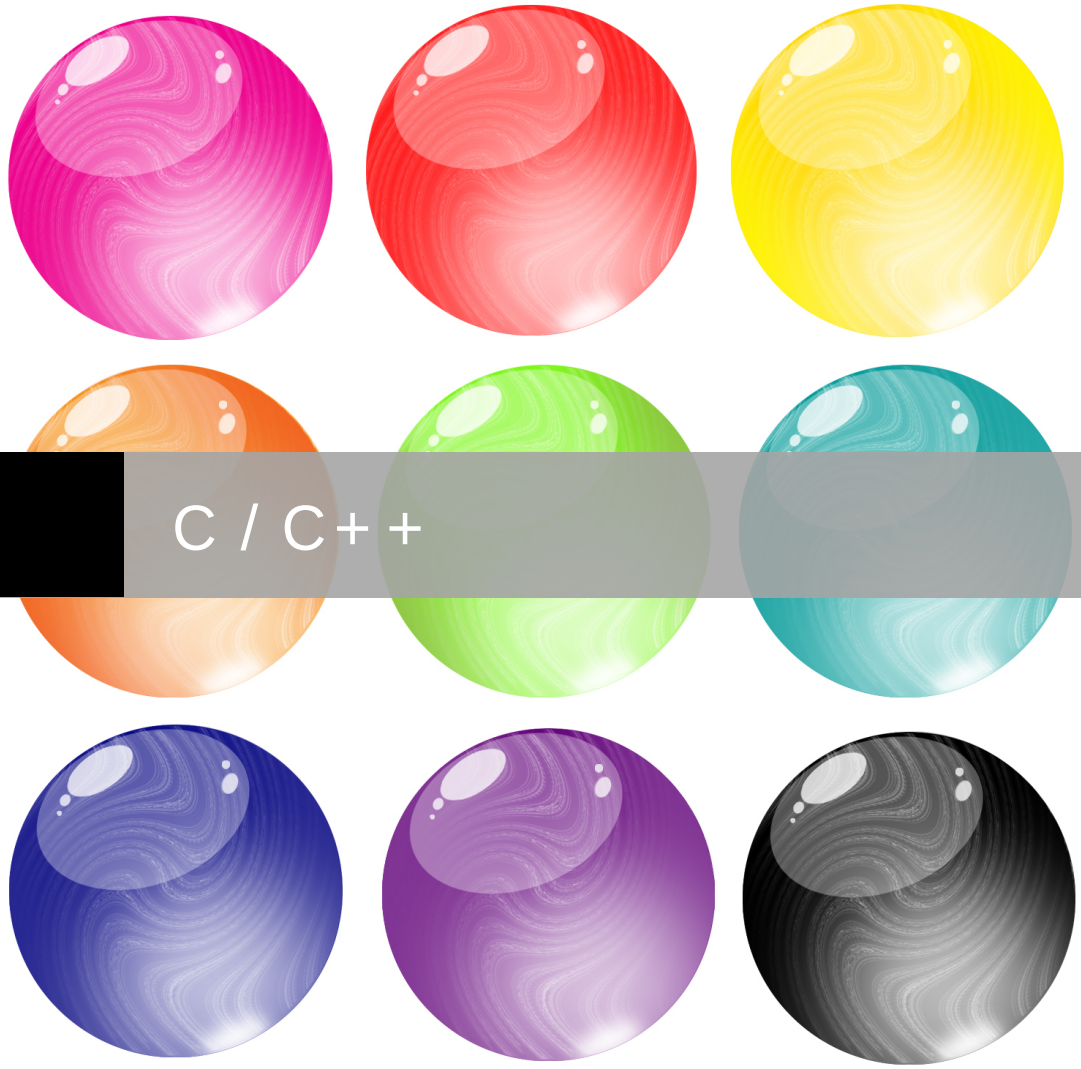
Heiko Gorski

C / C++ für Fortgeschrittene



C / C++

www.HeikoGorski.de

VORWORT ZUR 17. AUSGABE

An diesem Skript haben, über die Jahre verteilt, viele Menschen mitgewirkt – Freunde, Bekannte, Kollegen und natürlich Kursteilnehmer. Alle haben mich eifrig mit Verbesserungsvorschlägen – und nach mühseliger Detektivarbeit auch mit Fehlerkorrekturen versorgt. Ihnen allen gilt mein Dank für die wertvolle Unterstützung.

Ein ganz besonderer Dank gebührt meiner Frau, die es seit Jahren erträgt, dass ich jeden Urlaub und viele Wochenenden dazu nutze, dieses Skript zu überarbeiten und zu erweitern, so dass es von seiner „Erstausgabe“ mit knapp 20 Seiten auf den heutigen Umfang wachsen konnte.

Dipl. Inform Heiko Gorski, Januar 2012

INHALTSVERZEICHNIS

22. KLASSEN

Klassen sind das Herzstück des objektorientierten Ansatzes in C++. Sie erlauben es dem Entwickler der Programmiersprache C++ neue Datentypen „beizubringen“, die sich, wenn sie sorgfältig entwickelt wurden, nahtlos in das System einfügen und praktisch wie die eingebauten Typen (*int*, *long*, *double* etc.) verhalten. Klassen sind zumeist hochgradig komplexe Gebilde, basierend auf der Syntax eines zusammengesetzten Datentyps (*struct*).

Das Konzept der Klassen lässt sich an einem Beispiel wesentlich leichter erklären und verstehen, daher soll eine einfache Datenklasse *datum* als Anschauungsobjekt dienen. Zuerst benötigt man eine Klassendeklaration, die im Wesentlichen einer Strukturdeklaration ähnelt, aber mit dem Schlüsselwort *class* eingeleitet wird. Diese Klassenvereinbarung wird in eine Headerdatei (hier z.B. *datum.h*) gestellt, damit sie anderen Programmteilen verfügbar gemacht werden kann:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
   };
#endif
```

Man beachte das abschließende Semikolon, welches auf keinen Fall fehlen darf, da sonst die Klassendeklaration nicht abgeschlossen ist. Bereits diese Klassenvereinbarung ist ausreichend, um Variablen vom Typ *datum* zu erzeugen (im C++ spricht man davon, dass ein *datum*-Objekt instanziiert werden kann):

```
//============================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include "datum.h"

void main (void)
{
   datum x;
}
```

So wie bisher beschrieben, erfüllt die Klasse natürlich noch keinerlei Zweck, denn eine Möglichkeit in einer Instanz von *datum* Daten zu speichern ist noch nicht enthalten. Dennoch, so „leer“ wie die Klasse auf den ersten Blick erscheint, ist sie nicht. Tatsächlich enthält jede Klasse immer mindestens eine unsichtbare Zeigerkonstante (mit dem Namen *this*, siehe dazu auch den entsprechenden Abschnitt) und zwei unsichtbare Funktionen (*Constructor* und *Destructor* genannt). Die Bedeutung dieser unsichtbaren Klassenbestandteile soll etwas später erläutert werden, da sie für das Beispiel an dieser Stelle noch nicht relevant ist.

Um der Klasse *datum* Sinn zu verleihen, muss diese in der Lage sein ein vollwertiges Datum, d.h. Angaben für Tag, Monat und Jahr aufzunehmen. Dazu deklariert man, wie in einer Struktur, Komponenten. Komponenten (Variablen), die zu einer Klasse gehören nennt man *Member* oder *Membervariable*.

22.1. MEMBER-VARIABLEN

Beim Entwurf einer Klasse ist zunächst ihr Datenumfang zu klären – also die Menge an Daten, die unbedingt benötigt wird, um das Objekt (hier ein Datum) zu charakterisieren. Dazu stehen neben allen Grunddatentypen auch alle dem System bereits bekannten Klassen (!) zur Verfügung. Für eine Datumsklasse sieht ein typischer Entwurf wie folgt aus:

```
//=========================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      int Day;
      int Month;
      int Year;
   };
#endif
```

Wird ein Objekt instanziiert, so erhält es (genau wie eine Struktur) seinen eigenen Datenbereich. Die Klassenvereinbarung hat also noch immer große Ähnlichkeit mit einer Struktur, so dass man annehmen sollte, auch in gleicher Weise darauf zugreifen zu können:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
```

```
// in diesem Kapitel gezeigten Standes.
//============================================================

#include "datum.h"

void main (void)
{
   datum x;
   x.Day = 7;       // geht leider nicht!
}
```

An dieser Stelle trifft man auf den ersten Unterschied zur Struktur, der sogenannten *Datenkapselung* (*data encapsulation* oder *data hiding).* Grundsätzlich gilt für alle Klassen, dass Bestandteile, auf die der Zugriff nicht ausdrücklich erlaubt wird, dem Zugriff (Sichtbarkeit) des restlichen Programms entzogen sind. Solche „verborgenen“ Variablen gehören zu den sogenannten *private*-Variablen. Da *private* die Voreinstellung bei Klassen ist (*public* ist die Voreinstellung für Strukturen, s.u.), ist die bisherige Deklaration von *datum* mit der folgenden identisch:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      private:
         int Day;
         int Month;
         int Year;
   };
#endif
```

22.2. SICHTBARKEITSANGABEN - PRIVATE, PROTECTED, PUBLIC

Insgesamt gibt es in C++ drei verschiedene Stufen der Sichtbarkeit, die jeweils unterschiedliche Auswirkungen auf die Programmentwicklung und den Programmablauf haben.

Sichtbarkeitsangaben	
private	Variablen der Sichtbarkeit „private“ können nur innerhalb der Klasse verändert werden (von den sogenannten Methoden der Klasse, s.u.). Ein Zugriff auf diese Member von außerhalb ist nicht möglich, d.h. kein Programmteil außerhalb der Klasse kann direkten Einfluss auf diese Variable nehmen, um z.B. ihren Wert zu ändern oder abzufragen. Alle Member, die nicht den Schlüsselbegriffen „protected“ oder „public“ zugeordnet wurden, sind private (Standardeinstellung). Die „private“-Variablen werden nicht an abgeleitete Klassen vererbt (s.u.).
protected	Variablen des Sichtbarkeitstyps verhalten sich genauso wie Variablen des Typs „private“, werden jedoch an abgeleitete Klassen vererbt (s.u.).
public	Der Sichtbarkeitstyp „public“ macht Variablen der Klasse frei zugänglich, d.h. jedes Programm und jede Funktion, die ein Objekt kennt, kann die „public“-Variablen einer Instanz verändern, ohne dass das Objekt dieses bemerkt. Sie verhalten sich somit genau wie die Komponenten einer Struktur. Die Verwendung von „public“-Variablen vereinfacht zwar scheinbar die Programmierung, unterbindet jedoch die gewünschte Datenkapselung. Variablen des Typs „public“, werden an abgeleitete Klassen vererbt (s.u.).

Tabelle 1: Sichtbarkeitsangaben

Der einfachste Weg, den Zugriff auf die Variablen *Day*, *Month* und *Year* der Klasse *datum* zu ermöglichen, ist diese Member als *public* zu vereinbaren:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_
   class datum
   {
      public:
         int Day;
         int Month;
         int Year;
   };
#endif
```

Wie man leicht erkennen kann, ist es nicht nötig, alle Variablen mit dem entsprechenden Sichtbarkeitstyp zu versehen (hier: *public*), stattdessen werden üblicherweise Sichtbarkeitsgruppen gebildet. Mit Einführung des Schlüsselwortes *public* in die Deklaration der Klasse *datum* ist jetzt auch das oben beschriebene Programm korrekt:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include "datum.h"

void main (void)
{
   datum x;
   x.Day = 70;   // geht!
}
```

Der Nachteil ist im Beispiel sofort ersichtlich - der Klassenentwickler hat jetzt keine Möglichkeit mehr zu verhindern, dass ein Programmierer (oder Anwender) unsinnige Daten in das Objekt schreibt. Genau das ist aber der Sinn der Data-Encapsulation, die einen unkontrollierten Zugriff auf Datenobjekte unterbinden soll.

Da die Öffnung der Klasse mittels *public*-Member offensichtlich nicht zum gewünschten Ergebnis führt, muss es einen anderen Zugriffsweg geben. In C++ sind dies die Klassenmethoden. Im obigen Beispiel setzt man daher üblicherweise die Sichtbarkeit der Membervariablen auf *protected*, weil zur Implementationszeit noch nicht klar ist, ob von der Klasse *datum* andere Klassen abgeleitet werden sollen (siehe Abschnitt über Vererbung).

```
//=========================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;
   };
#endif
```

22.3. MEMBER-FUNKTIONEN (METHODEN)

Die *Methoden* (lokale Funktionen der Klasse) sind der entscheidende Unterschied zur Struktur. Statt wie letztere nur Daten zusammenzufassen, erlaubt es die Klasse zusätzlich Funktionen zu definieren, die nur im Zusammenhang mit dem Objektnamen gültig sind (so wie ein Komponenten- oder Membername). Dadurch ergibt sich die Möglichkeit, den gleichen Methodennamen in anderen Klassen wiederzuverwenden. So kann man z.B. alle Druckausgaben über eine – jeweils pro Klasse zu erstellende – Methode *print* handhaben.

Methoden unterliegen den gleichen Regeln zur Kapselung, wie die Membervariablen – d.h. auch den Methoden kann und sollte man Sichtbarkeitsangaben zuordnen. Da Funktionsprototypen in C++ Pflicht sind, werden auch die Methoden in der Klassendeklaration aufgeführt - Und weil diese neu einzuführenden Methoden den Zugriff auf die Membervariablen von „außen“ ermöglichen sollen, müssen diese zwangsläufig dem Typ *public* angehören:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;

      public:
         void setDay   (int t);
         void setMonth (int m);
         void setYear  (int j);

         int getDay   (void);
         int getMonth (void);
         int getYear  (void);
   };
#endif
```

Üblicherweise werden die Methoden einer Klasse in einer eigenen Quelltextdatei implementiert. Damit der Compiler bei der Übersetzung die richtige Methode für einen Funktionsaufruf auswählen kann, muss man die einzelnen Methoden, die ja gleiche Namen haben können (in unterschiedlichen Klassen), bei der Funktionsdefinition um den

Klassenbezug erweitern. Dies geschieht, indem man den Klassennamen vor den Methodennamen setzt und beide mit „*::*“ verbindet. Man nennt diese Beschreibung/Einschränkung der Gültigkeit den *Scopedelimiter* oder *Scope resolution operator*:

```
//============================================================
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include "datum.h"

//------------------------------------------------------------
// Tagesangabe holen
//------------------------------------------------------------
int datum::getDay (void)
{
   return (Day);
}

//------------------------------------------------------------
// Monatsangabe holen
//------------------------------------------------------------
int datum::getMonth (void)
{
   return (Month);
}

//------------------------------------------------------------
// Jahresangabe holen
//------------------------------------------------------------
int datum::getYear (void)
{
   return (Year);
}

//------------------------------------------------------------
// Tagesangabe setzen
//------------------------------------------------------------
void datum::setDay (int t)
{
   //---------------------------------------------------------
   // Alle Monate : min. Monatserster
   //---------------------------------------------------------
   if (t < 1) t = 1;
   switch (Month)
   {
      //------------------------------------------------------
      // Februar - es gilt immer:
      //------------------------------------------------------
      case 2 : if (t > 29) t = 29;
               //-----------------------------------------
               // für Nicht-Schaltjahre gilt immer:
               //-----------------------------------------
```

```
                if (Year % 4) { if (t > 28) t = 28; }
                else
                   //---------------------------------------
                   // durch 100 teilb.? kein Schaltj.
                   // durch 400 teilbar? doch nicht
                   //---------------------------------------
                   if ((!(Year % 100))&&(Year % 400))
                      if (t > 28) t = 28;
                break;
      //-------------------------------------------------------
      // Monate mit 30 Tagen
      //-------------------------------------------------------
      case 4 :
      case 6 :
      case 9 :
      case 11: if (t > 30) t = 30;
               break;
      //-------------------------------------------------------
      // Monate mit 31 Tagen bleiben übrig
      //-------------------------------------------------------
      default: if (t > 31) t = 31;
               break;
   }
   Day = t;
}

//-------------------------------------------------------------
// Monatsangabe setzen
//-------------------------------------------------------------
void datum::setMonth (int m)
{
   if (m > 12) m = 12;   // max. Dezember
   if (m < 1)  m = 1;    // min. Januar
   Month = m;
}

//-------------------------------------------------------------
// Jahresangabe setzen
//-------------------------------------------------------------
void datum::setYear (int j)
{
   if (j > 3000) j = 3000;   // max. bis Jahr 3000
   if (j < 1800) j = 1800;   // min. Jahr 1800
   Year = j;
}
```

Die Implementation der Methoden macht den Interface-Charakter gegenüber der Data-Encapsulation deutlich. Dadurch dass man z.B. die Methode *setMonth* benutzen muss, um den Wert der Membervariablen *Month* zu setzen, kann der Entwickler sicherstellen, dass niemals unsinnige Werte in die Member geschrieben werden. Man beachte, dass es innerhalb der Methoden einer Klasse völlig legal ist, auf die Member der eigenen Klasse zuzugreifen. Der Sichtbarkeitstyp ist dabei irrelevant,

er gilt immer nur für Nicht-Klassen-Mitglieder. Weil der Rechner zudem weiß, dass es sich an dieser Stelle (innerhalb einer Methode der Klasse *datum*) bei der Variablen *Day* um eine globale Variable oder einen Member der Klasse handeln muss, entfällt auch die Notwendigkeit den Scope anzugeben. Ein gültiges Datum kann jetzt erzeugt werden mit:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x;
   x.setYear  (1900);
   x.setMonth (2);
   x.setDay   (31);         // Korrektur nötig!

   //-----------------------------------------------------
   //  Ergebnis:28.2.1900
   //-----------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;
}
```

Die nächste Schwierigkeit taucht zwangsläufig bei der Implementation der Methode *setDay* auf. Der mögliche Wert für *Month* ist klar vorgegeben, er muss zwischen 1 und 12 liegen, auch das Jahr lässt sich jetzt (hier zwar recht willkürlich) eingrenzen. Wichtig ist allein, dass beide Werte völlig unabhängig von anderen Membervariablen sind. Anders sieht es bei der maximalen Tageszahl aus, sie ist abhängig vom Monat und vom Jahr (Schaltjahre). Plötzlich wird also die Reihenfolge, in der die Member belegt werden, zum Problem:

```
#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x;             // Korrektur nötig, erfolgt
   x.setDay   (31);     // aber nicht, da Month und
   x.setYear  (1900);   // Year noch unbesetzt oder
```

```
   x.setMonth (2);     // mit Zufallswerten besetzt sind!

   //-------------------------------------------------------
   // Ergebnis:31.2.1900
   //-------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;
}
```

Was geschieht, wenn die Methode *setDay* aufgerufen wird, bevor *Month* und *Year* einen nachprüfbaren Wert haben? – Bestenfalls sind die Member mit Null vorbelegt (wenn *x* als globale Instanz erzeugt wurde), zumeist wird der Inhalt aber eher aus Zufallswerten bestehen – ein Fall der in der *switch*-Anweisung von *setDay* definitiv nicht vorgesehen ist!

Bei der Klasse *datum* könnte man sich damit behelfen, die Methoden *setMonth* und *setYear* am Beginn von *setDay* aufzurufen. Dann würden ggf. vorhandene Zufallswerte in *Month* und *Year* korrigiert. Da es aber notwendig ist, die Methode *setDay* auch in *setMonth* und *setYear* aufzurufen (um ggf. die Tageszahl zu korrigieren, wenn man den Monat wechselt) läuft man in eine unendliche Rekursion! (*setDay* ruft *setMonth* auf, welcher wiederum *setDay* aufruft usw.):

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

//------------------------------------------------------------
// Monatsangabe setzen
//------------------------------------------------------------
void datum::setMonth (int m)
{
   if (m > 12) m = 12;   // max. Dezember
   if (m < 1)  m = 1;    // min. Januar
   Month = m;
   setDay (Day);         // ggf. Tag korrigieren
}

//------------------------------------------------------------
// Jahresangabe setzen
//------------------------------------------------------------
void datum::setYear (int j)
{
   if (j > 3000) j = 3000;   // max. bis Jahr 3000
   if (j < 1800) j = 1800;   // min. Jahr 1800
   Year = j;
   setDay (Day);             // ggf. Tag korrigieren
}
```

Abhilfe lässt sich auch schaffen, indem man die Prüfroutinen in einer einzelnen *check*-Methode zusammenfasst, die unter Umständen alle drei Werte korrigiert. Diese Methode wird nur klassenintern benötigt und es besteht daher kein Grund sie dem Entwickler direkt zur Verfügung zu stellen, eine *public*-Deklaration ist also unnötig. Um die Methode bei Bedarf vererben zu können (siehe Abschnitt Vererbung), sollte sie jedoch die Sichtbarkeit *protected* bekommen:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;
         void check (void);

      public:
         void setDay   (int t);
         void setMonth (int m);
         void setYear  (int j);

         int getDay   (void);
         int getMonth (void);
         int getYear  (void);
   };
#endif
```

Die Implementation der Klasse ändert sich dann wie folgt:

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

//------------------------------------------------------------
// Monatsangabe setzen
//------------------------------------------------------------
void datum::setMonth (int m)
{
   Month = m;
   check ();
}
```

```
//-------------------------------------------------------------
// Jahresangabe setzen
//-------------------------------------------------------------
void datum::setYear (int j)
{
   Year = j;
   check ();
}

//-------------------------------------------------------------
// Tagesangabe setzen
//-------------------------------------------------------------
void datum::setDay (int t)
{
   Day = t;
   check ();
}

//-------------------------------------------------------------
// Datum prüfen
//-------------------------------------------------------------
void datum::check (void)
{
   if (Month > 12) Month = 12;   // max. Dezember
   if (Month < 1)  Month = 1;    // min. Januar
   if (Year > 3000) Year = 3000; // max. bis Jahr 3000
   if (Year < 1800) Year = 1800; // min. Jahr 1800
   if (Day < 1) Day = 1;         // min. Monatserster
   switch (Month)
   {
      case 2 : // Februar - es gilt immer:
               if (Day > 29) Day = 29;
               // für Nicht-Schaltjahre gilt immer:
               if (Year % 4) { if (Day > 28) Day= 28;}
               else
                  // durch 100 teilbar dann kein Schaltjahr.
                  // durch 400 teilbar? doch Schaltjahr
                  if ((!(Year % 100))&&(Year % 400))
                     if (Day > 28) Day = 28;
               break;
      case 4 :
      case 6 :
      case 9 :
      case 11: if (Day > 30) Day = 30;
               break;
      default: if (Day > 31) Day = 31;
               break;
   }
}
```

Durch diesen Ansatz wird bei jedem setzenden Zugriff auf die Member eine Überprüfung des Inhaltes durchgeführt. Vollständige Korrektheit ist aber immer noch nicht erreicht:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x;

   //---------------------------------------------------------
   // Ausgabeergibt leider Unsinn!!
   //---------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;

   x.setYear  (1900);
   x.setMonth (2);
   x.setDay   (31);         // Korrektur nötig!

   //---------------------------------------------------------
   // Ergebnis: 8.2.1900
   //---------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;
 }
```

Noch immer ist der Zustand des *datum*-Objektes direkt nach der Erzeugung völlig unbestimmt, was zu schweren Fehlern führen kann, wenn keine der Setzfunktionen aufgerufen wird, bevor eine Auswertung über die *get*-Methoden (z.B. *getMonth*) erfolgt. Die einzig mögliche Lösung für alle Probleme dieser Art besteht darin, der Klasse vorzuschreiben, wie die einzelnen Membervariablen am Anfang, d.h. schon bei der Erzeugung des Objektes vorzubelegen sind.

Dies erfolgt im sogenannten *Constructor*.

22.3.1. CONSTRUCTOREN UND DESTRUCTOREN

Ein Constructor ist eine spezielle Methode, die automatisch bei der Erzeugung (also der Deklaration) eines Objektes aufgerufen wird. Da diese Aufrufe zwangsläufig von Außerhalb der Klasse erfolgt, müssen Constructoren (und auch Destructoren) als *public* vereinbart werden.

Der Constructor einer Klasse sorgt primär dafür, dass automatisch Speicherplatz in der Größe der Summe der Objectmember beim Rechner

angefordert wird. Somit sind auch die Namen der Standarddatentypen (*int*, *double*) nichts anderes als Constructoren.

Eine ausprogrammierte Constructor-Methode ermöglicht es aber auch, jedem Member der Klasse gleich zu Beginn einen bestimmten Wert zuzuweisen oder vorab Funktionen aufzurufen.

Der Constructor zeichnet sich dadurch aus, dass er als Methodennamen immer den Namen der Klasse erhält und keinen Returnwert kennt (intern ist der Returnwert identisch mit dem neu konstruierten Objekt):

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;

      public:
         datum (void);            // Constructor

         void setDay   (int t);
         void setMonth (int m);
         void setYear  (int j);

         int getDay   (void);
         int getMonth (void);
         int getYear  (void);
   };
#endif
```

Die Implementation eines Constructors gleicht grundsätzlich der Definition aller anderen Methoden:

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

//------------------------------------------------------------
// Constructor
//------------------------------------------------------------
```

```
datum::datum (void)
{
   Day   = 1;
   Month = 1;
   Year  = 1900;
}
```

Durch die Definition des Constructors kann es (im hier dargestellten Beispiel) nun keine *datum*-Objekte ohne Initialisierung mehr geben, so dass auch die oben noch fehlerhafte Ausgabe sinnvolle Werte liefert:

```
//============================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x;

   //-----------------------------------------------------
   // Ausgabe: 1.1.1900
   //-----------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;

   x.setYear  (1900);
   x.setMonth (2);
   x.setDay   (31);         // Korrektur nötig !

   //-----------------------------------------------------
   // Ergebnis:28.2.1900
   //-----------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;
}
```

Besondere Bedeutung kommt dem Constructor zu, wenn die Klasse dynamische Daten behandeln soll (z.B. Zeichenketten unterschiedlicher Länge), bei denen alle Speicherallokationen und Deallokationen per Hand erledigt werden müssen (siehe Abschnitt über dynamische Speicherverwaltung).

Wird für eine Klasse kein Constructor definiert, dann erzeugt C++ automatisch einen sogenannten Default-Constructor, der nichts weiter

tut, als den notwendigen Speicherplatz für das zu konstruierende Objekt bei Rechner abzufordern.

Der Destructor ist das genaue Gegenteil. Er wird aufgerufen, wenn das Objekt „vernichtet“, d.h. dessen Speicherplatz freigegeben wird. Die Freigabe, also der Aufruf des Destructors, erfolgt meist automatisch (bei globalen Objekten am Ende eines Programms, bei lokalen Objekten am Ende der Funktion oder Methode, bei dynamisch erzeugten Objekten mit Aufruf von *delete*). Ein explizites Aufrufen des Destructors ist nur sehr selten nötig.

Grundsätzlich kann es nur einen Destructor pro Klasse geben und dieser kann, im Gegensatz zu den Constructoren, keinerlei Parameter verarbeiten. Meistens muss man sich um den Destructor aber überhaupt nicht kümmern, da C++ diesen automatisch anlegt (genau wie beim Default-Constructor). Nur wenn innerhalb des Objektes dynamisch allokierten Daten verwaltet werden ist eine Definition des Destructors unvermeidlich, da der dynamisch belegte Speicher unbedingt freigegeben werden muss, bevor das Objekt „vergessen“ wird.

Genau wie der Constructor kennt der Destructor keinen Returnwert und folgt einer bestimmten Namenskonvention. Als Methodennamen dient ebenfalls immer der Klassenname, es wird jedoch noch eine Tilde („~“) vorangestellt.

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;

      public:
         datum (void);       // Constructor
         ~datum (void);      // Destructor

         void setDay   (int t);
         void setMonth (int m);
         void setYear  (int j);

         int getDay   (void);
         int getMonth (void);
         int getYear  (void);
```

```
   };
#endif
```

Die Implementation des Destructors für die Klasse *datum* ist trivial, da er keinerlei Anweisungen enthält. Man kann es daher bedenkenlos beim Default-Destructor belassen:

```
//=========================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

datum::~datum (void)
{
}
```

Ein ausführlicheres Beispiel, mit einer Klasse die auch dynamische Variablen verwaltet (und daher einen Destructor benötigt), ist im Kapitel über dynamische Speicherverwaltung (*new*) ausgeführt.

