, die Eingabe sowie den sogenannten wahlfreien Lese- und Schreibzugriff (*random access*) verwalten.

Datei-Streams werden genauso verwendet wie Objekte der Basisklassen. Für eine Funktion `void Out(ostream &out) { out << /* ... */ }` sind die beiden folgenden Aufrufe möglich:

```
ofstream myfile;       // Ausgabedatei (Stream)

// ...

Out(cout);          // Ausgabe auf cout

Out(myfile);        // Ausgabe auf die Datei myfile
```

Voraussetzung für den erfolgreichen Zugriff auf einen Datei-Stream ist das vorherige korrekte Öffnen. Im Falle der Standardstreams `cin`, `cout`, `cerr` und `clog` wird dies bereits automatisch beim Programmstart durchgeführt.

Zum Arbeiten mit Dateien (bzw. Datei-Streams) gehören im allgemeinen das Öffnen in einem korrekten Modus (bspw. `READONLY` zum Nur-Lesen), das Lesen und Schreiben einzelner Buffer-Einheiten (Zeichen, Zeilen, Datensätze) sowie gegebenenfalls das Positionieren sowie das Schließen.

## 10.1. Sequentielle Ein- und Ausgabe

Die aus C und anderen Programmiersprachen bekannte elementare sequentielle zeichenweise Ein- und Ausgabe geschieht mittels Streams der oben erwähnten Klassen `ifstream` und `ofstream`. Sehen wir uns zunächst ein kleines Beispiel an.

### 10.1.1. Beispiel

Der nachstehend abgedruckte Quelltext `iotest.cpp` ist ein kleines Kopierprogramm, das die Datei `testold.dat` zeichenweise in die Datei `testnew.dat` kopiert.

```
// ---------------------------------------------------------

// iotest.cpp - Ein erstes Demonstrationsprogramm für die

// Arbeit mit Datei-Streams.

// ---------------------------------------------------------
```

```
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>

// Die Dateinamen werden hier als Konstanten vereinbart.
const char INFILENAME[]  = "testold.dat";
const char OUTFILENAME[] = "testnew.dat";
int main()
{
  // Ein Stream für eine Eingabedatei wird bereitgestellt.
  ifstream infile(INFILENAME);

  // Hat das Öffnen (implizit beim Konstruktoraufruf durchgeführt) nicht
  // geklappt, so liefert der überladene !-Operator TRUE zurück.
  if (!infile)
    cerr << "Fehler: Datei " << INFILENAME << " läßt sich nicht lesen!"
      << endl, exit(EXIT_FAILURE);

  // Ein Stream für eine Ausgabedatei wird bereitgestellt.
  ofstream outfile;

  // Wie zuvor, nur wird diesmal mit der Methode open() der Stream
  // geöffnet.
  outfile.open(OUTFILENAME, ios::out);
  if (!outfile)
    cerr << "Fehler: Datei " << OUTFILENAME
      << " läßt sich nicht schreiben!"
      << endl, exit(EXIT_FAILURE);
```

```
  char c;
  // Sind beide Dateien erfolgreich geöffnet worden, so beginnt die
  // elementare Kopieraktion: eine kopfgesteuerte Schleife, die abbricht,
  // sobald im Eingabestrom keine Zeichen mehr zu finden sind - oder
  // aber die Ausgabedatei einen Fehler meldet.
  while (outfile && infile.get(c))
    outfile.put(c);
  cout << "Kopieren von " << INFILENAME << " nach "
    << OUTFILENAME << " beendet." << endl;

  infile.close();   // Technisch nicht erforderlich, da beim hier sowieso
  outfile.close();  // stattfindenden Destruktoraufruf automatisch die
                    // Streams geschlossen werden.
  return EXIT_SUCCESS;
} // end main
// end of file iotest.cpp
```

Das Öffnen einer Datei geschieht entweder mittels des Konstruktoraufrufes oder explizit mit der Klassenmethode `open()`; entsprechend wird ein Datei-Stream entweder durch einen Destruktoraufruf oder mit der parameterlosen `close()`-Methode geschlossen.

In der Basisklasse `ios` (siehe die Headerdatei `iostream.h`) sind folgende Konstanten deklariert, die beim Öffnen einer Datei verwendet werden können.[48)]

```
enum open_mode
{
   in=0x1,              // zum Lesen öffnen
```

48) Die Konstante `binary` heißt bei manchen DOS-basierten Compilern, z.B. beim Symantec C++ Compiler, `translated`. Dieser Begriff macht den eigentlichen Sinn dieses Wertes m.E. deutlicher als der Ausdruck `binary`.

```
   out=0x2,             // zum Schreben öffnen
   ate=0x4,             // Dateizeiger wird auf EOF gesetzt
   app=0x8,             // anhängendes Schreiben (impliziert out)
   trunc=0x10,          // löscht alten Dateiinhalt (impl. out)
   nocreate=0x20,       // Datei muß bereits existieren
   noreplace=0x40,      // Datei darf noch nicht existieren
   binary = 0x80        // Kein Konvertierung CR/LF -> NewLine bei
                        // input und umgekehrt bei output (wie bei
                        // MS-DOS üblich)
};
```

Diese Konstanten können – wie aus C gewohnt – ge*or*t[49)] werden. Eine Datei kann z.B. zum binären Lesen geöffnet werden durch die Anweisung

```
infile.open("file1", ios::in | ios::binary);
```

Der Fehlertest erfolgte im obigen Beispiel auf zweierlei Methoden. Zum einen mit dem überladenen `!`-Operator:

```
inline int ios::operator!() { return fail(); }
```

Zum zweiten gibt es die überladene Typkonvertierung (*overloaded casting*):

```
inline ios::operator void*() { return fail() ? 0 : this }
```

## 10.2. Wahlfreier Zugriff (Random Access)

Verschiedene Anwendungsgebiete, etwa der gesamte Bereich der Datenbanken, verlangen nach mehr Flexibilität als dies das rein sequentielle Lesen oder Schreiben bietet. Unter wahlfreiem Zugriff versteht man die Möglichkeit, den Dateizeiger (file pointer) beliebig innerhalb einer Datei zu setzen und in der Regel auch Lese- und Schreiboperationen simultan durchzuführen.

Die Elementfunktionen

---

49) Der deutsche Begriff ge*oder*t klingt vermutlich nicht besser?!

```
istream& istream::read(char *buf, int len);

ostream& ostream::write(char *buf, int len);
```

beginnen ihre Arbeit an der sogenannten aktuellen, der momentanen Position des Dateizeigers.

Das Setzen und Ermitteln des Dateizeigers übernehmen verschiedene Methoden – unterschieden nach Lese- und Schreib-Operationen (`get` bzw. `put`):

```
// absolutes Setzen (von Dateianfang aus)

istream& istream::seekg(streampos p);

ostream& ostream::seekp(streampos p);
```

```
// relatives Setzen

istream& istream::seekg(streamoff p, seek_dir d);

ostream& ostream::seekp(streamoff p, seek_dir d);
```

```
// Ermitteln der Dateizeigerposition

streampos istream::tellg(void)

streampos ostream::tellp(void)
```

Sowohl `streamoff` als auch `streampos` stehen für `int`-artige Typen, meistens `long`.

Als zweites Argument der `seekg()` bzw. `seekp()` Funktionen kann einer der Werte `ios::beg` (vom Dateibeginn aus gerechnet), `ios::cur` (von der aktuellen Cursorposition aus) und `ios::end` (vom Dateiende gerechnet) verwendet werden. Identisch sind also die folgenden beiden Aufrufe, die den Dateizeiger auf das 21.Byte – denn die Numerierung beginnt bei 0 – setzen.

```
infile.seekg(20);        // Relativadressierung ab ios::beg

infile.seekg(20,ios::beg);     // entspricht der Absolutadressierung
```

### 10.2.1. Beispiel

Das folgende Beispielprogramm `database.cpp` – angelehnt an ein Programm von Roman Gerike aus seinem Buch „*Weiter mit C++*" – demonstriert den wahlfreien Dateizugriff. In einer Textdatei (`database.dat`) stehen die folgenden minimal formatierten Daten.

```
0 Chomsky, Noam

1 Einstein, Albert
```

```
2 Krause, Jadwiga

3 Müller, Bernd

4 Schumann, Clara

5 Woolf, Virginia

9 Yondratschek, Alfons
```

Das Objekt `db` von der Klasse `DataBase` liest (im Konstruktor) zunächst diese Datei durch und merkt sich in dem Array `beg[]` die (physischen) Anfangsadressen der zu einer Indexnummer gehörenden Textzeile. In dem kleinen Testhauptprogramm werden dann die Datensatznummern 0 bis 9 abgerufen; im Falle der obigen Beispieldatei werden dabei die Sätze 6 bis 8 nicht gefunden, was auch das im Anschluß an das Programm abgedruckte Listing zeigt.

```
// ----------------------------------------------------------
// database.cpp
// Demonstrationsprogramm für die Arbeit mit wahlfreiem
// Dateizugriff; angelehnt an R. Gerikes MSGBASE.CPP
// ("Weiter mit C++", S. 212)
// ----------------------------------------------------------
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

const int MAXENTRIES = 128;          // Maximale Anzahl Datensätze
const int MAXENTRYLENGTH = 128;      // Maximale Datensatzlänge

// Rudimentäre Klasse für die Verwaltung von Datensätzen
class DATABASE
{
  private:
    unsigned int beg[MAXENTRIES];      // Index
    ifstream     file;           // File Stream für die "Datenbank"
    char         buf[MAXENTRYLENGTH]; // Buffer für Texte
  public:
    DataBase(char * name);   // Konstruktor, Name = Datenbankdateiname
    char * Msg(int nr);      // Zeiger auf Datensatz <nr>
    char * Msg();            // zuletzt gelieferter Datensatz
    istream& Stream()        // Datenbank-Stream
}; // end class DATABASE
```

```
DATABASE::DATABASE(char * name)
{
  file.open(name);     // Öffnen des Streams
  for (int i=0; i<MAXENTRIES; i++)
    beg[i]=0;          // "ungültig" als Vorbelegung
  while (file && !file.eof())
  {
    i=-1;
    file >> i >> ws;     // ws = whitespace (Trennzeichen)
    if (i>=0 && i<MAXENTRIES)
      beg[i]=int(file.tellg());
    file.ignore(MAXENTRYLENGTH,'\n');  // Überlesen bis Newline
  } // end while
  file.clear();         // eofbit löschen
} // end DATABASE::DATABASE
```

```
char * DATABASE::Msg(int nr)
{
  if (file && nr>=0 && nr<MAXENTRIES && beg[nr]) // Datensatz vorhanden
  {
    if (file.eof())
      file.clear();      // eofbit loeschen
    file.seekg(beg[nr]);
    file.getline(buf,MAXENTRYLENGTH);
  }
  else                   // Datensatz nicht vorhanden
    strcpy(buf,"Datensatz nicht vorhanden!");
  return buf;
} // end DATABASE::Msg(int)

char * DATABASE::Msg()
{
  return buf;
} // end DATABASE::Msg()

istream& DATABASE::Stream()
{
  return file;
} // end istream& DATABASE::Stream()



int main()
{
  // Testhauptprogramm
```

```
  DATABASE db("database.dat");
  if (!db.Stream())
  {
    cerr << "Datenbankfehler aufgetreten" << endl;
    return EXIT_FAILURE;
  }
  for (int nr=0; nr<10; nr++)
    cout << nr << ": " << db.Msg(nr) << endl;
  cout << "Letzter Datensatz: " << db.Msg() << endl;
  return EXIT_SUCCESS;
} // end main
// end of file database.cpp
```

Das Ablauflisting dieses Programms sieht so aus:

```
0: Chomsky, Noam

1: Einstein, Albert

2: Krause, Jadwiga

3: Müller, Bernd

4: Schumann, Clara

5: Woolf, Virginia

6: Datensatz nicht vorhanden!

7: Datensatz nicht vorhanden!

8: Datensatz nicht vorhanden!

9: Yondratschek, Alfons

Letzter Datensatz: Yondratschek, Alfons
```

# 11. Templates

C-Programmierer kennen die Vorzüge von Präprozessor-Makros. Das klassische Beispiel ist das Makro zur Bestimmung eines Minimums bzw. Maximums, wie es in `stdlib.h` zu finden ist.

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

Damit stehen durch eine einzige Zeile zahllose „Funktionen” bereit: `max()` kann für alle die Parameter(kombinationen) `a` und `b` aufgerufen werden, für die der Ersatzausdruck auf der rechten Seite der Makrodefinition Sinn macht.

Allerdings sind solche Makros auch fehleranfällig. Betrachten Sie dazu bitte die folgenden Programmzeilen.

```
char text1[]="Bergisch Gladbach", text2[]="Köln";

cout << min(a,b);
```

Was wird hier ausgegeben? – Es werden die Adressen (!) `text1` und `text2` miteinander verglichen[50], und entsprechend wird die Zeichenkette ausgegeben, die an der niedrigeren Adresse abgelegt ist.

Abhilfe gegen diese Problematik schaffen die sogenannten *Templates* (*Muster*, *Schablonen*, *Pattern*, *Vorlagen*), bei denen nicht der Präprozessor aktiv wird, sondern der C++-Compiler seine strengen Typüberprüfungen durchführen kann.

C++ kennt zwei Formen von Templates: Vorlagen für Funktionen und Klassen-Templates.

## 11.1. Template-Funktionen

Ein Template für eine Funktion ist eine Schablone, mit der formuliert wird, wie eine Funktion zu operieren hat. Dabei wird jedoch ein abstrakter, anonymer Datentyp verwendet, der dann im weiteren durch eine konkrete Verwendung erst realisiert wird.

Es werde ein Funktions-Template `func()` vereinbart wie folgt.

```
// Funktions-Template

template<class T> T func(T x)

{
```

---

[50] Hier werden nicht (inhaltlich) die Zeichenketten verglichen, denn `char text1[]` beschreibt (wie `char *text1`) C-typischerweise eine Adresse auf den Anfang der betreffenden Zeichenkette.

```
  // ... irgendwelche Aktionen mit x vom anonymen Typ T
  return x; // beispielsweise kann x zurückgegeben werden
}
```

Dann sind beispielsweise die folgenden Aufrufe korrekt.

```
int i=1, j;

j=func(i);

float f=1.23, g;

g=func(f);
```

Sehen wir uns dies in einem konkreten Programm an, in dem u.a. auch illustriert wird, wie Klassenobjekte bei solchen Funktions-Templates verwendet werden können.

### 11.1.1. Beispiel

Als einziges Beispiel für Funktions-Templates sei hier eine generische Minimum-Funktion angeführt.

```
// -----------------------------------------------------------

// templates1.cpp

// Beispiel für Funktions-Templates (Vorlagen, Schablonen).

// -----------------------------------------------------------


#include <iostream.h>

#include <stdlib.h>

#ifdef min

#   undef min

#endif

#ifndef BOOL

#   define BOOL int

#endif


// Funktions-Templates

template<class T> T min(T a, T b)

{

  return (a<b) ? a : b ;
```

```
} // end template<class T> T min(T a, T b)



// Zur Demonstration eine gekürzte Klassendefinition
class RATIONAL
{
  friend ostream& operator<<(ostream&, RATIONAL&); // Ausgabe mit <<-Op.

  private:
    int   z, n;           // Zähler, Nenner
  public:
    RATIONAL(int=0,int=1);       // Konstruktor (u.a. Default-Konstruktor)
    BOOL operator<(RATIONAL& q); // Vergleichsoperator
}; // end class RATIONAL




// Klassenimplementation RATIONAL (Hier nur eine Mini-Implementation)
RATIONAL::RATIONAL(int zz, int nn)
{
  z=zz; // Zuweisungen erfolgen hier der Kürze halber ungeprüft;
  n=nn; // die Werte werden aus demselben Grund auch nicht gekürzt.
} // end RATIONAL::RATIONAL(int,int)

BOOL RATIONAL::operator<(RATIONAL& q)
{
  return z*q.n < q.z*n;
```

```
} // end BOOL RATIONAL::operator<(RATIONAL& q)
// Ende der Klassenimplementation RATIONAL (Mini-Implementation)


// Freund-Funktion operator<< für die Ausgabe eines RATIONAL-
// Objektes
ostream& operator<<(ostream& out, RATIONAL& R)
{
  out << R.z << "/" << R.n;
  return out;
} // end ostream& operator<<(ostream&, RATIONAL&)


// -----------------------------------------------------------
// Hauptprogramm ---------------------------------------------
// -----------------------------------------------------------
int main()
{


  int i, i1=1, i2=2;
  cout << "Die kleinere int-Zahl von " << i1 << " und "
    << i2 << " ist " << min(i1,i2) << "." << endl;


  float f1=2.34, f2=3.45;
  cout << "Die kleinere float-Zahl von " << f1 << " und "
    << f2 << " ist " << min(f1,f2) << "." << endl;


  RATIONAL p(6,7), q(5,7);
  cout << "Die kleinere RATIONAL-Zahl von " << p << " und "
    << q << " ist " << min(p,q) << "." << endl;
```

```
/************************************************************

 Der Funktionsaufruf    min(p,i1)   stößt jedoch nicht auf den

 Beifall des Compilers (hier am Beispiel des Symantec C++ 6.11-

 Compilers):

        min(p,i1);

        ^

 templates1.cpp(81) :

    Error: no match for function 'min(RATIONAL,int )'

 ************************************************************/

  return EXIT_SUCCESS;

} // end main
```

Dieses Programm liefert das folgende Ablaufprotokoll.

```
Die kleinere int-Zahl von 1 und 2 ist 1.

Die kleinere float-Zahl von 2.34 und 3.45 ist 2.34.

Die kleinere RATIONAL-Zahl von 6/7 und 5/7 ist 5/7.
```

Noch eine Anmerkung: Das hier vorgestellte Template `min()` kann – anders als das Präprozessor-Makro – auch für Objekte von String-Klassen sinnvoll verwendet werden, sofern in einer solchen String-Klasse nur der Operator `<` korrekt überladen wird. Dann nämlich kann die Anweisung

```
return (a<b) ? a : b ;
```

für zwei String-Objekte `a` und `b` den inhaltlichen Vergleich anstelle des meist nicht erwünschten Adreßvergleichs bewirken.

Eine Aufgabenstellung hierzu finden Sie in Übung 18 auf Seite 0.

## 11.2. Klassen-Templates

Die zweite Form von Templates beschreibt generische Klassen. Darunter versteht man eine abstrakte Formulierung von Daten- und Funktionselementen einer Klasse, die dann für konkrete Datentypen realisiert werden können.

Dies kann dann so aussehen.

```
// Eine generische Klasse.
template <class T> class GENERIC
{
  private:
    T   member;
    int anyvalue;
  public:
    GENERIC();         // Default-Konstruktor
    GENERIC(T);          // Weiterer Konstruktor
    void AnyFunc(T);     // Eine weitere Klassenmethode
}; // end class GENERIC

template <class T> GENERIC<T>::GENERIC()
{
  anyvalue = 0;
} // end GENERIC::GENERIC()

template <class T> GENERIC<T>::GENERIC(T t)
{
  anyvalue = 1;
  member = t;
} // end GENERIC::GENERIC(T)
```

```
template <class T> void GENERIC<T>::AnyFunc(T t)
{
  // .. irgendwelche Zugriffe ..
} // end GENERIC::AnyFunc(T t)
```

Hiermit wird eine Familie von Klassen beschrieben. Konkret generiert werden solche Klassen dann durch ihre Verwendung. So werden mit

```
GENERIC<int>;
GENERIC<float>;
```

zwei komplette Klassen generiert, die einzeln und ausführlich geschrieben so aussehen[51] würden:

```
class GENERIC<int>
{
  private:
    int member;
    int anyvalue;
  public:
    GENERIC();          // Default-Konstruktor
    GENERIC(int);        // Weiterer Konstruktor
    void AnyFunc(int);  // Eine weitere Klassenmethode
}; // end class GENERIC<int>


class GENERIC<float>
{
  private:
    float member;
    int   anyvalue;
  public:
```

[51] Achtung: Dies ist keine korrekte C++-Syntax, sondern soll nur die Wirkung des Klassen-Templates `GENERIC` beschreiben!

```
    GENERIC();          // Default-Konstruktor
    GENERIC(float);     // Weiterer Konstruktor
    void AnyFunc(float);// Eine weitere Klassenmethode
}; // end class GENERIC<float>
```

### 11.2.1. Anmerkung: Containerklassen

Es gibt auch den Begriff der *Containerklasse*. Häufig wird er synonym mit Template-Klasse verwendet, streng genommen sind es jedoch unterschiedliche Dinge. Unter einer Containerklasse ist eine allgemeine, für verschiedenste Typen einsetzbare Klasse zu verstehen, die eine gewisse Datenstruktur umsetzt, z.B. einen Stack. Dies kann in C++ zum einen durch geschickt gewählte Präprozessordirektiven[52] erreicht werden, zum zweiten - und das ist der modernere, zu bevorzugende Weg - können Template-Klassen gebildet werden.

Bei Stroustrup findet sich in seinem Buch „The Design and Evoluton of C++" ein Beispiel einer Containerklasse für einen Stack, die ohne Templates formuliert ist.

```
// In der Datei stack.h:
class ELEM_STACK       // zitiert nach B. Stroustrup, leicht modifiziert
{
   private:
      ELEM *min, *top, *max;
      void new(int);
      void delete();
   public:
      ELEM_STACK(int =1024);
      void push(ELEM);
      ELEM pop();
}; // end class ELEM_STACK


// Im .cpp-File:
```

52) Es sei ausdrücklich betont, daß es das Bestreben von C++ ist, möglichst viel Arbeit vom „dummen" Präprozessor hin zum eigentlichen und strenger prüfenden Compiler zu verlagern.

```
#define ELEM long

#define ELEM_STACK long_stack

#include "stack.h"

#undef ELEM

#undef ELEM_STACK


class X; // irgendwo definiert

#define ELEM X

#define ELEM_STACK X_stack

#include "stack.h"

#undef ELEM

#undef ELEM_STACK


// nachfolgend werden nun zwei Stacks mit long-Werten (der Größe 1024) und

// ein Stack mit X-Objekten (der Größe 2048) angelegt.

// (Es geht also auch ohne Template-Klassen. - Aber wie!)

long_stack einLongStack(1024);

long_stack nochEinLongStack();

X_stack    einXStack(2048);
```

Wollen wir uns nach diesem wohl eher abschreckenden Beispiel, das die „Tricks" des Präprozessors benutzt, eine Containerklasse ansehen, die mit einer Template-Klasse realisiert wird.

## 11.2.2. Beispiel

Auch hier soll ein etwas komplexeres Gesamtprogramm vorgestellt werden, das einen generischen Stack implementiert[53]. Nach der Deklaration der generischen Stack-Klasse werden dann im Hauptprogramm zwei konkrete Stack-Klassen verwendet: eine, die int-Werte verwaltet (`STACK<int> anIntStack`), und eine, die einen Zeichen-Stack umsetzt (`STACK<char> aCharStack`). Zu Demonstratíonszwecken

[53] Ein solcher generischer Stack wurde bereits in 1.2.2.2. abstrakt vorgestellt.

wird in beiden Fällen der Stack mit Werten gefüllt und (vom Top-Of-Stack `tos` absteigend) auf den Bildschirm ausgegeben.

```
// ----------------------------------------------------------

// templates2.cpp

// Klassen-Templates am Beispiel eines Generischen Stacks.

// ----------------------------------------------------------


#include <iostream.h>

#include <stdlib.h>


const int STACKSIZE = 2048;


// ----------------------------------------------------------

// Template Stack -------------------------------------------

// ----------------------------------------------------------

template <class T> class STACK

{

  private:

     T    st[STACKSIZE]; // der Stack als einfaches Array

     int  tos;           // aktueller Stack-Index (Top-Of-Stack)

  public:

     STACK();            // Konstruktor: tos auf 0 setzen

     void Push(T);       // ein Element auf den Stack legen

     T Pop(void);         // oberstes Stack-Element holen

     int  Curr(void);    // aktuelle Stackgröße (=TOS)

}; // end class STACK
```

```
template <class T> STACK<T>::STACK()
{
  tos=0;
} // end STACK::STACK()

template <class T> void STACK<T>::Push(T c)
{
  st[tos++]=c;
} // end STACK::Push(char c)

template <class T> T STACK<T>::Pop(void)
{
  return(st[--tos]);
} // end char STACK::Pop(void)

template <class T> int STACK<T>::Curr(void)
{
  return(tos);
} // end int STACK::Curr(void)

// -----------------------------------------------------------
// Hauptprogramm
// -----------------------------------------------------------
int main()
{
  cout << "Ein char-Stack:" << endl;
  STACK<char> aCharStack;
  for (char c = 'A'; c<='Z'; c++)
```

```
    aCharStack.Push(c);
  cout << "TOS: " << aCharStack.Curr() << endl;
  for (int k=aCharStack.Curr()-1; k>=0; k--)
    cout << aCharStack.Pop();
  cout << endl << endl;

  cout << "Ein int-Stack:" << endl;
  STACK<int> anIntStack;
  for (int i=0; i<10; i++)
    anIntStack.Push(i);
  cout << "TOS: " << anIntStack.Curr() << endl;
  for (i=anIntStack.Curr()-1; i>=0; i--)
    cout << " " << anIntStack.Pop();
  cout << endl << endl;

  return EXIT_SUCCESS;
} // end of main
// ----------------------------------------------------------
// end of file templates2.cpp
// ----------------------------------------------------------
```

Nachstehend noch das Ablauflisting dieses Programms.

```
Ein char-Stack:
TOS: 26
ZYXWVUTSRQPONMLKJIHGFEDCBA

Ein int-Stack:
TOS: 10
```

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

Eine Aufgabenstellung zu Klassentemplates finden Sie in Übung 19 auf Seite 0.

# 12. Exception Handling

Die Reaktion auf eventuell auftretende Fehler ist im Rahmen der Programmierung ein ganz selbstverständliches Thema. Bisweilen jedoch führt die Fehlerbehandlung in der traditionellen prozeduralen Programmierung zu sehr unleserlichen, logisch tief verschachtelten Konstrukten.

Zur besseren Behandlung sogenannter Ausnahmesituationen (*exceptions*) stellen C und C++ verschiedene Hilfsmittel im Rahmen des *Exception Handling* bereit. Hier soll – da C++ im wesentlichen eine Obermenge von C darstellt – zunächst kurz an die Möglichkeiten von ANSI-C erinnert werden. Anschließend soll dann das in C++ verankerte Exception Handling vorgestellt werden.

## 12.1. Exception Handling in C

C kennt zwei verschiedene Möglichkeiten, auf Exceptions zu reagieren, einmal die ursprünglich aus der UNIX-Welt stammende Signalbehandlung, zum zweiten das Ausführen von nichtlokalen[54)] Sprüngen mit `setjmp()` und `longjmp()`.

### 12.1.1. Signalbehandlung

Eine spezielle Sorte von Ausnahmen sind *Signale*, die asynchron von außen an einen laufenden Prozeß geschickt werden können. Zu jedem Zeitpunkt kann beispielsweise unter UNIX ein ablaufendes Programm mit dem `kill`-Befehl abgebrochen werden, der das Signal `SIGTERM` an den Prozeß sendet. Üblicherweise kann auf dem PC oder unter UNIX mit der Tastenkombination [Strg]+[C] das Signal `SIGINT` zum Abbruch eines Programms gesendet werden.

Das nachstehende kleine Beispielprogramm soll in aller Kürze illustrieren, wie mit der Funktion `signal()` eigene Signal-Handler-Routinen implementiert werden können[55)].

```
/* except1.c - Exception Handling mit Signalverarbeitung */


#include <stdio.h>

#include <stdlib.h>
```

---

54) Ein *lokaler* Sprung innerhalb einer Funktion ist mit dem moralisch verwerflichen `goto`-Befehl möglich. Demgegenüber heißen Sprünge *nichtlokal*, wenn sie die aktuelle Funktion verlassen.

55) Auf die weiterführenden Möglichkeiten der anderen in `signal.h` definierten Funktionen (wie `sigemptyset()`, `sigfillset()` usw.) soll hier nicht eingegangen werden; hierzu sei auf den Bereich der UNIX-Systemprogrammierung verwiesen.
Siehe hierzu u.a. das Buch von Horn, *Systemprogrammierung unter UNIX*, Verlag Technik, Berlin 1994.

```
#include <signal.h>  /* Hier sind die Signal-Konstanten SIGxx definiert */
#include <string.h>

#define SIGNALS 33   /* Anzahl der Signale(+1) */
```

```
/*********************************************************************
   Ein rudimentärer Signal-Handler:
   hier wird allerdings nur mitgeteilt, welches Signal angekommen ist.
 *********************************************************************/
void Terminate(int sig)
{
  int i;
  /* Die Signale sollen auch mit ihrem Namen ausgegeben werden;
     aus rein stilistischen Gründen werden auch die Signalnummern, die
     hier konkret gar nicht abgefangen werden, mit initialisiert.
  */
  char signame[SIGNALS][13];
  for (i=0; i<SIGNALS; i++)
    strcpy(signame[i],"OTHER SIGNAL");
  strcpy(signame[SIGTERM],"SIGTERM");
  strcpy(signame[SIGINT],"SIGINT"); /* hier exemplarisch nur einige */
  strcpy(signame[SIGFPE],"SIGFPE"); /* Signale hier aufgeführt      */
  printf("\nTerminate(sig=%d=%s) ...\n",sig,signame[sig]);
  exit(EXIT_FAILURE);
} /* end Terminate */

int main(void)
{
  int ct;

  /* Die folgenden drei Signale werden abgefangen */
  signal(SIGTERM,Terminate);
  signal(SIGINT,Terminate);
```

```
  signal(SIGFPE,Terminate);


  printf("\nTestprogramm zur Signalbehandlung.");
  for (ct=1;;ct++)
  {
    printf("\nLooping %d...",ct);
    sleep(1);
    if (ct==3) /* Eine mutwillig erzeugte FPE (floating point exception)*/
      ct=1/(ct-3);
  }
  return EXIT_SUCCESS;
} /* end main */
```

Und hier noch einige Ablauflistings dieses Programms.

1.Durchlauf: Abbruch mit [Strg]+[C].

```
Testprogramm zur Signalbehandlung.
Looping 1...
Looping 2...
Terminate(sig=2=SIGINT) ...
```

2.Durchlauf: Abbruch durch die Floating Point Exception bei `ct=3`

```
Testprogramm zur Signalbehandlung.
Looping 1...
Looping 2...
Looping 3...
Terminate(sig=8=SIGFPE) ...
```

3.Durchlauf: Abbruch mit dem `kill`-Befehl

```
Testprogramm zur Signalbehandlung.

Looping 1...

Looping 2...

Terminate(sig=15=SIGTERM) ...
```

### 12.1.2. Nichtlokale Sprünge

Neben der Signalbehandlung bietet ANSI-C die Möglichkeit, nichtlokale Sprünge durchzuführen. Davon sollte in einer gut strukturierten Programmierung allerdings äußerst sparsam Gebrauch gemacht werden, denn die Übersichtlichkeit des Programms leidet unter solchen Sprüngen naturgemäß gewaltig!

Für nichtlokale Sprünge, also Sprünge, die auch nach außerhalb einer Funktion oder eines Moduls führen können, stehen zwei Funktionen (in `setjmp.h`) bereit: `setjmp()` und `longjmp()`.

Mit `setjmp()` wird der aktuelle Programmzustand in einer Variablen vom Typ `jmp_buf` festgehalten; mit `longjmp()` kann im Bedarfsfall von jeder beliebigen Stelle im Programm zu diesem Zustand zurückgekehrt werden.
Wichtig sind dabei zwei Details: Die Funktion `setjmp()` muß – natürlich – vor der ersten Verwendung von `longjmp()` aufgerufen werden, und die Funktion, aus der heraus `setjmp()` aufgerufen worden ist, muß zum Zeitpunkt des `longjmp()`-Aufrufes noch aktiv sein[56].

Auch hier soll ein kleines Demonstrationsprogramm weitere abstrakte Erläuterungen ersetzen.

```
/* except2.c - Exception Handling mit setjmp()/longjmp() */


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <setjmp.h>  /* Deklaration von jmp_buf, setjmp() und longjmp() */


jmp_buf buffer;  /* Programmzustand wird festgehalten */
```

---

[56] Andernfalls ist das Resultat per ANSI-C-Standard undefiniert!

```
int main(void)
{
  int i;

  printf("\nTestprogramm mit setjmp() und longjmp().\n");
  printf("setjmp() liefert %d\n",setjmp(buffer));
  printf("Bitte int-Wert eingeben: ");
  scanf("%d",&i);
  if (i < 0)
    longjmp(buffer,i);
  printf("Regulaere Weiterverarbeitung und Programmende.\n");
  return EXIT_SUCCESS;
} /* end main */
```

Auch hier soll ein kurzes Ablauflisting abgedruckt werden.

```
Testprogramm mit setjmp() und longjmp().
setjmp() liefert 0
Bitte int-Wert eingeben: -1
setjmp() liefert -1
Bitte int-Wert eingeben: -2
setjmp() liefert -2
Bitte int-Wert eingeben: 1
Regulaere Weiterverarbeitung und Programmende.
```

Anmerkung: Auf die über das oben beschriebene hinaus bestehende Möglichkeit, in ANSI-C mit dem Makro `assert()` Hilfe bei der Fehlerdiagnose zu erhalten, soll hier nicht eingangen werden.

## 12.2. Exception Handling in C++

ANSI-C++ sieht auch das Exception Handling mit vor, auch wenn es derzeit noch längst nicht von allen C++-Compilern umgesetzt wird. Im folgenden soll der bisherige AT&T-Standard 3.0 zu diesem Thema behandelt werden[57)], allerdings (wiederum aus Platzgründen) nur anhand von zwei einfachen Beispielen, die den grundlegenden Sachverhalt verdeutlichen sollen.