22.3.2. INLINE-METHODEN

In fast allen Klassen gibt es einige Methoden, deren Implementation derartig simpel ist, dass der Aufwand eines echten Funktionsaufrufes nicht lohnt. Für diese Funktionalitäten kann man auf *inline*-Methoden zurückgreifen. Ein gutes Beispiel hierfür sind die *get*-Methoden (*getDay* etc.) der Klasse *datum*, die nur aus einem einfachen *return* bestehen, aber auch der Destructor, der keinerlei Anweisungen im Funktionsrumpf besitzt:

```
inline int datum::getMonth (void)
{
   return (Month);
}
```

Wenn die Implementation so kurz ist wie in diesem Fall, kann man sogar soweit gehen, die Implementation in der Klassendeklaration gleich mit vorzunehmen. Methoden, die in der Klassenbeschreibung definiert werden sind immer *inline*, auch wenn dies nicht gesondert angegeben wird. Die Headerdatei hat dann das folgende Aussehen:

```
//=========================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================
```

```
#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;

      public:
         datum (void);       // Constructor !
         ~datum (void) {};   // Inline-Destructor

         //-------------------------------------------------
         // Inline-Methoden (Makros):
         //-------------------------------------------------
         void setDay  (int t) {Day=d;   check();};
         void setMonth(int m) {Month=m; check();};
         void setYear (int j) {Year=j;  check();};

         int getDay   (void)  {return (Day);};
         int getMonth (void)  {return (Month);};
         int getYear  (void)  {return (Year);};

         int check (void);
   };
#endif
```

Die restlichen Implementationen (z. B. der Methode check) verbleiben natürlich in der zugehörigen Quelltextdatei der Klasse, in diesem Fall also in *datum.cpp*.

22.3.3. METHODEN-OVERLOADING

Für Methoden gelten (bis auf Einschränkungen des Scope und der Sichtbarkeit) die gleichen Bedingungen wie für normale C++-Funktionen. D.h. Methoden und Constructoren (nicht Destructoren) können bei Bedarf auch überladen werden. Überladene Methoden haben, wie in C++ üblich, alle den gleichen Namen, müssen sich aber anhand der Parameterleiste (Parameteranzahl oder -typ) eindeutig unterscheiden.

Besonders häufig wird dieser Mechanismus für Constructoren benutzt, da das Overloading eine recht bequeme Möglichkeit zur Initialisierung durch den Entwickler bietet. Die Klasse *datum* hat bisher nur einen parameterlosen Constructor, der die Membervariablen auf den 1.1.1900 setzt. Angenehmer ist es natürlich, wenn man bei der Deklaration bereits Initialisierungswerte angeben kann (aber nicht muss), so dass man sich einen Aufruf der *set*-Methoden (*setDay* etc.) sparen kann. Dieses Verhalten kann man erreichen, in dem man den Constructor durch eine zusätzliche Variante überlädt. Der Vorteil einen weiteren Constructor

einzuführen (anstatt den bisherigen zu ändern) liegt darin, dass bereits geschriebene Programme, die den „alten“ Constructor verwenden, nicht geändert werden müssen:

```
//=========================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;

      public:
         datum (void);                  // Constructor
         datum (int t, int m, int j);   // Constructor
         ~datum (void) {};              // Inline-Destructor

         //-----------------------------------------------
         // Inline-Methoden (Makros):
         //-----------------------------------------------
         void setDay  (int t) {Day=d;   check();};
         void setMonth(int m) {Month=m; check();};
         void setYear (int j) {Year=j;  check();};

         int getDay   (void)  {return (Day);};
         int getMonth (void)  {return (Month);};
         int getYear  (void)  {return (Year);};

         int check (void);
   };
#endif
```

Für die Implementation des überladenen Constructors bietet sich nun die folgende Methodendefinition an:

```
//=========================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

datum::datum (int t, int m, int j)
{
```

```
   Day   = t;
   Month = m;
   Year  = j;
   check ();
}
```

Die Deklaration eines Objektes der Klasse *datum* mit Werten, die als Vorbelegung dienen sollen, hat im Programm dann folgendes Aussehen:

```
//============================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

void main (void)
{
   datum x (24, 12, 1996);

   //-----------------------------------------------------------
   // Ausgabe:24.12.1996
   //-----------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;

   x.setYear  (1972);
   x.setMonth (2);
   x.setDay   (31);         // Korrektur nötig!

   //-----------------------------------------------------------
   // Ergebnis:28.2.1900
   //-----------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;
}
```

22.3.4. PARAMETER-INITIALISIERUNG

Auch bei Methoden ist es erlaubt, den einzelnen Parametern Default Werte zuzuordnen, damit diese Parameter ggf. entfallen können, aber trotzdem belegt sind. Doch wie bei den allgemeinen Funktionen gibt es die gleichen Einschränkungen hinsichtlich der Verwendung. So dürfen Vorgabewerte nur von rechts beginnend (in der Parameterliste) vergeben werden und es dürfen keine „Lücken“ entstehen:

```
datum (int t,   int m,   int j=1900);   // erlaubt
datum (int t,   int m=1, int j=1900);   // erlaubt
datum (int t=1, int m=1, int j=1900);   // erlaubt
```

```
datum (int t=1, int m,   int j=1900);   // verboten
datum (int t,   int m=1, int j);        // verboten
datum (int t=1, int m,   int j);        // verboten
```

Zusätzlich muss der Entwickler darauf achten, dass keine Doppeldeutigkeiten entstehen. So ist im obigen Beispiel die Zeile mit drei Vorbelegungen zwar syntaktisch korrekt, führt in der Klasse *datum* aber unter Umständen zu Fehlern beim Linken, weil der Rechner (wie im folgenden Beispiel) nicht entscheiden kann, welcher Constructor gemeint ist. Solche Fehler meldet ein C++-Compiler üblicherweise als *ambiguity error*:

```
#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   // diese Zeile erzeugt einen Linkfehler, wenn die
   // Vorbelegung für datum alle Parameter vorbelegt:

   datum x;
   // URSACHE:
   // Der Rechner kann nicht entscheiden, welcher der
   // beiden folgenden Constructoren hier eingesetzt
   // werden muss:
   // datum ();
   // datum (1,1,1900);
}
```

Beschränkt man sich bei der Vorbelegung auf die letzten beiden Parameter, so hat man anschließend die folgenden, legalen Deklarationsmöglichkeiten für ein Objekt der Klasse *datum* (alternativ kann man natürlich auch den parameterlosen Constructor streichen):

```
//============================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
```

```
   datum x (24, 12, 1996);   // ist 24.12.1996
   datum y (24, 12);         // ist 24.12.1900
   datum z (24);             // ist 24. 1.1900
   datum a;                  // ist  1. 1.1900
}
```

22.3.5. OPERATOR OVERLOADING

Einer der Vorzüge von C++ ist das sogenannte *Operator-Overloading*. Damit ist gemeint, dass man einem Operator (z.B. dem „+“) in beschränktem Umfang eine neue Bedeutung geben kann.

Das Überladen von Operatoren ist beschränkt auf eigene Klassen, d.h. es ist z.B. nicht möglich die Bedeutung des „+“-Operator für die Addition von Integer Werten neu zu definieren.

Es können nur bereits bestehende Operatoren ergänzt werden, neue Operatoren (in der Form neuer Zeichenkombinationen) können nicht definiert werden.

Es sind nur bestimmte (also nicht alle) Operatoren überladbar.

Nicht überladbare Operatoren	
Operator	Bedeutung
.	(Punkt) Zugriff auf Struktur-/Unionkomponente, bzw. Klassenmember/Klassenmethode
*	Dereferenzierung von Zeigern (die Multiplikation ist überladbar)
::	C++ Scopedelimiter
?:	bedingte Bewertung
#	Präprozessorbefehl
##	Präprozessorbefehl
sizeof()	Größenoperator

Tabelle 2: nicht überladbare Operatoren

Alle anderen Operatoren (sogar die Indexklammern „*[]*“) sind durch eigene Methoden ergänzbar. Dabei sollte man immer beachten, dass die Bedeutung der überladenen Operatoren nach Möglichkeit auf den ersten Blick erkennbar bleiben sollte. Es ist zwar möglich, aber keineswegs sinnvoll z.B. den „+“-Operator für die Subtraktion komplexer Zahlen zu definieren.

Eine besondere Stellung nimmt auch der Zuweisungsoperator „=“ ein, denn dieser wird vom System automatisch erzeugt. Hierbei ist jedoch zu beachten, dass bei Pointer auch nur die Adresse kopiert wird.

D.h. bei der Verwendung dynamischer Member ist es auf jeden Fall nötig, den Zuweisungsoperator „von Hand“ zu überladen und die dynamischen Member selbst zu kopieren (siehe dazu auch das Beispiel einer dynamischen String-Klasse im Kapitel über dynamische Speicherverwaltung).

Für die Klasse *datum* soll hier als einfaches Beispiel jeweils ein binärer und ein unärer Operator definiert werden. Als binärer Operator soll die Addition einer Anzahl von Jahren dienen, als unärer Operator das „++“ (Fortschaltung um einen Monat). Die Bedeutung und Ausführung der Operatoren (Addition von Jahren / Monaten) ist willkürlich gewählt und folgt an dieser Stelle keinerlei Konventionen. Tatsächlich wäre eine Addition/Fortschaltung von Tagen sinnvoller, aber als Beispiel ungleich komplexer:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class datum
   {
      protected:
         int Day;
         int Month;
         int Year;
         void check (void);

      public:
         datum  (int t = 1, int m = 1, int j = 1900);
         ~datum (void) {};

         datum operator++ (void); // Prä-Inkrement
         datum operator++ (int);  // Post-Inkrement
         datum operator+  (int y);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};
   };
#endif
```

Die Implementation zu den entsprechenden Operatoren hätte dann das folgende Aussehen:

```
//=========================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

datum datum::operator+ (int y)
{
   datum Result;

   Result.Day   = Day;
   Result.Month = Month;
   Result.Year  = Year + y;
   return (Result);
}
```

Wie man leicht erkennen kann, ist das Ergebnis der Operation ein neues Objekt vom Typ *datum*. Dieses neue Objekt wird lokal erzeugt und somit am Ende der Methode wieder vernichtet. Über den automatisch erzeugten Zuweisungsoperator kann das Ergebnis auf ein beliebiges Objekt von Typ *datum* zugeweisen werden:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x (24, 12, 1996);
   datum y = x + 5;
   x++;

   //---------------------------------------------------------
   // Ausgabe:24.12.2001
   //---------------------------------------------------------
   cout << x.getDay () << "." << x.getMonth () << "."
        << x.getYear () << endl;

   //---------------------------------------------------------
   // Ausgabe: 24.12.2001
   //---------------------------------------------------------
   cout << y.getDay () << "." << y.getMonth () << "."
        << y.getYear () << endl;
}
```

Die Definition der unären Operatoren ist ähnlich, nur dass der Parameter darüber bestimmt, ob es sich um ein Prä- oder Post-Inkrment handelt. Wird kein Operator übergeben (*void*), dann handelt es sich um ein Prä-Inkrement. Zur Unterscheidung wird für das Post-Inkrement formal ein *int*-Parameter definiert, der aber nicht benutzt wird:

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

datum datum::operator++ (void)
{
   Year++;
   return (*this);
}

//------------------------------------------------------------
// beim Post-Inkrement muß der aktuelle Inhalt zurückgegeben
// werden, bevor der Wert erhöht wird. Das eht in dieser
// Reihenfolge natürlich nicht ohne vorher eine Kopie anzu-
// legen
//------------------------------------------------------------
datum datum::operator++ (int)
{
   // Kopie das aktuellen Zustands anlegen
   datum d (Day, Month, Year);
   Year++;     // Jetzt erst Wert erhöhen
   return (d); // Kopie zurückgeben
}
```

Wie die Methodendefinition des Prä-Inkrements zeigt, wird in diesem Fall das Objekt *x* selbst verändert und zurückgegeben. Dies geschieht über den *this*-Pointer des Objektes (s.u.).

22.3.6. AUTOMATISCHE TYPKONVERTIERUNG

Ein Überladen des Additionsoperators mit allen möglichen Datentypen ist natürlich sehr auswendig, denn nach dem oben gezeigten Beispiel ist es zwar möglich *int*-Variablen und Konstanten zu addieren, nicht aber solche vom Typ *double*. Um nicht eine schier unüberschaubare Anzahl von Überladungen schreiben zu müssen, gibt es die automatische Typkonvertierung. Diese wird vom System automatisch durchgeführt, wenn es einen entsprechenden Constructor gibt. Der Sinn dieser Konvertierung erschließt sich in vollem Umfang aber erst bei der Gestaltung eigener Zahlentypen, wie z.B. der komplexen Zahlen. Definiert man dann die Addition, Subtraktion, Division und Multiplikation für zwei komplexe Zahlen, sowie einen Constructor für die Initialisierung einer komplexen Zahl aus einem *int*, *long* und *double*, dann hat man zugleich die

Grundrechenarten für diese Datentypen mit definiert. Der Rechner versucht zunächst einen überladenen Operator (z.B. für eine Division komplexe Zahl / *double*) zu finden. Ist diese nicht explizit definiert, dann wird versucht die *double*-Zahl anhand eines Constructors in eine komplexe Zahl zu überführen und die Division zweier komplexer Zahlen ausgeführt.

22.4. DER THIS-POINTER

Der *this*-Pointer ist eine automatisch vom angelegte *const*-Variable, die in jeder Klasse vorhanden ist.

Genaugenommen ist es eine Adressreferenz, die in jedem Objekt enthalten ist, stets die Adresse des Objektes enthält und die immer den Namen *this* trägt. Der *this*-Zeiger wird vom Compiler automatisch in jeder Methode als erster Parameter übergeben, ohne dass der Programmierer dieses extra ausweisen oder vereinbaren muß. Die Variable ist also im Prinzip in jeder Methode vorhanden und verwendbar.

Der Sinn ist ebenso eindeutig, wie logisch:
Da die Methoden (im Gegensatz zu den Membervariablen und -konstanten) nur ein einziges Mal im Speicher vorhanden sind (alles andere wäre Platzverschwendung), muss der Computer der Methode mitteilen, für welches Objekt (Instanz) sie gerade aufgerufen wird, damit beim Zugriff die richtigen Member verwendet werden. Dies geschieht beim Aufruf (der ja mit einem Objekt zusammen erfolgt) über den *this*-Pointer. Das Objekt übergibt sozusagen immer seine Visitenkarte (Adresse). Das folgende Beispiel gibt die *this*-Konstante des *datum*-Objektes als Wert vom Typ *long* zurück:

```
//=============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class date
   {
      protected:
         int Day;
         int Month;
         int Year;
         void check (void);

      public:
         datum  (int t = 1, int m = 1, int j = 1900);
         ~datum (void) {};
```

```
         datum operator++ (void);
         datum operator+ (int y);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};

         long adress (void) {return ((long)this);};
   };
#endif
```

Der Aufruf der Methode *address* zeigt jetzt die Speicheradressen an:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x;
   datum y;

   cout << x.adress() << endl;
   cout << y.adress() << endl;
}
```

Die Tatsache, dass jeder Methodenaufruf um einen Parameter länger ist – nämlich um genau einen typisierten Pointer auf das aufrufende Objekt – bereitet aber auch Schwierigkeiten, die nur mit Hilfe des *friend*-Konzeptes umgangen werden können.

22.5. FRIEND-METHODEN UND OPERATOREN

Man sollte meinen, dass das oben dargelegte *Operator-Overloading* problemlos die Möglichkeit eröffnet, jetzt beliebig Zahlen zu Datumsangaben zu addieren und umgekehrt. Die Umkehrung ist besonders wichtig für eigene Klassen, die eigene Zahlenformate darstellen, wie z.B. Klassen für komplexe Zahlen oder Matrizen. Während man vielleicht einen Sinn in *Datum + Zahl* finden kann (hier hinzuzählen von Jahren) ist die Umkehrung *Zahl + Datum* weniger sinnvoll.

Andererseits ist nach den mathematischen Gesetzen zu erwarten, dass sowohl *Zahl + Matrix* wie auch *Matrix + Zahl* gültige Ergebnisse liefert:

```
//============================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#include "datum.h"

void main (void)
{
   datum x;
   datum y = x + 7;
   datum z = 7 + x;
}
```

Die Erwartung, dass mit einem *Operator-Overloading* beide Formen abgedeckt sind, wird enttäuscht. Durch die Reihenfolgenänderung auf *7 + x* ist die Addition nun nicht mehr eine Klassenmethode von *datum* sondern von *int* Der linke Operator gibt also die Klassenzugehörigkeit an.

In der Klasse *datum* kann die Methode mit den bisher bekannten Mitteln nicht definiert werden, da die neue Methode für die Addition *int + datum* als ersten Parameter (d.h. als *this*-Pointer) einen Zeiger auf *int* erwartet. Eine *datum*-Methode liefert in *this* aber immer einen Zeiger auf *datum.* Die Implementierung von *int* selbst kann man auch nicht direkt erweitern. Es wäre auf wenig sinnvoll die Erweiterung der *int*-Klasse hinzuzufügen, dies würde ja bedeuten, dass man zukünftig die Klasse *datum* immer als Teil des Systems mit einbinden müsste, auch wenn man *datum* nicht braucht. Dies ist auch nicht nötig, denn es ist nicht Bedingung in C++, dass alle Methoden einer Klasse in deren Klassendeklaration erweitert werden müssen. Zur Erweiterung einer fremden Klasse in einer eigenen dient die *friend*-Deklaration. Voraussetzung ist allerdings, dass es sich bei der zu erweiternden Klasse um eine der in C++ standardmäßig vorhandenen Klasse (*int*, *long* usw.) handelt oder dass man Zugriff auf den Aufbau der Klasse hat. Letzteres ist mit der Einbindung der Headerdatei immer gegeben:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class date
```

```
   {
      protected:
         int Day;
         int Month;
         int Year;
         void check (void);

      public:
         datum  (int t = 1, int m = 1, int j = 1900);
         ~datum (void) {};

         datum operator++ (void);
         datum operator++ (int);
         datum operator+ (int y);
         friend int operator+ (int & a, datum d);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};
   };
#endif
```

Die entsprechende Implementation dazu ist sehr einfach:

```
//=========================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

int operator+ (int & a, datum d)
{
   return (a + d.Year);
}
```

Im Programm kann man nun auch schreiben:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;
```

```
void main (void)
{
   datum x;

   x = x + 10;
   cout << x.getYear() << endl;    // Ergebnis ist 1910
   cout << (10 + x) << endl;       // Ergebnis ist 1920
}
```

22.5.1. ÜBERLADEN DER STREAM-IO

Die gleiche Methodik wird verwendet, um den Streamausgabe-Operator << dergestalt zu überladen, dass man einfach den Objektnamen übergibt, statt (wie bisher) die einzelnen Member umständlich aneinanderreihen zu müssen. Im Headerfile ist lediglich eine Zeile zu ergänzen:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_
   #include <iostream>

   using namespace std;

   class date
   {
      protected:
         int Day;
         int Month;
         int Year;
         void check (void);

      public:
         datum  (int t = 1, int m = 1, int j = 1900);
         ~datum (void) {};

         datum operator++ (void);
         datum operator++ (int);
         datum operator+  (int y);

         friend int operator+       (int & a, datum d);
         friend ostream& operator<< (ostream& s,datum d);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
```

```
          int getYear   (void)  {return (Year);};
   };
#endif
```

Auch die Definition ist recht simpel:

```
//=========================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

ostream & operator<< (ostream & s, datum d)
{
   return (s << d.Day << "." << d.Month << "." << d.Year);
}
```

Der Vereinfachungseffekt bei der Ausgabe eines Datums ist erheblich:

```
//=========================================================
// Teilimplementation des Testprogramms. Die Datei TEST.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//=========================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x (20, 4, 1986);

   cout << x << endl;    // Ergebnis ist 20.4.1986
}
```

Der Operator der Ausgabeaufreihung << erwartet immer eine Referenz auf ein *ostream*-Objekt als zweiten (rechten) Operanden. Der erste (linke) Operand ist ohnehin immer ein *ostream*-Objekt, da die <<-Verknüpfung von links nach rechts ausgewertet wird. Die beiden folgenden Zeilen sind daher inhaltlich identisch:

```
cout << 7 << " " << "Hallo" << endl;
((((cout << 7) << " ") << "Hallo") << endl);
```

Es ist leicht zu erkennen, dass das Ergebnis jeder Ausgabe mit << wieder ein Ausgabestrom sein muss. Weil aber der erste Operand vom

Typ *ostream* sein muss, ist das *Operator-Overloading* Teil der *ostream*-Klasse und nicht Teil der selbstdefinierten Klasse *datum.*

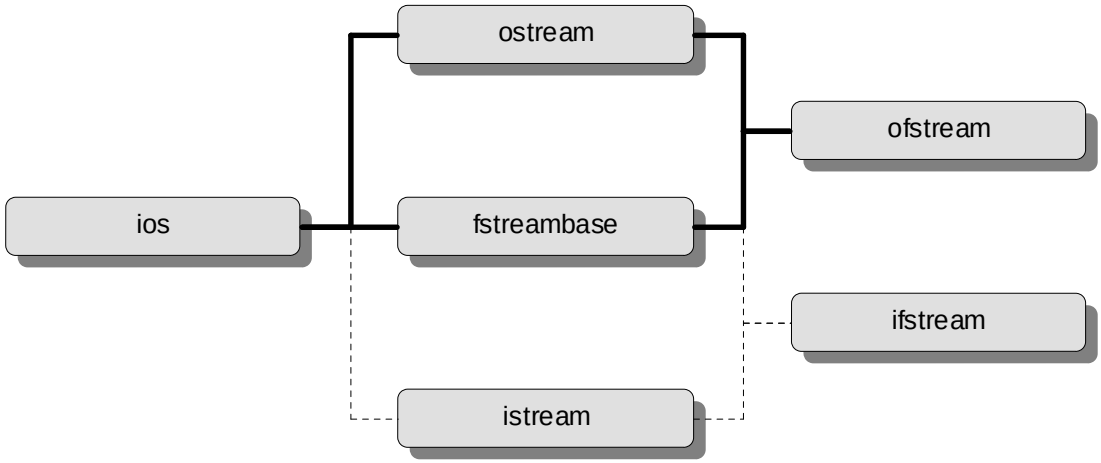


Abbildung 22-1 – Hierarchiebaum der IO-Stream-Klassen

Die Deklaration als *friend* ermöglicht es dem *ostream*-Operator sogar auf die *private*-Variablen der Klasse *datum* zuzugreifen, was die Implementation der Ausgabemethode erheblich vereinfacht. Leider werden *friend*-Deklarationen aber aus genau diesem Grunde niemals vererbt, denn mit der Vererbung entfällt ja der Zugriff auf *private*-Variablen (siehe Abschnitt Vererbung) und somit würden ggf. auch die Zugriffe der *friend*-Methode illegal werden.

22.5.2. EIGENE MANIPULATOREN

Ein Manipulator ist nicht mehr, als eine Methode, die innerhalb eines Streams aufgerufen wird und diesen in seinen Eigenschaften (ggf. nur temporär) verändert.

Mittels eines eigenen Manipulators kann nun auch endlich das Problem der Formatierung von Fließkommazahlen (mit genauer Angabe der Anzahl von Vor- und Nachkommastellen) gelöst werden:

Achtung:
Die Implementation des LFOUT-Manipulators kann abhängig vom verwendeten Compiler leicht unterschiedlich ausfallen.