### 12.2.1 Beispiel

Dieses Beispiel illustriert die Verwendung der drei neuen C++-Schlüsselworte `try`, `throw` und `catch`. In einem `try`-Block wird eine Aktion formuliert, die nach Möglichkeit durchgeführt werden soll. Dabei werden in auftretenden Fehlersituationen Fehlermeldungen mit dem `throw`-Befehl „geworfen" – und zwar an einen (anschliessenden) `catch`-Block, der diese Fehlermeldungen „einzusammeln" hat. Bitte beachten Sie: wird ein `throw` ausgeführt, so wird der betreffende `try`-Block vollständig verlassen.

Im folgenden kleinen Programm wird versucht, eine nicht existente Datei `does.not.exist` zum Lesen zu öffnen; anschließend soll die Datei `/etc/passwd` zum Schreiben geöffnet werden.

```
// except3.cpp - Exception Handling in C++

// Compilation unter UNIX mit "CC +eh except3.cpp"



#include <iostream.h>

#include <stdlib.h>



const int FILENOTEXIST = 101;

const int FILENOWRITE  = 102;



// Die Fehlerbehandlungsroutine

void Caught(int errstatus)

{

  cerr << "catch: ";

  switch(errstatus)
```

---

57) Vgl. hierzu den Beitrag "*Fehlerfang*" aus der Zeitschrift *iX*, Ausgabe 4/94, S. 196.

```
  {
    case FILENOTEXIST:
      cerr << "Die Datei existiert nicht "
        << "oder darf nicht gelesen werden!" << endl;
      break;
    case FILENOWRITE:
      cerr << "Die Datei kann nicht geschrieben werden!"
        << endl;
      break;
    default: // kann hier konkret allerdings nicht auftreten
      cerr << "Sonstiger Fehler aufgetreten!" << endl;
      break;
  } // end switch
} // end Caught


int main()
{
  // try - Wir versuchen etwas...
  try
  {
    FILE *fp; // lokale Gültigkeit innerhalb des Blockes
    cout << "Öffnen der Datei does.not.exist:" << endl;
    if ((fp=fopen("does.not.exist","r"))==NULL)
      throw FILENOTEXIST;
  } // end try
  // catch - Die Ausnahmebehandlungen zum obigen try-Block
  catch(int i)
```

```
{
  Caught(i);
} // end catch
```

```
  // try - Wir versuchen noch mehr...
  try
  {
    FILE *fp;
    cout << "Öffnen der Datei /etc/passwd zum Schreiben:" << endl;
    if ((fp=fopen("/etc/passwd","w"))==NULL)
      throw FILENOWRITE;
    cout << "Diese Ausgabe schließt den zweiten try-Block ab." << endl;
  }

  // catch - Die Ausnahmebehandlungen...
  catch(int i)
  {
    Caught(i);
  } // end catch(int)
  return EXIT_SUCCESS;
} // end main
```

Das Ablauflisting dieses Programms sieht so aus:

```
Öffnen der Datei does.not.exist:
catch: Die Datei existiert nicht oder darf nicht gelesen werden!
Öffnen der Datei /etc/passwd zum Schreiben:
catch: Die Datei kann nicht geschrieben werden!
```

Generell können `try`-Blöcke auch geschachtelt werden. In einem solchen Fall wird bei einem `throw` im inneren Block zunächst in dem inneren `catch`-Block nach einer passenden Fehlerbehandlung gesucht; wird dort keine gefunden, so wird das äußere `catch` berücksichtigt.

## 12.2.2. Noch ein Beispiel

Das folgende Beispiel ist ein wenig komplexer. Hier wird im Ansatz gezeigt, wie z.B. für eine `ARRAY`-Klasse bei Indexüber- oder unterschreitungen das Exception Handling implementiert werden könnte. Dabei werden nebenbei auch lokale Klassen – also Klassen, die innerhalb einer anderen Klasse definiert sind – vorgestellt: innerhalb der Klasse `ARRAY` wird die Klasse `AERROR` (Array-Error-Klasse) definiert, die für das Exception Handling innerhalb der Array-Klasse zuständig ist[58)].

Die Klasse `ARRAY` besteht hier der Übersichtlichkeit wegen nur aus einem Datenelement `data` mit fünf `int`-Komponenten sowie einer Methode `Set()` zum Setzen dieser Speicherplätze.

Die Klasse `ARRAY::AERROR` besitzt ein privates Datenelement `error`, das inhaltlich die Fehlerursache verwaltet; im Konstruktor wird es initialisiert. Daneben gibt es eine Funktion `Explain()`, die für die Ausgabe eines Fehlertextes auf den Fehlerkanal `cerr` sorgt.

```
// except4.cpp - Exception Handling in C++

// Compilation unter UNIX mit "CC +eh except4.cpp"


#include <iostream.h>

#include <stdlib.h>


const int MAX = 5;              // Arraygröße

const int IDXOVERFLOW  = 1;     // selbstsprechende Konstanten

const int IDXUNDERFLOW = 2;


class ARRAY

{

  private:

    int data[MAX];

  public:

    void Set(int, int); // Setzen eines Elementes
```

58) Bitte beachten: die Klasse `AERROR` ist keine von `ARRAY` abgeleitete Klasse!

```
    class AERROR        // eingelagerte Fehlerbehandlungsklasse
    {
      private:
        int error;
      public:
        AERROR(int i)  { error=i; }; // Konstruktor
        void Explain(void);          // Fehlermeldungsroutine
    }; // end class ARRAY::AERROR

}; // end class ARRAY

void ARRAY::Set(int pos, int val)
{
  if (pos >= MAX)
    throw AERROR(IDXOVERFLOW);
  if (pos < 0)
    throw AERROR(IDXUNDERFLOW);
  data[pos]=val;
} // end ARRAY:Set

void ARRAY::AERROR::Explain(void)
{
  cerr << "Fehler: ";
  if (error==IDXOVERFLOW)
    cerr << "Indexüberlauf aufgetreten!";
  else if (error==IDXUNDERFLOW)
    cerr << "Indexunterlauf aufgetreten!";
  else // tritt hier aktuell nicht auf
```

```
    cerr << "Absolut überraschender sonstiger Fehler aufgetreten!";
  cerr << endl;
} // end ARRAY::AERROR::Explain
```

```
int main()
{
  try
  {
    ARRAY anArray;
    anArray.Set(MAX,3);
  } // end try
  catch(ARRAY::AERROR aerror)
  {
    aerror.Explain();
    return(EXIT_FAILURE);
  } // end catch
  return EXIT_SUCCESS;
} // end main
```

Das zu erwartende Ablaufprotokoll dieses Programms:

```
Fehler: Indexüberlauf aufgetreten!
```

Diese Seite wurde absichtlich leer gelassen.

# 13. Ausblick

Abschließend noch ein paar Worte als Ausblick, wie es weiter gehen könnte.

Zunächst einmal war es in den vorigen zwölf Kapiteln natürlich nur möglich, eine prinzipielle Einführung in die wesentlichen Aspekte der Programmiersprache C++ zu geben. Dabei konnte, wie es bei ordentlichen Einführungen in C++ immer der Fall sein sollte, das Bibliothekskonzept[59)] zwar vorgestellt werden, aus Platzgründen ist jedoch leider die praktische Arbeit mit Klassenbibliotheken, wie z.B. den Motif-Klassenbibliotheken[60)] unter dem X Window System oder der unter den verschiedenen Microsoft Windows Derivaten eingesetzten Microsoft Foundation Class (kurz *MFC*), zu kurz gekommen.

Auch auf die im Juli 1994 vom ANSI-Komitee als Bestandteil des C++ Standards verabschiedete Standard Template Library (*STL*) konnte leider nicht eingegangen werden. Die STL ist eine u.a. von Hewlett Packard entwickelte Container-Klassenbibliothek, die mit sehr allgemeinen („generischen”) Algorithmen arbeitet, welche für die verschiedensten konkreten Klassen oder andere Container eingesetzt werden können. Sie umfaßt unter anderem Vektoren, verkettete Listen, Stacks, Queues und Dequeues, Mengen und Warteschlangen. Hierzu sei auf die im Literaturverzeichnis genannten Bücher (S. 0) verwiesen. Weitere Informationen zur STL findet man im World Wide Web[61)] unter den folgenden Adressen.

a) http://www.cs.rpi.edu/~musser/stl-what.html - Was ist die STL? (Unter der Adresse http://www.cs.rpi.edu/~musser/doc.ps liegt dieses Dokument im PostScript-Format vor.)

b) http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html - Einführung in die STL für Neulinge.

c) http://web2.airmail.net/markn/stl/stl.htm - Ein Programmer's Guide in die STL.

Bereits im Vorwort[62)] erwähnte ich auch die Frequently Asked Questions (*FAQ*s), die im World Wide Web z.B. an den folgenden Stellen gefunden werden können.

a) Unter der Adresse http://www.cerfnet.com/~mpcline/C++-FAQs-Lite/ ist die HTML-Version von Marshall Cline zu finden.

---

59) Dieses Konzept ist allerdings bereits in C für die Praxis erforderlich, weshalb wir in diesem Rahmen von Grundkenntnissen diesbezüglich ausgingen.

60) Man spricht auch hier von einem Klassenkonzept: obschon die basierende Programmiersprache C nicht objektorientiert ist, ist es aber die Architektur der Bibliothek.

61) Mag man auch nichts vom „Surfen” im World Wide Web halten: etliche Informationen hält es nun wirklich bereit, wenn man nur weiß, wo!

62) Doch: wer liest schon das Vorwort?

b) FAQ-Ressourcen zum Download via ftp:
ftp://rtfm.mit.edu/pub/usenet-by-group/comp.lang.c++/C++_FAQ:_posting_#1_4
ftp://rtfm.mit.edu/pub/usenet-by-group/comp.lang.c++/C++_FAQ:_posting_#2_4
ftp://rtfm.mit.edu/pub/usenet-by-group/comp.lang.c++/C++_FAQ:_posting_#3_4
ftp://rtfm.mit.edu/pub/usenet-by-group/comp.lang.c++/C++_FAQ:_posting_#4_4

c) Und ideal für den Download und zum Offline-Lesen als ein einziges HTML-Dokument: http://www.connobj.com/cpp/cppfaq.htm.

d) Zu Microsoft Visual C++ und den MFC gibt es eine eigene FAQ-Liste:
http://www.unx.com/~scot/mfc_faq.html

e) Zur STL gibt es bei Hewlett Packard eine sehr gute FAQ-Liste:
ftp://butler.hpl.hp.com/stl/stl.faq

f) Zur Thematik plattformunabhängiger graphischer Benutzeroberflächen, den berühmt-berüchtigten *graphical user interfaces* (*GUI*s), gibt es ebenfalls eine FAQ-Liste:
http://www.zeta.org.au/~rosko/pigui.htm

Außerdem sei noch auf die World Wide Web C++ Virtual Library am DESY in Hamburg unter der Adresse http://info.desy.de/user/projects/C++.html hingewiesen, wo ebenfalls eine ganze Menge Informationen zum Thema C++ vorliegen.

Abschließend sei noch auf die Webseiten des Autors hingewiesen: unter der Adresse http://www.bg.bib.de/~dozbm/oop/cpp/ werden künftig aktualisierte Links auf den Gebieten C++ und Objektorientierte Programmierung zu finden sein.

# Übungen

In diesem Kapitel sollen dem fleißigen Leser einige Übungsaufgaben als Anregung mit auf den Weg gegeben werden.

## Übung 1: Das erste Programm

Nachstehend sehen Sie das kleine Programm helloworld.cpp; compilieren Sie es bitte und starten Sie das fertige ausführbare Programm.

Wofür steht die Formulierung `<< endl` in dem Programm? Wenn es Ihnen nicht unmittelbar klar sein sollte: Finden Sie es bitte heraus, indem Sie diesen Teil auskommentieren.

```
// helloworld.cpp

// Ein erstes Demonstrationsprogramm in C++


#include <iostream.h>

#include <stdlib.h>


int main()

{

    cout << "Hello, world!" << endl;

    return EXIT_SUCCESS;


} // end main

// end of file helloworld.cpp
```

## Übung 2: Überladene Funktionen

Schreiben Sie bitte zwei Funktionen Ausgabe() mit den Prototypen

```
void Ausgabe(char c);
```

bzw.

```
void Ausgabe(char *s);
```

die dafür sorgen, daß entsprechend ein einzelnes Zeichen oder aber eine ganze Zeichenkette[63] schön durch Sterne eingerahmt ausgegeben wird. (Zur Zeichenketten-problematik, die Sie aus C kennen dürften, folgen noch einige Übungen: wir werden hier zu einer C++-Klasse `STRING` übergehen, die uns ein sichereres Arbeiten mit Zeichenketten erlaubt als die `char *` Konstrukte von C.)

So soll der Aufruf

```
Ausgabe('A');
```

dafür sorgen, daß folgende Ausgabe auf dem Bildschirm erscheint:

```
*****

* A *

*****
```

Analog soll

```
Ausgabe("Autobahnraststätte");
```

für die nachstehende Bildschirmausgabe sorgen:

```
**********************

* Autobahnraststätte *

**********************
```

## Übung 3: Überladene Funktionen

Erweitern Sie das Programm aus Aufgabe 2 bitte um die folgende dritte Version von Ausgabe() mit dem Prototypen

```
void Ausgabe(char *s, char zeichen);
```

wobei die Arbeitsweise analog wie zuvor sein soll, nur daß statt des Sterns als Einrahmungszeichens das in der Variablen `zeichen` übergebene Zeichen verwendet werden soll.

Beispiel: Der Aufruf

---

[63] Sicherheitshalber sei an dieser Stelle noch einmal darauf hingewiesen, daß `char *` (wie schon bei C) natürlich im technischen Sinne nicht eine Zeichenkette selbst repräsentiert sondern nur einen Zeiger darauf!

```
Ausgabe("C++ ist wundervoll!",'+');
```

führt zu der folgenden Ausgabe.

```
+++++++++++++++++++++++

+ C++ ist wundervoll! +

+++++++++++++++++++++++
```

Der Funktionsaufruf

```
Ausgabe("Die Renten sind sicher!",'?');
```

dagegen liefert die folgende Ausgabe.

```
??????????????????????????

? Die Renten sind sicher!?

??????????????????????????
```

Zusatzfrage: Können Sie nach Implementation dieser Erweiterung Ihr Programm wieder vereinfachen? Haben Sie eine Idee? Gelingt es Ihnen eventuell, mehrere dieser Funktionen zu einer zusammenzufassen?

## Übung 4: Tauschen

Schreiben Sie eine C++-spezifische Variante der einfachen Funktion `Tauschen()`: es sollen die Inhalte von zwei übergebenen int-Variablen ausgetauscht werden. Verwenden Sie hierfür die Übergabe per Referenz, die in ANSI-C bekanntlich nicht zur Verfügung steht.

Ein korrekter Aufruf der Funktion `Tauschen()` kann also etwa sein:

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

Tauschen(a,b);          // Kein Tippfehler!
```

Erweitern Sie Ihr Programm anschließend so, daß auch die Werte zweier float-Variablen `x` und `y` durch den Aufruf

```
Tauschen(x,y);
```

ausgetauscht werden können.

## Übung 5: Tastatureingabe in C++

Es seien folgende Deklarationen gegeben.

```
const int SIZE = 128;

int i, j, k;

char c;

char txt[SIZE];     // Das waere uebrigens in C nicht moeglich: ein Array

          // mit einem const int zu dimensionieren!

float f;
```

Mit der Syntax

```
cin >> i;

cin >> c;

cin >> txt;

cin >> f;
```

können erwartungsgemäß die entsprechenden Variablen interaktiv von Tastatur eingelesen werden. Testen Sie dies bitte an einem kleinen Programm aus. Probieren Sie auch die Wirkung der beiden folgenden Anweisungen aus.

```
cin >> i >> j >> k;

cin >> c >> i;
```

## Übung 6: Rechnen mit Brüchen

Nachstehend finden Sie die Datei brueche.cpp, in der eine einfache, rudimentäre Klasse BRUCH definiert wird.

a) Nehmen Sie eine Klassenfunktion `Eingabe()` in die Klasse `BRUCH` auf, die für die korrekte Eingabe eines gültigen Bruches sorgt. Das heißt: Zähler und Nenner können beliebige int-Werte sein, der Nenner darf jedoch nicht 0 sein! Sollte ein Anwender für den Nenner 0 eingeben, so ist die Eingabe des Nenners zu wiederholen.

b) Nachfolgend sehen Sie (auf Seite 0) die beiden Hilfsfunktionen `Kuerzen()` und `GGT()` (vgl. hierzu auch die Kommentare im Quelltext). Nehmen Sie diese Funktionen bitte mit auf in die Klasse `BRUCH` und sorgen Sie dafür, dass von nun an alle Brüche, mit denen Sie arbeiten, stets gekürzt sind! - In welchen Klassenfunktionen müssen Sie dies berücksichtigen?

c) Erweitern Sie die Klasse `BRUCH` bitte um die drei naheliegenden Methoden `Subtrahiere()`, `Multipliziere()`, `Dividiere()`. Sie dürfen die Funktionen (Methoden) auch umbenennen, wir ersetzen diese später sowieso durch die harmonischere Operatorschreibweise!

d) Ergänzen Sie zu Testzwecken die Klasse `BRUCH` bitte so, daß in jedem Konstruktor eine kurze Bildschirmausgabe erscheint („Hier meldet sich der Konstruktor...”).

e) Ergänzen Sie die Klasse dann noch um einen Destruktor, der als einzige Aktion ebenfalls eine kleine Meldung („Hier meldet sich der Destruktor...”) ausgibt.

```
// ----------------------------------------------------------------------

// brueche.cpp

// ----------------------------------------------------------------------


#include <iostream.h>


class BRUCH

{

   private:

      int z, n;            // Zaehler und Nenner
```

```
  public:
    BRUCH();              // Default-Konstruktor
    BRUCH(int,int);       // Konstruktor
    BRUCH Addiere(BRUCH);
    void Ausgabe();
};
```

```
// Implementation der Klassen-Methoden (Mitgliedsfunktionen)
// ----------------------------------------------------------


BRUCH::BRUCH()              // Default-Konstruktor
{
   z=0;
   n=1;
} // end BRUCH::BRUCH()



BRUCH::BRUCH(int zaehler, int nenner) // weiterer Konstruktor
{
   // Minimale Notfallbehandlung bei nenner==0 !
   if (nenner!=0)
   {
      z=zaehler;    // ausfuehrlich:  this->z=zaehler;
      n=nenner;     // this ist ein Zeiger auf das aktuelle Objekt
   }
   else
   {
      z=0;
      n=1;
   }
} // end BRUCH::BRUCH(int,int)



BRUCH BRUCH::Addiere(BRUCH q1)
{
```

```
   BRUCH q;
   q.z = z*q1.n + q1.z*n;
   q.n = n * q1.n;
   return(q);
} // end BRUCH::Addiere



void BRUCH::Ausgabe()
{
   cout << z << "/" << n;
} // end BRUCH::Ausgabe



// ----------------------------------------------------------

void main()                        // synonym fuer void main(void)
{
   BRUCH q1(1,1), q2(1,2), q3;      // q3 ist defaultmaessig 0/1

   // Die Anweisung q1.z=1; ist nun illegal, da z ein privates
   // Datenelement des Objektes q1 ist!

   // Addieren von q1 und q2 in die Variable q3
   q3=q1.Addiere(q2);

   // Ausgabe der Summe
   q1.Ausgabe();
   cout << " + ";
```

```
   q2.Ausgabe();

   cout << " = ";

   q3.Ausgabe();

   cout << endl;

} // end main

// end of file brueche.cpp
```

Nachfolgend die Prototypen und eine mögliche Implementation der Funktionen `GGT()` und `Kuerzen()`, die Sie für die gekürzte Darstellung eines Bruches einsetzen können.

```
// -----------------------------------------------------------------------

// Prototypen

// -----------------------------------------------------------------------

int GGT(int, int);

void Kuerzen(int&, int&);

// -----------------------------------------------------------------------



// -----------------------------------------------------------------------

// Implementation

// -----------------------------------------------------------------------

int GGT(int a, int b)

{

  if (a==0 || b==0)

     return(1);

  if (a<0)

     a *= (-1);

  if (b<0)
```

```
      b *= (-1);

   while (a!=b)
   {
      if (a>b)
         a-=b;
      else // if (b>a)
         b-=a;
   }
   return(a);
} // end GGT(int, int)
```

```
void Kuerzen(int& z, int& n)
{
  int ggt;
  if ((ggt=GGT(z,n))>1)
  {
     z/=ggt;
     n/=ggt;
  }
  if (n<0)  // Nenner stets positiv
  {
     z = -z;
     n = -n;
  }
} // end Kuerzen(int&,int&)
```

## Übung 7: Konstruktoren und Destruktoren

a) Erweitern Sie die in Übung 6 verwendete Klasse `BRUCH` um einen Kopierkonstruktor, der naheliegenderweise lediglich dafür sorgen soll, daß die Komponenten z und n (Zähler und Nenner) explizit kopiert werden.
Zur Erinnerung: Der Prototyp dieses Kopierkonstruktors ist `BRUCH::BRUCH(const BRUCH&);` oder `BRUCH::BRUCH(BRUCH&);`

b) Ergänzen Sie alle Konstruktoren und Destruktoren um eine kleine Ausgabe der Form „Hier meldet sich der Konstruktor...” etc. Sehen Sie sich damit an, wann und wie oft welche Konstruktoren bzw. wie oft der Destruktor aufgerufen wird.

## Übung 8: Statische Klassenmitglieder

Ergänzen Sie die Klasse `BRUCH` (siehe Übung 6 und Übung 7) um ein statisches Element `static int anzahl;` das zunächst auf 0 initialisiert wird. Setzen Sie es in jedem Konstruktor um 1 hoch, im Destruktor subtrahieren Sie 1 davon.

Erweitern Sie Ihre Bildschirmausgaben der *struktoren[64] um die Ausgabe des momentanen Wertes von `anzahl`. Beobachten Sie wiederum, wie die *struktoren implizit aufgerufen werden!

## Übung 9: Operator Overload

Ändern Sie in der Klasse `BRUCH` die Methoden `Addiere()`, `Subtrahiere()`, `Multipliziere()` und `Dividiere()` ab, so daß daraus überladene Operatoren (`operator+`, `operator-`, `operator*` und `operator/`) werden.

## Übung 10: Operator Overload

a) Erweitern Sie unsere Beispielklasse `BRUCH` um Operatoren `operator+`, `operator-`, `operator*` und `operator/` dergestalt, daß für einen `BRUCH q` und eine int-Zahl `i` sowie eine long-Zahl `longval` die folgenden Rechenausdrücke zu bilden sind:

```
q+i   q-i   q*i   q/i   // falls i != 0
q+longval q-longval q*longval q/longval // für longval!=0L
```

   Das Ergebnis soll natuerlich wieder vom Typ `BRUCH` sein.

b) Und nun möchten wir gerne noch Brüche mit float-Zahlen verrechnen können. Ergänzen Sie also die Klasse `BRUCH` um Operatoren `+`, `-`, `*` und `/`, mit denen ein `BRUCH q` mit einer float-Zahl `f` verarbeitet werden kann: die Ausdrücke `q+f`, `q-f`, `q*f` und `q/f` (für `f` ungleich 0.0) sollen somit alle sinnvoll sein und naturgemäß etwas vom Typ float zurückliefern.

c) Zusatzfrage zum Verständnis: wieso akzeptiert der Compiler trotz Ihrer obigen Bemühungen in a) und b) noch immer keine Ausdrücke der Form `i+q` oder `f*q`?

## Übung 11: Referenzen

Was sind Referenzen in C++? Erläutern Sie kurz und präzise die Unterschiede und die Gemeinsamkeiten zwischen Pointern und Referenzen in C++.

## Übung 12: Klassendesign

In einer Firma sollen Abteilungen verwaltet werden. Entwerfen Sie das Design einer C++-Klasse `ABTEILUNG`, so daß die nachstehend genannten Anforderungen erfüllt werden können.

---

[64] Mit **struktoren* meinen wir natürlich die Konstruktoren und den Destruktor!

Sie sollen allerdings an dieser Stelle nicht die Klassenmethoden implementieren, sondern lediglich die Klassendeklaration (Datenelemente, Prototypen der Methoden) vornehmen! Datenelemente müssen hierbei im Schutzbereich `private` verwaltet werden.

a) Jede Abteilung hat eine Bezeichnung, die maximal dreißig Zeichen lang sein darf und bereits beim Anlegen der Abteilung feststeht.

b) Zu jeder Abteilung sollen (genau) vier Mitarbeiter gehören.

c) Zu jedem Mitarbeiter werden der Name, die (ganzzahlige, maximal fünfstellige) Personalnummer sowie das Geburtsdatum verwaltet.

d) In jeder Abteilung gibt es genau eine/n Abteilungsleiter/in, der/die natürlich einer der o.g. Mitarbeiter dieser Abteilung ist.

Bedenken Sie bei Ihrem Design (Auswahl der Zugriffsmethoden), welche Anfragen an eine Abteilung bei den oben aufgeführten Datenelementen sinnvoll gestellt werden können.

## Übung 13: Ein fehlerhaftes Programm

Sehen Sie sich bitte das nachstehende kleine Programm an. Welche(n) Fehler enthält es?

```
1  // Ein fehlerhaftes Programm.

2

3  #include <iostream.h>

4  #include <stdlib.h>

5

6  int& Rechne();                        // Prototyp

7

8  int main()                            // Hauptprogramm

9  {

10   int  i=10, j;

11   j=Rechne(i);

12   cout << "j=" << j << endl;

13   return EXIT_SUCCESS;
```

```
14 } // end main

15

16

17 int& Rechne(int startwert)            // Implementierung

18 {

19   return startwert*startwert;

20 } // end Rechne()
```

## Übung 14: Vererbungslehre - Klasse ANGESTELLTER

Im nachfolgenden Quelltext vererbung.cpp auf Seite 0 finden Sie ein Beispielprogramm zur einfachen und mehrfachen Vererbung.

a) Welche Bedeutung hat das Schlüsselwort `virtual` in Zeile 60 (auf Seite 0) bei der Deklaration der Klasse `LTDANGEST`? Wie verändert sich ein Objekt der Klasse `LTDANGEST`, wenn die Klasse `LTDANGEST` von `ANGEST` ohne `virtual` abgeleitet wird? Hinweis: Betrachten Sie insbesondere mit `sizeof` die Größe eines Objektes der Klasse `LTDANGEST`!
Was spricht dafür, das Schlüsselwort `virtual` wegzulassen? Was spricht dafür, es stehen zu lassen?

b) Implementieren Sie eine Klasse `CHEF`, die den Firmenchef modelliert. Ein `CHEF` ist insbesondere ein Leitender Angestellter, verfügt jedoch neben dem Dienstwagen auch noch über einen Chauffeur, der mit seinem Namen (maximal 30 Zeichen) abgespeichert werden soll.

c) Wenn Sie später feststellen[65], daß in der zu modellierenden Firma neben den Angestellten auch noch Arbeiter/innen beschäftigt sind, was ändern Sie an dem bis hierher vorhandenen Klassendesign?
Wir wollen hierbei davon ausgehen, daß ein Arbeiter - genau wie ein Angestellter - einen Namen und eine Personalnummer besitzt, daß daneben allerdings auch noch abgespeichert wird, in welchem der drei Werke Bergisch Gladbach, Aachen oder Düren der betreffende Arbeiter beschäftigt ist. Er soll in genau einem dieser Werke tätig sein!

d) Nachdem Sie den vorherigen Teil c) implementiert haben: erweitern Sie die Informationen, die zu einem Angestellten verwaltet werden, um den Namen der Abteilung, in der er arbeitet: Lohn, Vertrieb, Hotline oder Verwaltung. Wieder soll nur genau eine Eintragung vorgenommen werden (können).

```
1  // ------------------------------------------------------------------
```

[65] Natürlich wissen Sie das in der Praxis schon vorher, aber das ist ja auch nur ein Beispiel...

```
2  // vererbung.cpp - // Kleines Beispiel zur Vererbung (Inheritance)
3  // -------------------------------------------------------------------
4  //
5  // Übersicht über die Vererbungshierarchie:
6  //
7  //                            ANGESTELLTER
8  //                                 |
9  //                +-----------------------------------+
10 //                |                                   |
11 //      LEITENDER ANGESTELLTER             VERTRIEBSANGESTELLTER
12 //                |                                   |
13 //                |                                   |
14 //                +-----------------------------------+
15 //                                 |
16 //                 LEITENDER ANGESTELLTER IM VERTRIEB
17 //
18 // Namen im Source:
19 // ANGEST .......... ANGESTELLTER
20 // LTDANGEST ........LEITENDER ANGESTELLTER
21 // VERTRANGEST ......VERTRIEBSANGESTELLTER
22 // LTDVERTRANGEST ...LEITENDER ANGESTELLTER IM VERTRIEB
23 //
24 // Die Klasse LTDVERTRANGEST ist mehrfach abgeleitet von LTDANGEST
25 // und von VERTRANGEST. Mehrfachvererbung existiert nicht bei allen
26 // objektorienterten Programmiersprachen, insbesondere nicht bei
27 // Smalltalk.
28 //
29 // -------------------------------------------------------------------
```

```
30
31 #define DEVELOP  // nur während der Entwicklungsphase aktivieren:
32                  // dient für Diagnose, die in der endgültigen
33                  // Programmfassung nicht mehr erforderlich sind.
34
35 #include <iostream.h>
36 #include <stdlib.h>
37 #include <string.h>
38
39 // --- Deklaration der Klasse ANGEST -------------------------------
40 class ANGEST
41 {
42    protected:
43       // Wenn man die Konstante(n) in die Klasse integrieren (und damit
44       // verbergen) möchte, dann ist der enum-Weg möglich:
45       enum interneKonstantennamen { STRINGLENGTH=20 };
46       // (Zu Demonstrationszwecken hier nur auf 20 gesetzt.)
47
48       // Eigentliche Datenelemente
49       char name[STRINGLENGTH];
50       int  persnr;
51    public:
52       friend ostream& operator<<(ostream&,ANGEST&);
53       ANGEST(char * = "kein Name", int = 0);
54       ~ANGEST();
55
56 }; // end class ANGEST
57
```

```
58
59 // --- Deklaration der Klasse LTDANGEST -----------------------------
60 class LTDANGEST : virtual public ANGEST
61 {
62    protected:
63       char  polkennzeichen[12];
64                            // Der Dienstwagen wird hier einfach nur
65                            // über die Autonummer modelliert. In
66                            // der Praxis gäbe es z.B. eine Klasse
67                            // DIENSTWAGEN, in der alles geregelt ist...
68    public:
69       friend ostream& operator<<(ostream&,LTDANGEST&);
70       LTDANGEST(char *, int, char * = "undefiniert");
71       ~LTDANGEST();
72 }; // end class LTDANGEST
73
74
75 // --- Deklaration der Klasse VERTRANGEST ---------------------------
76 class VERTRANGEST : virtual public ANGEST
77 {
78    protected:
79       char laptop[ANGEST::STRINGLENGTH]; // Kennung des Laptops
80    public:
81       friend ostream& operator<<(ostream&,VERTRANGEST&);
82       VERTRANGEST(char *, int, char * = "kein Laptop");
83       ~VERTRANGEST();
84 }; // end class VERTRANGEST
85 // --- Deklaration der Klasse LTDVERTRANGEST ------------------------
```

```
86 class LTDVERTRANGEST : public LTDANGEST, public VERTRANGEST
87 {
88    public:
89       friend ostream& operator<<(ostream&,LTDVERTRANGEST&);
90       LTDVERTRANGEST(char *, int,
91                      char * = "undefiniert", char * = "kein Laptop");
92       ~LTDVERTRANGEST();
93
94 }; // end class LTDVERTRANGEST
95
96
97
98 // --- Implementation der Klasse ANGEST -----------------------------
99 // Anm.: Einige C++ Entwickler geben den Übergabeparametern
100 //       in diesem Kontext denselben Namen wie den zu setzenden
101 //       Datenelementen, erweitert lediglich um einen Unterstrich.
102 //       Diese oder ähnliche Konventionen sind sehr sinnvoll, damit
103 //       der Code leserlich bleibt!
104 ANGEST::ANGEST(char * _name, int _persnr)
105 {
106    strncpy(name,_name,STRINGLENGTH-1);
107    name[STRINGLENGTH-1]='\0';
108    persnr=_persnr;
109 #ifdef DEVELOP // nur solange DEVELOP definiert ist...
110    cout << "Konstruktor ANGEST meldet sich: name=" << name
111       << " persnr=" << persnr << endl;
112 #endif
113 }
```

```
114
115 ANGEST::~ANGEST()
116 {
117 #ifdef DEVELOP
118    cout << "Destruktor ANGEST meldet sich: name=" << name
119       << " persnr=" << persnr << endl;
120 #endif
121 }
122
123
124 // --- Implementation der Freundfunktionen von ANGEST ---------------
125 ostream& operator<<(ostream& out,ANGEST& angest)
126 {
127    out << angest.persnr << " (" << angest.name << ")" << endl;
128    return(out);
129 }
```

```
130 // --- Implementation der Klasse LTDANGEST -------------------------
131 LTDANGEST::LTDANGEST(char * name, int persnr, char * _polkennzeichen):
132    ANGEST(name,persnr)
133 {
134    strncpy(polkennzeichen,_polkennzeichen,12);
135    polkennzeichen[12-1]='\0';
136 #ifdef DEVELOP // nur solange DEVELOP definiert ist...
137    cout << "Konstruktor LTDANGEST meldet sich: polkennzeichen="
138       << polkennzeichen << endl;
139 #endif
140 }
141
142 LTDANGEST::~LTDANGEST()
143 {
144 #ifdef DEVELOP
145    cout << "Destruktor LTDANGEST meldet sich: polkennzeichen="
146       << polkennzeichen << endl;
147 #endif
148 }
149
150 // --- Implementation der Freundfunktionen von LTDANGEST ------------
151 ostream& operator<<(ostream& out,LTDANGEST& ltdangest)
152 {
153    out << ltdangest.persnr << " (" << ltdangest.name << ")  "
154       << ltdangest.polkennzeichen << endl;
155    return(out);
156 }
157
```