```
//============================================================
// Headerdatei LFOUT.H
//============================================================

#ifndef _LFOUT_H_
#define _LFOUT_H_

#include <iostream>
#include <float.h>
```

```
using namespace std;

//-------------------------------------------------------------
// Manipulationsklasse für LONG DOUBLE – Variable
// Da es ein automatisches CASTING (Überführen) von FLOAT und
// DOUBLE nach LONG DOUBLE gibt, funktioniert diese Klasse
// automatisch auch mit FLOAT und DOUBLE-Variablen
//-------------------------------------------------------------
class lfout
{
   //-------------------------------------------------------
   // Diese Variablen sind protected und können außerhalb der
   // Klasse nicht direkt angesprochen werden
   //-------------------------------------------------------
   protected:
      long double llf;  // zur Formatierung übergebener Wert
      int NK;           // Anzahl der Nachkommastellen
      int VK;           // Anzahl der Vorkommastellen

   //-------------------------------------------------------
   // Diese Methoden (Funktionen) werden von außerhalb
   // verwendet, um die Formatierung aufzurufen. Da die lfout
   // (...) Methoden sehr klein sind, werden sie hier als
   // INLINE-Methoden (Makros) definiert.
   //-------------------------------------------------------
   public:

      //-------------------------------------------------------
      // normale Formatierung, tut nichts - nur der
      // Vollständigkeit halber -1 hat Bedeutung von "NICHT
      // GESETZT", da NULL ein gültiger Wert ist
      //-------------------------------------------------------
      lfout (long double lf)
            { set (lf, -1,  -1);   }

      //-------------------------------------------------------
      // nur Nachkommastellen angegeben
      //-------------------------------------------------------
      lfout (long double lf, int prec)
            { set (lf, -1, prec); }

      //-------------------------------------------------------
      // Vor- und Nachkommastellen angegeben
      //-------------------------------------------------------
      lfout (long double lf, int vks, int prec)
            { set (lf, vks, prec); }

      //-------------------------------------------------------
      // Überladen des OSTREAM-Operators << für Klasse LFOUT
      //-------------------------------------------------------
      friend ostream& operator << (ostream& out, lfout& f);

   //-------------------------------------------------------
   // Diese Methode kann außerhalb der Klasse nicht direkt
   // angesprochen werden -
```

```
   //-------------------------------------------------------
   protected:

      //-------------------------------------------------------
      // Setzen der Klassenvariablen
      //-------------------------------------------------------
      void set (long double lf, int vks, int prec);
};
#endif
```

```
//============================================================
// Programm LFOUT.CPP
//============================================================

#include <iostream>
#include <iomanip>
#include <math.h>
#include "lfout.h"

using namespace std;

//------------------------------------------------------------
// Setze Member gemäß der an Klasse LFOUT übergebenen Werte
//------------------------------------------------------------
void lfout::set (long double lf, int vks, int prec)
{
   //-------------------------------------------------------
   // Setzen der LONG DOUBLE-Variablen auf den im Parameter
   // übergebenen Wert
   //-------------------------------------------------------
   llf = lf;

   //-------------------------------------------------------
   // Setzen Nachk.-Variable, Korrektur illegaler Werte
   // (kleiner -1).
   // -1 bedeutet Standardwert (d.h. 6 Nachkommastellen)
   // 0 bedeutet keine Nachkommast. ist daher gültiger Wert
   // Da damit die Anzahl der Nachkommastellen feststeht,
   // kann diese in NK gespeichert werden
   //-------------------------------------------------------
   if (prec < 0)
   {
      prec = -1;
   }

   NK = prec;

   //-------------------------------------------------------
   // Setze Vork.-Variable, Korrektur illegaler Werte
   // (kleiner als -1)
   // -1 bedeutet Standard (d.h. benötigte Anz. Vork.stellen)
   // die Anzahl der Vorkommastellen ist nur dann von
   // Interesse, wenn die Anzahl der benötigten Stellen
   // KLEINER ist als die Anzahl der gewünschten Stellen.
   //-------------------------------------------------------
```

```
   if (vks  < 0)
   {
      vks = -1;
   }

   //-------------------------------------------------------
   // Wenn Nachkommastellen gewünscht und die angegebene Anz.
   // GRÖSSER als die GESAMTZAHL der gewünschten Stellen also
   // z.B. lfout(x,3,9) und die Gesamtanzahl der Stellen wird
   // übergeben (ungleich -1). Dann immer benötigte Anzahl an
   // Vorkommastellen ausgeben (wie -1)
   //-------------------------------------------------------
   if (prec > 0)
   {
      if ((prec >= vks) && (vks != -1))
      {
         vks = -1;
      }
   }

   //-------------------------------------------------------
   // Wenn Nachkommastellen gewünscht sind und die angegebene
   // Gesamtanzahl an Stellen wurde übergeben (ungleich -1),
   // dann ist die Anzahl der Vorkommastellen zu berechnen.
   // Sonst entspricht die Anzahl der Vorkommastellen der
   // Anzahl der Gesamtstellen
   //-------------------------------------------------------
   if ((prec > 0) && (vks != -1))
   {
      VK = vks - prec - 1;
   }
   else
   {
      VK = vks;
   }
}

//-------------------------------------------------------------
// Formatierte Ausgabe eines LFOUT-Objekts in einem OSTREAM
//-------------------------------------------------------------
ostream& operator << (ostream& out, lfout& f)
{
   long double rundung = 0.0;

   //-------------------------------------------------------
   // Wert der Vorkommastellen werden durch einfaches
   //  Abschneiden der Nachkommastellen gebildet
   //-------------------------------------------------------
   long double tmpvk = floorl (f.llf);

   //-------------------------------------------------------
   // Nachkommastellen erstmal auf Null setzen
   //-------------------------------------------------------
   long double tmpnk = 0.0;

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

```
   // Keine Exponentialausgaben zulassen! Dazu muss ein Flag
   // gesetzt werden, der vorherige Wert des Flags wird
   // zwischengespeichert und vor Funktionsende restauriert
   //-------------------------------------------------------
   long oldflag = out.setf(ios::fixed, ios::floatfield);

   //-------------------------------------------------------
   // Gleiches gilt für die Ausgabepräzision, die hier auf
   // Null (keine Nachkommastellen) gesetzt wird, die
   // Nachkommastellen machen wir selbst.
   //-------------------------------------------------------
   int  oldprec = out.precision(0);

   //-------------------------------------------------------
   // Wenn definierte Anzahl von Nachkommastellen gewünscht
   // ist, dann diese in Vorkommastellen einer anderen
   // Variablen umwandeln
   // Die Addition von +0.1 vermeidet Rundungsfehler bei der
   // Subtraktion ist der Wert für Nachkommastellen gleich
   // -1, dann 6 Stellen ausgeben
   //-------------------------------------------------------
   if (f.NK != -1)
   {
      tmpnk = ((f.llf - tmpvk) * (pow10 (f.NK)));
   }
   else
   {
      tmpnk = ((f.llf - tmpvk) * (pow10 (6)));
   }

   //-------------------------------------------------------
   // jetzt gemäß der alten printf-Regeln runden. Dazu den
   // nicht mehr anzuzeigenden Restanteil der Nachkomma-
   // stellen herauslösen. Hier entspricht dies den Nachkomma
   // stellen der Floatingpointzahl, die die Nachkommastellen
   // repräsentiert. Ein eventuell entstehender Übertrag ist
   // ggf. in Vor- oder Nachkommazahl zu übertragen
   //-------------------------------------------------------
   rundung = tmpnk - floor (tmpnk);
   tmpnk = floor (tmpnk);

   if (rundung >= 0.5)
   {
      tmpnk += 1.0;
   }

   if (tmpnk / pow10(f.NK) == 1.0)
   {
      tmpvk += 1.0;
      tmpnk  = 0.0;
   }

   //-------------------------------------------------------
   // gibt es weder für Vor- noch für Nachkommastellen eine
   // Vorgabe, dann die Flags wiederherstellen und ganz
   // normal ausgeben
```

```
   //-------------------------------------------------------
   if ((f.NK == -1) && (f.VK == -1))
   {
      out.precision (oldprec);
      out.setf (oldflag, ios::floatfield);
      out << f.llf;
      return (out);
   }

   //-------------------------------------------------------
   // Ausgabe der Vorkommastellen gemäß Anzahl der
   // Stellen oder die benötigten Stellen (ELSE-Fall)
   //-------------------------------------------------------
   if ((f.VK > 0) && (f.NK != -1))
   {
      out << setw (f.VK) << tmpvk;
   }
   else
   {
      out << tmpvk;
   }

   //-------------------------------------------------------
   // Ausgabe der Nachkommastellen gemäß Anzahl der
   // gewünschten Stellen oder die benötigten Stellen (ELSE-
   // Fall). Dazu muss man mit Nullen auffüllen!
   //-------------------------------------------------------
   if (f.NK > 0)
   {
      char fillbuff = out.fill ('0');
      out << "." << setw(f.NK) << tmpnk;
      out.fill (fillbuff);
   }
   if (f.NK < 0)
   {
      out << "." << tmpnk;
   }

   //-------------------------------------------------------
   // Flags wiederherstellen
   //-------------------------------------------------------
   out.precision (oldprec);
   out.setf (oldflag, ios::floatfield);

   return (out);
}
```

```
//=========================================================
// Headerdatei LFOUTTST.CPP
//=========================================================

#include <iostream>
#include <iomanip>
#include <float.h>
#include "lfout.h"
```

```
#include <conio.h>
#include <stdio.h>
#include <fstream>

using namespace std;

void main (void)
{
   ofstream fout ("c:\\lfouttst.txt");

   cout << "Ergebnisse mit cout und lfout" << endl;
   cout << lfout (9.9999,     15, 3) << endl;
   cout << lfout (9.9999,     15, 5) << endl;
   cout << lfout (3.004,      15, 2) << endl;
   cout << lfout (1.00009999, 15, 4) << endl;
   cout << lfout (0.0,        15, 2) << endl;

   //-------------------------------------------------------
   // Da die objektorientierte Datei-Ausgabeklasse von
   // OSTREAM erbt, funktioniert der Manipulator LFOUT auch
   // mit Objekten der Klasse OFSTREAM
   //-------------------------------------------------------
   f << lfout (9.9999,     15, 3) << endl;
   f << lfout (9.9999,     15, 5) << endl;
   f << lfout (3.004,      15, 2) << endl;
   f << lfout (1.00009999, 15, 4) << endl;
   f << lfout (0.0,        15, 2) << endl;

   cout << "Vergleichsergebnisse mit printf()" << endl;
   printf ("%15.3lf\n", 9.9999);
   printf ("%15.5lf\n", 9.9999);
   printf ("%15.2lf\n", 3.004);
   printf ("%15.4lf\n", 1.00009999);
   printf ("%15.2lf\n", 0.0);
}
```

Der Manipulator muss lediglich für die Basisklasse der Stream-Ausgabe definiert werden und kann dann, wie im Testprogramm gezeigt, auch in den anderen Ausgabe-Streams benutzt werden. Der folgende Ausschnitt aus dem Hierarchiebaum verdeutlicht die Vererbung.

Der Vererbungsprozess in C++ wird im nachfolgenden Kapitel ausführlich erklärt.

22.6. STATIC MEMBER

Wie man die Eigenschaften einer Klasse festhalten kann, sollte an den vorangegangenen Beispielen deutlich geworden sein – zumeist bedient man sich einfacher Bitfelder, die als Flags dienen (wie in den Streamklassen *ostream* und *istream*). Ist nur eine speicherwürdige Eigenschaft vorhanden, so tut es auch ein einfacher *int*. Die Eigenschaften sind (wie alle Membervariablen) zumeist vom

Sichtbarkeitstyp *protected* oder *private* und werden über Methoden gesetzt.

Für die Beispielklasse *datum* wäre z.B. eine Eigenschaft nützlich, welche die Ausgabe formatiert und bestimmt ob ein Datum in der Form „1.1.1960“ oder „01.01.1960“ ausgegeben wird:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_
   #include <iostream>

   using namespace std;

   class date
   {
      protected:
         int Day;
         int Month;
         int Year;
         int Formated;
         void check (void);

      public:
         datum  (int t = 1, int m = 1, int j = 1900);
         ~datum (void) {};

         datum operator++ (void);
         datum operator+ (int y);
         friend int operator+ (int & a, datum d);
         friend ostream& operator<< (ostream & s, datum d);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};
   };
#endif
```

Die neue Membervariable *Formated* übernimmt genau diese Aufgabe. Wenn sie gesetzt ist (also ungleich Null ist), soll die Ausgabe formatiert, d.h. immer zehnstellig erfolgen. Damit *Formated* immer einen gültigen Wert hat (Default-Wert soll Null sein), muss die Membervariable in allen Constructoren gesetzt werden:

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

datum::datum (int t, int m, int j)
{
   Day   = t;
   Month = m;
   Year  = j;
   Formated = 0;
   check ();
}
```

An dieser Stelle wäre zu überlegen, ob man die Parameterleiste des Constructors um die Angabe von Formated erweitert und diesen Parameter mit Null vorbelegt:

datum (int t = 1, int m = 1, int j = 1900, int f = 0);

Dies gäbe dem Entwickler die Möglichkeit gleich bei der Deklaration festzulegen, ob eine Variable formatiert oder unformatiert ausgegeben werden soll.

Zu ergänzen sind jetzt noch die Zugriffsmethoden auf die neue Membervariable:

```
//============================================================
// Teilimplementation des Programmheaders. Die Datei DATUM.H
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_
   #include <iostream>

   using namespace std;

   class date
   {
      protected:
         int Day;
         int Month;
         int Year;
         int Formated;
         static int Text;
         void check (void);
```

```
      public:
         datum  (int t=1,int m=1,int j=1900, int f=0);
         ~datum (void) {};

         datum operator++ (void);
         datum operator+ (int y);
         friend int operator+ (int & a, datum d);
         friend ostream& operator<< (ostream & s, datum d);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};
         void setForm  (int f) {Formated=f;};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};
         int getForm   (void)  {return (Formated);};
   };
#endif
```

Die Ausgabemethode erhält dann die folgende Form:

```
//============================================================
// ÄNDERUNG
// Teilimplementation der Klasse Datum. Die Datei DATUM.CPP
// enthält die vollständige Implementation gemäß des letzten
// in diesem Kapitel gezeigten Standes.
//============================================================

ostream & operator<< (ostream & s, datum d)
{
   int oldfill;
   if (!d.Formated)
      return (s << d.Day << "." << d.Month << "." << d.Year);

   oldfill = s.fill ('0');
   s << setw(2) << d.Day << "." << setw (2) << d.Month
     << "." << d.Year;
   s.fill (oldfill);
   return (s);
}
```

Jetzt kann für jedes einzelne Objekt festgelegt werden, welche Eigenschaft (formatierte oder unformatierte Ausgabe) es besitzen soll.

Allerdings trifft man häufig auch auf eine etwas andere Notwendigkeit, nämlich bestimmte *Eigenschaften* global festzulegen. Wünschenswert wäre z.B. dass alle Warntexte eines Systems in den gleichen Farben ausgegeben werden. Dazu ist es notwendig, dass sich alle Objekte die Eigenschaftsvariable teilen, d.h. alle Objekte greifen zur Ermittlung der Eigenschaft auf die gleiche Speicherstelle zu.

Denkbar für die Klasse *datum* wäre eine globale Einstellung, ob der Monat in der textuellen Form („Jan“, „Feb“ etc.) oder als Zahl ausgegeben werden soll.

Um ein und dieselbe Klassenvariable allen Instanzen dieser Klasse zugänglich zu machen, muss diese als *static* deklariert werden:

```
//============================================================
// Headerdatei DATUM.H
//============================================================

#ifndef _DATUM_H_
   #define _DATUM_H_
   #include <iostream>

   using namespace std;

   class date
   {
      protected:
         int Day;
         int Month;
         int Year;
         int Formated;
         static int Text;
         void check (void);

      public:
         datum  (int t=1,int m=1,int j=1900,int f=0);
         ~datum (void) {};

         datum operator++ (void);
         datum operator++ (int);
         datum operator+ (int y);
         friend int operator+ (int & a, datum d);
         friend ostream& operator<< (ostream & s, datum d);

         void setDay   (int t) {Day=t;   check();};
         void setMonth (int m) {Month=m; check();};
         void setYear  (int j) {Year=j;  check();};
         void setForm  (int f) {Formated=f;};

         int getDay    (void)  {return (Day);};
         int getMonth  (void)  {return (Month);};
         int getYear   (void)  {return (Year);};
         int getForm   (void)  {return (Formated);};
   };
#endif
```

Eine Membervariable vom Typ *static* darf natürlich nicht innerhalb eines Constructors initialisiert werden, da sonst der in diesem Member gespeicherte Wert bei jeder Instanzendeklaration überschrieben wird. Stattdessen muss eine solche Variable außerhalb, möglichst in der

Klassendefinitionsdatei, gesetzt werden. Für das Beispiel *datum* wird daher (außerhalb aller Methoden) in *datum.cpp* ergänzt:

```
int datum::Text = 0;
```

Der Scopedelimiter *datum::* ist unbedingt notwendig, da es auch global oder in anderen Klassen eine Variable mit dem Namen *Text* geben könnte und der Compiler wissen muss, dass die Variable *Text* aus der Klasse *datum* gemeint ist. Zugleich ist so sichergestellt, dass die Membervariable nur ein einziges Mal, bei Programmstart, gesetzt wird. Die Implementation der Ausgabe ändert sich jetzt in:

```
//============================================================
// ÄNDERUNG Implementation der Klasse Datum (DATUM.CPP)
//============================================================

ostream & operator<< (ostream & s, datum d)
{
   int oldfill;
   char Mon [4] = "";

   if (d.Text)
      switch (d.Month)
      {
         case 1 : strcpy (Mon, "Jan"); break;
         case 2 : strcpy (Mon, "Feb"); break;
         case 3 : strcpy (Mon, "Mär"); break;
         case 4 : strcpy (Mon, "Apr"); break;
         case 5 : strcpy (Mon, "Mai"); break;
         case 6 : strcpy (Mon, "Jun"); break;
         case 7 : strcpy (Mon, "Jul"); break;
         case 8 : strcpy (Mon, "Aug"); break;
         case 9 : strcpy (Mon, "Sep"); break;
         case 10: strcpy (Mon, "Okt"); break;
         case 11: strcpy (Mon, "Nov"); break;
         case 12: strcpy (Mon, "Dez"); break;
      }

   if ((!d.Formated) && (!d.Text))
      return (s << d.Day << "." << d.Month << "." << d.Year);

   if ((!d.Formated) && (d.Text))
      return (s << d.Day << "." << Mon << "." << d.Year);

   //-----------------------------------------------------------
   // ab hier die formatierte Ausgabe
   //-----------------------------------------------------------
   oldfill = s.fill ('0');

   if (!d.text)
      s << setw(2) << d.Day << "." << setw (2) << d.Month
        << "." << d.Year;
   else
```

```
      s << setw(2) << d.Day << "." << Mon << "." << d.Year;

   s.fill (oldfill);
   return (s);
}
```

Wird jetzt die Eigenschaft *Text* gesetzt, so gilt dies für die Ausgabe aller *datum*-Objekte.

Hier die zusammengefasste, vollständige Implementation der Klasse Datum:

```
//============================================================
// Implementation der Klasse Datum (DATUM.CPP)
//============================================================

#include "datum.h"
#include <iomanip>

//------------------------------------------------------------
// Constructor
//------------------------------------------------------------
datum::datum (int t, int m, int j, int f)
{
   Day   = t;
   Month = m;
   Year  = j;
   Formated = f;
   check ();
}

//------------------------------------------------------------
// Operator: Addition mit einem Integer
//------------------------------------------------------------
datum datum::operator+ (int y)
{
   datum Result;

   Result.Day   = Day;
   Result.Month = Month;
   Result.Year  = Year + y;
   return (Result);
}

//------------------------------------------------------------
// Operator: Post-Inkrement
//------------------------------------------------------------
datum datum::operator++ (void)
{
   Year++;
   return (*this);
}

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

```
// Operator: Prä-Increment
//------------------------------------------------------------
datum datum::operator++ (void)
{
   Datum d (Day, Month, Year);
   Year++;
   return (d);
}

//------------------------------------------------------------
// Friend Operator: Addition mit einem Integer
//------------------------------------------------------------
int operator+ (int & a, datum d)
{
   return (a + d.Year);
}

//------------------------------------------------------------
// Friend Operator: Stream-Output
//------------------------------------------------------------
ostream & operator<< (ostream & s, datum d)
{
   int oldfill;
   if (!d.Formated)
      return (s << d.Day << "." << d.Month << "." << d.Year);

   oldfill = s.fill ('0');
   s << setw(2) << d.Day << "." << setw (2) << d.Month
     << "." << d.Year;
   s.fill (oldfill);
   return (s);
}

//------------------------------------------------------------
// Datum prüfen
//------------------------------------------------------------
void datum::check (void)
{
   if (Month > 12) Month = 12;   // max. Dezember
   if (Month < 1)  Month = 1;    // min. Januar
   if (Year > 3000) Year = 3000; // max. bis Jahr 3000
   if (Year < 1800) Year = 1800; // min. Jahr 1800
   if (Day < 1) Day = 1;         // min. Monatserster
   switch (Month)
   {
      case 2 : // Februar - es gilt immer:
               if (Day > 29) Day = 29;
               // für Nicht-Schaltjahre gilt immer:
               if (Year % 4) { if (Day > 28) Day= 28;}
               else
                  // durch 100 teilb.? kein Schaltj.
                  // durch 400 teilbar? doch nicht
                  if ((!(Year % 100))&&(Year % 400))
                     if (Day > 28) Day = 28;
               break;
      case 4 :
```

```
      case 6 :
      case 9 :
      case 11: if (Day > 30) Day = 30;
               break;
      default: if (Day > 31) Day = 31;
               break;
   }
}
```

```
//============================================================
// Implementation des Testprogramms TEST.CPP
//============================================================

#include <iostream>
#include <iomanip>
#include "datum.h"

using namespace std;

void main (void)
{
   datum x (20, 4, 1986);
   datum y = x + 5;
   datum z;

   cout << x << endl;
   cout << y << endl;
   cout << z << endl;

   x.setYear  (1900);
   x.setMonth (2);
   x.setDay   (31);          // Korrektur nötig !
   cout << x << endl;

   y = x + 7;
   z = 7 + x;
   x++;
   cout << x << endl;
   cout << y << endl;
   cout << z << endl;
}
```

23. VERERBUNG, ABGELEITETE KLASSEN

Einer der zentralen Ansätze der objektorientierten Programmierung ist die sogenannte Vererbung (engl. *Inheritance*). Hinter diesem, auf den ersten Blick etwas verwirrenden Begriff verbirgt sich im Grunde genommen wenig mehr als eine Erweiterung der ursprünglichen Klasse durch zusätzliche Daten.

23.1. ABLEITUNG

Die Vererbung (Klassenableitung) ist vergleichbar mit der Erweiterung von Strukturen. Die folgenden Zeilen zeigen eine sehr einfache Personendefinition in ANSI-C:

```
struct Person {char Name [40]; char Vname [40];};
struct Kunde  {Person P; int KdNr;};
```

Wie man sieht, dient die Struktur *Person* als Grundlage für die Struktur *Kunde*. Überträgt man das gleiche Verfahren auf die Klassen in C++, so erhält man den Mechanismus der Vererbung (hier gleich erweitert um die entsprechenden *Inline*-Zugriffsmethoden):

```
//============================================================
// Headerdatei PERSON.H
//============================================================

//------------------------------------------------------------
// BASECLASS = BASISKLASSE = VATERKLASSE
//------------------------------------------------------------

#ifndef _PERSON_H_
   #define _PERSON_H_

   #include <string.h>

   class Person
   {
      private:
         int Zustand;
         int checkPerson (void);

      protected:
         char Name  [40];
         char VName [40];

      public:
         char *getName  (void)    {return (Name);};
         char *getVName (void)    {return (VName);};
         void setName   (char *t) {strcpy (Name, t);};
         void setVName  (char *t) {strcpy (VName, t);};
   };
#endif
```

```
//============================================================
// Headerdatei KUNDE.H
//============================================================

//------------------------------------------------------------
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//------------------------------------------------------------

#ifndef _KUNDE_H_
   #define _KUNDE_H_

   #include "person.h"      // BASISKLASSE EINBEZIEHEN

   class Kunde : public Person // ANGABE DER BASECLASS
   {
      private:
         int checkKunde (void);

      protected:
         int KdNr;

      public:
         int getKdNr (void) {return (KdNr);};
         int setKdNr (int n) {KdNr = n; return (KdNr);};
};
#endif
```

Die Klasse *Kunde* erbt von der Klasse *Person*, d.h. alle *public* und *protected*-Member von *Person* sind auch in *Kunde* enthalten, ohne dass sie erneut aufgeführt werden müssen. Das gleiche gilt auch für alle Methoden, die in der Klasse *Person* als *public* oder *protected* vereinbart wurden.

Abbildung 23-1 – Einfache Vererbung

D.h. die abgeleitete Klasse *Kunde* besteht aus den Membervariablen *Name*, *Vorname* und *KdNr*, sowie den Methoden *getName*, *setName*, *getVName*, *setVName*, *getKdNr* und *setKdNr*. Nicht enthalten sind die Member/Methoden *Zustand* und *checkPerson*.

Neben den Membern und Methoden werden auch Operatoren an die abgeleitete Klasse vererbt. Die einzige Ausnahme stellt dabei – logischerweise – der Zuweisungsoperator dar, der nicht mit vererbt wird.

```
//=============================================================
// Testprogramm
//=============================================================

#include "person.h"

void main (void)
{
   Person a;

   a.setName ("Kohl");   // Ok
   a.setVName("Helmut"); // Ok
   a.setKdNr(10);        // Fehler! KdNr kein Teil von Person
}
```

```
//=============================================================
// Programm KDTEST1.CPP
//=============================================================

#include "kunde.h"

void main (void)
{
   Kunde b;

   b.setName  ("Kohl");         // Ok, von Person geerbt
   b.setVName ("Hannelore");    // Ok, von Person geerbt
   b.setKdNr  (10);             // Ok
}
```

Als „nicht zugreifbar“ vererbt werden also *private*-Member und -Methoden bzw. *friend*-Methoden. Dies ermöglicht es allgemeine Beschreibungen für Objekte zu finden und zu vererben. Besonderheiten einer Klasse hingegen, die sich nicht weitervererben lassen, können über *private*-Member und -Methoden hinzugefügt werden, ohne dass die Verallgemeinerung dadurch zerstört wird.

Tatsächlich beinhaltet die abgeleitete Klasse die Basisklasse (genau wie bei einer Strukturerweiterung). So wird z.B. auch automatisch der Constructor der Basisklasse aufgerufen, bevor der Constructor der Tochterklasse ausgeführt wird. dadurch ist sichergestellt, dass die abgeleitete Klasse die Voreinstellungen der Vaterklasse bei Bedarf überschreiben kann. Beim Destructor ist es logischerweise umgekehrt, hier wird der Destructor der Basisklasse zuletzt ausgeführt. Erwartet der Constructor der Basisklasse irgendwelche Parameter, so müssen diese in der abgeleiteten Klasse angegeben werden. Dadurch kann bei mehreren Constructoren (in der Basisklasse) auch die gewünschte Constructor-Methode ausgewählt werden.

```
//=============================================================
// Header der Klasse Father (FATHER.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//=============================================================

#ifndef _FATHER_H_
   #define _FATHER_H_

   class Father
   {
      protected:
         int fathervar;

      public:
         Father (int param = 1) {fathervar = param;};
   };
#endif
```

```
//=============================================================
// Header der Klasse Daughter (DAUGHTER.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//=============================================================

#ifndef _DAUGHTER_H_
   #define _DAUGHTER_H_

   #include "father.h"            // BASISKLASSE EINBEZIEHEN

   class Daughter : public Father // ANGABE BASECLASS
   {
      protected:
         int daughtervar;

      public:
         Daughter (int para = 2);
   };
#endif
```

```
//=============================================================
// AUSSCHNITT AUS DAUGHTER.CPP
// Implementation des Contructors der Klasse Daughter, mit
// Aufruf des Constructors der Basisklasse
//=============================================================
#include "daughter.h"

Daughter::Daughter (int para) : Father (0)
{
   daughtervar = para;
}
```

23.1.1. PRIVATE UND PUBLIC-ABLEITUNG

In C++ werden zwei Arten der Vererbung unterschieden, die *public* und die *pivate*-Ableitung (eine *protected*-Ableitung gibt es nicht). Der Unterschied beider Klassen besteht in der Auswirkung auf den Sichtbarkeitstyp der abgeleiteten Member und Methoden.

Vererbungsformen		
Basisklasse	abgeleitete Klasse	
Ausgangs-Sichtbarkeit der Member / Methoden	Sichtbarkeit bei der public Ableitung	Sichtbarkeit bei der private Ableitung
public	public	private
protected	protected	private
Private	verborgen	

Tabelle 3: Vererbungsformen

Die *private*-Ableitung ist also fast zwangsläufig die letzte Ableitung einer Kette, da diese Vererbung alle noch vorhandenen *public* und *protected* der Basisklasse in den Sichtbarkeitstyp *private* ändert.

23.1.2. GEZIELTE ABLEITUNG

Lässt sich die gewünschte Ableitung nicht durch eine generelle *public* oder *private*-Vererbung erzielen, so kann man die einzelnen Sichtbarkeitsableitungen auch von Hand für jede Membervariable oder jede Methode bestimmen:

```
//=========================================================
// Header der Klasse Eins (EINS.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//=========================================================

#ifndef _EINS_H_
   #define _EINS_H_

   class Eins
   {
      private:
         int a

      protected:
         int b;
         int c;

      public:
         int d;
         int e;
   };
#endif
```

```
//=========================================================
// Header der Klasse Zwei (ZWEI.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//=========================================================

#ifndef _ZWEI_H_
   #define _ZWEI_H_

   #include "eins.h"        // BASISKLASSE EINBEZIEHEN

   class Zwei : private Eins    // ANGABE DER BASECLASS
   {
      private:
         int f;

      protected:
         int Eins::d;   // eigentlich private

      public:
         int Eins::e;   // eigentlich private
   };
#endif
```

Auch mit diesem Mechanismus ist es allerdings nicht möglich die Membervariable *a* (Typ *private*) an die Klasse *Zwei* zu vererben. Da die Variable aber natürlich Teil der Klassenstruktur bleibt, kann sie indirket, d.h. über ihre getter- bzw. Setter-Methode weiterhin verwendet werden. Es ist also nur die direkte Verwendung und Veränderung nicht mehr erlaubt.

23.2. VIRTUELLE METHODEN

Im Personen/Kundenbeispiel oben wird eine Methode *checkPerson* und eine Methode *checkKunde* definiert, die besagen soll, ob alle Daten der Klasse gefüllt wurden. Diese Art der Programmierung hat einen Nachteil, denn der Entwickler muss genau wissen, welchen Typ er vor sich hat. Eine Verallgemeinerung der Methode *check*, genau zugeschnitten auf die jeweils abgeleitete Klasse, aber aufrufbar über den allgemeinen Zeiger der Basisklasse würde eine erhebliche Vereinfachung darstellen. Dieser Mechanismus wird in C++ mit den sogenannten „*virtuellen Methoden*“ zur Verfügung gestellt:

```
//=========================================================
// Header der Klasse Person (PERSON.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//=========================================================

#ifndef _PERSON2_H_
   #define _PERSON2_H_

   class Person
   {
```

```
      private:
         int Zustand;

      protected:
         char Name  [40];
         char VName [40];

      public:
         virtual int check (void);
         char *getName  (void)    {return (Name);};
         char *getVName (void)    {return (Vname);};
         void setName   (char *t) {strcpy (Name, t);};
         void setVName  (char *t) {strcpy (VName, t);};
   };
#endif
```