```
158 // --- Implementation der Klasse VERTRANGEST ------------------------
159 VERTRANGEST::VERTRANGEST(char * name, int persnr, char * _laptop) :
160    ANGEST(name,persnr)
161 {
162    strncpy(laptop,_laptop,ANGEST::STRINGLENGTH-1);
163    laptop[ANGEST::STRINGLENGTH-1]='\0';
164 #ifdef DEVELOP
165    cout << "Konstruktor VERTRANGEST meldet sich: laptop="
166       << laptop << endl;
167 #endif
168 }
169
170 VERTRANGEST::~VERTRANGEST()
171 {
172 #ifdef DEVELOP
173    cout << "Destruktor VERTRANGEST meldet sich: laptop="
174       << laptop << endl;
175 #endif
176 }
```

```
177 // --- Implementation der Freundfunktionen von VERTRANGEST ----------
178 ostream& operator<<(ostream& out,VERTRANGEST& vertrangest)
179 {
180    out << vertrangest.persnr << " (" << vertrangest.name << ")  "
181       << vertrangest.laptop << endl;
182    return(out);
183 }
184
185
186 // --- Implementation der Klasse LTDVERTRANGEST ---------------------
187 LTDVERTRANGEST::LTDVERTRANGEST(char * name, int persnr,
188                   char * polkennzeichen, char * laptop) :
189    LTDANGEST(name,persnr,polkennzeichen),
190    VERTRANGEST(name,persnr,laptop),
191    ANGEST(name,persnr)
192 {
193 #ifdef DEVELOP
194    cout << "Konstruktor LTDVERTRANGEST meldet sich..." << endl;
195 #endif
196 }
197
198 LTDVERTRANGEST::~LTDVERTRANGEST()
199 {
200 #ifdef DEVELOP
201    cout << "Destruktor LTDVERTRANGEST meldet sich..." << endl;
202 #endif
203 }
204
```

```
205 // --- Implementation der Freundfunktionen von LTDVERTRANGEST -------
206 ostream& operator<<(ostream& out,LTDVERTRANGEST& ltdvertrangest)
207 {
208    out << ltdvertrangest.persnr << " (" << ltdvertrangest.name <<") "
209       << ltdvertrangest.polkennzeichen << "  Laptop: "
210       << ltdvertrangest.laptop << endl;
211    return(out);
212 }
213
214 // === Hauptprogramm ================================================
215 int main(void)
216 {
217    cout << "*** vererbung.cpp: Demonstrationsprogramm zur Vererbung"
218       << endl << endl;
219
220    cout << "*** Der Angestellte Meier betritt die Buehne:" << endl;
221    ANGEST einAngestellter("Meier",12001);
222    cout << einAngestellter << endl;
223
224    cout << "*** Der Angestellte Wieselmann betritt die Buehne:"<<endl;
225    ANGEST einWeitererAngestellter("Wieselmann",12002);
226    cout << einWeitererAngestellter << endl;
227    cout << "*** Der leitende Angestellte Brutzelbach-Weissenberger "
228       << "erscheint:" << endl;
229    LTDANGEST einLeitenderAngestellter("Brutzelbach-Weissenberger",
230       25001,"GL-AA 111");
231    cout << einLeitenderAngestellter << endl;
232
```

```
233     cout << "*** Die Vertriebsangestellte Schmittke kommt: " << endl;
234     VERTRANGEST vertrangest("Schmittke",14711,"ComPack SuperLite P75");
235     cout << vertrangest << endl;
236
237     cout << "*** Der leitende Vertriebsangestellte Brummer kommt: "
238       << endl;
239     LTDVERTRANGEST ltdvertrangest("Brummer",30911,
240                         "GL-ZZ 999","HP OmniBook");
241     cout << ltdvertrangest << endl;
242
243     cout << "*** Ende des Programms" << endl;
244
245     return EXIT_SUCCESS;
246 } // end main
247
248 // end of file vererbung.cpp
```

## Übung 15: Weitere Operatoren in der Klasse BRUCH

Nehmen Sie in die Klasse `BRUCH` (vgl. Übung 6 auf Seite 0) weitere Operatoren auf, z.B. `+=`, `-=`, `*=` und `/=` zur kompakten Berechnung-und-Zuweisung in einem, so wie Sie sie von `int` her schon kennen.

So soll folgender Code-Ausschnitt möglich werden und den Bruch `b` von 3/5 auf 7/5 erhöhen.

```
BRUCH b(3,5), b2(4,5);
b += b2;
```

## Übung 16: Eine String-Klasse

Sie kennen die Probleme, die C (und damit auch C++) mit den Zeichenketten der Form `char*` bzw. `char[]` einem Programmierer bereitet.

Überlegen Sie sich eine Klasse `String`, die mit den Mitteln von C++ alles besser macht. Implementieren Sie Ihren Entwurf und verbessern Sie ihn sukzessive.

Hierzu einige Anregungen:

a) Nehmen Sie zur schnelleren Abfrage auf die Stringlänge ein Datenelement `length` auf, das natürlich dann auch je nach Notwendigkeit stets aktualisiert wird.

b) Gestalten Sie in der Klasse `String` eine (ziemlich) sichere Eingabe einer Zeichenkette. Verwenden Sie dabei möglichst die Streams aus `iostream.h` und nicht die C-Funktionen zur Zeichenketteneingabe.

c) Überlegen Sie sich bitte, wie ein `operator==` aussehen könnte, mit dem der einfache (inhaltliche) Vergleich zweier Strings möglich ist!

Darüber hinaus können Sie natürlich noch weiter nachdenken, welche Funktionalität Sie in eine solche `String`-Klasse legen möchten...

## Übung 17: Noch einmal eine String-Klasse

In Weiterentwicklung von Übung 16 soll es in dieser Aufgabe um das Erarbeiten einer Klasse `String` gehen, die die Anforderungen des nachstehend abgedruckten Hauptprogramms stringmain.cpp erfüllt. Dabei soll dann das auf Seite 0 abgedruckte Ablauflisting erzeugt werden.

Wenn Sie die Möglichkeit hierzu haben, dann sollten Sie für diese Aufgabe am besten eine Arbeitsgruppe bilden. Gerade die Designphase einer (C++-)Klasse lebt sehr stark vom Austausch der Gedanken. Vielleicht können Sie außerdem auch einiges an Tipparbeit sparen, wenn nicht jeder alles zu schreiben und in den Rechner einzugeben hat...

Weitere Klassenmethoden können Sie natürlich gerne nach Lust und Laune hinzunehmen; Minimalanforderung ist jedoch das Funktionieren des hier gezeigten Hauptprogramms.

Hinweis: Da es sicherlich nicht ganz trivial ist, sofort sämtliche Anforderungen zu erfüllen, kommentieren Sie zunächst viele Teile des Hauptprogramms aus und implementieren Sie Schritt fuer Schritt: erst die Konstruktoren und den Destruktor, dann weitere Methoden... Und testen Sie...

```
// -----------------------------------------------------------------------
// stringmain.cpp
// Hauptprogramm fuer eine zu implementierende String-Klasse String.
//
// Die hier zu implementierende String-Klasse soll alle Anforderungen
// erfuellen, die im untenstehenden Hauptprogramm implizit formuliert
// sind.
// Insbesondere soll natuerlich das Arbeiten mit dieser Klasse String
// "sicher" sein, es also insbesondere keine Programmabstuerze wegen
// fehlerhafter Speicherzugriffe, z.B. bei String-Index-Overflow,
// (mehr) geben (wie einst in C)!
// -----------------------------------------------------------------------

// Natuerlich duerfen auch mehr oder weniger als diese Headerfiles
// eingebunden werden. Insbesondere ist das Verwenden der String-
// Routinen aus C (innerhalb der Klasse String) erlaubt.
#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

// .... hier muss die Klassendeklaration und -implementation fuer
//      die Klasse String hin!
//      (Wer Lust hat, kann dies auch mit Auslagerung in weitere
//       Dateien und Erstellen eines eigenen Makefiles tun!)

void main()  // Testhauptprogramm
{
```

```
cout << "Entwicklung einer String-Klasse: Test-Hauptprogramm." << endl;

// Hiermit werden fuenf Strings erzeugt. s2 soll bereits mit der
// Zeichenkette "bib" initialisiert werden, s4 soll zu Beginn zehn
// Leerzeichen enthalten, und s5 bestehe aus fuenf Sternchen
// (gefolgt natuerlich von '\0' als String-Ende-Zeichen wie bei C).
String s1, s2("bib"), s3(s2), s4(10), s5('*',5);

// Die folgenden Testausgaben sollen moeglich sein.
cout << "s1: [" << s1 << "]" << endl;
cout << "s2: [" << s2 << "]" << endl;
cout << "s3: [" << s3 << "]" << endl;
cout << "s4: [" << s4 << "]" << endl;
cout << "s5: [" << s5 << "]" << endl;

// Auch folgendes soll durchfuehrbar sein: Eingabe zweier Zeichen-
// ketten und Konkatenieren (=Aneinanderhaengen) der beiden.
cout << "Bitte eine Zeichenkette eingeben: ";
String s6;
cin >> s6;                    // Interaktive Eingabe einer Zeichenkette
cout << "Bitte eine weitere Zeichenkette: ";
String s7;
cin >> s7;
cout << "Danke." << endl;
cout << "s6: [" << s6 << "]" << endl;
cout << "s7: [" << s7 << "]" << endl;
cout << "Konkateniert: [" << s6+s7 << "]" << endl;
```

```
// Auch Zuweisungen mit konstanten Zeichenketten sollen funktionieren.
String s8=s7+"!!!";                // In dieser Reihenfolge -
String s9="Hallo - " + s7;         // und in dieser!
cout << "s8: [" << s8 << "]" << endl;
cout << "s9: [" << s9 << "]" << endl;

// Und die Stringlaenge soll abfragbar sein.
cout << "s8 ist " << s8.StrLen() << " Byte(s) lang." << endl;
cout << "s9 ist " << s9.StrLen() << " Byte(s) lang." << endl;

// Vergleichen zweier Zeichenketten soll moeglich sein.
String s10("bib"), s11('*',12);
cout << "s10: [" << s10 << "]" << endl;
cout << "s11: [" << s11 << "]" << endl;
if (s10 == s11)
   cout << "s10 und s11 sind gleich." << endl;
else
   cout << "s10 und s11 sind nicht gleich." << endl;



// Auch eine solche Zuweisung soll moeglich und korrekt sein.
s11="bib";
cout << "s11: [" << s11 << "]" << endl;

// Vergleiche auf Ungleichheit sollen ebenfalls moeglich sein.
if (s10 != s11)
   cout << "s10 und s11 sind nun nicht gleich." << endl;
else
```

```
   cout << "s10 und s11 sind nun gleich." << endl;


// Auch eine Ordnung soll implementiert werden: die Operatoren
// <, >, <= und >= werden benoetigt.
cout << "Bitte eine Zeichenkette eingeben: ";
cin >> s10;
cout << "Bitte nochmal eine Zeichenkette eingeben: ";
cin >> s11;
cout << "s10: [" << s10 << "]" << endl;
cout << "s11: [" << s11 << "]" << endl;
if (s10 < s11)
   cout << "s10 [" << s10 << "] < s11 [" << s11 << "]" << endl;
if (s10 > s11)
   cout << "s10 [" << s10 << "] > s11 [" << s11 << "]" << endl;
if (s10 <= s11)
   cout << "s10 [" << s10 << "] <= s11 [" << s11 << "]" << endl;
if (s10 >= s11)
   cout << "s10 [" << s10 << "] >= s11 [" << s11 << "]" << endl;


// Und auch die Zuweisung einer (ganzen) Zahl soll erlaubt sein.
// Aus der binaer abgespeicherten Zahl 1101(dual) == 13(dezimal)
// soll dann die Zeichenkette '1' '3' '\0' werden.
String s12;
s12=4711;
cout << "s12: [" << s12 << "]" << endl;
cout << "Bitte eine ganze Zahl eingeben: ";
int value;
cin >> value;
```

```
   s12=value;

   cout << "s12: [" << s12 << "]" << endl;

   cout << "Ende des Programms." << endl;


} // end main
```

Und das Ablauflisting dazu:

```
Entwicklung einer String-Klasse: Test-Hauptprogramm.

s1: []

s2: [bib]

s3: [bib]

s4: [          ]

s5: [*****]

Bitte eine Zeichenkette eingeben: Eine Zeichenkette

Bitte eine weitere Zeichenkette: Noch eine Zeichenkette

Danke.

s6: [Eine Zeichenkette]

s7: [Noch eine Zeichenkette]

Konkateniert: [Eine ZeichenketteNoch eine Zeichenkette]

s8: [Noch eine Zeichenkette!!!]

s9: [Hallo - Noch eine Zeichenkette]

s8 ist 26 Byte(s) lang.

s9 ist 31 Byte(s) lang.

s10: [bib]

s11: [************]

s10 und s11 sind nicht gleich.

s11: [bib]

s10 und s11 sind nun gleich.

Bitte eine Zeichenkette eingeben: Guten Tag
```

```
Bitte nochmal eine Zeichenkette eingeben: Guten Tag

s10: [Guten Tag]

s11: [Guten Tag]

s10 [Guten Tag] <= s11 [Guten Tag]

s10 [Guten Tag] >= s11 [Guten Tag]

s12: [4711]

Bitte eine ganze Zahl eingeben: 1234

s12: [1234]

Ende des Programms.
```

## Übung 18: Funktionstemplates

Implementieren Sie bitte die folgenden Funktionstemplates.

a) Das Template `Max(a,b)` liefert das Maximum zweier Parameter `a` und `b` desselben Typs `T (<class T>)` zurück.

b) Das Template `Max(a,b,c)` liefert analog das Maximum von drei Parametern desselben Typs `T` zurück.

c) Ein weiteres Template namens `Max()` soll zu einem übergebenen Array (bzw. einer übergebenen Liste) von `T`-Objekten das maximale Objekt zurückliefern.

Wenn Sie diese Templates implementiert haben, dann sollen die folgenden Aufrufe bei den genannten Deklarationen möglich sein.

```
int a=1, b=2, c=3;

char c1='a', c2='b', c3='z';

BRUCH b1(1,2), b2(3), b3;  // hier wird unsere Klasse BRUCH vorausgesetzt

int A[5]={ 1,23,2,18,4 };

String s1("Haus"), s2("Helgoland"); // vgl. Übung 17


cout << Max(a,b) << endl;        // Ausgabe des Wertes 2

cout << Max(a,b,c) << endl;      // Ausgabe des Wertes 3

cout << Max(c1,c2,c3) << endl;   // Ausgabe von 'z'
```

```
cout << Max(A) << endl;          // Ausgabe der 23

cout << Max(b1,b2,b3) << endl;  // Ausgabe von 3 in der Klasse BRUCH

cout << Max(b1,b3) << endl;     // Ausgabe von 1/2 in der Klasse BRUCH

cout << Max(s1,s2) << endl;     // Ausgabe von "Helgoland" i.d.Kl. String
```

## Übung 19: Ein Klassentemplate

Schreiben Sie ein Klassentemplate für lineare Listen:

```
template <class T> class LISTE

{

    // ... hier auffüllen

};
```

Damit soll es möglich sein, dynamisch verwaltete lineare Listen von den verschiedensten Datentypen zu realisieren. So sollen beispielsweise `LISTE<int>` und `LISTE<char>` lineare Listen von `int`- bzw. `char`-Werten sein. Aber auch `LISTE<String>` oder `LISTE<BRUCH>` sind natürlich möglich, sofern die Klassen `String` und `BRUCH` implementiert worden sind.

Entwerfen und programmieren Sie zumindest die Grundroutinen einer Linearen Liste:

a) Im Default-Konstruktor das Anlegen einer leeren Liste.

b) Eine Methode `Add()`, die ein Element an das Ende der Liste anhängt. Die Listen sollen also nicht sortiert verwaltet werden.

c) Eine Methode `Show()`, die die gesamte Liste ausgibt. Wenn Sie möchten, koennen Sie dies auch als `operator<<()` umsetzen.

d) Die Methode `Delete()` löscht ein Element aus der Liste.

e) Die Methode `Count()` liefert die aktuell eingetragene Anzahl Elemente zurück.

Anmerkung: Die dynamische Speicherplatzallokation soll mit den C++-typischen Mitteln `new` und `delete` erfolgen. Notfalls sehen Sie hierzu bitte im Abschnitt 4.2 (Seite 0) nach.

Diese Seite wurde absichtlich leer gelassen.

# A. Anhang I: Weitere Beispielprogramme

Im folgenden sollen zum vertiefenden Verständnis noch einige etwas umfangreichere C++-Beispielprogramme vorgestellt werden.

## A.1. Beispielklasse KTime: Zeit und Datum

Das nachfolgende Beispielprogramm `timec.cpp` illustriert die mögliche Verwendung des Klassenkonzeptes. Die Klasse `KTime` verwaltet eine hier als `private` deklarierte `time_t`-Komponente, die die Anzahl der Sekunden seit dem 1.1.1970[66] angibt. Dazu werden in den Bereichen `protected` und `public` Zugriffsmethoden bereitgestellt. Hierbei werden eine ganze Reihe von Operatoren überladen; insbesondere wird gezeigt, wie die Präfix- und die Postfix-Variante des unären[67] Inkrementoperators `++` (bzw. `--`) überladen werden können.

```
// ---------------------------------------------------------------------

// ktime.cpp - Eine Zeit- und Datumsklasse

// ---------------------------------------------------------------------



// #define DIAGNOSE 1          // Für Trace-Modus aktivieren.



#include <iostream.h>

#include <iomanip.h>

#include <stdlib.h>

#include <time.h>



// asTage und asTageShort können natürlich auch in die Klasse Ktime

// aufgenommen werden.

char asTage[][11]     = { "Sonntag", "Montag", "Dienstag", "Mittwoch",

                          "Donnerstag", "Freitag", "Samstag" };

char asTageShort[][3] = { "So", "Mo", "Di", "Mi", "Do", "Fr", "Sa" };
```

---

[66] Die Zeit in Sekunden seit dem 1.1.1970 zu verwalten, ist gängige UNIX-Tradition.

[67] *unär* bedeutet einstellig, das heißt: dieser Operator erwartet nur ein Argument. Der bekannteste unäre Operator ist wohl der Vorzeichenoperator `-` .

```
// Klasse KTime: Deklaration

class KTime

{

   private:

      time_t tim;              // Anzahl der Sekunden seit 1.1.70



      // Interne Hilfsroutinen

      void getdata(void);            // Berechnen der Komponenten-Daten
                     // aus dem Element tim
```

```
    char * stringcopy(char *, const char *);
                  // Eigene String-Kopierroutine,
                  // damit string.h nicht eingebunden
                  // werden muß

    void showdigit(int);          // Ausgabe eines int-Wertes auf 2
                  // Stellen


#ifdef DIAGNOSE

    int objnr;                    // zu Diagnosezwecken: Instanznummer

#endif



  protected:            // Die Komponenten, die der Spezifika-
                  // tion von struct tm in time.h
                  // entsprechen

    int  day, month, year,
     doy /* day of year */,

     dow /* day of week */,
     hour, min, sec,
     isdst /* is daylight-saving-time (Sommerzeit) */ ;

    char times[6];            // Zeit-String der Form "12:34" (Uhr)

    char dates[9];            // Datum-String der Form "24.12.95"

  public:

#ifdef DIAGNOSE

    static int classcount;        // zur Diagnose: Instanzenzähler

#endif



                  // Freundoperatoren zur Ein- und
                  // Ausgabe von KTime-Objekten

    friend ostream & operator<< (ostream &, KTime );

    friend istream & operator>> (istream &, KTime & );



                  // Konstruktoren ("Ctors") und
```

```
Ktime();                          // Destruktoren ("Dtors")

KTime(int,int,int,int=0,int=0,int=0);

                  // initialisiert Tag/Monat/Jahr und
                  // optional Stunde/Minute/Sekunde

Ktime(KTime&);                    // Kopierkonstruktor ("Copy Ctor")

~KTime();            // Destruktor ("Dtor")


void operator=(KTime);            // Zuweisungsoperator

KTime operator+(int);             // Hinzu-Addieren von Tagen

KTime operator-(int);             // Subtrahieren von Tagen

int operator-(KTime);             // Differenz zweier Daten

void operator+=(int);             // Hinzu-Addieren von Tagen zu *this

void operator-=(int);             // Subtrahieren von Tagen von *this

KTime operator++();               // Hinzuaddieren von 1 Tag (Präfix)

KTime operator--();               // Subtrahieren von 1 Tag (Präfix)

KTime operator++(int);            // Addieren von 1 Tag (Postfix)

KTime operator--(int);            // Subtrahieren von 1 Tag (Postfix)


                  // Vergleichsoperatoren

int operator==(KTime);

int operator<(KTime);

int operator<=(KTime);

int operator>(KTime);

int operator>=(KTime);

int operator!=(KTime);

                  // Stunden/Tage/Minuten/Sek. addieren

void Addiere(time_t,time_t=0,time_t=0,time_t=0);

                  // Komponentenwerte zurückliefern
```

```
    int  GetDay();     // Tag im Monat (1-31)
    int  GetMonth();         // Monat im Jahr (1-12)
    int  GetYear();          // Jahr (vierstellig)
    int  GetHour();          // Stunde am Tag (0-23)
    int  GetMinute();        // Minute (0-59)
    int  GetSecond();        // Sekunde (0-59)
    int  GetDayOfYear();           // Tag im Jahr (1-366)
    int  GetDayOfWeek();           // Tag der Woche (0-6, 0=Sonntag)
    char * GetDate(char *);        // Datum(stext) zurückliefern
    char * GetTime(char *);        // Zeit(text) zurückliefern
    char * GetWeekday(char *);     // Wochentag zurückliefern ("Montag")
    char * GetWeekdayShort(char *);// Wochentagskürzel holen ("Mo")
    time_t GetTime();        // time_t-Darstellung tim
                             // zurückliefern

}; // end class KTime

// Diagnosehilfe
#ifdef DIAGNOSE
int KTime::classcount=0;  // Allokierung des statischen Klassenelementes
#endif

// Klasse KTime: Implementation

KTime::KTime()           // Default-Konstruktor
{
   tim=time(NULL);
   getdata();
```

```
#ifdef DIAGNOSE

   objnr=classcount++;

   cout << "Konstruktor KTime() für Objekt " << objnr << " aufgerufen."
  << endl;

#endif

} // end KTime::KTime()
```

```
KTime::KTime(int dd, int mm, int yy, int hs, int mn, int sc)
{
   // Hinweis: verschiedene für die Praxis sinnvolle Überprüfungen wurden
   // hier aus Platzgründen nicht implementiert!
   struct tm strtm;
   strtm.tm_mday=dd;
   strtm.tm_mon=mm-1;
   strtm.tm_year=yy-1900;           // struct tm verwaltet nur die Jahre
   strtm.tm_hour=hs;                // seit 1900
   strtm.tm_min=mn;
   strtm.tm_sec=sc;
   strtm.tm_isdst=0;  // Achtung: Sommerzeitregelung nicht implementiert!
   strtm.tm_wday=0;
   strtm.tm_yday=0;
   tim=mktime(&strtm);
   getdata();
#ifdef DIAGNOSE
   objnr=classcount++;
   cout << "Konstruktor KTime(int6) für Objekt "
      << objnr << " aufgerufen." << endl;
#endif
} // end KTime::KTime(int dd, int mm, int yy, int hs, int mn, int sc)

KTime::KTime(KTime & ktime)    // Kopierkonstruktor
{
   tim=ktime.tim;
   getdata();
#ifdef DIAGNOSE
```

```
   objnr=classcount++;
   cout << "Konstruktor KTime(KTime&) für Objekt "
      << objnr << " aufgerufen." << endl;
#endif
} // end KTime::KTime(KTime & ktime)

KTime::~KTime()                // (i.w. leerer) Destruktor
{
#ifdef DIAGNOSE
   cout << "Destruktor für Objekt " << objnr << " aufgerufen." << endl;
   classcount--;
#endif
} // end KTime::~KTime()

void KTime::operator=(KTime ktime)
{
   tim=ktime.tim;
   getdata();
} // end void KTime::operator=(KTime ktime)
```

```
KTime KTime::operator+(int tage)
{
   KTime tmp;
   tmp.tim=tim;
   if (tage > 0)              // Keine sonstige Überprüfung,
   {                    // keine Fehlermeldung!
      tmp.Addiere(tage,0,0);
   }
   return(tmp);
} // end KTime KTime::operator+(int tage)


KTime KTime::operator-(int tage)
{
   KTime tmp;
   tmp.tim=tim;
   if (tage > 0)
   {
      tmp.tim -= time_t(24L*3600L*tage);
      tmp.getdata();
   }
   return(tmp);
} // end KTime KTime::operator-(int tage)


int KTime::operator-(KTime ktime)
{
   return(difftime(tim,ktime.tim) / (24L*3600L));
} // end int KTime::operator-(KTime)
```

```
void KTime::operator+=(int tage)
{
   if (tage > 0)              // Wieder ohne Fehlerbehandlung...
   {
      Addiere(tage,0,0);
   }
} // end void KTime::operator+=(int tage)

void KTime::operator-=(int tage)
{
   if (tage>0)
   {
      tim -= time_t(24L*3600L*tage);
      getdata();
   }
} // end void KTime::operator-=(int tage)

KTime KTime::operator++()
{                      // Präfix-Variante
   Addiere(1,0,0);      // oder: *this += 1;
   return *this;
} // end void KTime::operator++()
KTime KTime::operator--()
{                      // Präfix-Variante
   *this -= 1;
   return *this;
} // end void KTime::operator--()
```

```
KTime KTime::operator++(int)
{                       // Postfix-Variante
   KTime tmp = *this;          // Der ursprüngliche Wert von *this
   Addiere(1,0,0);       // wird zurückgeliefert; *this aber
   return tmp;           // dennoch inkrementiert!
} // end void KTime::operator++(int)

KTime KTime::operator--(int)
{                       // Postfix-Variante
   KTime tmp = *this;
   *this -= 1;
   return tmp;
} // end void KTime::operator--(int)

int KTime::operator==(KTime ktime)
{
   return(tim==ktime.tim);
} // end int operator==(KTime)

int KTime::operator<(KTime ktime)
{
   return(tim<ktime.tim);
} // end int operator<(KTime)

int KTime::operator<=(KTime ktime)
{
   return(tim<=ktime.tim);
} // end int operator<=(KTime)
```

```
int KTime::operator>(KTime ktime)
{
   return(tim>ktime.tim);
} // end int operator>(KTime)

int KTime::operator>=(KTime ktime)
{
   return(tim>=ktime.tim);
} // end int operator>=(KTime)

int KTime::operator!=(KTime ktime)
{
   return(tim!=ktime.tim);
} // end int operator!=(KTime)

void KTime::getdata()
{
   struct tm *ptm=localtime(&tim);
   hour=ptm->tm_hour;
   min=ptm->tm_min;
   sec=ptm->tm_sec;
   day=ptm->tm_mday;
   month=ptm->tm_mon+1;        // Bereich 1..12 herstellen!
   year=ptm->tm_year+1900;     // vierstellige Jahreszahlen!
   dow=ptm->tm_wday;
   doy=ptm->tm_yday;
   isdst=ptm->tm_isdst;
```

```
                    // Time-String herstellen
   times[0]='0'+hour/10;      // (ohne Verwendung von sprintf()!)
   times[1]='0'+hour%10;
   times[2]=':';
   times[3]='0'+min/10;
   times[4]='0'+min%10;
   times[5]='\0';
                    // Date-String herstellen
   dates[0]='0'+day/10;             // (ohne Verwendung von sprintf())
   dates[1]='0'+day%10;
   dates[2]='.';
   dates[3]='0'+month/10;
   dates[4]='0'+month%10;
   dates[5]='.';
   dates[6]='0'+year%100/10;
   dates[7]='0'+year%10;
   dates[8]='\0';
} // end void KTime::getdata()


char * KTime::stringcopy(char *s, const char *others)
{
   while(*others)
   {
      *s++ = *others++;
   }
   *s='\0';
   return s;
} // end char * KTime::stringcopy(char *s, char *others)
```

```
void KTime::showdigit(int d)
{
   if (0<=d && d<10)
      cout << '0';       // Einstellige positive Zahlen werden
   cout << d;            // mit führender Null geschrieben
} // end void KTime::showdigit(int d)
```

```
void KTime::Addiere(time_t tage, time_t stunden,
             time_t minuten, time_t sekunden)

{

   tim += time_t(sekunden+60L*(minuten+60L*(stunden+24L*tage)));

   getdata();

} // end void KTime::Addiere(time_t, time_t, time_t, time_t)




int KTime::GetDay()

{

   return(day);

} // end int KTime::GetDay()


int KTime::GetMonth()

{

   return(month);

} // end int KTime::GetMonth()


int KTime::GetYear()

{

   return(year);

} // end int KTime::GetYear()


int KTime::GetHour()

{

   return(hour);

} // end int KTime::GetHour()
```

```
int KTime::GetMinute()
{
   return(min);
} // end int KTime::GetMinute()

int KTime::GetSecond()
{
   return(sec);
} // end int KTime::GetSecond()

char * KTime::GetDate(char * s)
{
   return(stringcopy(s,dates));
} // end char * KTime::GetDate(char * s)

char * KTime::GetTime(char * s)
{
   return(stringcopy(s,times));
} // end char * KTime::GetTime(char * s)
```

```
int KTime::GetDayOfYear()
{
   return(doy);
} // end int KTime::GetDayOfYear()

int KTime::GetDayOfWeek()
{
   return(dow);
} // end int KTime::GetDayOfWeek()

char * KTime::GetWeekday(char * s)
{
   return(stringcopy(s,asTage[dow]));
} // end char * KTime::GetWeekday(char * s)

char * KTime::GetWeekdayShort(char * s)
{
   return(stringcopy(s,asTageShort[dow]));
} // end char * KTime::GetWeekdayShort(char * s)

time_t KTime::GetTime()
{
   return(tim);
} // end time_t KTime::GetTime()

// Freunde der Klasse KTime
ostream & operator<< (ostream & out, KTime ktime)
{                       // Formatierte Ausgabe eines Klassen-
```

```
    ktime.showdigit(ktime.day);      // objektes
    out << ".";
    ktime.showdigit(ktime.month);
    out << ".";
    ktime.showdigit(ktime.year);
    out << ", ";
    ktime.showdigit(ktime.hour);
    out << ":";
    ktime.showdigit(ktime.min);
    out << " ";
    return out;
} // end ostream & operator<< (ostream &, KTime );

istream & operator>> (istream & in, KTime & ktime)
{
   int day,month,year;         // Eingabe - nur minimal geprüft!
   do                    // So ist etwa ein 31.Februar
   {                     // hier noch möglich!
      cout << "Tag[1-31]:    ";
      in >> day;
   }
   while (day<1 || day>31);
   do
   {
      cout << "Monat[1-12]:  ";
      in >> month;
   }
   while (month<1 || month>12);
```

```
   cout << "Jahr:         ";        // Werte unter
   in >> year;



   if (year < 70)             // Jahresangaben zwischen 0 und 69
      year+=2000;             // werden als 2000 bis 2069 verstan-
   else if (year < 100)             // den, zwischen 70 und 99 als
      year+=1900;             // 1970 bis 1999.

   KTime tmp(day,month,year);
   ktime=tmp;
   return in;
} // end istream & operator>> (istream & in, KTime ktime)



// Ein kleines Test-Hauptprogramm
int main()
{
   cout << "KTime - Zeit- und Datumsklasse. Testprogramm" << endl;
   cout << "--------------------------------------------" << endl;

#ifdef DIAGNOSE
   cout << "*** Es existieren zur Zeit " << KTime::classcount
  << " Objekt(e)." << endl;
#endif

   KTime aTimeObject;
   cout << "Heute ist der " << aTimeObject << endl;
```

```
#ifdef DIAGNOSE
   cout << "*** Es existieren zur Zeit " << KTime::classcount
  << " Objekt(e)." << endl;
#endif


   char ws[11];
   aTimeObject.GetWeekday(ws);
   cout << "Wochentag: " << ws << endl;


#ifdef DIAGNOSE
   cout << "*** Es existieren zur Zeit " << KTime::classcount
  << " Objekt(e)." << endl;
#endif
   cout << "Und nun bitte ein Datum eingeben: " << endl;
   KTime newone;
   cin >> newone;
   cout << "newone: " << newone << endl;
   newone.GetWeekday(ws);
   cout << newone << " ist ein " << ws << "." << endl;


   KTime tmp;
   tmp=newone+28;
   cout << "4 Wochen später:    tmp: " << tmp << endl;
   int tagedazwischen=tmp-newone;
   cout << "Dazwischen liegen " << tagedazwischen
  << " Tage." << endl;


   tmp.GetWeekday(ws);
```

```
   cout << tmp << " ist ein " << ws << "." << endl;

   tmp-=21;

   cout << "3 Wochen zurück ist der: " << tmp << endl;

   tmp.GetWeekdayShort(ws);

   cout << tmp << " ist ein " << ws << "!" << endl;

   --tmp;

   cout << "Der Tag davor: " << tmp << " mit tim=" << tmp.GetTime()
  << endl;

   tmp--;

   cout << "Und der Tag davor: " << tmp << " mit tim="
  << tmp.GetTime() << endl;

   if (tmp==newone)

      cout << tmp << " und " << newone << " sind gleich. " << endl;

   else

      cout << tmp << " und " << newone << " sind nicht gleich. " << endl;

   if (tmp<newone)

      cout << tmp << " < " << newone << endl;

   else if (tmp>newone)

      cout << tmp << " > " << newone << endl;


#ifdef DIAGNOSE

   cout << "*** Es existieren zur Zeit " << KTime::classcount
  << " Objekt(e)." << endl;

#endif

   return EXIT_SUCCESS;

} // end main


// --------------------------------------------------------------------

// end of file ktime.cpp
```

```
// ----------------------------------------------------------------------
```

Ein Ablaufprotokoll zu diesem Programm:

```
KTime - Zeit- und Datumsklasse. Testprogramm

--------------------------------------------

Heute ist der 16.11.1996, 13:45

Wochentag: Samstag

Und nun bitte ein Datum eingeben:

Tag[1-31]:     24

Monat[1-12]:   12

Jahr:          95

newone: 24.12.1995, 00:00

24.12.1995, 00:00  ist ein Sonntag.