```
//============================================================
// Header der Klasse Person (KUNDE.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _KUNDE2_H_
   #define _KUNDE2_H_

   #include "person2.h"        // BASISKLASSE EINBEZIEHEN

   class Kunde : public Person    // ANGABE DER BASECLASS
   {
      private:
         int checkKunde (void);

      protected:
         int KdNr;

      public:
         int check (void);
         int getKdNr (void)  {return (KdNr);};
         int setKdNr (int n) {KdNr = n; return (KdNr);};
   };
#endif
```

Die Implementationen der Methode *check* könnte jeweils wie folgt aussehen:

```
//============================================================
// AUSSCHNITT AUS PERSON2.CPP
// Implementation der Methode CHECK der Klasse Person
//============================================================

#include "person2.h"

int Person::check (void)
{
```

```
   if ((strlen (Name) > 0) && (strlen (Vname) > 0))
      return (1);
   return (0);
}
```

```
//============================================================
// AUSSCHNITT AUS KUNDE2.CPP
// Implementation der Methode CHECK der Klasse Kunde
//============================================================

#include "kunde2.h"

int Kunde::check (void)
{
   if ((strlen (Name)>0) && (strlen (Vname)>0) && (KdNr>0))
      return (1);
   return (0);
}
```

Beide *check*-Methoden dienen dazu, festzustellen, ob alle relevanten Daten der jeweiligen Klasse eingegeben worden sind. Durch die *virtual*-Deklaration der *check*-Methode in der Klasse *Person* darf diese in der abgeleiteten Klasse *Kunde* durch eine geeignetere Methode ersetzt werden. Wird die Methode in der abgeleiteten Klasse nicht ersetzt, so wird im Bedarfsfall natürlich die Methode der Basisklasse aufgerufen – virtuelle Methoden zu ersetzen ist demnach also eine Möglichkeit, keine Pflicht.

Der folgende Zugriff macht den Vorteil von virtuellen Methoden etwas deutlicher:

```
//============================================================
// Testprogramm
//============================================================

#include "person2.h"
#include "kunde2.h"

void main (void)
{
   Person p;          // Objekt vom Typ Person
   Kunde  k;          // Objekt vom Typ Kunde
   Person *pptr;      // Zeiger auf Objekt vom Typ Person

   pptr = &p;         // Zeiger auf Personenobjekt p
   pptr->check();     // Person::check() aufrufen

   pptr = &k;         // geht weil jeder Kunde auch eine
                      // Person ist (vererbt)
   pptr->check ();    // Aufruf von Kunde::check(), die
                      // virt. Methode Person::check()
```

```
                    // wird von Kunde::check verdeckt.
}
```

Ein klassisches Beispiel für die Verwendung von virtuellen Methoden sind z.B. objektorientierte Zeichenprogramme. Diese setzen Zeichnungen aus einer Reihe von Grundobjekten zusammen (Kreis, Quadrat usw.). Das Zeichnen dieser Elemente erfordert immer die gleichen Grundangaben (z.B. Angabe der linken, oberen Ecke der Bounding-Box als Koordinate). Verwaltet werden die Einzelelemente, aus denen eine Zeichnung besteht, als verkettete Liste. Dazu wird eine Basisklasse geschaffen, die die einzelnen Elemente gar nicht kennt, sondern nur die übergeordneten Eigenschaften und Methoden, die allen Elementen der Zeichnung zueigen sind. Wenn es z.B. notwendig wird, die gesamte Zeichnung neu aufzubauen, so geht das Programm Schritt für Schritt durch die Liste und ruft für jedes Element (ganz gleich welchen Typs) eine „virtuelle“-Methode „zeichne“ auf. Diese wird erst in der abgeleiteten Klasse definiert und verdeckt die Methode „zeichne“ der Basisklasse. Jedes einzelne Element weiß also durch seine Klassenzugehörigkeit wie es sich zu zeichnen hat.

Der klassische (strukturierte) Programmansatz hätte an dieser Stelle eine Fallunterscheidung machen müssen, um festzustellen, von welchem Typ das Zeichnungselement ist und anschließend eine Funktion „zeichneKreis“ oder „zeichneQuadrat“ aufrufen müssen. Der Vorteil des objektorientierten Ansatzes liegt jetzt darin, dass man weitere Elemente von der Basisklasse ableiten kann, ohne die Basisklasse ändern zu müssen, denn die Programmierung aller Abweichungen erfolgt ja in der abgeleiteten Klasse, in der die virtuellen Methoden „überschrieben“ werden können (aber nicht müssen!).

In der strukturierten Programmierung hingegen müsste man einen neuen Fall in die oben angeführte Fallunterscheidung einfügen, die Basis müsste also jedesmal verändert werden.

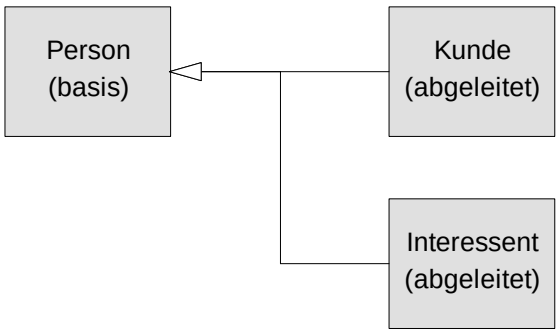


Abbildung 23-2– Wiederverwendung durch Vererbung

Die Verwendung von virtuellen Methoden führt insbesondere dann zu erheblichen Zeitgewinnen, wenn die Basisklasse fertig ist und mehrere Programmierer nunmehr – unabhängig voneinander – abgeleitete Klassen entwickeln können.

Die in der abgeleiteten Klasse neu definierte, virtuelle Methode kann natürlich in der Tochterklasse wiederum als *virtual* deklariert werden, um eine erneute Ableitung mit Redefinition zu ermöglichen.

23.2.1. PURE VIRTUELLE METHODEN

Gelegentlich kann es vorkommen, dass es keinen allgemeingültigen Algorithmus gibt, den man in der Basisklasse implementieren könnte. In solchen Fällen kann man den Entwickler zwingen eine virtuelle Methode in der abgeleiteten Klasse zu redefinieren. Virtuelle Methoden, die eine Redefinition erzwingen heißen „*pure virtual*“ und werden wie folgt definiert:

```
//============================================================
// Header der Klasse GraphElem (GRAPHELE.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//============================================================

#ifndef _GRAPHELE_H_
   #define _GRAPHELE_H_

   class GraphElem
   {
      protected:
         GraphElement *Next;                // werden vererbt
         double Xkoord;
         double Ykoord;

      public:
         virtual void Zeichne(void) = 0;    // pure virtual
   };
#endif
```

```
//============================================================
// Header der Klasse Kreis (KREIS.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _KREIS_H_
   #define _KREIS_H_

   #include "graphele.h"

   class Kreis : public GraphElem     // ANGABE BASECLASS
   {
      protected:
         double rad;                  // Radius
```

```
      public:
         virtual void Zeichne (void); // MUSS definiert
werden
   };
#endif
```

```
//============================================================
// Header der Klasse Quadrat (QUADRAT.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _QUADRAT_H_
   #define _QUADRAT_H_

   #include "graphele.h"

   class Quadrat : public GraphElem   // ANGABE BASECLASS
   {
      protected:
         double width;                // Seitenlänge

      public:
         virtual void Zeichne (void); // MUSS definiert
werden
   };
#endif
```

```
//============================================================
// Header der Klasse Rechteck (RECHTECK.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _RECHTECK_H_
   #define _RECHTECK_H_

   #include "quadrat.h"

   class Rechteck : public Quadrat    // ANGABE BASECLASS
   {
      protected:
         double height;               // Seitenhöhe

      public:
         virtual void Zeichne (void); // KANN definiert
werden
   };
#endif
```

Die Implementationen sind an dieser Stelle ohne Belang und werden daher nicht ausgeführt. Zu beachten ist hier, dass die „*pure virtual*“-Vererbung natürlich nur Auswirkungen auf die nächste Vererbungsstufe

hat. Aber natürlich kann man eine weitere Klassenebene einziehen, indem man die Rechtecke erneut verallgemeinert.

23.3. MEHRFACHE VERERBUNG

Dass man Klassen „in Reihe“ voneinander ableiten kann, ist aus dem obigen Beispiel zu *GraphElem* ersichtlich. Es gibt aber auch die Möglichkeit, von mehreren Basisklassen abzuleiten (engl. *multiple Inheritance*). Das ermöglicht es dem Entwickler viele, letztlich voneinander unabhängige – aber kombinierbare – Basisklassen zu schaffen.

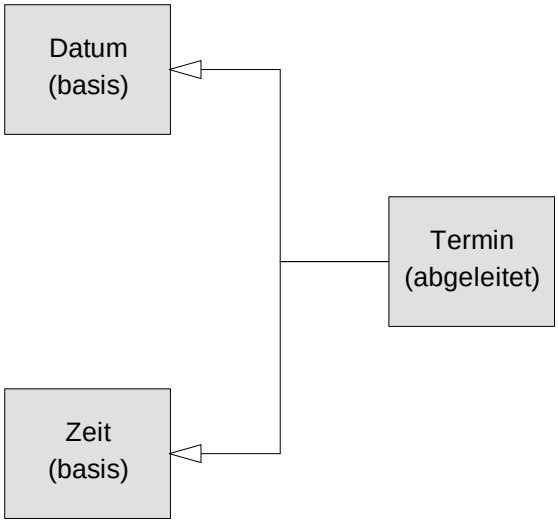


Abbildung 23-3 - Mehrfachvererbung

Nimmt man als Beispiel an, man hätte eine Klasse *Datum* (unabhängig definiert) und eine Klasse *Zeit* (ebenfalls unabhängig), dann ließe sich daraus eine neue Klasse *Termin* ableiten, indem man beide Klassen kombiniert:

```
//=========================================================
// Header der Klasse Datum (DATUM.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//=========================================================

#ifndef _DATUM_H_
   #define _DATUM_H_

   class Datum
   {
      protected:
         int Tag;
         int Monat;
         int Jahr;

      public:
```

```
         // Hier die üblichen Methoden: Constructor /
         // Destructor Zugriffsmethoden etc...
   };
#endif
```

```
//============================================================
// Header der Klasse Zeit (ZEIT.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//============================================================

#ifndef _ZEIT_H_
   #define _ZEIT_H_

   class Zeit
   {
      protected:
         int Stunde;
         int Minute;
         int Sekunde;

      public:
         // Hier die üblichen Methoden: Constructor /
         // Destructor Zugriffsmethoden etc...
   };
#endif
```

```
//============================================================
// Header der Klasse Termin (TERMIN.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _TERMIN_H_
   #define _TERMIN_H_

   #include "datum.h"
   #include "zeit.h"

   class Termin : public Datum, public Zeit
   {
      // Termin erbt Tag, Monat, Jahr, Stunde, Minute und
      // Sekunde, sowie die Methoden von Datum und Zeit.
      protected:
         int Dauer;
         char *Terminart [60];

      public:
         // Hier die üblichen Methoden: Constructor /
         // Destructor Zugriffsmethoden etc...
   };
#endif
```

23.4. VERERBUNGSPROBLEME

Die Verwendung von Klassen beseitigt weitgehend eines der grundsätzlichen Probleme komplexer Anwendungssysteme – die Notwendigkeit eindeutiger Variablen und Funktionsnamen. Durch die Tatsache, dass der Name eines Members oder einer Methode nur im Zusammenhang mit dem Namen des instanziierten Objekts gültig ist, kann man ohne Schwierigkeiten den gleichen Namen – in unterschiedlichen Klassen bzw. Strukturen – mehrfach verwenden.

Wie alle „Lösungen“ hat aber auch diese ihre Grenzen, die in diesem Fall genau dann erreicht ist, wenn in einer Vererbungskette ein Name mehrfach auftaucht (hier die Membervariable *na* und die Methode *setna*):

```
//============================================================
// Header der Klasse a (A.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//============================================================

#ifndef _A_H_
   #define _A_H_

   class a
   {
      protected:
         int na;

      public:
         void setna (int y) {na = y;}
   };
#endif
```

```
//============================================================
// Header der Klasse b (B.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _B_H_
   #define _B_H_

   #include "a.h"

   class b : public a
   {
      protected:
         int nb;

      public:
         void setnb (int y) {nb = y;}
   };
#endif
```

```
//=========================================================
// Header der Klasse c (C.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//=========================================================

#ifndef _C_H_
   #define _C_H_

   #include "b.h"

   class c : public b
   {
      protected:
         int na;

      public:
         void setna (int y) {na = y;}
   };
#endif
```

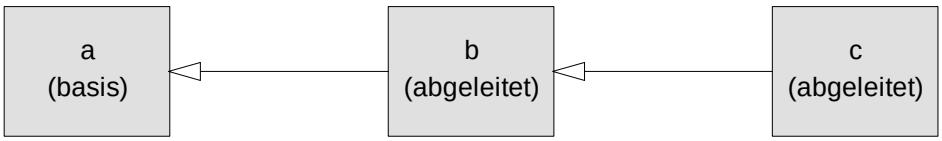


Abbildung 23-4 - Ableitungssequenz

Die oben dargestellte Ableitungssequenz scheint problemlos, zumal der Compiler weder Fehler (Error) noch Warnung (Warning) generiert. Tatsächlich behandelt C++ die Membervariable *na* und die Methode *setna ()* wie eine Überdeckung (siehe dazu auch das Kapitel über Funktionen - lokale Variablen). Es bedeutet in der Konsequenz aber auch, dass bestimmte Funktionalitäten verdeckt werden und nicht mehr zur Verfügung stehen.

23.4.1. PROBLEME BEI MEHRFACHER VERERBUNG

Die Überdeckung ist nicht das einzige Problem, welches bei der Vererbung entstehen kann. Noch problematischer ist die Uneindeutigkeit (Ambiguity), welche entsteht, wenn im Zuge einer Mehrfachvererbung die gleiche Basisklasse mehrfach in Erscheinung tritt, wie im folgenden Beispiel:

```
//=========================================================
// Header der Klasse a (A.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//=========================================================

#ifndef _A_H_
   #define _A_H_
```

```
   class a
   {
      protected:
         int na;

      public:
         void setna (int y) {na = y;}
   };
#endif
```

```
//============================================================
// Header der Klasse b1 (B1.H)
// Klasse b1 erbt alle Member/Methoden aus a
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _B1_H_
   #define _B1_H_

   #include "a.h"

   class b1 : public a
   {
      protected:
         int nb1;

      public:
         void setnb1 (int y) {nb1 = y;}
   };
#endif
```

```
//============================================================
// Header der Klasse b2 (B2.H)
// Klasse b2 erbt alle Member/Methoden aus a
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _B2_H_
   #define _B2_H_

   #include "a.h"

   class b2 : public a
   {
      protected:
         int nb2;

      public:
         void setnb2 (int y) {nb2 = y;}
   };
#endif
```

```
//============================================================
// Header der Klasse c (C.H)
// Klasse c erbt alle Member/Methoden aus b1 und b2
// da b1 und b2 public von a abgeleitet sind, enthält c die
// Member/Methoden von a zweimal.
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _C_H_
   #define _C_H_

   #include "b1.h"
   #include "b2.h"

   class c : public b1, public b2
   {
      protected:
         int nc;

      public:
         void setnc (int y) {nc = y;}
   };
#endif
```

```
//============================================================
// TESTPROGRAMM
// instanziiert ein Objekt der Klasse c und ruft anschließend
// alle ererbten Methoden auf
//============================================================

#include "c.h"

void main (void)
{
   c x;

   //-----------------------------------------------------------
   // kein Problem, da setnc() eindeutig aus der Klasse c ist
   //-----------------------------------------------------------
   x.setnc (1);

   //-----------------------------------------------------------
   // kein Problem, setnb1() eindeutig aus Klasse b1 geerbt
   //-----------------------------------------------------------
   x.setnb1 (2);

   //-----------------------------------------------------------
   // kein Problem, setnb2() eindeutig aus Klasse b2 geerbt
   //-----------------------------------------------------------
   x.setnb2 (3);

   //-----------------------------------------------------------
   // Fehler: member is ambiguous
   // Da setna() sowohl über Klasse b1 wie b2 vererbt wurde
   //-----------------------------------------------------------
```

```
   x.setna (4);
}
```

Die nachstehende, graphische Darstellung des Ableitungsbaumes macht das Problem noch etwas deutlicher.

Wie unschwer zu erkennen ist, sind alle Elemente der Klasse a doppelt vorhanden. Beim Zugriff auf Methoden der Klasse a ist die Compiler daher nicht mehr in der Lage zu entscheiden, welche der Methoden gemeint ist und bricht mit einem Ambiguity-Fehler ab.

Es ist einfach einzusehen, dass solche Fehler umso häufiger auftreten, je komplexer die Ableitungshierarchie zwischen Klassen wird. Es ist daher in solch komplexen Systemen oft klüger nicht mehrfach abzuleiten, sondern stattdessen innerhalb einer neuen Klasse Objekte der gewünschten Basisklassen zu erzeugen.

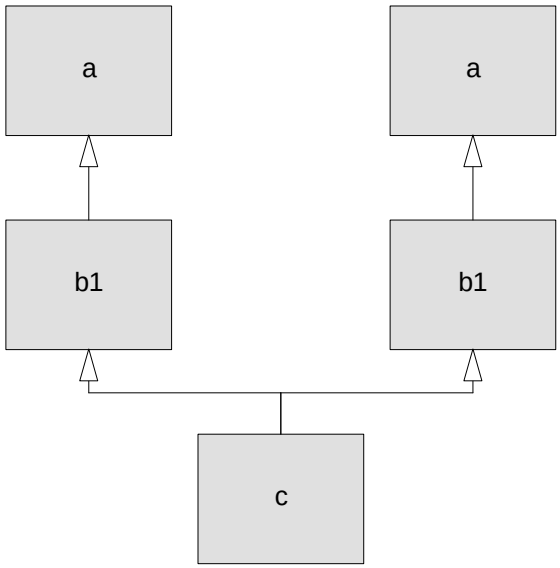


Abbildung 23-5 - Ambiguityfehler bei der Vererbung

Alternativ (und nur für einfachere Ableitungshierarchien zu empfehlen) gibt es die Möglichkeit der virtuellen Vererbung. Bei dieser Vererbungsform wird eine gemeinsam vorhandene Basisklasse nur einfach eingebunden.

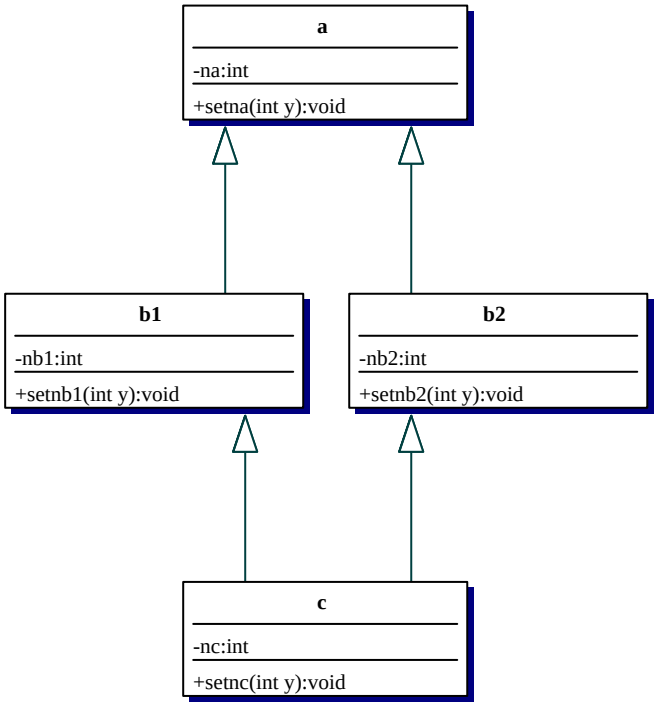


Abbildung 23-6 – UML-Diagramm einer Vererbung ohne Ambiguity

```
//=============================================================
// Headerdatei A.H
//=============================================================

#ifndef _A_H_
   #define _A_H_

   class a
   {
      protected:
         int na;

      public:
         void setna (int y) {na = y;}
   };
#endif
```

```
//=============================================================
// Headerdatei B1.H
//=============================================================

#ifndef _B1_H_
   #define _B1_H_

   #include "a.h"

   class b1 : virtual public a
   {
      protected:
         int nb1;
```

```
      public:
         void setnb1 (int y) {nb1 = y;}
   };
#endif
```

```
//============================================================
// Headerdatei B2.H
//============================================================

#ifndef _B2_H_
   #define _B2_H_

   #include "a.h"

   class b2 : virtual public a
   {
      protected:
         int nb2;

      public:
         void setnb2 (int y) {nb2 = y;}
   };
#endif
```

```
//============================================================
// Headerdatei C.H
//============================================================

#ifndef _C_H_
   #define _C_H_

   #include "b1.h"
   #include "b2.h"

   class c : virtual public b1, virtual public b2
   {
      protected:
         int nc;

      public:
         void setnc (int y) {nc = y;}
   };
#endif
```

```
//============================================================
// Programm CTEST.CPP
//============================================================

#include "c.h"

void main (void)
```

```
{
   c x;

   //-------------------------------------------------------
   // kein Problem, da setnc() eindeutig aus der Klasse c ist
   //-------------------------------------------------------
   x.setnc (1);

   //-------------------------------------------------------
   // kein Problem, setnb1() eindeutig aus Klasse b1 geerbt
   //-------------------------------------------------------
   x.setnb1 (2);

   //-------------------------------------------------------
   // kein Problem, setnb2() eindeutig aus Klasse b2 geerbt
   //-------------------------------------------------------
   x.setnb2 (3);

   //-------------------------------------------------------
   // kein Problem, da setna() virtuell vererbt wurde
   //-------------------------------------------------------
   x.setna (4);
}
```

Aufgrund der oben aufgezeigten Schwierigkeiten wird vielfach davon abgeraten die mehrfache Vererbung überhaupt zu verwenden. In vielen Programmiersprachen, die sich als Nachfolger von C++ betrachten (z.B. JAVA), fehlt die Möglichkeit der Mehrfachvererbung völlig.

24. POLYMORPHIE

Polymorphie ist ein Schlagwort, welches eine bestimmte Vorgehensweise in der objektorientierten Programmierung unter intensiver Verwendung von virtuellen Methoden beschreibt.

Sehr gut verdeutlichen lässt sich dies am Beispiel eines objektorientierten Graphikprogramms mit etwa folgendem Aufbau:

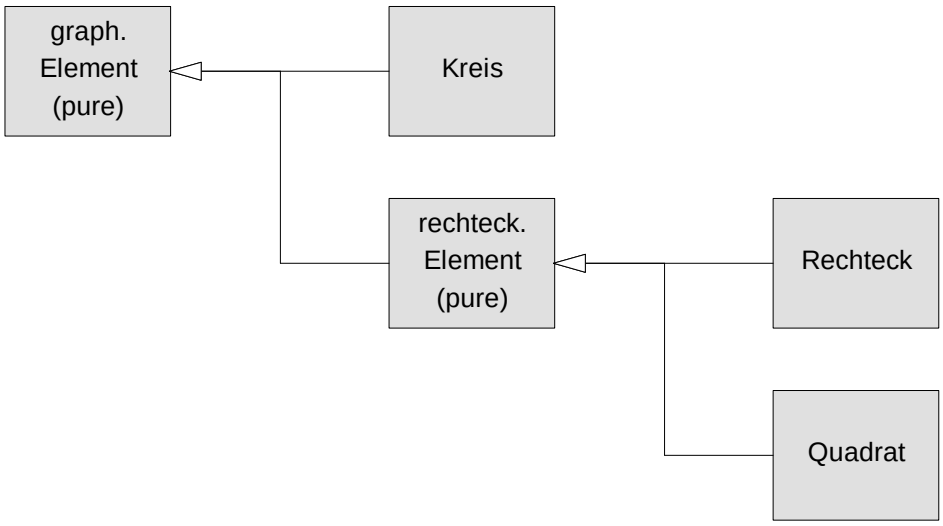


Abbildung 24-1 – Vererbungsschema eines OOP-Graphikprogramms

Solche objektorientierten Programme verwalten Graphikelemente als verkettete oder hierarchisch gruppierte Listen. Um nun nicht immer die komplette Listenverwaltung ändern zu müssen, wenn ein neuer, abgeleiteter *GraphElement* Typ (z.B. *Oval*) hinzuzufügen ist, bedient man sich der Polymorphie.

Das Verfahren ist eigentlich denkbar einfach, es wird lediglich eine Liste von Zeigern auf Instanzen der Basisklasse angelegt und alle Graphikelemente werden allein über virtuelle Methoden angesprochen.

Das folgende Beispiel, eine Teilimplementation des oben gezeigten Graphikprogramms, verdeutlicht die Vorgehensweise:

```
//============================================================
// Programm RTTI1.CPP
//============================================================

//------------------------------------------------------------
// Defines zur besseren Lesbarkeit
//------------------------------------------------------------
#define PURE        0
#define ANZELEMENTE 5
#define NULL        0L

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

```
// Basisklasse GRAPHELEMENT, stellt alle Methoden virtuell
zur
// Verfügung
//-------------------------------------------------------------
class GraphElement
{
   public:
      virtual ~GraphElement (void) {};
      virtual void Draw (void) = PURE;
};

//-------------------------------------------------------------
// abgeleitete Klasse KREIS, MUSS die Methoden oben vorsehen,
// da diese PURE sind
//-------------------------------------------------------------
class Kreis : public GraphElement
{
   public:
      virtual ~Kreis (void) {};
      virtual void Draw (void) {};
};

//-------------------------------------------------------------
// Testprogramm
//-------------------------------------------------------------
void main (void)
{
   GraphElement *Zeichnung [ANZELEMENTE];
   int i;

   //----------------------------------------------------------
   // Vorbelegung
   //----------------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      Zeichnung[i] = NULL;
   }

   //----------------------------------------------------------
   // Objekte erzeugen
   //----------------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      Zeichnung[i] = new Kreis;
   }

   //----------------------------------------------------------
   // Daten wieder löschen
   //----------------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      delete Zeichnung[i];
      Zeichnung[i] = NULL;
   }
}
```

Wie bei der Speicheranforderung mit *new* zu erkennen ist, wird jeder Zeiger auf eine abgeleitete Klasse als Zeiger auf seine Basisklasse behandelt.

Dies ist syntaktisch erlaubt, da eine abgeleitete Klasse technisch (von der Speicherbelegung her) nichts weiter ist als eine Struktur, die wiederum eine Struktur vom Typ der Basisklasse enthält. Die folgenden beiden Deklarationen sind daher vom Aufbau im Speicher identisch und unterscheiden sich nur in der Zugriffsmethodik:

```
//============================================================
// Klassen a und b
//============================================================

class a
{
   public:
      int avar;
};

class b
{
   public:
      int bvar;
};
```

```
//============================================================
// Strukturen a und b
//============================================================

struct a { int avar; };
struct b { a Basis; int bvar; };
```

Speicheraufbau der Klasse a			
Adresse	Wert	Name	Anmerkung
10000	unbekannt	int avar	Datenbereich der Klasse a

Speicheraufbau der Klasse b			
Adresse	Wert	Name	Anmerkung
10000	unbekannt	int avar	Übernommen aus Klasse a
10002	unbekannt	int bvar	Datenbereichsanteil der Klasse b

Speicheraufbau der Struktur a			
Adresse	Wert	Name	Anmerkung
10000	unbekannt	int avar	Datenbereich der Struktur a

Speicheraufbau der Struktur b			
Adresse	Wert	Name	Anmerkung
10000	unbekannt	int a.avar	Übernommen aus Struktur a
10002	unbekannt	int bvar	Datenbereichanteil der Struktur b

Weist man nun einem Pointer auf ein Objekt vom Typ *a* die Adresse eines Objektes vom Typ *b* zu, so zeigt der Pointer automatisch auf den im *b*-Objekt enthaltenen *a*-Anteil.