4 Wochen später:    tmp: 21.01.1996, 00:00

Dazwischen liegen 28 Tage.

21.01.1996, 00:00  ist ein Sonntag.

3 Wochen zurück ist der: 31.12.1995, 00:00

31.12.1995, 00:00  ist ein So!

Der Tag davor: 30.12.1995, 00:00  mit tim=820278000

Und der Tag davor: 29.12.1995, 00:00  mit tim=820191600

29.12.1995, 00:00  und 24.12.1995, 00:00  sind nicht gleich.

29.12.1995, 00:00  > 24.12.1995, 00:00
```

## A.2. Beispielklassen KElement und  KListe: Lineare Listen

Das nachstehende Beispielprogramm `liste1.cpp` implementiert eine einfache Form von Linearen Listen, in diesem Falle von generell unsortierten Listen von int-Werten; in eine solche Liste kann bei der hier vorliegenden Implementation ein Wert auch mehrfach eingetragen werden.

Die Klasse `KElement` beschreibt einen einzelnen Elementeintrag (Knoten) einer Linearen Liste, die dann in der Klasse `KListe` deklariert wird. Damit innerhalb der Klasse `KListe` auf die Elementdaten eines einzelnen Knotens zugegriffen werden kann, muß `KElement` die Klasse `KListe` zu ihrem Freund erklären.

Um das Beispielprogramm nicht zu sehr aufzublähen, wurde hier nur ein notwendiges Minimum für das kurz gehaltene Demonstrationshauptprogramm implementiert.

```
// ---------------------------------------------------------------------

// liste1.cpp - (Unsortierte) Lineare Liste von int-Werten

// ---------------------------------------------------------------------


#include <iostream.h>

#include <stdlib.h>


class KListe;             // Vorwärtsdeklaration, damit in
                        // KElement der operator<<

                        // zum Freund erklärt werden kann.


                        // Klasse KElement: Deklaration

class KElement

{

   protected:

      int  data;

      KElement * next;

   public:

      friend class KListe;
```

```
    friend ostream & operator<<(ostream&,KListe&);

    KElement(int wert=-1)      { data=wert; next=NULL; };

}; // end class KElement
```

```
                    // Klasse KListe: Deklaration
class KListe
{
   protected:
      KElement * head;         // Zeiger auf erstes Element der Liste
      KElement * tail;         // Zeiger auf letztes Element der Liste
      void purge_rest();             // Hilfsmethode zum Löschen der
                    // restlichen Liste
   public:
      friend ostream & operator<<(ostream&,KListe&); // Ausgabe-Operator
      KListe();                // Default-Konstruktor
      ~KListe();         // Destruktor
      KListe& operator=(KListe&); // Zuweisung einer ganzen Liste
      void operator+=(int);          // Anhängen eines Wertes
      void operator-=(int);          // Entfernen eines Wertes
                    // (sofern dieser vorhanden ist)
}; // end class KListe


                    // Klasse KListe: Implementation
KListe::KListe()
{
   head=tail=NULL;
} // end KListe::KListe()


KListe::~KListe()
{
   purge_rest();
} // end KListe::~KListe()
```

```
void KListe::purge_rest()
{
   KElement *tmp=head;
   while (tmp)
   {
      head=head->next;
      delete tmp;
      tmp=head;
   } // end while
   head=tail=NULL;
} // end void KListe::purge_rest()
```

```
KListe& KListe::operator=(KListe& old)
{                       // Zuweisung der gesamten Liste,
                        // d.h. insbesondere Bereitstellen
                        // der erforderlichen Speicherplätze
                        // auf dem Heap mit new
    purge_rest();        // gegebenenfalls werden die alten
                        // Listenelemente zuvor gelöscht
    KElement *current=old.head;
    while (current)      // d.h. while (current != NULL) ...
    {
        (*this)+=current->data;      // operator+=(current->data);
        current=current->next;
    } // end while
    return(*this);
} // end KListe& KListe::operator=(KListe&)


void KListe::operator+=(int neu)     // Anhängen eines Wertes an das Ende
{                       // der Liste
   KElement *pelem;
   pelem = new KElement;
   pelem->data=neu;
   pelem->next=NULL;
   if (tail==NULL)       // Liste ist leer
   {
      head=tail=pelem;
   }
   else if (head==tail)        // genau ein Eintrag existiert bereits
   {
```

```
      head->next=tail=pelem;
   }
   else                    // es existieren bereits mehrere
   {                       // Einträge
      tail->next=pelem;
      tail=tail->next;
   } // end if
} // end void KListe::operator+=(int neu)

void KListe::operator-=(int kill)    // Entfernen eines Wertes
{                       // (sooft er vorkommt)
   while (head!=NULL && head->data==kill)
   {                       // Entfernen vom Beginn der Liste
      KElement *tmp=head;
      if (head==tail)
        tail=head->next;
      head=head->next;
      delete tmp;
   } // end while

   if (head==NULL)         // Die Liste wäre (nun) leer
      return;
   KElement *prev, *curr;      // Die Liste hat noch Elemente...
   prev=head;                        // head->data ist jedoch ungleich kill
   curr=prev->next;
   while (curr)
   {
      if (curr->data==kill)
```

```
      {
         prev->next=curr->next;
         if (curr==tail)
            tail=prev;
         delete curr;
         curr=prev->next;
      }
      else
      {
         prev=curr;
         curr=curr->next;
      }
   } // end while

} // end void KListe::operator-=(int kill)

                       // Freundoperator der Klasse KListe
ostream & operator<<(ostream& out,KListe& liste)
{
   if (liste.head==NULL)
   {
      out << "Die Lineare Liste ist leer.";
      return(out);
   }
                       // Die Liste ist nicht leer
   KElement * current;
   current=liste.head;
   while (current!=NULL)
```

```
    {
       out << " " << current->data;
       current=current->next;
    } // end while
    return(out);
} // end ostream & operator<<(ostream& out,KListe liste)
```

```
// Hauptprogramm
int main()
{
   cout << "LISTE1 - Einfache Lineare Listen. Testprogramm" << endl;
   cout << "----------------------------------------------" << endl;

   KListe liste1;
   cout << "Zu Beginn Liste1: " << liste1 << endl;
   for (int i=1; i<10; i++)
   {
      liste1+=i;
   } // end for i
   cout << "Zum Abschluß Liste1: " << liste1 << endl;

   KListe liste2;
   cout << "Zu Beginn Liste2: " << liste2 << endl;
   liste2=liste1;
   liste2+=10;
   liste2+=11;
   liste2+=12;
   liste2+=13;
   liste2+=14;
   liste2+=15;
   cout << "Nach Zuweisung liste2=liste1 und Aufnahme von 10 bis 15 "
     << endl << "in Liste2: " << liste2 << endl;
   for (i=0; i<16; i+=2)  // geradzahlige Elemente entfernen
     liste2-=i;
   cout << "Nach Entfernen der geradzahligen Elemente Liste2: "
```

```
    << liste2 << endl;
  for (i=1; i<16; i+=2)  // ungeradzahlige Elemente entfernen
    liste2-=i;
  cout << "Zum Abschluß Liste2: " << liste2 << endl;

  return EXIT_SUCCESS;
} // end main
// -----------------------------------------------------------------------
// end of file liste1.cpp
// -----------------------------------------------------------------------
```

Und hier wieder das Ablauflisting des Programms:

```
LISTE1 - Einfache Lineare Listen. Testprogramm
----------------------------------------------
Zu Beginn Liste1: Die Lineare Liste ist leer.
Zum Abschluß Liste1:  1 2 3 4 5 6 7 8 9
Zu Beginn Liste2: Die Lineare Liste ist leer.
Nach Zuweisung liste2=liste1 und Aufnahme von 10 bis 15
in Liste2:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nach Entfernen der geradzahligen Elemente Liste2:  1 3 5 7 9 11 13 15
Zum Abschluß Liste2: Die Lineare Liste ist leer.
```

## A.3. Templateklassen KElement<T> und KListe<T>

Anknüpfend an das Beispielprogramm `liste1.cpp` des vorigen Abschnitts wird in dem nachfolgenden Programm `liste2.cpp` zur abstrakten Realisierung linearer Listen mit zwei Template-Klassen(familien) `KElement<T>` und `KListe<T>` gearbeitet. Dabei beschreibt `T` einen (nahezu) beliebigen[68] Datentyp, von dem dann Lineare Listen gebildet und verwaltet werden können.

Um auch hier das Beispielprogramm nicht zu sehr auszudehnen, wurden hier im Hauptprogramm nur zwei Realisierungen der Template-Klassen (mit `T`=int und `T`=float) codiert. Der Quelltext wurde spartanisch kommentiert, ggf. hilft ein Blick in das Programm `liste1.cpp` im vorherigen Abschnitt.

```
// ---------------------------------------------------------------------
// liste2.cpp - (Unsortierte) Lineare Liste-Template-Klasse(n)
// ---------------------------------------------------------------------

#include <iostream.h>
#include <stdlib.h>

template<class T> class KListe;      // Vorwärtsdeklaration, damit in
                       // KElement der operator<<
                       // zum Freund erklärt werden kann.

                       // Klasse KElement: Deklaration
template<class T> class KElement
{
   protected:
      T data;
      KElement * next;
```

[68] Es ist lediglich darauf zu achten, daß die eingesetzten Operationen auf Werten des Typs `T` Sinn ergeben. Wird etwa mit `T a,b;` eine Zuweisung `a=b;` formuliert, so ist bei Pointertypen (z.B. `char*`) darauf zu achten, daß hier kein neuer Speicherplatz allokiert sondern lediglich die Adresse, die in `b` steht, in den Pointer `a` hineinkopiert wird.

```
   public:
      friend class KListe<T>;
      friend ostream & operator<<(ostream&,KListe<T> &);
      KElement()           { data=0; next=NULL; };
      KElement(T wert)     { data=wert; next=NULL; };
}; // end template<class T> class KElement
```

```
                     // Klasse KListe: Deklaration
template<class T> class KListe
{
   protected:
      KElement<T> * head;
      KElement<T> * tail;
      void purge_rest();
   public:
      friend ostream& operator<<(ostream&, KListe<T> &);
      KListe();
      ~KListe();
      KListe& operator=(KListe&);
      void operator+=(T);
      void operator-=(T);
}; // end template<class T> class KListe

                     // Klasse KListe: Implementation
template<class T> KListe<T>::KListe()
{
   head=tail=NULL;
} // end template<class T> KListe<T>::KListe()


template<class T> KListe<T>::~KListe()
{
   purge_rest();
} // end template<class T> KListe<T>::~KListe()
```

```
template<class T> void KListe<T>::purge_rest()
{
   KElement<T> *tmp=head;
   while (tmp)
   {
      head=head->next;
      delete tmp;
      tmp=head;
   } // end while
   head=NULL;
   tail=NULL;
} // end void KListe::purge_rest()
```

```
template<class T> KListe<T>& KListe<T>::operator=(KListe<T>& old)
{
    purge_rest();
    KElement<T> *current=old.head;
    while (current)
    {
        (*this)+=current->data;
        current=current->next;
    } // end while
    return(*this);
} // end KListe& template<class T> KListe<T>::operator=(KListe)


template<class T> void KListe<T>::operator+=(T neu)
{
   KElement<T> *pelem;
   pelem = new KElement<T>;
   pelem->data=neu;
   pelem->next=NULL;
   if (tail==NULL)
   {
      head=tail=pelem;
   }
   else if (head==tail)
   {
      head->next=tail=pelem;
   }
   else
```

```
    {
       tail->next=pelem;
       tail=tail->next;
    } // end if
} // end void template<class T> KListe<T>::operator+=(T neu)


template<class T> void KListe<T>::operator-=(T kill)
{
    while (head!=NULL && head->data==kill)
    {
       KElement<T> *tmp=head;
       if (head==tail)
         tail=head->next;
       head=head->next;
       delete tmp;
    } // end while

    if (head==NULL)
       return;


    KElement<T> *prev, *curr;
    prev=head;
    curr=prev->next;
    while (curr)
    {
       if (curr->data==kill)
```

```
      {
         prev->next=curr->next;
         if (curr==tail)
            tail=prev;
         delete curr;
         curr=prev->next;
      }
      else

      {
         prev=curr;
         curr=curr->next;
      }
   } // end while
} // end void template<class T> KListe<T>::operator-=(T kill)
                       // Freundoperator der Klasse
                       // KListe<int>
ostream & operator<<(ostream& out, KListe<int> & liste)
{
   if (liste.head==NULL)
   {
      out << "Die Lineare int-Liste ist leer.";
      return(out);
   }
   // Liste ist nicht leer:
   KElement<int> * current;
   current=liste.head;
   while (current!=NULL)
```

```
   {
      out << " " << current->data;
      current=current->next;
   } // end while
   return(out);
} // end ostream & operator<<(ostream& out,KListe<int> liste)


                        // Freundoperator der Klasse
                        // KListe<float>
ostream & operator<<(ostream& out, KListe<float> & liste)
{
   if (liste.head==NULL)
   {
      out << "Die Lineare float-Liste ist leer.";
      return(out);
   }
   // Liste ist nicht leer:
   KElement<float> * current;
   current=liste.head;
   while (current!=NULL)
   {
      out << " " << current->data;
      current=current->next;
   } // end while
   return(out);
} // end ostream & operator<<(ostream& out,KListe<float> liste)
```

```
KListe<int>;             // Erzeugen der Realisierungen
KListe<float>;
// Hauptprogramm
int main()
{
   cout << "LISTE2 - Einfache Lineare Listen (Template-Version)" << endl;
   cout << "---------------------------------------------------" << endl;

   KListe<int> iliste;         // Eine int-Liste wird generiert...
   cout << "Zu Beginn int-Liste: " << iliste << endl;
   for (int i=1; i<10; i++)
   {
      iliste+=i;
   } // end for i
   cout << "Zum Abschluß int-Liste: " << iliste << endl;

   KListe<float> fliste;       // Eine float-Liste wird generiert...
   cout << "Zu Beginn float-Liste: " << fliste << endl;
   fliste+=1.5;
   fliste+=2.5;
   fliste+=3.5;
   cout << "Zum Abschluß float-Liste: " << fliste << endl;

   return EXIT_SUCCESS;
} // end main
// ---------------------------------------------------------------------
// end of file liste2.cpp
// ---------------------------------------------------------------------
```

Und auch hier wieder das erwartungsgemäße Ablauflisting:

```
LISTE2 - Einfache Lineare Listen (Template-Version)

---------------------------------------------------

Zu Beginn int-Liste: Die Lineare int-Liste ist leer.

Zum Abschluß int-Liste:  1 2 3 4 5 6 7 8 9

Zu Beginn float-Liste: Die Lineare float-Liste ist leer.