D.h. bei der Ableitung im Beispiel oben ist ein Zeiger auf *Kreis* gleichzeitig auch ein Zeiger auf *GraphElement*.

Genau diese Eigenschaft wird bei der Polymorphie ausgenutzt, um mit Hilfe der Basisklassenzeiger Listen (Arrays oder verkettete Strukturen) über abgeleitete Elemente zu bilden.

Man kann dann allgemeine Verwaltungsfunktionen (z.B. über *GraphElement*-Listen) schreiben, die allein mit den Zeigern dieser Basisklasse arbeiten – trotzdem kann man diese Funktionen auch mit den Zeigern von unterschiedlichsten, abgeleiteten Klassen benutzen:

```
//============================================================
// Programm RTTI2.CPP
//============================================================

#include <iostream>

using namespace std;

//------------------------------------------------------------
// Defines zur besseren Lesbarkeit
//------------------------------------------------------------
#define PURE        0
#define ANZELEMENTE 5

//------------------------------------------------------------
// Basisklasse GRAPHELEMENT, stellt alle Methoden virtuell
// zur Verfügung
//------------------------------------------------------------
class GraphElement
{
   public:
      virtual ~GraphElement (void) {};
      virtual void Draw (void) = PURE;
};

//------------------------------------------------------------
// abgeleitete Klasse KREIS, MUSS die Methoden oben vorsehen,
// da diese PURE sind
//------------------------------------------------------------
class Kreis : public GraphElement
{
```

```
   public:
      virtual ~Kreis (void) {};
      virtual void Draw (void) {};
};

//------------------------------------------------------------
// abgeleitete Klasse QUADRAT, MUSS die Methoden oben
// vorsehen, da diese PURE sind
//------------------------------------------------------------
class Quadrat : public GraphElement
{
   public:
      virtual ~Quadrat (void) {};
      virtual void Draw (void) {};
};

//------------------------------------------------------------
// Testprogramm
//------------------------------------------------------------
void main (void)
{
   GraphElement *Zeichnung [ANZELEMENTE];
   char Eingabe [80] = "";
   int i;

   //---------------------------------------------------------
   // Vorbelegung
   //---------------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      Zeichnung[i] = NULL;
   }

   //---------------------------------------------------------
   // Eingaben
   //---------------------------------------------------------
   cout << "Bitte 'Q' für Quadrat oder 'K' für Kreis"
        << " eingeben: " << endl;

   i=0;
   while (i<ANZELEMENTE)
   {
      cout << (i+1) << ".tes Element :";

      cin.getline (Eingabe, 80);
      switch (Eingabe[0])
      {
         case  'Q' :
         case  'q' : Zeichnung [i] = new Quadrat;
                     i++;
                     break;
         case  'K' :
         case  'k' : Zeichnung [i] = new Kreis;
                     i++;
                     break;
      }
```

```
   }


//#############################################################
//#    auf welchen Datentyp zeigt an dieser Stelle
//#    Zeichnung[3] ???
//#############################################################

   //-----------------------------------------------------
   // Daten wieder löschen
   //-----------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      delete Zeichnung[i];
      Zeichnung[i] = NULL;
   }
}
```

20.1 VIRTUELLE DESTRUCTOREN

Für die Nutzung von Polymorphie sind virtuelle Destructoren unverzichtbar, damit sichergestellt ist, dass der korrekte Destructor aufgerufen wird. Das folgende, sehr einfache Beispiel verdeutlicht den Vorgang:

```
//============================================================
// Header der Klasse aclass (ACLASS.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//============================================================

#ifndef _ACLASS_H_
   #define _ACLASS_H_

   #include <iostream>

   using namespace std;

   class aclass
   {
      protected:
         int avar;

      public:
         ~aclass (void) {cout << "Destructor aclass"; }
   };
#endif
```

```
//============================================================
// Header der Klasse bclass (BCLASS.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _BCLASS_H_
```

```
   #define _BCLASS_H_

   #include <iostream>
   #include <bclass.h>

   using namespace std;

   class bclass : public aclass
   {
      protected:
         int bvar;

      public:
         ~bclass (void) {cout << "Destructor bclass"; }
   };
#endif
```

```
//============================================================
// TESTPROGRAMM
//============================================================

#include <aclass.h>
#include <bclass.h>

void main (void)
{
    aclass *aptr = new bclass;
    delete aptr;
}
```

Wie bereits oben erwähnt, kann an jeden Pointer einer Basisklasse ein Objekt einer davon abgeleiteten Klasse angehängt werden. Leider wird jedoch beim delete der nicht-virtuelle Destructor der Basisklasse aufgerufen, so dass ein Teil der dynamisch reservierten Speicherbereiche (im Beispiel genau der Platzbedarf von *bvar*) nicht freigegeben wird und somit als Memory-Leakage (Speicher-Leck) hängen bleibt.

Wird stattdessen der Destructor von *aclass* als virtuell gekennzeichnet, tritt dieses Problem nicht mehr auf:

```
//============================================================
// Header der Klasse aclass (ACLASS.H)
// BASECLASS = BASISKLASSE = VATERKLASSE
//============================================================

#ifndef _ACLASS_H_
   #define _ACLASS_H_

   #include <iostream>

   using namespace std;
```

```
   class aclass
   {
      protected:
         int avar;

      public:
         virtual ~aclass (void) {cout << "Destructor aclass"
                                      << endl;}
   };
#endif
```

```
//============================================================
// Header der Klasse bclass (BCLASS.H)
// DERIVED CLASS = ABGELEITETE KLASSE = TOCHTERKLASSE
//============================================================

#ifndef _BCLASS_H_
   #define _BCLASS_H_

   #include <iostream>
   #include "aclass.h"

   using namespace std;

   class bclass : public aclass
   {
      protected:
         int bvar;

      public:
         virtual ~bclass (void) {cout << "Destructor bclass"
                                      << endl;}
   };
#endif
```

```
//============================================================
// TESTPROGRAMM VIRTDEST.CPP
//============================================================

#include "aclass.h"
#include "bclass.h"

void main (void)
{
    aclass *aptr = new bclass;
    delete aptr;
}
```

Um nicht unbeabsichtigt ein Speicherleck zu erzeugen sollten Sie grundsätzlich alle Destructoren als virtuell deklarieren. Zu Problemen kann dies nicht führen. Sie haben einzig und allein den

zusätzlichen Schreibaufwand um einen leeren Destructor zu deklarieren und definieren, falls für einen Destructor (noch) keine echte Notwendigkeit besteht. Es gehört jedoch zu einem guten Programmierstil immer einen virtuellen Destructor zur Verfügung zu stellen.

20.2 RUNTIME TYPE INFORMATION (RTTI)

In C++ besteht die Möglichkeit, den Typnamen einer Klasse zur Laufzeit zu ermitteln. Dies ist besonders nützlich, wenn (wie oben) mit Pointern auf polymorphe Objekte gearbeitet wird, man also gar nicht genau weiß, um welche Art von abgeleiteter Klasse es sich handelt.

Der Compiler kann angewiesen werden, für jede zu übersetzende Klasse im Programm Typinformationen (d.h. den Klassennamen) zur Verfügung zu stellen. Um diese dann zu nutzen, muss die Headerdatei *<typeinfo>* mit eingebunden werden.

Achtung:
RTTI ist eine Option, die nicht bei allen Compilern voreingestellt ist. Schlagen Sie die notwendigen Optionen und Einstellungen bitte gegebenenfalls im Handbuch ihres C++ Compilers nach.

```
//============================================================
// Programm RTTI3.CPP
//============================================================

#include <iostream>
#include <typeinfo>

using namespace std;

//------------------------------------------------------------
// Defines zur besseren Lesbarkeit
//------------------------------------------------------------
#define PURE        0
#define ANZELEMENTE 5

//------------------------------------------------------------
// Basisklasse GRAPHELEMENT, stellt alle Methoden virtuell
// zur Verfügung
//------------------------------------------------------------
class GraphElement
{
   public:
      virtual ~GraphElement (void) {};
      virtual void Draw (void) = PURE;
};

//------------------------------------------------------------
// abgeleitete Klasse KREIS, MUSS die Methoden oben vorsehen,
// da diese PURE sind
//------------------------------------------------------------
```

```
class Kreis : public GraphElement
{
   public:
      virtual ~Kreis (void) {};
      virtual void Draw (void) {};
};

//---------------------------------------------------------
// abgeleitete Klasse QUADRAT, MUSS die Methoden oben
// vorsehen, da diese PURE sind
//---------------------------------------------------------
class Quadrat : public GraphElement
{
   public:
      virtual ~Quadrat (void) {};
      virtual void Draw (void) {};
};

//---------------------------------------------------------
// Testprogramm
//---------------------------------------------------------
void main (void)
{
   GraphElement *Zeichnung [ANZELEMENTE];
   char Eingabe [80] = "";
   int i;

   //------------------------------------------------------
   // Vorbelegung
   //------------------------------------------------------
   for (i=0; i<ANZELEMENTE; i++)
   {
      Zeichnung[i] = NULL;
   }

   //------------------------------------------------------
   // Eingaben
   //------------------------------------------------------
   cout << "Bitte 'Q' für Quadrat oder 'K' für Kreis"
        << " eingeben: " << endl;

   i=0;
   while (i<ANZELEMENTE)
   {
      cout << (i+1) << ".tes Element :";

      cin.getline (Eingabe, 80);
      switch (Eingabe[0])
      {
         case  'Q' :
         case  'q' : Zeichnung [i] = new Quadrat;
                     i++;
                     break;
         case  'K' :
         case  'k' : Zeichnung [i] = new Kreis;
                     i++;
```

```
                    break;
            }
        }

        //-------------------------------------------------------
        // Namen der polymorph angebundenen Klassen
        //-------------------------------------------------------
        for (i=0; i<ANZELEMENTE; i++)
        {
            if (Zeichnung[i])
            {
                cout << typeid(Zeichnung[i]).name() << endl;
                cout << typeid(*Zeichnung[i]).name() << endl;
            }
        }

        //-------------------------------------------------------
        // Daten wieder löschen
        //-------------------------------------------------------
        for (i=0; i<ANZELEMENTE; i++)
        {
            delete Zeichnung[i];
            Zeichnung[i] = NULL;
        }
}
```

Es ist zu beachten, dass die erste Programmzeile in der Ausgabe der Typnamen:

```
cout << typeid(Zeichnung[i]).name() << endl;
```

nicht das gewünschte Ergebnis (den Typ des polymorph angebundenen Objekts) liefert, sondern nur den Typ des Pointers und der ist natürlich in allen Fällen vom Typ *GraphElement **.

Um nun an den gewünschten Klassennamen zu gelangen, muss dieser Pointer (wie in der folgenden Zeile) dereferenziert werden:

```
cout << typeid(*Zeichnung[i]).name() << endl;
```

Typeid() kann nicht nur mit Objektinstanzen (also tatsächlich vorhandenen Objekten) arbeiten, sondern auch mit Datentypen selbst.

VORSICHT:
Auf gar keinen Fall darf typeid() für einen NULL-Pointer aufgerufen werden, dies hat einen sofortigen Programmabsturz zur Folge.

Neben der Methode *typeid(Objekt).name()* sind noch einige weitere Methoden und Operatoren für Typen definiert.

Bei den Operatoren sind zwischen zwei Typenidentifikationen die Gleichheit und die Ungleichheit vorgesehen (*operator==* und *operator!=*)

```
//============================================================
// Programm TYPIDBSP.CPP
//============================================================

#include <iostream>
#include <typeinfo>

using namespace std;

class abc
{
};

class def : public abc
{
};

//------------------------------------------------------------
// Testprogramm
//------------------------------------------------------------
void main (void)
{
   abc eins, zwei;
   def drei;

   //-----------------------------------------------------------
   // Vergleich zweier Instanzen
   //-----------------------------------------------------------
   if (typeid(eins) == typeid(zwei))
   {
      cout << "Typen sind identisch" << endl;
   }
   else
   {
      cout << "Typen sind nicht identisch" << endl;
   }

   if (typeid(eins) == typeid(drei))
   {
      cout << "Typen sind identisch" << endl;
   }
   else
   {
      cout << "Typen sind nicht identisch" << endl;
   }

   //-----------------------------------------------------------
   // Vergleich zwischen Instanz und Typ
   //-----------------------------------------------------------
```

```
   if (typeid(eins) == typeid(abc))
   {
      cout << "Typen sind identisch" << endl;
   }
   else
   {
      cout << "Typen sind nicht identisch" << endl;
   }
}
```

Als weitere Methode, neben *name()*, steht für die Überprüfung noch eine Funktion mit der Bezeichnung *before()* zur Verfügung.

Der Name ist leider ein wenig irreführend, da man zunächst annehmen möchte, dass sich *before()* auf die Vererbungshierarchie bezieht. Leider ist dies nicht korrekt, *before()* stellt lediglich fest, ob ein Klassenname lexikalisch kleiner ist als ein zweiter Klassenname, d.h. ob er alphabetisch vor dem anderen liegt.

```
if (typeid(abc).before(typeid(def)))
{
    cout << "ist lexikalisch kleiner";
}
```

Ob es dieser Funktionalität bedurft hätte (und was der Anwender eigentlich mit der lexikalischen Reihenfolge anfangen soll) mag dahingestellt bleiben, schließlich kann man diese Auswertung ohne Probleme auch mit *strcmp()* und *typeid(Objekt).name()* durchführen.

20.3 HIERARCHIE ZUR LAUFZEIT PRÜFEN (DYNAMIC CASTING)

Leider gibt es keinen besonders einfachen Weg in C++, um zur Laufzeit festzustellen, ob eine Klasse von einer bestimmten Basisklasse abgeleitet ist, oder ob man ein Objekt gefahrlos auf einen anderen Typ umstellen darf (Typecasting). Letzteres benötigt man z.B. wenn man ein Objekt erweitern möchte, dessen Quellcode einem nicht zur Verfügung steht.

Ausgehend vom Beispiel des objektorientierten Graphikprogramms wäre z.B. denkbar, dass es sich bei der Graphikbibliothek um ein gekauftes Produkt handelt, welches die grundlegenden Graphikelemente beschreibt und eine Verwaltungsgrundlage mitliefert. Dies ist eine sehr häufig anzutreffende Konstellation bei gekauften Objektbibliotheken.

Ziel soll es nun sein, diese Bibliothek um eigene 3D-Varianten zu ergänzen. Da 3D-Objekte erheblich mehr grundlegende Methoden, Daten und Parameter brauchen, müsste man dazu sehr wahrscheinlich die Basisklasse ergänzen. So muss z.B. eine Methode zum Verschieben eines Objektes im Raum drei statt zwei Koordinaten enthalten.

Der beste Weg wäre sicherlich die Methode „Verschieben“ der Basisklasse um einen Parameter zu erweitern (z-Achse) und diesen Übergabewert für 2D-Objekte mit Null vorzubelegen (Defaultparameter). Dadurch könnten bereits existierende Programme ohne Änderung übernommen werden.

Dieser Weg steht aber leider nicht offen, da in diesem angenommenen Beispiel der Quellcode nicht abgeändert werden kann.

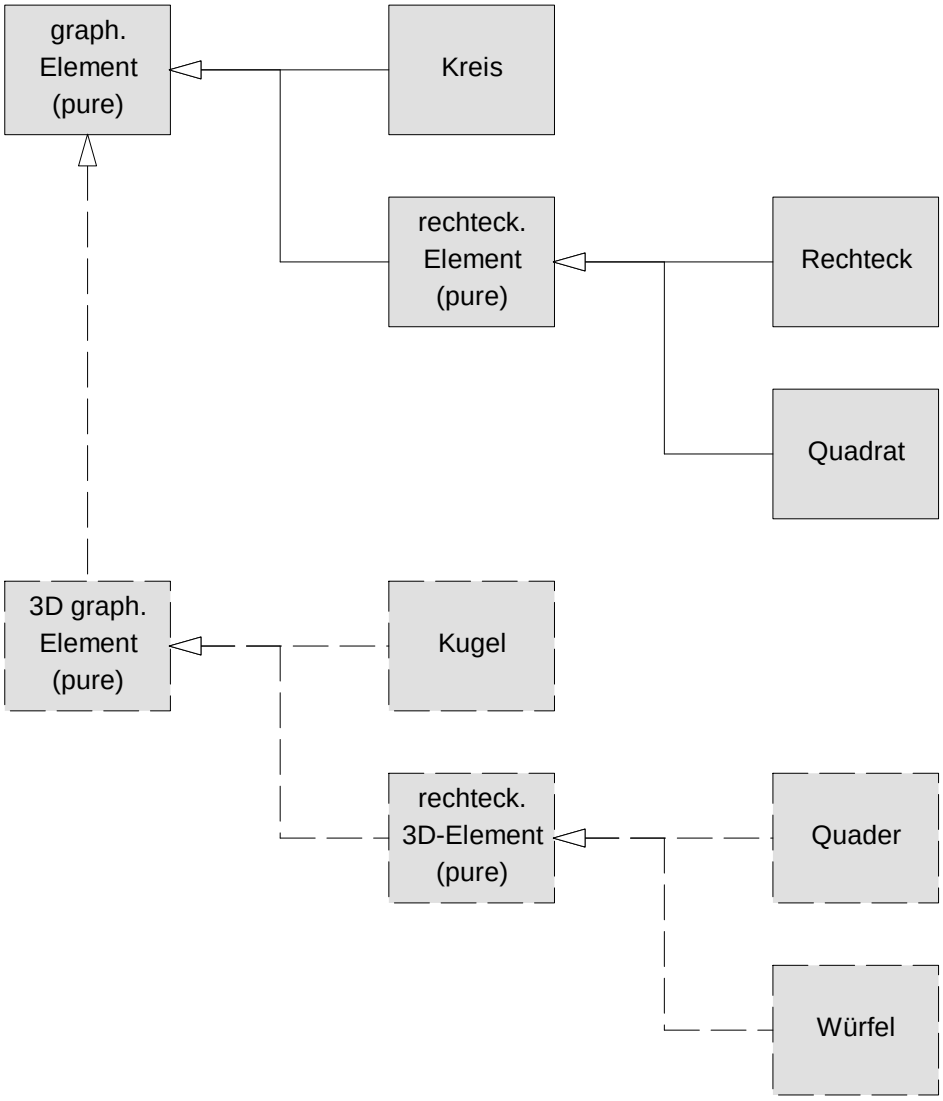


Abbildung 24-2 – Vererbungsschema eines OOP-Graphikprogramms

Es bleibt daher nur der Weg über eine abgeleitete Klasse, wie in der oben dargestellten Abbildung.

Da die Basisklasse gleich bleibt, können über die nun entstandene polymorphe Liste (s.o.) die spezifischen 3D-Methoden nicht aufgerufen werden, da es in der 2D-Basisklasse dafür keine virtuellen Prototypen gibt.

Dies leistet zwar die 3D-Basisklasse, man darf aber nicht alle Einträge der polymorphen Liste einfach als 3D-Elemente betrachten, da die 2D-Elemente (wie aus der Graphik ersichtlich) weiterhin die 3D-Methoden nicht kennen.

Wandelt man den Kreis über eine Typumwandlung (Casting) in eine Kugel um und ruft die 3D-Methode zum Verschieben auf, so ist ein Absturz unvermeidlich.

Faktum ist, dass man zunächst feststellen muss, ob ein dynamisch (polymorph) angebundenes Objekt zum 2D- oder 3D-Strang der Vererbungshierarchie gehört.

Dazu stellt C++ leider keine direkte Methode zur Ermittlung bereit, man hat jedoch die Möglichkeit eine solche Typumwandlung „probehalber“ durchzuführen und das Laufzeitsystem zu befragen, ob es dabei zu Problemen gekommen ist. Diese Vorgehensweise wird üblicherweise als *dynamic casting* oder *save downcast* bezeichnet.

Voraussetzung ist allerdings, dass alle Objekttypen mit der RTTI -Option übersetzt worden sind.

Eine typsichere Umwandlung nach obigem Muster hätte jetzt das folgende Aussehen:

```
//============================================================
// Programm SAVEDOWN.CPP
//============================================================

#include <iostream>
#include <typeinfo>

using namespace std;

//------------------------------------------------------------
// 2D-Basisklasse
//------------------------------------------------------------
class GraphElement
{
   public:
      virtual void Verschiebe (void)
                  { cout << "Verschieben" << endl;}
};

//------------------------------------------------------------
// Beispiel für abgeleitetes 2D-Objekt
//------------------------------------------------------------
class Kreis : public GraphElement
{
   public:
      virtual void Verschiebe (void)
                  { cout << "Kreis Verschieben" << endl;}
};
```

```
//-------------------------------------------------------------
// 3D-Basisklasse
//-------------------------------------------------------------
class Graph3D : public GraphElement
{
   public:
      virtual void Verschiebe (void)
                   { cout << "2D-Verschieben" << endl;}
      virtual void Verschiebe3D (void)
                   { cout << "3D-Verschieben" << endl;}
};

//-------------------------------------------------------------
// Beispiel für abgeleitetes 3D-Objekt
//-------------------------------------------------------------
class Kugel : public Graph3D
{
   public:
      virtual void Verschiebe (void)
                   { cout << "2D-Kugel Verschieben" << endl;}
      virtual void Verschiebe3D (void)
                   { cout << "3D-Kugel Verschieben" << endl;}
};

//-------------------------------------------------------------
// Testprogramm
//-------------------------------------------------------------
void main (void)
{
   int i;

   //----------------------------------------------------------
   // Erzeugung einer (sehr) einfachen polymorphen Liste
   //----------------------------------------------------------
   Kreis         aKreis;
   Kugel         aKugel;
   GraphElement *Zeichnung[2] = {&aKreis, &aKugel};

   //----------------------------------------------------------
   // Hilfpointer für 3D-Objekte
   //----------------------------------------------------------
   Graph3D *p3DObjekt = NULL;

   //----------------------------------------------------------
   // Über gesamte Liste gehen
   //----------------------------------------------------------
   for (i=0; i<2; i++)
   {
      //-------------------------------------------------------
      // 2D-Objektverschiebung geht immer
      //-------------------------------------------------------
      Zeichnung[i]->Verschiebe();

      //-------------------------------------------------------
      // feststellen, ob es ein 3D-Objekt ist
```

```
      //-----------------------------------------------------
      p3DObjekt = dynamic_cast <Graph3D *>(Zeichnung[i]);
      if (p3DObjekt)
      {
         //-----------------------------------------------------
         // 3D-Verschiebung nur mit 3D-Objekten
         //-----------------------------------------------------
         p3DObjekt->Verschiebe3D();
      }
   }
}
```

Der Befehl *dynamic_cast* versucht im angeführten Beispiel den Pointer *Zeichnung[i]* in einen *Graph3D*-Pointer zu überführen. Gelingt dieses, so gibt *dynamic_cast* die Adresse des Objektes zurück, sonst einen *NULL*-Pointer.

In für C/C++ typischer Art und Weise kann man auf den *Graph3D*-Pointer als Zwischenspeicher natürlich auch verzichten und die Typumwandlung in der *if*-Anweisung direkt ausführen:

```
for (i=0; i<2; i++)
   {
      //-----------------------------------------------------
      // 2D-Objektverschiebung geht immer
      //-----------------------------------------------------
      Zeichnung[i]->Verschiebe();

      //-----------------------------------------------------
      // feststellen, ob es ein 3D-Objekt ist
      //-----------------------------------------------------
      if (dynamic_cast <Graph3D *>(Zeichnung[i]))
      {
         //-----------------------------------------------------
         // 3D-Verschiebung nur mit 3D-Objekten
         //-----------------------------------------------------
         ((Graph3D *)Zeichnung[i])->Verschiebe3D();
      }
   }
```

Ein wenig eleganter ist es natürlich, die Abfrage in eine Funktion zu packen, zumal es dann möglich ist, alle Änderungen der Abfrage zentral vorzunehmen:

```
Graph3D *IsGraph3D (GraphElement *x)
{
   return dynamic_cast <Graph3D *>(x);
}
```

Der Schreibweise des *dynamic_cast* (in spitzen Klammern mit einem Datentyp darin) ist zu entnehmen, dass es sich bei dieser Anweisung um ein Template-Makro handelt.

25. DYNAMISCHE SPEICHERVERWALTUNG

Da es in C – wie bei fast allen Compiler-Sprachen – keine Möglichkeit gibt dynamische Felder anzulegen (anders als bei den meisten Interpretersprachen), bleibt lediglich die dynamische Speicherallokation, um diese Beschränkung zu umgehen. Dazu wird der benötigte Speicherplatz erst zur Laufzeit des Programms berechnet und angefordert.

25.1. DYNAMISCHE SPEICHERVERWALTUNG IN ANSI-C

Wie aus den nachfolgenden Beispielen ersichtlich wird, ist die dynamische Speicherverwaltung unter ANSI-C recht unhandlich. Mit C++ wurde die Anforderung von Speicher erheblich vereinfacht, so dass man – sofern möglich – stets versuchen sollte, die C++ Variante (bestehend aus den Befehlen *new* und *delete*) zu verwenden.

Die ANSI-C Befehle werden hier der Vollständigkeit halber aber ebenfalls erläutert.

25.1.1. DIE FUNKTION MALLOC

Die angesprochene Lösung des Problems, oder zumindest seine Umschiffbarkeit liegen in den *malloc*-Funktionen, die zum Umfang der Standardbibliothek gehört. Mit dem Aufruf der *malloc*-Funktion (manchmal auch *alloc* genannt) wird vom Betriebssystem des Rechners ein Speicherblock bestimmter Größe angefordert.

```
Syntax:
   malloc (sizeof (Variablentyp));
```

Anders als bei den Feldern ist es dabei nicht notwendig, dass die Größe dieses Speicherblocks schon zur Übersetzungszeit bekannt sein muss.

```
//============================================================
// Programm MALLOC.CPP
//============================================================

#include <stdlib.h>
#include <malloc.h>

typedef struct Zeile
{
   char *String;          // Zeiger auf Text
   struct Zeile *Prev;    // Zeiger vorherige Zeile
   struct Zeile *Next;    // Zeiger nächste Zeile
} TEXTZEILE, *TEXTPTR;

TEXTPTR NeueZeile (void)
{
   TEXTPTR pt = NULL;
```

```
   pt = (TEXTPTR) malloc (sizeof(TEXTZEILE));
   return pt;
}

void main (void)
{
   TEXTPTR Start = NULL;

   Start = NeueZeile ();
   if (Start != NULL)
   {
      free (Start);
   }
}
```

Das Beispiel zeigt die dynamische Allokation einer Liste für Textzeilen, wie man sie z.B. in einem Editor-Quellcode erwarten könnte. Mit dem Aufruf der Funktion *malloc* wird die Größe des gewünschten Speichers als Parameter übergeben (hier *sizeof (TEXTZEILE)*). Da die gewünschte Länge hier implizit übergeben wird, reicht bei einer Änderung des Aufbaus der Struktur *TEXTZEILE* ein Neuübersetzen aus, um die Änderungen wirksam zu machen.

Als Ergebnis des Funktionsaufrufes erhält der Programmierer einen Pointer auf den nun reservierten Speicherbereich.

C interessiert sich hierbei nicht für den Datentyp, der in diesem Bereich gespeichert werden soll, daher erfolgt die Angabe der Größe in Bytes (Typ *void*, hier gleich mit Casting in *char* umgewandelt) und der Pointer ist vom Typ „Pointer to char“. Der zurückgegebene Pointer wird hier durch ein Casting (explizite Typumwandlung) in den gewünschten Typ überführt. Der erhaltene Pointer darf auf gar keinen Fall verloren gehen, da sonst auf den belegten Speicherbereich weder zugegriffen, noch dieser Bereich wieder freigegeben werden kann.

Ist nicht mehr genügend Speicherplatz frei, um einen Block der geforderten Größe zu reservieren, so wird ein Pointer auf NULL zurückgegeben. Dies muss nicht unbedingt heißen, dass der erforderliche Speicherplatz nicht ausreicht - unter Umständen ist die Speicherbelegung nur durch häufiges Anfordern und Freigeben von Speicherbereichen so stark fragmentiert, dass kein zusammenhängender Block entsprechender Größe mehr zur Verfügung steht.

Dieses Phänomen ist stark abhängig vom verwendeten Betriebssystem. Einige Systeme führen in solchen Fällen automatisch eine sogenannte Garbage Collection (Müllabfuhr) durch und bauen die Speicherbelegung völlig neu auf. Bei einfacheren Betriebssystemen, die keine Garbage Collection durchführen, kann es hilfreich sein, alle Daten auf ein Speichermedium auszulagern, die belegten Speicherbereiche freizugeben

und die Daten dann wieder einzulesen. Beide Formen Arten der „Entsorgung“ benötigen einiges an Zeit.

Betriebssysteme mit einer sehr guten Speicherverwaltung können auch nichtzusammenhängende Speicherbereiche reservieren oder in sogenannte virtuelle Speicherbereiche auslagern, was allerdings den Verwaltungsaufwand beträchtlich erhöht. Grundsätzlich bleibt die Problematik dieser Art der Speicherreservierung aber erhalten, da vom C-Compiler selbst keinerlei Laufzeitunterstützung in dieser Hinsicht geliefert wird.

So muss der Programmierer auch selbst darauf achten, dass die abgespeicherten Daten nicht über die Grenzen des Speicherblocks hinweg geschrieben werden, wo sie unter Umständen bei der nächsten Speicheranforderung überschrieben werden (sei es durch einen weiteren *malloc*-Aufruf oder die Reservierung von Speicher für lokale Variable während eines Funktionsaufrufes) oder selbst wichtige Daten überschreiben. Wird der reservierte Speicherbereich nicht mehr benötigt, so kann er mit der Funktion *free* wieder freigegeben und später, gegebenenfalls mit veränderter Größe, neu angefordert werden.

```
Syntax:
   free (Pointer);
```

Beachten Sie bitte, dass Sie unter keinen Umständen einen nicht belegten Pointer mit *free* freigeben dürfen (siehe *if*-Abfrage in der Funktion *main* im Beispiel oben), da dies in den meisten Betriebssystemen (insbesondere bei Multitaskingsystemen) zu Systemabstürzen führt.

25.1.2. DIE FUNKTION CALLOC

Sollen nun nicht einfache Daten (z.B. Texte) gespeichert werden, sondern Felder, so wird der dynamische Aufbau von Arrays von der Funktion *calloc* besser unterstützt, als durch die Standardfunktion *malloc*. Wie *malloc* gibt *calloc* einen zu castenden Pointer zurück, bzw. einen Pointer auf *NULL*, wenn nicht genügend Speicherplatz vorhanden ist.

```
Syntax:
   calloc (Anzahl, sizeof (Variablentyp));
```

Bei *calloc* liegt der Vorteil darin, dass der Pointer anschließend wie eine Feldvariable behandelt werden kann, auf die mittels eines Zeigers zugegriffen wird. Außerdem erspart sich der Programmierer durch Angabe der Feldelementanzahl und der Größe der Feldinhalte (in Bytes) die Berechnung des benötigten Speicherplatzes.

```
//=============================================================
// Programm CALLOC.CPP
//=============================================================

#include <stdlib.h>
#include <malloc.h>

typedef struct Zeile
{
   char *String;          // Zeiger auf Text
   struct Zeile *Prev;    // Zeiger vorherige Zeile
   struct Zeile *Next;    // Zeiger nächste Zeile
}  TEXTZEILE, *TEXTPTR;

TEXTPTR NeueZeile (int Count)
{
   TEXTPTR pt = NULL;

   pt = (TEXTPTR)calloc (Count, sizeof(TEXTZEILE));

   return pt;
}

void main (void)
{
   TEXTPTR Start = NULL;

   Start = NeueZeile (5);

   if (Start != NULL)
   {
      free (Start);
   }
}
```

Im Beispiel stellt *Count* die Anzahl der zu reservierenden Feldelemente dar, in diesem Fall vom Typ *TEXTZEILE*, daher wird die Größe des Einzelelementes mit *sizeof (TEXTZEILE)* übergeben. Die Übergabe mittels *sizeof* hat den Vorteil, dass der Programmierer unabhängig von der tatsächlich auf der Maschine implementierten Variablenlänge bleibt (z.B. bei Integer). Das Casting wandelt den Pointer in einen Zeiger auf *TEXTZEILE* um, so dass dieser problemlos zugewiesen werden kann, ohne eine Compilermeldung auszulösen. Anschließend kann über die Variable *Start* wie mit einem Zeiger auf eine Feldvariable zugegriffen werden.

```
Aktuell = Start;
Zuletzt = Start;
Aktuell++;                  // Zeigt auf 2. Zeile!
Start->Prev = NULL;
```

```
Start->Next = Aktuell;     // untereinander verbind.
Aktuell->Prev = Start;
Zuletzt = Aktuell;
Aktuell++;
Zuletzt->Next = Aktuell;   // usw, usw...
```

Die mit Zeigern verbundenen Strukturen nennt man verkettete Liste (hier eine doppelt verkettete Liste, da Zeiger sowohl auf das nachfolgende wie auf das vorausgehende Listenelement existieren).

Die Freigabe des belegten Speicherplatzes erfolgt wie bei *malloc* über die Funktion *free*. Man sollte allerdings strikt beachten, dass niemals ein Speicherbereich freigegeben werden darf, der nicht zuvor mit *malloc* oder *calloc* reserviert wurde, da jeder Zugriff auf eine solche (lokale oder globale) Variable oft mit dem Absturz des Rechners endet.

Außerdem ist zu beachten, dass ein einzelnes Glied einer verketteten Liste erst vollständig aus der Verkettung gelöst werden muss (Alle Pointer zeigen auf *NULL*), bevor es sicher freigegeben werden kann, da sonst (bei vielen Compilern) alle damit noch verbundenen Kettenglieder ebenfalls gelöscht werden.

25.2. DYNAMISCHE SPEICHERVERWALTUNG IN C++

Die dynamische Speicherverwaltung in C++ ist – gegenüber ANSI-C – erheblich verbessert worden, so entfällt z.B. die umständliche, von Hand zu durchzuführende, Berechnung der benötigten Größe in Bytes.

25.2.1. DER OPERATOR NEW

Der in C++ definierte *new*-Operator löst die *malloc*- und *calloc*-Funktionen ab. Die Größe des zu erzeugenden Objekts wird dabei automatisch ermittelt und auch die Angabe von Arrays ist möglich. Besonderen Wert wurde darauf gelegt, dass *new* auch mit allen selbstdefinierten Klassen problemlos funktioniert, so dass man ein einheitliches Konzept für die dynamische Speicherverwaltung vorliegen hat.

```
Syntax:
   new Variablentyp;
   new Variablenarray [n];
```

Da es sich bei *new* um einen Standardoperator handelt, steht dieser überall zur Verfügung, ohne dass man zuerst eine Headerdatei einbinden muss:

```
//============================================================
// Programm NEWDEL.CPP
//============================================================

#include <stdlib.h>

class datum              // eigene Datumsklasse als Beispiel
{
   int Tag;
   int Monat;
   int Jahr;
};

struct Zeile
{
   char *String;         // Zeiger auf Text
   struct Zeile *Prev;   // Zeiger vorherige Zeile
   struct Zeile *Next;   // Zeiger nächste Zeile
};

void main (void)
{
   Zeile *Start    = new Zeile;      // einfache Struktur
   int   *Intptr   = new int [20];   // Feld
   datum *Datumptr = new datum;      // eigene Klasse

   if (Start    != NULL) delete Start;      // Freigabe
   if (Intptr   != NULL) delete [] Intptr;  // Feldfreigabe
   if (Datumptr != NULL) delete Datumptr;   // Freigabe
}
```

Das Beispiel zeigt die dynamische Allokation einer Liste für Textzeilen, wie man sie z.B. in einem Editor-Quellcode erwarten könnte. Wie man erkennen kann, erleichtern die Neuerungen von C++ die Handhabung dynamischen Codes erheblich – So entfallen z.B. die Typdeklarationen mit *typedef*. Mit dem Aufruf von *new* wird automatisch die Größe des gewünschten Speicherobjekts ermittelt und als Parameter übergeben. Als Ergebnis des Funktionsaufrufes erhält man einen Pointer auf den nun reservierten Speicherbereich.

Im Gegensatz zu ANSI-C ist C++ an dieser Stelle stärker typisiert, d.h. es wird nicht ein *void*-Pointer zurückgegeben, sondern ein bereits auf den Objekttyp abgestimmter (casted) Zeiger (siehe auch Casting). Wie bei ANSI-C darf der erhaltene Pointer auf gar keinen Fall verloren gehen, da sonst auf den belegten Speicherbereich weder zugegriffen, noch dieser Bereich wieder freigegeben werden kann.

Ist nicht mehr genügend Speicherplatz frei, um einen Block der geforderten Größe zu reservieren, so wird ein Pointer auf NULL zurückgegeben. Dies muss nicht unbedingt heißen, dass der erforderliche Speicherplatz nicht ausreicht - unter Umständen ist die Speicherbelegung

nur durch häufiges Anfordern und Freigeben von Speicherbereichen so stark fragmentiert, dass kein zusammenhängender Block entsprechender Größe mehr zur Verfügung steht.

Dieses Phänomen ist stark abhängig vom verwendeten Betriebssystem. Einige Systeme führen in solchen Fällen automatisch eine sogenannte Garbage Collection (Müllabfuhr) durch und bauen die Speicherbelegung völlig neu auf. Bei einfacheren Betriebssystemen, die keine Garbage Collection durchführen, kann es hilfreich sein, alle Daten auf ein Speichermedium auszulagern, die belegten Speicherbereiche freizugeben und die Daten dann wieder einzulesen. Beide Formen Arten der „Entsorgung“ benötigen einiges an Zeit.

Betriebssysteme mit einer sehr guten Speicherverwaltung können auch nichtzusammenhängende Speicherbereiche reservieren oder in sogenannte virtuelle Speicherbereiche auslagern, was allerdings den Verwaltungsaufwand beträchtlich erhöht und die Ausführungsgeschwindigkeit senkt. Grundsätzlich bleibt die Problematik dieser Art der Speicherreservierung aber erhalten, da auch vom C++ Compiler keinerlei Laufzeitunterstützung in dieser Hinsicht geliefert wird.

So muss der Programmierer auch weiterhin selbst darauf achten, dass die abgespeicherten Daten nicht über die Grenzen des Speicherblocks hinweg geschrieben werden.

25.2.2. DER OPERATOR DELETE

Wird der reservierte Speicherbereich nicht mehr benötigt, so kann er mit dem Operator *delete* bzw. *delete []* (bei Feldern) wieder freigegeben und später, gegebenenfalls mit veränderter Größe, neu angefordert werden.

```
Syntax:
   delete Objektpointer;
   delete [] Objektfeldpointer
```

Es ist zu beachten, dass man unter keinen Umständen einen nicht belegten Pointer mit *delete* oder *delete []* freigeben darf (siehe *if*-Abfrage oben, in der Funktion *main*), da dies in den meisten Betriebssystemen (insbesondere bei Multitaskingsystemen) zu Systemabstürzen führt.

25.3. BEISPIEL: EINFACHE, DYNAMISCHE STRINGKLASSE

Der klassische Fall einer Klasse mit dynamischen Elementen ist eine Stringklasse mit optimaler Längenausnutzung. Eine entsprechende Klasse hat in etwa das folgende Aussehen (hier nur in den grundlegenden Methoden implementiert):

```
//============================================================
// Programm DYNSTR.H
//============================================================

#ifndef _DYNSTR_H_
#define _DYNSTR_H_

class dynstr
{
   //-----------------------------------------------------
   // Member-Variablen, nach Möglichkeit immer PRIVATE !
   //-----------------------------------------------------
   private:
      char *ptr;

   //-----------------------------------------------------
   // (Methoden) Member-Funktionen
   //-----------------------------------------------------
   public:
      //-----------------------------------------------------
      // Constructoren und Destructor
      //-----------------------------------------------------
       dynstr ();             // Constructor
       dynstr (dynstr& x);    // Constructor
      ~dynstr ();

      //-----------------------------------------------------
      // Methoden
      //-----------------------------------------------------
      char *get     (void) const       {return (ptr);};
      void  set     (char *toset);
      int   len     (char *tocount) const;
      void  concat  (char *toadd);
      void  lower   (void);
      void  upper   (void);
      char *ltrim   (void);
      char *rtrim   (void);
      char *alltrim (void);

      //-----------------------------------------------------
      // Operatoren
      //-----------------------------------------------------
      char *operator=  (dynstr & tocopy);
      char *operator=  (char *tocopy);
      int   operator== (dynstr & tocheck);
      int   operator== (char *tocheck);
};
#endif
```

```
//============================================================
// Programm DYNSTR.H
//============================================================
```

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include "dynstr.h"

using namespace std;

#define ENDSTR '\0'

//============================================================
// Constructor der dynamischen Stringklasse
//============================================================
dynstr::dynstr ()
{
   //-----------------------------------------------------
   // Alle Ptr sollten stets einen gültigen Wert haben
   //-----------------------------------------------------
   ptr = NULL;
}

//============================================================
// Copy-Constructor der dynamischen Stringklasse
//============================================================
dynstr::dynstr (dynstr& x)
{
   set (x.ptr);
}

//============================================================
// Destructor der dynamischen Stringklasse
//============================================================
dynstr::~dynstr ()
{
   //-----------------------------------------------------
   // Freigabe ist nur dann sinnvoll, wenn auch was enthalten
   // ist! - VORSICHT, wenn Null-Pointer freigegeben wird
   // besteht Absturzgefahr !
   //-----------------------------------------------------
   if (ptr) delete ptr;
}

//============================================================
// String in Kleinschreibung umwandeln
//============================================================
void dynstr::lower (void)
{
   char *help = ptr;

   //-----------------------------------------------------
   // Wenn keine Daten vorhanden, kann man nichts tun
   //-----------------------------------------------------
   if (!ptr) return;
```

```
   //-----------------------------------------------------
   // Wenn Daten vorhanden sind, dann umsetzen (insbesondere
   // deutsche Sonderzeichen)
   //-----------------------------------------------------
   for (; *help; help++)
   {
      if ((*help > 64) && (*help < 91)) *help += (char)32;
      else switch (*help)
      {
         case 'Ä' : *help = 'ä'; break;
         case 'Ö' : *help = 'ö'; break;
         case 'Ü' : *help = 'ü'; break;
      }
   }
}

//=========================================================
// String in Großschreibung umwandeln
//=========================================================
void dynstr::upper (void)
{
   char *help = ptr;

   //-----------------------------------------------------
   // Wenn keine Daten vorhanden, kann man nichts tun
   //-----------------------------------------------------
   if (!ptr) return;

   //-----------------------------------------------------
   // Wenn Daten vorhanden sind, dann umsetzen (insbesondere
   // deutsche Sonderzeichen)
   //-----------------------------------------------------
   for (; *help; help++)
   {
      if ((*help > 96) && (*help < 123)) *help -= (char)32;
      else switch (*help)
      {
         case 'ä' : *help = 'Ä'; break;
         case 'ö' : *help = 'Ö'; break;
         case 'ü' : *help = 'Ü'; break;
      }
   }
}

//=========================================================
// Leerzeichen links abschneiden
//=========================================================
char *dynstr::ltrim (void)
{
   char *help = ptr;
   char *newp = NULL;
   char *nptr = NULL;

   //-----------------------------------------------------
   // Wenn keine Daten vorhanden, kann man nichts tun
```

```
   //-------------------------------------------------------
   if (!ptr) return (NULL);

   //-------------------------------------------------------
   // Wenn der ASCII-Wert des aktuellen Zeichens kleiner ist
   // als 33, dann ist es ein Whitespace-Zeichen und soll
   // abgeschnitten werden. Hier wird nach dem ersten nicht-
   // Whitespace gesucht
   //-------------------------------------------------------
   while ((*help < 33) && (*help != '\0')) help++;

   //-------------------------------------------------------
   // Wenn help jetzt auch '\0'-Zeichen steht, dann enthält
   // die Zeichenkette nur Whitespace-Zeichen und wird
   // zurückgesetzt
   //-------------------------------------------------------
   if (*help == ENDSTR)
   {
      ptr[0] = ENDSTR;
      return (ptr);
   }

   //-------------------------------------------------------
   // Wenn die Zeichenkette verwertbare Zeichen enthält, dann
   // neuen Speicher für die Restlänge (ohne führende
   // Whitespaces) anfordern
   //-------------------------------------------------------
   nptr = new char [len (help) + 1];

   //-------------------------------------------------------
   // Kein Speicher verfügbar? dann alten String zurück
   //-------------------------------------------------------
   if (!nptr) return (ptr);

   //-------------------------------------------------------
   // Wenn Speicher verfügbar, dann Zeichenweise kopieren und
   // mit String-Ende-Zeichen abschließen
   //-------------------------------------------------------
   for (newp = nptr;*help;help++,newp++) *newp = *help;
   *newp = 0;

   //-------------------------------------------------------
   // Nach dem kopieren den alten Speicher wieder freigeben
   // und den internen Stringpointer auf den neuen
   // Speicherbereich zeigen lassen. Zum Schluß  noch den
   // Stringpointer zurückgeben
   //-------------------------------------------------------
   delete ptr;
   ptr = nptr;
   return (ptr);
}

//=========================================================
// Leerzeichen rechts abschneiden
//=========================================================
char *dynstr::rtrim (void)
```

```
{
   char *help = ptr;
   char *nptr = NULL;
   char *newp = NULL;
   int i = len (ptr);

   //-------------------------------------------------------
   // Wenn keine Daten vorhanden, kann man nichts tun
   //-------------------------------------------------------
   if ((!ptr) || (i < 0)) return (NULL);
   while ((help [i] < 33) && (i >= 0)) i--;
   help[i+1] = 0;

   //-------------------------------------------------------
   // Wenn die Zeichenkette verwertbare Zeichen enthält, dann
   // neuen Speicher für die Restlänge (ohne folgende
   // Whitespaces) anfordern
   //-------------------------------------------------------
   nptr = new char [len (ptr) + 1];

   //-------------------------------------------------------
   // Kein Speicher verfügba? dann alten String zurück
   //-------------------------------------------------------
   if (!nptr) return (ptr);

   //-------------------------------------------------------
   // Wenn Speicher verfügbar, dann Zeichenweise kopieren und
   // mit String-Ende-Zeichen abschließen
   //-------------------------------------------------------
   for (newp = nptr, help = ptr; *help; help++, newp++)
      *newp = *help;
   *newp = 0;

   //-------------------------------------------------------
   // Nach dem kopieren den alten Speicher wieder freigeben
   // und den internen Stringpointer auf den neuen
   // Speicherbereich zeigen
   // lassen. Zum Schluß  noch den Stringpointer zurückgeben
   //-------------------------------------------------------
   delete ptr;
   ptr = nptr;
   return (ptr);
}

//=========================================================
// Führende und abschließende Leerzeichen wegschneiden
//=========================================================
char *dynstr::alltrim (void)
{
   rtrim();
   ltrim();
   return (ptr);
}

//=========================================================
// Setzen des Strings
```

```
//=============================================================
void dynstr::set (char *toset)
{
   int size = len (toset) + 1;
   char *help = NULL;

   //-------------------------------------------------------
   // Wenn schon Daten vorhanden sind, müssen diese zunächst
   // freigegeben werden
   //-------------------------------------------------------
   if (ptr) delete ptr;

   //-------------------------------------------------------
   // neuen Speicher in der benötigten Größe besorgen
   //-------------------------------------------------------
   ptr = new char [size];

   //-------------------------------------------------------
   // Wenn wir den Speicher bekommen haben, aus Parameter
   // herüberkopieren
   //-------------------------------------------------------
   if (!ptr) return;
   for (help=ptr;*toset;help++,toset++) *help = *toset;

   //-------------------------------------------------------
   // und zum Schluß das Stringende setzen
   //-------------------------------------------------------
   ptr [size-1] = 0;
}

//=============================================================
// Anhängen an den gespeicherten String
//=============================================================
void dynstr::concat (char *toadd)
{
   int  size;
   char *nptr = NULL, *nhelp = NULL, *ohelp = NULL;

   //-------------------------------------------------------
   // Wenn keine Daten vorhanden, kann man nichts tun
   //-------------------------------------------------------
   if (!ptr) return;

   //-------------------------------------------------------
   // der benöigte Speicherplatz ist die Länge des alten
   // Strings plus die Länge des neuen Strings
   //-------------------------------------------------------
   size = len (ptr) + len (toadd) + 1;

   //-------------------------------------------------------
   // Platz für den zusammengesetzten String besorgen
   //-------------------------------------------------------
   nptr = new char [size];

   //-------------------------------------------------------
   // Wenn nicht genügend Speicher vorhanden ist, Adios
```

```
   //-------------------------------------------------------
   if (!nptr) return;

   //-------------------------------------------------------
   // zunächst den alten String kopieren, dann den neuen Teil
   // dahinter und zuletzt mit String-Ende-Symbol beenden
   //-------------------------------------------------------
   for (nhelp=nptr,ohelp=ptr;*(ohelp);nhelp++,ohelp++)
       *nhelp=*ohelp;
   for (; *(toadd); nhelp++, toadd++) *nhelp = *toadd;
   *nhelp = 0;

   //-------------------------------------------------------
   // Nun noch den alten String freigeben und den internen
   // Stringpointer auf den neuen Speicherbereich zeigen
   // lassen
   //-------------------------------------------------------
   delete ptr;
   ptr = nptr;
}

//============================================================
// Länge eines Strings ermitteln
//============================================================
int dynstr::len (char *tocount) const
{
   char *help = tocount;

   //-------------------------------------------------------
   // Startadresse minus Endadresse ist natürlich die Größe
   // in Bytes (ohne String-Ende-Symbol)
   //-------------------------------------------------------
   for (; *help; help++);
   return ((int)(help-tocount));
}

//============================================================
// Zuweisungsoperator überladen, da sonst der Pointer kopiert
// wird und nicht der Inhalt des Objekts
//============================================================
char *dynstr::operator= (dynstr & tocopy)
{
   set (tocopy.ptr);
   return (ptr);
}

//============================================================
// Zuweisungsoperator überladen, da sonst der Pointer kopiert
// wird und nicht der Inhalt des Objekts
//============================================================
char *dynstr::operator= (char *tocopy)
{
   set (tocopy);
   return (ptr);
}
```

```
//=============================================================
// Vergleichsoperator überladen, da sonst die Stringadressen
// verglichen werden, nicht die Objektinhalte
//=============================================================
int dynstr::operator== (dynstr & tocheck)
{
   int nlen = len (ptr);
   if (nlen != len (tocheck.ptr)) return (0);

   while (nlen >= 0)
   {
      if (tocheck.ptr[nlen] != ptr[nlen]) return (0);
      nlen--;
   }
   return (1);
}

//=============================================================
// Vergleichsoperator überladen, da sonst die Stringadressen
// verglichen werden, nicht die Objektinhalte
//=============================================================
int dynstr::operator== (char *tocheck)
{
   int nlen = len (ptr);
   if (nlen != len (tocheck)) return (0);

   while (nlen >= 0)
   {
      if (tocheck[nlen] != ptr[nlen]) return (0);
      nlen--;
   }
   return (1);
}

//=============================================================
// Beispielprogramm
//=============================================================

int main (void)
{
   dynstr x;
   dynstr y;

   x.set (" Alltrim Test ");
   y.set (" Alltrim Test ");
   cout << "SET      [" <<x.get () << "]" << endl;

   cout << "Vergleich : " << (x == y) << endl;

   x.alltrim ();
   cout << "Vergleich : " << (x == y) << endl;
   cout << "Vergleich : " << (x=="Alltrim Test")<< endl;
   cout << "ALLTRIM  [" <<x.get () << "]" << endl;
   cout << (y = x) << endl;
   return (0);
}
```

26. TYPÄNDERUNGEN (CASTING)

Unter Casting versteht man in C eine explizit vorgenommene Typumwandlung durch den Programmierer. Im Allgemeinen werden Typumwandlungen implizit vorgenommen. Wenn z.B. eine Integervariable mit einer Variable vom Typ *float* verknüpft wird, so muss intern die Integervariable zunächst in das Fließkommaformat überführt werden. Das Ergebnis wird, abhängig vom Typ der Variablen auf die das Ergebnis zugewiesen wird, gegebenenfalls erneut umgewandelt.

Da Pointer unterschiedlichen Typs immer die gleiche Länge haben, sind eigentlich alle Zeiger einander zuweisbar. Da C jedoch die Pointerfortschaltung um ein Element durch den Inkrementoperator erlaubt, muss die Länge dieses Elementes bekannt sein. Daher gibt es in C nur typisierte Zeiger. Eine Zuweisung zwischen unterschiedlichen Zeigertypen wird daher zumindest zu einer Compilerwarnung führen, die besagt, dass zwischen den Pointern eine Typinkompatibilität besteht. Diese Warnung werden zur Fehlervermeidung ausgegeben (z.B. kommt es relativ häufig vor, dass man einen Zeiger auf Integer mit einer Integervariablen verwechselt).

Soll dennoch ein Pointer- oder Variablentyp in einen anderen überführt werden, ist dies durch ignorieren der Warnung (unsaubere Methode) oder mittels Casting möglich. Ein Casting wird durchgeführt, indem der Zieltyp in Klammern vor die umzuwandelnde Variable oder den Ausdruck gesetzt wird:

```
Syntax :
   (Zieltyp) Variable_oder_Ausdruck;
```

Sehr hilfreich ist das Casting auch, wenn eine Funktion (z.B. die Wurzelfunktion *sqrt*) einen Wert vom Typ *double* erwartet, die zu verarbeitende Variable jedoch z.B. Integerformat hat. Statt vorher den Wert auf eine Zwischenvariable zuzuweisen kann man auch mittels Casting den Parameter in das erforderliche Format umwandeln und spart dadurch Speicherplatz und Zeit.