Zum Abschluß float-Liste:  1.5 2.5 3.5
```

# B. Anhang II: Ergänzungen

In diesem Teil des Anhangs sollen ein paar technische Ergänzungen (Übersicht über die Digraphen, Trigraphen und die C++ Schlüsselwörter) sowie einige Tips zum objektorientierten Design gegeben werden.

## B.1. Digraphen

Der ANSI/ISO C++ Draft (Diskussionsentwurf) von Februar 1995 sieht sogenannte Digraphen[69] (Zweizeichenfolgen) vor, die hilfreich sein können, wenn auf einer Tastatur einige Sonderzeichen nicht direkt erreichbar sind.

<table>
<tr><th>Zweizeichenfolge (Digraph)</th><th>...ersetzt das Zeichen</th></tr>
<tr><td><%</td><td>{</td></tr>
<tr><td>%></td><td>}</td></tr>
<tr><td><:</td><td>[</td></tr>
<tr><td>:></td><td>]</td></tr>
<tr><td>%:</td><td>#</td></tr>
<tr><td>%:%:</td><td>##</td></tr>
<tr><td>and</td><td>&&</td></tr>
<tr><td>bitor</td><td>|</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>compl</td><td>~</td></tr>
<tr><td>bitand</td><td>&</td></tr>
<tr><td>and_eq</td><td>&=</td></tr>
<tr><td>or_eq</td><td>|=</td></tr>
<tr><td>xor_eq</td><td>^=</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>not_eq</td><td>!=</td></tr>
</table>

[69] Bereits in ANSI-C wurden die sogenannten Trigraphen definiert, drei Zeichen lange Ersatzdarstellungen für eventuell auf einer Tastatur nicht verfügbare Zeichen. Wenn Sie eine Seite weiterblättern, sehen Sie sie...

## B.2. Trigraphen

Bereits ANSI-C hat die oben erwähnten Trigraphen (Dreizeichenfolgen) definiert. Zur Erinnerung werden diese neun Ersetzungsmöglichkeiten hier aufgelistet. Allerdings zeigt die Erfahrung, daß nicht mehr alle C++-Compiler dieses Konzept unterstützen. Man sollte allerdings wissen, daß es diese Trigraphen gibt, und daß eventuell auch hierin Fehlerquellen begründet sein können.

<table>
<tr><th>Dreizeichenfolge (Trigraph)</th><th>...ersetzt das Zeichen</th></tr>
<tr><td>??=</td><td>#</td></tr>
<tr><td>??(</td><td>[</td></tr>
<tr><td>??)</td><td>]</td></tr>
<tr><td>??/</td><td>\</td></tr>
<tr><td>??'</td><td>^</td></tr>
<tr><td>??<</td><td>{</td></tr>
<tr><td>??></td><td>}</td></tr>
<tr><td>??!</td><td>|</td></tr>
<tr><td>??-</td><td>~</td></tr>
</table>

## B.3. Schlüsselwörter

Der ANSI/ISO C++ Draft sieht einige weitere Schlüsselwörter für C++ vor, auf die in diesem Manuskript teilweise nicht eingegangen werden konnte. Der Vollständigkeit halber seien nachfolgend sämtliche Schlüsselwörter aufgelistet, die gemäß diesem Standard als reservierte Worte nicht anderweitig verwendet werden dürfen. Vorausschauende Programmierung bedeutet auch, daß diese geplanten Schlüsselwörter schon jetzt nicht mehr als selbstdefinierte Namen eingesetzt werden, auch wenn dies die heutigen C++-Compiler noch akzeptieren würden.

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

## B.4. Vererbung und Objektorientiertes Design - Einige Tips

Aus dem Buch von Scott Meyers, *Effektiv C++ programmieren: 50 Möglichkeiten zur Verbesserung Ihrer Programme*, Addison-Wesley 1995, werden abschließend noch ein paar (etwas gekürzte) Hinweise zum Zusammenhang zwischen Vererbung und Objektorientiertem Design aufgelistet.

1. Eine gemeinsame Basisklasse steht für gemeinsame Aufgaben.
   Wenn zwei Klassen B und C die Klasse A als Basisklasse besitzen, dann bedeutet dies, daß B und C gewisse Datenelemente und/oder Elementfunktionen gemeinsam haben.

2. Öffentliche Vererbung bedeutet „ist ein" (IS-A).
   Wenn die Klasse B öffentlich (public) von einer Basisklasse A abgeleitet ist, bedeutet dies, daß jedes Objekt des Typs B auch ein Objekt des Typs A ist, nicht jedoch umgekehrt.

3. Private Vererbung bedeutet „ist implementiert mit".
   Wenn die Klasse B privat von der Klasse A abgeleitet ist, dann bedeutet das nur, daß Objekte vom Typ B mit einem Objekt vom Typ A implementiert sind. Eine Beziehung auf der Ebene des Programmentwurfs besteht zwischen diesen Klassen nicht.

4. Layering bedeutet „hat ein" oder „ist implementiert mit".
   Wenn eine Klasse A ein Datenelement vom Typ B besitzt, haben Objekte des Typs A entweder eine Komponente vom Typ B oder sind mit einem Objekt vom Typ B implementiert.

5. Eine rein virtuelle Funktion bedeutet, daß nur die Schnittstelle geerbt wird.
   Wenn eine Klasse A eine rein virtuelle Elementfunktion mf deklariert, so müssen die Unterklassen von A die Schnittstelle von mf erben und für sie eine eigene Implementation bereitstellen.

6. Eine virtuelle Funktion bedeutet, daß die Schnittstelle und eine Standard-implementation geerbt werden.
   Wenn eine Klasse A eine einfache virtuelle Elementfunktion mf deklariert, müssen die Unterklassen von A die Schnittstelle von mf und können wahlweise auch noch eine Standardimplementation erben.

7. Eine nichtvirtuelle Funktion bedeutet, daß die Schnittstelle und eine obligatorische Implementation geerbt werden.
   Wenn eine Klasse A eine nichtvirtuelle Elementfunktion mf deklariert, so müssen die Unterklassen von A die Schnittstelle von mf und ihre Implementation erben. Damit definiert mf eine Invariante über Spezialisierung von A.

# Literaturhinweise

An dieser Stelle soll keine ausführliche Auflistung aller möglichen Bücher zum Thema OOP bzw. C++ erfolgen, denn mittlerweile tummeln sich auf diesem Gebiet *sehr* viele Werke. Stattdessen nur einige exemplarische Verweise auf Bücher, die für verschiedene Zwecke interessant sein können.

Zum einen ist da die Referenz vom Erfinder von C++ schlechthin:

- Stroustrup, Bjarne
  Die C++ Programmiersprache
  Addison-Wesley, 1992-1994 (2. Auflage, 4.Nachdruck)

Außerdem gibt es von ihm auch das neuere Werk – mittlerweile auch in deutscher Sprache –

- Stroustrup, Bjarne
  The Design and Evolution of C++
  Addison-Wesley, 1994

Für intime Kenner und Liebhaber der schönen Programmiersprache Pascal bietet sich das folgende Buch an:

- Gerike, Roman R.
  Weiter mit C++
  Eine Einführung für Pascal- und C-Programmierer
  Heise-Verlag, 1992

Ein etwas älteres, recht praxisnahes Buch, in dem der Autor die konkrete Arbeit mit mehreren Compilern (u.a. dem Zortech C++ Compiler, heute Symantec C++) sowie die Portabilität des erzeugten Codes in den Vordergrund stellt, ist

- Huckert, Edgar
  Programmieren in C++
  Markt & Technik Verlag, 1990

Recht preisgünstig zu haben ist ein kleines Büchlein aus dem bhv-Verlag, das jedoch nicht in die Tiefen von C++ eindringt:

- Herglotz, Walter
  Das Einsteigerseminar: C++
  bhv Verlag, 1994

Und wenn Sie in einiger Zukunft vertraut sein werden mit den grundlegenden Aspekten von C++, dann empfehle ich Ihnen ein Buch mit fünfzig und eines mit 161 guten Tips (und einem schönen Buchtitel):

- Meyers, Scott
  Effektiv C++ programmieren: 50 Möglichkeiten zur Verbesserung Ihrer Programme
  Addison-Wesley, 1995

- Holub, Allen I.
  Enough Rope to Shoot Yourself in the Foot
  McGraw-Hill, 1995

Zur auf Seite 0 erwähnten Standard Template Library ein englischsprachiger und ein deutschsprachiger Buchhinweis:

- Nelson, Mark
  C++ Programmer's Guide to the Standard Template Library
  IDG Books Worldwide, Inc. Foster City, 1995

- Breymann, Ulrich
  Die C++ Standard Template Library - Einführung, Anwendungen, Konstruktion neuer Komponenten
  Addison-Wesley, 1996

Außerdem sei noch ein englischsprachiges Buch erwähnt, das mehr grundlegend das komplexe Thema Objektorientierung, jedoch mit Beispielen in C++, behandelt.

- Mullin, Mark
  Object Oriented Program Design
  Addison-Wesley, 1989

Ebenfalls erwähnt sei das in Abschnitt 1.2 zur Datenabstraktion zitierte Buch von Balzert.

- Balzert, Helmut
  Die Entwicklung von Software-Systemen
  Spektrum Akademischer Verlag, 1982

# Stichwortverzeichnis


**~ // << >> .**

--, 61

->*, 31

#define, 22

++, 61

., 55

.*, 31, 55

.a, 24

.c, 2

.CPP, 2

.dll, 24

.h, 2

.lib, 24

.o, 24

.obj, 24

.sl, 24

//, 12

~, 38

::, 55

<, 16, 64, 68

>>, 64, 68

?:, 55

**A**

ABC, 104

abgeleitete Klasse, 79

abstract base class, 104

abstrakte Basisklasse, 104

Abstrakte Klassen, 8

abstrakter Datentyp, 2, 5, 19

abstrakter Stack, 3

abstraktes Datenobjekt, 8

ADT, 2, 5

Alias, 17, 41

Alias-Bezeichnung, 40

ANSI/ISO C++ Draft, 183, 185

app, 115

asm, 14

Assembler, 26

assert(), 132

AT&T, 132

ate, 115

auto, 14

automatische Typumwandlung, 76

**B**

base class, 79

Basisklasse, 79

befreundet, 61

Bibliothek, 1

Bibliothekskonzept, 137

binary, 115

bool, 185

Borland, 95

Botschaft, 7

break, 14

C

C, 16

C++, 7

call by reference, 40

call by value, 40, 42, 96

case, 14

Casting, 76

catch, 14, 132

cerr, 63, 68, 113

char, 14

char * const, 17

cin, 63, 64, 113

class, 14

Class Hierarchy Browser, 9, 79

clog, 63, 68, 113

close(), 114

Code-Generierung, 26, 32

COMPLEX, 76

const, 14, 17

const char *, 17

const_cast, 185

Container, 137

Containerklasse, 125, 126

continue, 14

cout, 63, 64, 68, 113

Ctor, 162

D

Datei-Stream, 113

Datenabstraktion, 2

Datenkapselung, 8

Datentyp, 19

default, 14

Default-Konstruktor, 35, 37

Default-Parameter, 13

delete, 14, 49

derived class, 79

Destruktor, 35, 38

Digraph, 183

do, 14

DOS, 24

double, 14

Dreizeichenfolge, 184

Dtor, 162

dynamic_cast, 185

dynamisch, 49

dynamische Bindung, 11

E

early binding, 102

else, 14

endl, 65

enum, 14

Exception Handling, 129, 132

Exemplarvariable, 8

Exit-Code, 13

EXIT_SUCCESS, 22

explicit, 185

extern, 14

## F

false, 185

FAQ, 137

Fehlerbehandlung, 129

float, 14

floating point exception, 130

for, 14

FPE, 130

free(), 49

Frequently Asked Questions, 137

Freund, 61, 71, 74, 173

Freund-Klassen, 61

Freundfunktion, 61

friend, 14, 61, 71

frühe Bindung, 11, 102

fstream, 113

FSTREAM.H, 113

## G

Geburt, 35

Geheimhaltungsprinzip, 8

generisch, 122, 124

generischer Stack, 6

goto, 14

graphical user interface, 138

graphische Benutzeroberfläche, 138

Gültigkeit, 51

Gültigkeitsbereich, 40, 51

GUI, 138

## H

Heap, 39, 49, 175

Hierarchie, 7

HP-UX, 2, 24

HP-UX C++ Compiler, 2

## I

if, 14

ifstream, 113

Implementation, 23, 24

in, 115

Information Hiding, 8

Initialisierung, 35

Inkrementoperator, 61, 161

inline, 14, 25, 45, 115

Instanz, 8, 10, 19

Instanzmethode, 8

Instanzvariable, 8, 10

int, 14

iomanip.h, 65

ios, 64, 113

ios::beg, 116

ios::cur, 116

ios::end, 116

iostream, 64

iostream.h, 16, 64

IS-A, 186

istream, 64

## J

jmp_buf, 131

## K

Kanal, 63

Kapselung, 8

kill, 129

Klasse, 8, 19, 21

Klassen-Template, 124

Klassenbibliothek, 9, 24, 137

Klassenbildung, 21

Klassenhierarchie, 9

Klassenmethode, 8

Klassenvariable, 8

Knoten, 173

Kommentar, 12

konstante Instanzvariable, 47

Konstante Klassenmitglieder, 47

Konstruktor, 35, 37

Kopierkonstruktor, 42, 96

## L

late binding, 11, 102

Laufzeit-Information, 102

Layering, 186

Lebensdauer, 40, 51

Libraries, 1

Library, 24

Lineare Liste, 173

lokale Klasse, 134

long, 14

longjmp(), 131

## M

Makefile, 23

Makro, 121

malloc(), 49

Manipulator, 65

max(a,b), 121

Mehrfache Vererbung, 11

Mehrfachvererbung, 92

message, 7

Methode, 7, 19

Methoden Interface, 7

MFC, 137, 138

Microsoft Foundation Class, 137

Motif, 137

MS-Windows, 24

multiple inheritance, 11, 92

Muster, 121

mutable, 185

## N

Name Mangling, 14

Namensraum, 22

namespace, 22, 185

new, 14, 49, 175

Nichtlokale Sprünge, 131

nocreate, 115

noreplace, 115

NULL, 49

## O

Oberklasse, 9

Object Windows Library, 95

Objekt, 2, 7

objektorientierte Programmiersprache, 9

objektorientierte Programmierung, 7

Objektorientierung, 188

Offset, 32

ofstream, 113

OOP, 9, 187

open(), 114

open_mode, 115

operator, 14, 15

operator++, 166

ostream, 64

out, 115

overloaded casting, 115

overloading, 10, 55, 68

OWL, 95

## P

Parameterliste, 16

Pascal, 1

Pattern, 121

Pointer, 40

Pointer-Arithmetik, 49

Polymorphie, 7, 11, 16

Polymorphismus, 11, 101

pop, 3

Postfix, 161

Präfix, 161

Präprozessor-Makro, 121

printf(), 63

Priorität, 15

private, 14, 22, 25, 80, 83

Projekt, 1

Projektdatei, 23

protected, 14, 22, 25, 80, 83

Prototyping, 13

public, 14, 22, 25, 80, 83

pur virtuelle Methode, 104

push, 3

## R

random access, 113, 115

Referenz, 17, 40, 58

Referenzparameter, 40

register, 14

reinterpret_cast, 185

rekursiv, 25

reservierten Worte, 14

return, 14

reusability, 11, 24

reusable components, 79

## S

scanf(), 63

Schablone, 9

Schablonen, 121

Schiebeoperator, 64

Schlüsselwörter, 14, 185

Schnittstelle, 7, 19, 23

Schutzbereich, 25

seekg(), 116

seekp(), 116

sequentiell, 113

setjmp(), 131

short, 14

SIGINT, 129

Signal, 129

Signal-Handler, 129

Signalbehandlung, 129

signed, 14

SIGTERM, 129

sizeof, 14

Smalltalk, 7, 9

Software-Baustein, 83

späte Bindung, 11, 102

Sprünge, 129

Stack, 3

Stack-Frame, 25

Stack, generischer, 6

Standard-Stream, 63

Standard Template Library, 137

static, 14

static_cast, 185

statisch, 163

statische Bindung, 11

Statische Klassenmitglieder, 43

statisches Klassenelement, 163

statisches Klassenmitglied, 58

stdaux, 63

stderr, 63

stdin, 63

stdio.h, 63

stdout, 16, 63

stdprn, 63

STL, 137

Stream, 63

Stream-Klasse, 64

Stroustrup, 76, 187

struct, 14

Struktur, 19

Subklasse, 9, 79

Superklasse, 9, 79

switch, 14

Symantec, 26, 45, 187

Symantec C++, 26

## T

Taxonomie, 9

template, 6, 9, 14, 178

Template-Funktion, 121

Templates, 121

temporär, 42

this, 14, 37, 45, 56

throw, 14, 132

Tilde, 38

time_t, 161

Top-Of-Stack, 127

top of stack, 3

tos, 3, 127

transitiv, 10

Trigraph, 184

true, 185

trunc, 115

try, 14, 132

typedef, 14

typeid, 185

typename, 185

Typprüfung, 22

## U

Überladen, 10, 13, 37, 55, 61, 68

überladene Typkonvertierung, 115

Umlenkung, 63

union, 14

UNIX, 2, 45, 63, 129

unsichtbare Daten, 7

unsigned, 14

Unterklasse, 9

using, 185

## V

Vererbung, 8, 9, 24, 79

Vererbungsarten, 80

virtual, 14, 102

Virtual Reality, 101

virtuell, 76, 186

virtuelle Funktion, 102

Virtuelle Methoden-Tabelle, 103

Visual C++, 138

VMT, 103

void, 14

volatile, 14

Vorlagen, 121

## W

wahlfrei, 113

Wahlfreier Zugriff, 115

wchar_t, 185

while, 14

Wiederverwendbarkeit, 24

Wiederverwendbarkeit von Code, 11, 79

## X

X::X(), 37

X::X(const X&), 43

X::X(X&), 43

X::~X(), 38

## Z

Zeiger auf Klassenelemente, 31

Zeilenkommentar, 16

Zeilenvorschub, 65

Zortech, 187

Zweizeichenfolge, 183

Dokument: CPPBUCH2.LWP vom 05.04.1995-06.01.2006

Erstellt mit Lotus Word Pro.

© 1995-2006 P.Baeumle-Courth, Bergisch Gladbach