```
//=============================================================
// Programm CASTING1.CPP
//=============================================================

#include <iostream>
#include <math.h>

using namespace std;

int  Zahlen [5] = {1, 2, 3, 4, 5};
long index;
```

```
void main (void)
{
   for (index=0; index<5; index++)
   {
      //-----------------------------------------------------
-
      // erzeugt bei einigen Compilern Warnings, wegen des
      // long als index
      //-----------------------------------------------------
-
      cout << (index+1) << ".te Zahl = " <<
Zahlen[(int)index]
           << endl;
   }
}
```

Soll das Casting zwischen Pointertypen erfolgen, so muss noch ein Verweisoperator ('*') angefügt werden:

```
//============================================================
// Programm CASTING2.CPP
//============================================================

#include <float.h>
#include <malloc.h>

double *Pointer;
int Size;

void main (void)
{
   Pointer = (double *) malloc (Size);
   free (Pointer);
}
```

Bei selbstgeschriebenen Klassen kann man eine abgeleitete Klasse problemlos auf eine Basisklasse casten, z.B. um eine Funktion aufzurufen, die einen Basistyp verlangt.

Aber Vorsicht! – wird das Objekt über *Call by Value* übergeben, so wird auch nur der Basisanteil kopiert. Unter Umständen führt des zu ungewollten Nebeneffekten. Dieses Problem wird auch *Slicing* genannt.

27. SICHERES PROGRAMMIEREN

Den Anwendungsentwickler vor versehentlich programmierten Abstürzen zu bewahren ist – mit Sicherheit nicht – die Stärke von C/C++.

Die Sprache selbst ist, insbesondere wenn man sich die Grundelemente betrachtet (die Anweisungen die bereits im K&R-C enthalten waren), primär auf Ausführungsgeschwindigkeit ausgelegt, nicht auf Sicherheit. Allein zur Geschwindigkeitssteigerung haben Kernighan & Ritchie eine ganze Reihe von Schlüsselbefehlen sehr dicht an den zugrundeliegenden Assemblerbefehlen orientiert (z.B. *switch*), die Typsicherheit unterlaufen (Gleichstellung von *char* und *int*) und Seiteneffekte in Kauf genommen (Zuweisungsketten wie a = b = c).

Dennoch ist es auch in C und C++ möglich, über bestimmte Mechanismen und Befehle, die Sicherheit von Programmen bereits bei der Entwicklung zu erhöhen und sich Fehlersuche ein wenig einfacher zu gestalten.

27.1. ZUSICHERUNGEN (ASSERT)

Ein *assert()* (eine Zusicherung) ist die einfachste Form die Programmsicherheit etwas zu verbessern. Die Zusicherung besteht im Wesentlichen aus einem Makro, welches eine *if* Anweisung enthält und – bei Verletzung der enthaltenen Schutzbedingung – das Programm beendet.

Das *assert()*-Makro ist in seiner Wirkungsweise eindeutig für den Zeitraum der Entwicklung gedacht, nicht für eine auszuliefernde Version.

Das Prinzip ist einfach, man übergibt dem Makro *assert()* lediglich einen logischen Ausdruck. Ist die Bedingung wahr (TRUE), so passiert nichts, das Programm wird fortgesetzt. Sobald die Auswertung des Ausdrucks jedoch FALSE ergibt, wird das Programm unter Nennung der fehlgeschlagenen Bedingung, des Moduls (d.h. Name der Quelltextdatei) und der Anweisungszeile (Quelltextzeile in der das *assert()* steht) abgebrochen.

Praktisch ist, dass die *assert()*-Anweisungen meist durch eine Compileroption oder einen globalen *#define* (bedingte Compilierung) abgeschaltet werden können. Die *assert*()-Anweisungen müssen aus dem fertigen Programm also nicht wieder entfernt werden.

Um *assert()* nutzen zu können muss die zugehörige Header-Datei eingebunden werden:

```
//=============================================================
// Programm ASSRTBSP.CPP
//=============================================================

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

using namespace std;

#define ARRAYSIZE 5

class Alpha
{
   public:
      void Test (void) { cout << "Alpha" << endl; }
};

//-------------------------------------------------------------
// Testprogramm
//-------------------------------------------------------------
void main (void)
{
   char sEingabe [80];
   int i;
   int nIndex;
   int nArray [ARRAYSIZE];
   Alpha *pAlpha = NULL;

   for (i=0; i<ARRAYSIZE; i++)
   {
      nArray [i] = i+1;
   }

   cout << "Bitte angeben, welches Arrayelement gelesen "
           "werden soll: ";
   cin.getline (sEingabe, 80);
   nIndex = atoi (sEingabe);

   //-----------------------------------------------------------
   // Zusicherungen. Die Asserts kann man auch zusammenfassen
   // zu:
   // assert ((nIndex < ARRAYSIZE) && (nIndex >= 0))
   // dann weiß man aber im Falle eines Fehlers nicht, welche
   // der beiden Bedingungen verletzt wurde
   //-----------------------------------------------------------
   assert (nIndex < ARRAYSIZE);
   assert (nIndex >= 0);

   cout << "Arrayelement : " << nArray [nIndex] << endl;

   //-----------------------------------------------------------
   // typische Zusicherung: der Pointer darf noch einen Wert
   // haben
   //-----------------------------------------------------------
   assert (NULL == pAlpha);
```

```
   cout << "Geben Sie 'T' für eine korrekte Objektallokation
"
           "ein - oder einen anderen Buchstaben um einen "
           "ASSERT auszulösen :";
   cin.getline (sEingabe, 80);
   switch (sEingabe[0])
   {
      case 'T' :
      case 't' : pAlpha = new Alpha;
   }

   //---------------------------------------------------------
   // typische Zusicherung: der Pointer muß einen Wert haben
   //---------------------------------------------------------
   assert (pAlpha);

   pAlpha->Test();
   delete pAlpha;
   pAlpha = NULL;
}
```

27.2. EXCEPTION HANDLING (TRY, CATCH UND THROW)

Mit den Exceptions (Ausnahmen) hat man unter C++ einen verbindlichen Mechanismus der Fehlerverarbeitung zur Laufzeit eingeführt. Einige Hersteller bieten eine entsprechende Funktionalität auch unter ANSI-C an.

Unter C++ muss, um die sogenannte Ausnahmebehandlung zu ermöglichen, die Headerdatei *excpt.h* eingebunden werden, eventuell zusätzlich die Datei *except.h*.

Achtung:
Behandelt werden immer nur interne Exceptions. Von außen ausgelöset Exceptions (z.B. Programmabbruch durch CTRL-C bzw. STRG-C) lösen keine Exception aus.

Die Ausnahmebehandlung soll verhindern, dass ein Programm (wie z.B. bei *assert*) einfach abgebrochen wird, wenn eine Fehlersituation auftritt. In fast allen Programmen gibt es Funktionen und Objekte, die einen Fehlerstatus zurückgeben und verarbeiten können (meist realisiert über Returncodes oder globale Variablen). Allerdings obliegt es dann dem Anwendungsprogrammierer, ob er den Rückgabewert auch beachtet oder den Fehlerstatus des Objektes abfragt. Ausgehend von der Vermutung, dass die Disziplin des durchschnittlichen Programmierers in dieser Hinsicht eher als dürftig zu betrachten ist, bietet C++ mit dem Exceptionhandling die Möglichkeit eine vorgesehene Ausnahmebehandlung (sofern es eine geben kann) automatisch aufzurufen. Tritt zur Laufzeit eine Fehlersituation auf, so wird die weitere Kontrolle an

einen dafür vorgesehenen Programmteil übergeben, der speziell für die Bearbeitung dieses Problems entworfen wurde. Im einfachsten Fall wird eine Fehlermeldung ausgegeben und das Programm (wie bei *assert*) beendet. Wenn möglich sollte das Programm jedoch versuchen zu retten was zu retten ist – und ggf. sogar Korrekturen an den übergebenen Daten vornehmen, wenn diese Veränderungen eindeutig, nachvollziehbar und gewünscht sind.

Die Vorteile dieser Vorgehensweise sind:

- Die Verarbeitung einer Funktion oder Methode wird nicht einfach abgebrochen und (wie bei `goto`) auf ein Fehlerprogramm verzweigt (mit allen negativen Konsequenzen für den Call-Stack), sondern das Laufzeitsystem kopiert den mit `throw` erzeugten Ausdruck an eine neutrale Stelle im System und versucht dann sauber den `try`-Block zu beenden (löscht also auch alle lokalen Objekte wieder). Danach ruft das System die zugehörigen `catch()`-Blöcke auf. Sind mehrere `try-catch`-Blöcke ineinander verschachtelt, so wird der nächstliegende mit dem passenden Ausdruckstyp angesprungen.
- Nach der Ausführung der Fehlerbehandlungsroutine wird das Programm nicht abgebrochen, sondern an der nächsten Anweisung Stelle hinter der letzten Behandlungsroutine (`catch`) des `try` fortgesetzt.
- Der objektorientierte Ansatz (das Objekt selbst „weiß“ alles über sich) bleibt erhalten – im Gegensatz zu einer Fehlerbehandlung außerhalb des Objektes (wie z.B. im klassischen Vorgehensfall bei Fehler-Returncodes).
- Eine Fehlerbehandlung kann nicht von einem anderen Anwendungsprogrammierer vergessen oder ignoriert werden (zumindest wenn man sie in das Objekt integriert hat).
- Die Fehlerverwaltung erfolgt zentral, d.h. es gibt nicht diverse Stellen im Sourcecode, an denen immer wieder die gleichen oder sehr ähnliche Fehlermeldungen stehen.
- Fehlerbedingungen können geändert und die Ausnahmebehandlung verbessert werden, ohne dass andere Programmteile ebenfalls abgeändert werden müssen.

Wie man sieht, kann das Exception-Handling – richtig eingesetzt – den modularen Programmaufbau unterstützen.

Für die Ausnahmebehandlung wurden in C++ insgesamt drei neue Schlüsselworte eingeführt: *try*, *throw* und *catch*.

try Der Anweisung *try* folgt immer ein Anweisungsblock. Dieser Block stellt das überwachte Codestück dar, für den eine mögliche Exception abgefangen werden soll. Beispielsweise der Zugriff auf eine Datenbank oder die Berechnung einer Funktion. Auf den *try*-Block folgen unmittelbar ein oder mehrere *catch()*-Blöcke

catch Der Anweisung *catch()* folgt immer ein Anweisungsblock. Dieser Block stellt den sogenannten Exception-Handler dar, eine Reihe von Anweisungen, die durchlaufen werden, wenn im überwachten Block eine Exception ausgelöst wurde

throw Auch als Auslösepunkt der Ausnahmebehandlung bezeichnet. Der Anweisung *throw* folgt immer ein Ausdruck. Dieser Ausdruck kann von einem beliebigen, eindeutigen Typ sein (*int*, *double*, Stringpointer etc.). Der Typ des Ausdrucks bestimmt, welcher *catch()*-Block aufgerufen wird. Dieser Ausdruck kann im *catch()*-Block ausgewertet werden (s.u.). Wird zum angegebenen *throw*-Ausdruck kein entsprechender *catch()*-Block gefunden, so erfolgt ein normaler Programmabbruch. Ein Standard *catch()*-Block für einen try (ähnlich dem *default* bei einer *switch*-Anweisung) kann mit *catch(...)* definiert werden.

Das folgende (noch sehr einfache) Beispiel zeigt den Umgang mit Exceptions anhand der Fakultätsfunktion. Aus demonstrativen Zwecken ist die Ausnahmebehandlung hier geteilt, beide *throw* Anweisungen gehören natürlich in die Funktion:

```
//============================================================
// Programm TRYBSP1.CPP
//============================================================

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

using namespace std;

//------------------------------------------------------------
// Funktion mit Exceptionhandling
//------------------------------------------------------------
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;
```

```
   //-------------------------------------------------------
   // Fakultät von negativen Werten ist nicht definiert:
   // Exception auslösen
   //-------------------------------------------------------
   if (Param < 0) throw 1;

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}

//-------------------------------------------------------------
// Testprogramm
//-------------------------------------------------------------
void main (void)
{
   char sEingabe [80];
   int fak = 0;

   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      //----------------------------------------------------
      // Die Exception kann hier ODER in einer Unterfunktion
      // ausgelöst werden
      //----------------------------------------------------
      if (fak > 50) throw 1;
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird. catch (...) ist der
   // Handler für jede beliebige C++ Exception
   //-------------------------------------------------------
   catch ( ... )
   {
      cout << "Es wurde eine Exception ausgelöst";
   }

   cout << endl << "Programmende" << endl;
}
```

Das oben gezeigte Beispiel löst eine korrekte Ausnahmebehandlung aus, die aber keine genauen Angaben über den Fehler zulässt, da immer nur die Ausgabe „Es wurde eine Exception ausgelöst“ erscheint.

Der *catch*-Block sollte daher besser lauten:

```
catch (int nError)
{
   cout << "Es wurde eine Exception ausgelöst : " << nError;
}
```

Nun hat man die Möglichkeit den beiden Fehlerzuständen (Param ist kleiner als Null oder größer als 50) unterschiedliche Fehlernummern zuzuweisen und erhält diese Fehlernummern in der Ausgabe:

```
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;

   if (Param < 0)  throw 1;
   if (Param > 50) throw 2;

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}
```

Noch eleganter ist es, statt einer Fehlernummer einen Fehlertext auszugeben:

```
//============================================================
// Programm TRYBSP2.CPP
//============================================================

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

using namespace std;

//------------------------------------------------------------
// Funktion mit Exceptionhandling
//------------------------------------------------------------
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;

   //---------------------------------------------------------
   // Exception auslösen
   //---------------------------------------------------------
   if (Param > 50) throw "Parameter zu groß";
   if (Param < 0)  throw "negative Werte nicht erlaubt";

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}
```

```
//-------------------------------------------------------
// Testprogramm
//-------------------------------------------------------
void main (void)
{
   char sEingabe [80];
   int fak = 0;

   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird
   //-------------------------------------------------------
   catch (char *pError)
   {
      cout << pError;
   }

   cout << endl << "Programmende" << endl;
}
```

Achtung:
Da try immer einen nachstehenden Block benötigt, eignet sich der try-catch Mechanismus nicht um eine Objektdefinition (Aufruf des Objekt-Constructors) zu kapseln, denn durch den try-Block wird das Objekt nur lokal erzeugt und am Ende des Blocks wieder zerstört.

27.2.1. UNLÖSBARE FEHLERSITUATIONEN

In den bisherigen Beispielen zu *try-catch*-Blöcken hat das Programm stets die Abarbeitung sauber beendet, d.h. die letzte Anweisung im Programm wurde immer ausgeführt:

```
cout << endl << "Programmende" << endl;
```

Wenn zur Laufzeit aber eine Fehlersituation auftritt, bei der eine sinnvolle Fortführung des Programms nicht mehr möglich ist, sollte das Programm im *catch()*-Block beendet werden.

Dazu kann natürlich die Standardfunktion *exit()* aufgerufen werden, was auch problemlos funktioniert. Die Funktion *exit()* ist aber eigentlich für eine korrekte Beendung eines Programms mit einem speziellen *return*-Wert vorgesehen (wie in der Batch-Programmierung üblich). Dieser *return*-Wert kann zwar auch einen Fehlerwert darstellen, *try-catch()* geht aber von einem unbeabsichtigten Programmabbruch aus.

Um einen „anormalen“ Programmabbruch durchzuführen stellt C++ die Funktion *terminate()* bereit, welche die Fehlermeldung „*abnormal program termination*“ erzeugt.

Der folgende Programmcode erweitert das Beispielprogramm um einen weiteren *catch()*-Block, indem unterschiedliche *throw*-Ausdrücke erzeugt werden. Man beachte, dass die letzte Zeile des Programms im Falle des terminierenden *catch()*-Blocks nicht mehr erreicht wird.

Um *terminate()* verwenden zu können muss die Header-Datei *except.h* eingebunden werden:

```
//============================================================
// Programm TRYBSP3.CPP
//============================================================
#include <excpt.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

//------------------------------------------------------------
// Funktion mit Exceptionhandling
//------------------------------------------------------------
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;

   //---------------------------------------------------------
   // Exception auslösen
   //---------------------------------------------------------
   if (Param > 50) throw "Parameter zu groß";
   if (Param < 0)  throw 1;

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}

//------------------------------------------------------------
// Testprogramm
//------------------------------------------------------------
void main (void)
{
   char sEingabe [80];
```

```
   int fak = 0;

   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird
   //-------------------------------------------------------
   catch (char *pError)
   {
      cout << pError;
   }
   catch (int nError)
   {
      cout << "Schwerer Fehler: negative Fakultät";
      terminate();
   }

   cout << endl << "Programmende" << endl;
}
```

Achtung:
Ein Programm sollte immer alle Exceptions die es auslöst (definiert durch die verschiedenen Variablentypen der throw-Anweisungen) auch abarbeiten, sonst wird das Programm mit einer „abnormal program termination“ beendet

Die so erlangte Kontrolle über den anormalen Programmabbruch hat einen enormen Vorteil.

Unterstellt man ein komplexes Softwarepaket, z.B. basierend auf einer Datenbank, so muss bei jedem Programmabbruch geprüft werden, ob z.B. die im Hintergrund laufende Datenbank noch beendet werden muss – oder noch dynamisch belegter Speicher freizugeben ist.

Trotz aller Mühe wird man sicherlich den einen oder anderen Fall übersehen, der dann zu einer Exception führt, die man in dieser Form nicht vorgesehen oder nicht abgefangen hat. Anstatt das Programm nun direkt nun direkt auf den *terminate*() Befehl laufen zu lassen, kann man das Laufzeitsystem den Umweg über eine eigene Routine gehen lassen, indem man eigene *terminate*() und *unexpected*() Funktion angibt

(s.u.). In diesen Funktionen können alle nötigen Nacharbeiten geleistet werden, wie. z.B. das Beenden der oben genannten Datenbank:

```
//=============================================================
// Programm TRYBSP4.CPP
//=============================================================
#include <excpt.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

//-------------------------------------------------------------
// Eigene terminate-function
//-------------------------------------------------------------
void myTerminate ()
{
   cout << "myTerminate";
   abort();
}

void myUnexpected ()
{
   cout << "unerwartete Exception";
   abort();
}

//-------------------------------------------------------------
// Funktion mit Exceptionhandling
//-------------------------------------------------------------
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;

   //----------------------------------------------------------
   // Exception auslösen
   //----------------------------------------------------------
   if (Param > 100) throw 'P';
   if (Param > 50) throw "Parameter zu groß";
   if (Param < 0)  throw 1;

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}

//-------------------------------------------------------------
// Testprogramm
//-------------------------------------------------------------
void main (void)
{
   char sEingabe [80];
   int fak = 0;
   terminate_function  OldTerminate;
   unexpected_function OldUnexpected;
```

```
   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Eigene Termination und Unexpected
   //-------------------------------------------------------
   OldTerminate  = set_terminate  (myTerminate);
   OldUnexpected = set_unexpected (myUnexpected);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird
   //-------------------------------------------------------
   catch (char *pError)
   {
      cout << pError;
   }
   catch (int nError)
   {
      cout << "Schwerer Fehler: negative Fakultät";
      terminate();
   }
   //-------------------------------------------------------
   // alle anderen Exceptiontypen laufen jetzt in
   // myTerminate oder myUnexpected
   //-------------------------------------------------------

   cout << endl << "Programmende" << endl;
   set_terminate  (OldTerminate);
   set_unexpected (OldUnexpected);
}
```

Die Verwendung von *abort()* statt von *terminate()* bewirkt, dass die Standardmeldung „*abnormal program termination*“ nicht erscheint. Im Beispiel wird stattdessen eine eigene Fehlermeldung ausgegeben, die Standardmeldung aus der *terminate()* Funktion ist also überflüssig. Auch *terminate()* tut nichts weiter, als die Meldung auszugeben und *abort()* aufzurufen.

Achtung:
Im Aufruf einer eigenen terminate() oder unexpected() Funktion muß das Programm beendet werden, da das Programm sich in einem undefinierten Zustand befindet.

27.2.2. MEHRERE CATCH()-BLÖCKE

Das vorangegangene Beispielprogramm zeigt den Umgang mit unterschiedlichen *catch()*-Blöcken zu einem *try*-Block. Der C++ Compiler verwendet ein Overloading-Verfahren, um die *catch()*-Blöcke voneinander zu unterscheiden. Es wird immer nur der *catch()*-Block aufgerufen, der im Typ zum auslösenden *throw*-Ausdruck passt – oder der durch eine erlaubte Typumwandlung erreicht werden kann. Dies ist z.B. der Fall, wenn *throw* einen *int* erzeugt, aber nur ein *catch()* für *long* zur Verfügung steht. Da der Compiler weiß, wie *int* in *long* umzuwandeln ist, kann der *catch()*-Block für *long* aufgerufen werden.

Die *catch()*-Blöcke werden in der Reihenfolge ihrer Definition nach dem *try*-Block abgearbeitet. Die Exception gilt als abgefangen, sobald eine passende Routine zum Typ des *throw* gefunden wurde, weitere *catch()* Anweisungen werden dann nicht mehr geprüft.

Die Reihenfolge der Definitionen der *catch()*-Anweisungen erfordert daher unter Umständen ein wenig Fingerspitzengefühl.

Da der Block *catch(...)* jede Exception – unabhängig vom Typ des *throw*-Ausdrucks – behandelt, muss er zwangsläufig, damit die spezifischen Blöcke aufgerufen werden können, als letzte catch()-Anweisung stehen.

27.2.3. EXCEPTIONS MIT SPEZIELLEN EXCEPTIONKLASSEN

In den bisherigen Beispielen wurden im *throw* nur Ausdrücke verwendet, die interne Datentypen darstellen (*int*, *char**). Bei komplexen Programmen ist zu erwarten, dass dazu eine Vielzahl von Exceptions bestehen, die alle in einen gemeinsamen *catch()*-Block laufen, wo sie behandelt werden sollen. Beispielsweise könnte nicht nur die Fakultätsfunktion eine Exception 1 auslösen, sondern auch eine im gleichen *try*-Block aufgerufene andere Funktion. Letztlich ist es dann unmöglich diese Exceptions voneinander zu unterscheiden und im Programm noch korrekt zu reagieren (z.B. eine konkrete Fehlermeldung auszugeben).

Aus diesem Grunde erzeugt man zumeist spezielle Exceptionklassen, die nur dazu dienen, den richtigen *catch()* aufzurufen:

```
//============================================================
// Programm TRYBSP5.CPP
//============================================================
#include <excpt.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

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

```
// Fakultäts-Fehler-Objekt
//------------------------------------------------------------
class FakulError
{
   protected:
      int error;

   public:
      FakulError (int i = 0) {error = i;}
      void GetError (void);
};

//------------------------------------------------------------
// Fakultäts-Fehlermethode
//------------------------------------------------------------
void FakulError::GetError (void)
{
   cout << "Fakultät: ";
   switch (error)
   {
      case 1 : cout << "negativer Wert";
               break;
      case 2 : cout << "Parameter zu groß";
               break;
      default: cout << "unbekannter Fehler";
               break;
   }
}

//------------------------------------------------------------
// Funktion mit Exceptionhandling
//------------------------------------------------------------
double fnFakultaet (int Param)
{
   double Ergebnis = 1.0;

   //---------------------------------------------------------
   // Exception auslösen
   //---------------------------------------------------------
   if (Param > 50) throw FakulError(2);
   if (Param < 0)  throw FakulError(1);

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}

//------------------------------------------------------------
// Testprogramm
//------------------------------------------------------------
void main (void)
{
   char sEingabe [80];
   int fak = 0;
```

```
   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird
   //-------------------------------------------------------
   catch (FakulError& x)
   {
      x.GetError();
   }

   cout << endl << "Programmende" << endl;
}
```

Die Ausgestaltung eines Exceptionobjektes unterliegt den normalen Regeln für Klassen, auch hier wäre es z.B. denkbar eine Exception-Basisklasse zu implementieren, von der alle Exception-Objekte erben, so daß alle Ausnahmebehandlungen gleichartig funktionieren.

Da die Auslösung einer Exception immer eine lokale Kopie des Objektes erzeugt, kann es unter Umständen sinnvoll sein einer Copyconstructor zu schreiben. Unvermeidlich wird ein solcher Contructor, wenn das Exceptionobjekt dynamische allozierte Daten enthält.

27.2.4. UNTERPROGRAMME MIT VEREINBARTEN EXCEPTIONS

Möchte man die möglichen Exceptions, die ein Unterprogramm auslösen darf vorab festlegen, so gibt es in C++ die Möglichkeit dies im Kopf der Funktion/Methode – nach der Parameterleiste – ausdrücklich zu vereinbaren:

```
//-------------------------------------------------------
// Funktion mit Exceptionhandling
//-------------------------------------------------------
double fnFakultaet (int Param) throw (FakulError)
{
   double Ergebnis = 1.0;

   //-------------------------------------------------------
   // Exception auslösen
   //-------------------------------------------------------
   if (Param > 100) throw 100;
```

```
   if (Param > 50)  throw FakulError(2);
   if (Param < 0)   throw FakulError(1);

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}
```

Wird für die oben gezeigte Funktion nun als Parameter ein Wert größer als 100 übergeben, so wird eine *int*-Exception ausgelöst, die aber nicht vereinbart wurde.

Daher wird die Funktion aufgerufen, die für eine *unexpected*-Exception vorgesehen ist. Im Normalfall ist dies die Funktion *terminate()*, es sei denn es wurde eine gesonderte Funktionalität bereitgestellt.

Wurde für diesen Fall eine Routine mit *set_unexpected* vereinbart (siehe Beispiel oben), dann wird die angegebene Funktion aufgerufen, ansonsten – sofern vereinbart – das mit *set_terminate* vereinbarte Unterprogramm. Sind beide Routinen nicht vom Entwickler mit eigenen Unterprogrammen belegt worden, wird *terminate()* aufgerufen

Wenn eine Funktion in der Lage sein soll, mehr als eine Art von Exceptions auszulösen, werden diese einfach in der *throw*-Vereinbarung aufgezählt:

```
//============================================================
// Programm TRYBSP6.CPP
//============================================================
#include <excpt.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

//------------------------------------------------------------
// Fakultäts-Fehler-Objekt
//------------------------------------------------------------
class FakulError
{
   protected:
      int error;

   public:
      FakulError (int i = 0) {error = i;}
      void GetError (void);
};

//------------------------------------------------------------
// Fakultäts-Fehlermethode
//------------------------------------------------------------
```

```
void FakulError::GetError (void)
{
   cout << "Fakultät: ";
   switch (error)
   {
      case 1 : cout << "negativer Wert";
               break;
      case 2 : cout << "Parameter zu groß";
               break;
      default: cout << "unbekannter Fehler";
               break;
   }
}

//------------------------------------------------------------
// Eigene terminate-function
//------------------------------------------------------------
void myTerminate ()
{
   cout << "myTerminate";
   abort();
}

//------------------------------------------------------------
// Eigene unexpected-function
//------------------------------------------------------------
void myUnexpected ()
{
   cout << "unerwartete Exception";
   abort();
}

//------------------------------------------------------------
// Funktion mit Exceptionhandling
//------------------------------------------------------------
double fnFakultaet (int Param) throw (FakulError, int)
{
   double Ergebnis = 1.0;

   //---------------------------------------------------------
   // Exception auslösen
   //---------------------------------------------------------
   if (Param > 200) throw "P";  // erzeugt unexpected Except.
   if (Param > 100) throw 100;
   if (Param > 50)  throw FakulError(2);
   if (Param < 0)   throw FakulError(1);

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}

//------------------------------------------------------------
// Testprogramm
//------------------------------------------------------------
```

```
void main (void)
{
   char sEingabe [80];
   int fak = 0;

   //-------------------------------------------------------
   // Eigene Termination und Unexpected
   //-------------------------------------------------------
   terminate_function  OldTerminate;
   unexpected_function OldUnexpected;

   OldUnexpected = set_unexpected (myUnexpected);
   OldTerminate  = set_terminate  (myTerminate);

   cout << "Parameter : ";
   cin.getline (sEingabe, 80);
   fak = atoi (sEingabe);

   //-------------------------------------------------------
   // Die folgenden Codezeilen überwachen (try-Block)
   //-------------------------------------------------------
   try
   {
      cout << fnFakultaet (fak) << endl;
   }
   //-------------------------------------------------------
   // Exceptionhandler, wird aufgerufen, wenn im try-Block
   // eine Exception ausgelöst wird
   //-------------------------------------------------------
   catch (FakulError& x)
   {
      x.GetError();
   }
   catch (int x)
   {
      cout << "integer Exception";
   }

   cout << endl << "Programmende" << endl;
   set_terminate  (OldTerminate);
   set_unexpected (OldUnexpected);
}
```

Wenn eine Funktion keine definierten Exceptions auslösen soll, übergibt man eine leere *throw*-Liste:

```
//-------------------------------------------------------
// Funktion mit leerer Exceptionhandling-Liste
//-------------------------------------------------------
double fnFakultaet (int Param) throw ()
{
   double Ergebnis = 1.0;

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

```
   // Exception auslösen
   //-----------------------------------------------------
   if (Param > 100) throw 100;
   if (Param > 50)  throw FakulError(2);
   if (Param < 0)   throw FakulError(1);

   if (0 == Param) Param = 1;
   for (int i=Param; i>1; i--) Ergebnis *= i;

   return Ergebnis;
}
```

Jede Exception, die jetzt innerhalb der Funktion ausgelöst wird, gilt als *unexpected()* und läuft in eines der oben beschriebenen Unterprogramme.

28. DATEIEN UND STREAMS

C++ kennt drei Klassen, die einen Stream-Zugriff auf Dateien ermöglichen – *ifstream*, *ofstream* und *fstream*.

Alle drei Klassen werden in der Headerdatei <*fstream*> deklariert. Die Aufteilung ist ähnlich wie bei der klassischen Methode über *FILE* und dem Zugriffsparameter, welcher festlegt, ob man lesend, schreibend oder mit beiden Formen zugreifen kann.

Dateien, die über einen *ifstream* geöffnet werden, sind reine Eingabeströme, genau wie *istream* bei der Tastatureingabe.
Entsprechend sind *ofstream*-Objekte reine Ausgabeströme (wie *ostream*). Ziel und Quelle sind allerdings nicht Tastatur und Bildschirm sondern Dateien. Eine Erweiterung gegenüber der einfachen Stream-IO ist die Klasse *fstream*, die sowohl Ein- wie Ausgabe auf der Datei zuläßt, bei Bedarf also die Eigenschaften von *ifstream* und *ofstream* kombiniert. Da beide Gruppen (*ostream* und *istream*, wie auch *ofstream*, *ifstream* und *fstream*) von der gleichen Basisklasse (*ios*) erben, sind ihre Methoden und Operatoren weitgehend identisch.
So erfolgt auch hier die Ein- bzw. Ausgabe mit den Operatoren >> und <<, jeweils bezogen auf die aktuelle Position innerhalb der Datei. In den folgenden Abschnitten stehen daher mehr die Erweiterungen und Abweichungen im Vordergrund. Bei gleichem Verhalten oder gleichartiger Handhabung wird auf das entsprechende Kapitel bei der Standard-IO verwiesen.

28.1. ÖFFNEN UND SCHLIESSEN VON DATEISTREAMS

Genau wie die *FILE*-Strukturen beim Dateihandling unter ANSI-C sind die Datei-Streams logische Dateien, die zunächst mit physikalischen Dateien verbunden weden müssen. Praktischerweise kann man dies gleich bei der Deklaration des Streamobjekts erledigen, da es einen entsprechenden Constructor gibt:

```
Syntax (Constructoren):

1) ofstream streamvar(char *name,int mode,int prot);
   ifstream streamvar(char *name,int mode,int prot);
   fstream  streamvar(char *name,int mode,int prot);

2) ofstream streamvar (char *name, int mode);
   ifstream streamvar (char *name, int mode);
   fstream  streamvar (char *name, int mode);

3) ofstream streamvar (char *name);
   ifstream streamvar (char *name);

4) ofstream streamvar (void);
   ifstream streamvar (void);
```

```
   fstream  streamvar (void);

5) ofstream streamvar (int fd);
   ifstream streamvar (int fd);
   fstream  streamvar (int fd);

6) ofstream streamvar(int fd,char *buf,int len);
   ifstream streamvar(int fd,char *buf,int len);
   fstream  streamvar(int fd,char *buf,int len);
```

1. Der erste Constructor öffnet gleichzeitig die angegebene Datei im angegebenen Modus und mit den in *prot* aufgeführten Zugriffsrechten. Letztere sind Compiler und Implementations-spezifisch (ggf. auch abhängig vom Betriebssystem). Sie müssen daher im Handbuch des Compilers nachgeschlagen werden. Es ist daher für portierbare Programme günstiger den zweiten Consructor zu verwenden und die Voreinstellung des Parameters *prot* zu nutzen. Die nachstehenden Modi sind Teil der Klasse *ios*, der Basisklasse zu *ifstream*, *ofstream* und *fstream*.

2. Der zweite Constructor verwendet für den Parameter *prot* die (compilerspezifische) Voreinstellung, ermöglicht es dem Entwickler jedoch das Verhalten durch Angabe von Flags in weiten Teilen zu bestimmen.

3. Der dritte Constructor verwendet für beide Parameter (*prot* und *mode*) die entsprechenden Voreinstellungen. Die Voreinstellung von *mode* für ein *ofstream*-Objekt ist *ios::out* und für *ifstream*-Objekte *ios::in*. Dieser Constructor existiert für *fstream* nicht, da für diese Streamklasse festgelegt werden muß, ob eine Datei zum Lesen (*ios::in*), Schreiben (*ios::out*) oder für Lesen/Schreiben (*ios::in* | *ios::out*) geöffnet wird.

4. Der vierte Construktor erzeugt einen Stream, der noch mit keiner physikalischen Datei verbunden ist (z.B. weil der Anwender den Namen der Datei erst noch eingeben soll). Das Verbinden mit der physikalischen Datei kann später mit der Methode *open* (siehe unten) erfolgen.

5. Der fünfte Constructor setzt voraus, daß eine physikalische Datei schon mit einem ANSI-C Filepointer geöffnet wurde (*FILE*). Dieser kann dann als Grundlage für ein Streamobjekt dienen. An diesem Umstand kann man erkennen, daß die interne Repräsentation einer Datei sich in C und C++ nicht grundlegend unterscheidet. In C++ wird die *FILE*-Struktur aus C lediglich in einer Klasse gekapselt und um gängige Öffnungsvarianten erweitert, was den Zugriff ein wenig erleichtert.

6. Der sechste Constructor setzt ebenfalls voraus, daß eine physikalische Datei schon mit einem ANSI-C Filepointer geöffnet wurde (*FILE*). Im Gegensatz zum 5. Constructor kann hier zusätzlich ein Pufferbereich für das IO-Handling angegeben werden, der somit von den Standardeinstellungen abweichen kann.

Modusangaben für Dateistreams		
Modus	*Flagwert*	*Bedeutung*
ios::in	1	Datei wird zum Lesen geöffnet, Voreinstellung für ifstream-Objekte. Kann auch (zusätzlich) mit fstream benutzt werden, um Lesen (und Schreiben) zu ermöglichen
ios::out	2	Datei wird zum Schreiben geöffnet, Voreinstellung für ofstream-Objekte. Kann auch (zusätzlich) mit fstream benutzt werden, um Schreiben (und Lesen) zu ermöglichen
ios::ate	4	Dateipointer wird zu Beginn am Ende der Datei positioniert (kann später überall stehen)
ios::app	8	Beim Schreiben werden alle neuen Daten an das Dateiende angehängt (append)
ios::trunc	16	Dateiinhalt löschen, wenn die Datei bereits existiert. Voreinstellung bei Modus out, wenn weder ate noch app zusätzlich spezifiziert ist (truncate)
ios::nocreate	32	Nur öffnen, wenn die Datei bereits existiert, sonst Fehler melden. Voreinstellung ist, daß nicht existente Dateien automatisch angelegt werden
ios::noreplace	64	Nicht öffnen wenn Datei schon existiert und weder ate noch app zusätzlich spezifiziert sind

Tabelle 4: Dateistream-Modi

Um das korrekte Schließen von Streams muß man sich in C++ weniger Gedanken machen als beim Dateihandling in ANSI-C, denn beim Aufruf des Destructors wird die noch offene Datei automatisch geschlossen. Auch die potentielle Fehlerquelle, daß man eine neue Datei öffnet, bevor die bisher verwendete Datei geschlossen wurde – und man die Dateipointer verliert, so daß man die Datei im aktuellen Programmdurchlauf nicht mehr schließen kann (eventuell sogar ein Neustart des Rechners notwendig wird), existiert nicht mehr. Bei der Stream-IO läßt der Rechner einen Dateiwechsel nicht zu, solange noch eine Datei geöffnet ist. Will man die aktuelle Datei schließen, um eine andere zu öffnen, so muß zunächst die *close*-Methode aufgerufen werden (siehe unten).

Das folgenden Beispiele zeigen die einfache Handhabung von Streams, beginnend mit einem reinen Eingabestream. Die verwendeten Methoden werden in den nachfolgenden Abschnitten erklärt:

```
//============================================================
// Programm FSTREAM1.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   char buff [300] = "";

   //------------------------------------------------------------
   // die folgenden beiden Zeilen sind inhaltlich identisch !
   //------------------------------------------------------------
// ifstream mystream ("c:\\autoexec.bat", ios::in);
   ifstream mystream ("c:\\autoexec.bat");

   cout << "---- AUSGABE AUTOEXEC.BAT ----" << endl;

   if (mystream.fail())
   {
      cout << "Fehler1" << endl;
      return;
   }

   while (!mystream.eof())
   {
      mystream.getline (buff, 300);
      cout << buff << endl;
   }
   mystream.close ();

   mystream.open ("c:\\config.sys");
   cout <<endl << "---- AUSGABE CONFIG.SYS ----" << endl;
```

```
   if (mystream.fail())
   {
      cout << "Fehler2" << endl;
      return;
   }

   //-------------------------------------------------------
   // Jetzt Dateiinhalt lesen und ausgeben
   //-------------------------------------------------------
   while (!mystream.eof())
   {
      mystream.getline (buff, 300);
      cout << buff << endl;
   }

   //-------------------------------------------------------
   // Kein close notwendig, Datei des Streams wird über
   // Destructor automatisch geschlossen, aber sauberer
   // ist es trotzdem !!!
   //-------------------------------------------------------
   mystream.close ();
}
```

Das nächste Beispiel zeigt die einfache Handhabung eines reinen *ofstream*-Ausgabestreams:

```
//============================================================
// Programm FSTREAM2.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   //-------------------------------------------------------
   // die folgenden beiden Zeilen sind inhaltlich identisch !
   //-------------------------------------------------------
// ofstream mystream ("c:\\test.txt", ios::out);
   ofstream mystream ("c:\\test.txt");

   if (mystream.fail())
   {
      cout << "Datei nicht zu öffnen !" << endl;
   }
   else
   {
      mystream << " Dies ist c:\\test.txt" << endl;
   }
   mystream.close ();

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

```
   // Jetzt noch eine Zeile ans Ende anhängen !
   //-------------------------------------------------------
   mystream.open ("c:\\test.txt", ios::out | ios::ate);

   if (mystream.fail())
   {
      cout << "Datei nicht zu öffnen !" << endl;
   }
   else
   {
      mystream << " Zweite Zeile !!" << endl;
      mystream << " Dritte Zeile !!" << endl;
      mystream << " Vierte Zeile !!" << endl;
      cout << "Datei geschrieben !" << endl;
   }
   mystream.close ();
}
```

Das dritte Beispiel zeigt, wie man mit einem *fstream* Lesen und Schreiben kann:

```
//============================================================
// Programm FSTREAM3.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   char buff [300];
   fstream mystream ("c:\\test.txt", ios::in | ios::out);

   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   mystream.getline (buff, 300);

   cout << buff << endl;
   mystream << "oops" << endl;
   mystream.close ();
}
```

28.2. DATEISTREAM-METHODEN

Um das Dateihandling von ANSI-C abzulösen sind natürlich auf den Streams eine ganze Reihe von Methoden definiert, die im folgenden kurz

behandelt werden. Die beschriebenen Methoden sind, sofern nicht anders beschrieben, für alle Datei-Streamklassen gültig.

28.2.1. ÖFFNEN EINER DATEI

```
Syntax:
   void streamvar.open (const char *name, int mode,
                        int prot);
   void streamvar.open (const char *name, int mode);
   void streamvar.open (const char *name);
```

Mit der *open*-Methode kann einem Stream eine physikalische Datei zugeordnet werden. Ob das Öffnen mit erfolg abgeschlossen wurde, kann man über die Methode *fail* (s.u.) erfragen, die einen Wert ungleich Null zurückgibt (den Fehlercode), wenn beim Öffnen der Datei Probleme auftraten.

Bevor man die Datei für eine Stream wechseln kann, muß man eine bereits geöffnete Datei explizit schließen (siehe Methode *close*). Die Parameter der *open*-Methode entsprechen den Parametern des Constructors:

```
#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   fstream mystream;

   mystream.open ("c:\\test.txt", ios::in | ios::out);

   // Fehler: mystream wird nicht geschlossen !!!
}
```

28.2.2. SCHLIESSEN EINER DATEI

```
Syntax:
   void streamvar.close (void);
```

Mit der Methode *close* wird die mit dem Stream verbundene, geöffnete Datei wieder geschlossen. Der Stream kann anschließend mit einer anderen Datei verbunden werden. Die *close*-Methode wird im Destructor der Streamklasse automatisch aufgerufen, so daß Dateien, die über

lokale Streamvariablen geöffnet wurden, nicht erst von Hand geschlossen werden müssen:

```
#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   fstream mystream;

   mystream.open ("c:\\test.txt", ios::in | ios::out);
   mystream.close ();
}
```

28.2.3. FEHLERSTATUS

```
Syntax:
   int streamvar.fail (void);
   int streamvar.good (void);
   int streamvar.bad (void);
```

Die Methode *fail* gibt darüber Auskunft, ob eine Dateioperation erfolgreich war. Trat ein Fehler auf, so gibt *fail* einen Wert ungleich Null zurück. Der Zurückgegebene Wert entspricht dann dem Zustand der Bits *ios::failbit*, *ios::badbit* und *ios::hardfail* aus der Membervariable *ios::state* (Member der Basisklasse *ios*).

```
//============================================================
// Programm FSTREAM5.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   fstream mystream ("c:\\test.txt", ios::in);

   if (mystream.fail())
   {
      cout << "Datei nicht zu öffnen !" << endl;
      return;
   }
   else
   {
      mystream.close();
   }
```

```
}
```

Die Methode *bad* gibt einen Wert ungleich Null zurück, falls bei der Abfrage von *ios::badbit* oder *ios::hardfail* in *ios::state* ein Fehler erkannt wird.

```
//=============================================================
// Programm FSTREAM5.CPP
//=============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   char buff [300];
   fstream mystream ("c:\\test.txt", ios::in | ios::out);

   while (1)  // ENDLOSSCHLEIFE !!!
   {
      mystream.getline (buff, 300);
      if (mystream.bad ())
      {
         cout << "Fehler bei Dateioperation" << endl;
         mystream.close ();
         return;
      }
      else
      {
         cout << buff << endl;
      }

      if (mystream.eof ())
      {
         mystream.close ();
         return;
      }
   }
}
```

Die Umkehrung zu *bad*, die Methode *good*, liefert einen Wert ungleich Null zurück, falls keines der Statusbits in *ios::state* gesetzt, also kein Fehler aufgetreten ist .

```
//=============================================================
// Programm FSTREAM6.CPP
//=============================================================

#include <iostream>
#include <fstream>
```

```
using namespace std;

void main (void)
{
   char buff [300];
   fstream mystream ("c:\\test.txt", ios::in | ios::out);

   while (1)  // ENDLOSSCHLEIFE !!!
   {
      mystream.getline (buff, 300);
      if (mystream.good ())
      {
         cout << buff << endl;
      }
      else
      {
         cout << "Fehler bei Dateioperation" << endl;
         mystream.close ();
         return;
      }

      if (mystream.eof ())
      {
         mystream.close ();
         return;
      }
   }
}
```

28.2.4. DATEIENDE ERKENNEN

```
Syntax:
   int streamvar.eof (void);
```

Die *eof*-Methode gibt einen Wert ungleich Null zurück, wenn das Ende der datei erreicht worden ist.

```
//============================================================
// Programm FSTREAM7.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   char buff [300] = "";
   fstream mystream ("c:\\autoexec.bat", ios::in);
```

```
   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   while (!mystream.eof())
   {
      mystream.getline (buff, 300);
      cout << buff << endl;
   }
   mystream.close ();
}
```

28.2.5. BUFFERANWEISUNGEN

Bei Bedarf kann man dem Stream-Constructor (siehe oben) einen eigenen IO-diesen Puffer bzw. ermöglichen eine nachträgliche Zuweisung. Wird als Puffergröße der Wert Null zugewiesen, so wird das ungepufferte IO-Handling eingeschaltet, welches erheblich langsamer ist, als die gepufferte Ein-/Ausgabe, gelegentlich aus Sicherheitsgründen aber angebracht erscheint.

Die Methode *flush* schreibt alle noch in den Puffern befindlichen Daten in die zugehörigen, physikalischen Dateien.

```
Syntax:
   filebuf *streamvar.rdbuf (void);
   void streamvar.setbuf (char *Bufferpointer,
                          int Bufferlen);
   void streamvar.flush (void);
```

```
//=============================================================
// Programm FSTREAM8.CPP
//=============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   char buff [300] = "";
   char *oldbuff = NULL;
   fstream mystream ("c:\\autoexec.bat", ios::in);

   cout << "Bufferadresse:" << (long)mystream.rdbuf()
        << endl;
   mystream.setbuf (buff, 300);
   cout << "Buffergröße  : " << sizeof(buff) << endl;
}
```

28.2.6. FORMATANWEISUNGEN

für die Stream-IO gibt es eine ganze Reihe von Methoden, die bereits im Abschnitt über die Standard-IO abgehandelt wurden und dort nachzulesen sind. Es handelt sich dabei um die folgenden Methoden:

```
Syntax:
   long streamvar.flags (long flagstoset);
   long streamvar.flags (void);

   long streamvar.setf (long ios::flagname);
   long streamvar.setf  (long ios::flagname, long
                          ios::flaggruppe);
   long streamvar.unsetf (long ios::flagname);

   int streamvar.fill ();
   int streamvar.fill (char Füllzeichen);

   int streamvar.width ();
   int streamvar.width (int Anzahlzeichen);

   int streamvar.flush ();

   int streamvar.precision ();
   int streamvar.precision (int Anzahlzeichen);
```

Diese Methoden beeinflussen die Flags des Streams, mit denen die Ein- und Ausgabeeigenschaften des Streamobjektes festgelegt werden. Eine Liste der beeinflußbaren Flags ist zu finden im Abschnitt über die Stream-IO zu finden. Die Handhabung der Methoden ist identisch mit dem Handling für die Klassen *ostream* und *istream*.

Natürlich werden nicht nur die gleichen Methoden und Operatoren vererbt, sondern auch die Manipulatoren (u.a. *setprecision*, *setw*, *setfill*, *endl*, *ends*).

```
//============================================================
// Programm FSTREAM9.CPP
//============================================================

#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

void main (void)
{
   double x = 3.14;
   int    y = 7;
   ofstream mystream ("c:\\test2.txt");
```

```
   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   mystream << setfill ('0') << setw (5) << y << endl;
   mystream << setfill (' ') << setprecision (10) << x
            << endl;
   mystream.close ();
}
```

28.3. EIN- UND AUSGABEANWEISUNGEN

Neben den Operatoren << und >> existieren bei der Datei-IO die gleichen Ein- und Ausgabeanweisungen wie bei der Standard-IO:

28.3.1. AUSGABE MIT DATEI-STREAMS

Die Methode *put* schreibt ein einzelnes Zeichen in den Ausgabestrom, wobei die Methode *put* hat den Vorteil hat, daß mit dieser auch Steuerzeichen an den Ausgabestrom gegeben werden können.

Die Methode *write* hingegen dient zur Ausgabe von Zeichenketten unter Angabe einer Ausgabelänge.

```
Syntax:
   streamtyp& streamvar.put (char Zeichen);
   streamtyp& streamvar.put (signed char Zeichen);
   streamtyp& streamvar.put (unsigned char Zeichen);

   streamtyp& write (const char *Zeichenkette,
                     int Anzahlzeichen);
   streamtyp& write (const signed char *String,
                     int Anzahlzeichen);
   streamtyp& write (const unsigned char *String,
                     int Anzahlzeichen);
```

Besonders anzumerken ist, daß die Funktion *write* die angegebene Ausgabelänge auf jeden Fall einhält. Ist die Zeichenkette länger, dann ist die Sache recht unproblematisch, denn es wird nur der Anfang der Zeichenkette (entsprechend der angegebenen Länge) ausgegeben. Ist die Zeichenkette jedoch kürzer, so kümmert sich der C++-Compiler sich nicht um das Stringendesymbol *'\0'* (siehe auch Abschnitt über Zeichenketten), sondern gibt immer die angegebene Länge an Zeichen aus – d.h. zumeist jede Menge Unsinn.

```
//============================================================
// Programm FSTREAMA.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   ofstream mystream ("c:\\test2.txt");

   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   mystream.put   ('\t');
   mystream.write ("Hallo", 5);
   mystream.close ();
}
```

28.3.2. EINGABE MIT DATEI-STREAMS

Von der Methode *get* gibt es mehrere Varianten (die Methode ist also mehrfach „überladen“), jede mit einer leicht anderen Handhabung und unterschiedlichen Parameterleisten. Die Methode *get* akzeptiert, im Gegensatz zum Operator >> auch Leerzeichen als Eingaben.

```
Syntax:
   int streamvar.get ();
```

Die einfachste Form der *get*-Methode liest das nächste Zeichen aus dem Eingabestrom und gibt dessen ASCII-Wert zurück. Da ASCII-Wert und Character sich in ANSI-C und C++ nicht unterscheiden, kann man das Ergebnis entweder als Zahl oder als gelesenes Zeichen betrachten.

```
Syntax:
   streamtyp& streamvar.get (char& charvar);
```

Auch diese Form der *get*-Methode liest nur das nächste Zeichen aus dem Eingabestrom. Das Zeichen selbst wird jedoch nicht zurückgegeben, sondern auf die Variable geschrieben, die als Parameter übergeben wurde.

```
Syntax:
   streamtyp& streamvar.get (char* buf, int len,
                             char Delim = '\n');
```

Diese Form des *get* ist dazu geeignet Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiendezeichen stößt. Es werden jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *get*-Methode sorgt zudem dafür, daß stets ein Stringende-Zeichen (*'\0'*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muß nicht angegeben werden, wenn es sich um die Return-Taste handeln soll, da dies ist die Voreinstellung ist (siehe auch Abschnitt über Default-Parameter).

Die Methode *getline* ist identisch mit der *get*-Variante, die in der Lage ist Zeichenketten zu lesen. Ein Unterschied zwischen *getline* und der angesprochenen *get*-Methode besteht nicht. Die Verwendung von *getline* ist trotzdem vorzuziehen, da sich die Bedeutung im Unterschied zum einfachen *get* leichter erschließt.

```
Syntax:
   streamtyp& istreamvar.getline (char* buf,
                          int len, char Delim = '\n');
```

Die Methode *getline* dient, wie bereits erwähnt, dazu Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiendezeichen stößt. Es werden jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *getline*-Methode sorgt zudem dafür, daß stets ein Stringende-Zeichen (*'\0'*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muß nicht angegeben werden, wenn es sich um die Returntaste handeln soll, da dies ist die Voreinstellung ist.

Die Methode *peek* dient dazu das nächste Zeichen zu betrachten, ohne es aus dem Datenstrom zu entnehmen, d.h. das Zeichen bleibt im Eingabepuffer (meist der Tastaturpuffer) und wird erst beim nächsten *get* oder *getline* entnommen.

```
Syntax:
   int istreamvar.peek ();
```

Die Methode *putback* hat die Aufgabe ein bereits auf dem Stream gelesenes (oder ein beliebig anderes) Zeichen wieder in diesen zurückzustellen (oder hinzuzufügen).

```
Syntax:
   streamtyp & istreamvar.putback(char);
```

```
//=============================================================
// Programm FSTREAMB.CPP
//=============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   int c = 0;
   char buff [300] = "";
   ifstream mystream ("c:\\test2.txt");

   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   c = mystream.get ();
   mystream.putback ((char)c);
   cout << c << endl;
   c = mystream.peek ();
   cout << c << endl;

   mystream.getline (buff, 300);
   cout << buff << endl;
   mystream.close ();
}
```

28.4. POSITIONIERUNG IN STREAMS

Bereits die Klaasen *istream* und *ostream* beinhalten Methoden zur Positionierung des „Dateizeigers“ (Filepointers) im Stream. Dort erfüllen sie relativ wenig Sinn, es sei denn, man möchte sich durch den Tastatur- oder Bildschirmpuffer bewegen. Um so wichtiger ist die tatsache, daß die Positionierungsmethoden an die Klassen *ifstream*, *ofstream* und *fstream* vererbt werden (siehe auch Abschnitt über Vererbung).

Im Gegensatz zu ANSI-C, daß für diese Aufgabe nur die Funktionen *seek* und *tell* besitzt, beinhalten die Streamobjekte unterschiedliche Methoden für Ein- und Ausgabeströme (*seekg* und *tellg* für die Eingabe – das *g* steht für *get* – und *seekp* und *tellp* für die Ausgabe – das *p* steht für *put*).

Die Unterteilung ist durchaus sinnvoll, da man sich vorstellen kann, daß man z.B. eine Klasse sowohl von von *ifstream*, wie auch von *ofstream* ableitet (siehe auch Abschnitt über mehrfache Vererbung) und dann den Lese- und die Schreibzeiger unabhängig von einander in der Datei bewegen kann. Die Standardklasse *fstream* leitet sich zwar ebenfalls von beiden Basisklassen (*istream* und *ostream*) ab, man hat in dieser aber beide Pointer synchronisiert, so daß sie immer auf die gleiche Dateiposition verweisen. Ein unabhängiges Verschieben der Pointer ist also nicht möglich.

Die Methoden *seekg* und *seekp* setzen den Streampointer (entspricht hier dem Filepointer aus ANSI-C) auf die angegebene, relative Position.

```
Syntax:
   istream& seekg (long position);
   istream& seekg (long offset, int relativzu);

   ostream& seekp (long position);
   ostream& seekp (long offset, int relativzu);
```

Zusätzlich kann man angeben, von wo aus die Verschiebung erfolgen soll (analog zu *seek* in C). Wird kein Positionierungsmodus angegeben, so erfolgt die Verschiebung relativ zum Dateianfang. Die nachstehende Tabelle enthält die erlaubten Schlüsselworte für die Verschiebung, die anstelle der in ANSI-C üblichen *defines* verwendet werden:

Positionierungsmodi für Streams		
Modus	*Wert*	*Bedeutung*
ios::beg	0	Positionierung relativ zum Dateibeginn, Voreinstellung
ios::cur	1	Positionierung relativ zur aktuellen Dateiposition
ios::end	2	Positionierung relativ zum Dateiende

Tabelle 5: Stream-Positionierungsmodi

Die Methoden *tellg* und *tellp* wiederum geben die aktuelle Position (relativ zum Dateianfang) des Streampointers zurück:

```
Syntax:
   long tellg (void);
   long tellp (void);
```

```
//============================================================
// Programm FSTREAMC.CPP
//============================================================

#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
   long pos;
   char buff [300] = "";
   ifstream mystream ("c:\\test.txt");

   if (mystream.fail())
   {
      cout << "Fehler" << endl;
      return;
   }

   mystream.getline (buff, 300); // erste Zeile lesen
   cout << buff << endl;
   // Position hinter der ersten Zeile merken !
   pos = mystream.tellg ();

   mystream.getline (buff, 300); // zweite Zeile lesen
   cout << buff << endl;

   // Hinter die erste Zeile zurückspringen
   mystream.seekg (pos, ios::beg);
   mystream.getline (buff, 300); // 2.Zeile erneut lesen
   cout << buff << endl;
}
```

Tritt bei der Positionierung ein Fehler auf, so wird der Streamstatus entsprechend gesetzt und kann bei Bedarf über die Methoden *fail*, *bad* und *good* (s.o.) abgefragt werden.

28.5. AUFGABENTEIL

Die folgende Aufgabe, wendet sich an den fortgeschrittenen Programmierer, bzw. an engagierte Einsteiger.

28.5.1. AUFGABE 1

Schreiben Sie ein Programm, welches maximal 20 Adressen einliest und in einer Datei speichert. Schreiben Sie danach Funktionen, mit denen diese kleine Datenbank verwaltet werden kann (Hinzufügen von Datensätzen, Lesen der Datensätze, Löschen von Datensätzen, Ändern der Datensätze, Suchen von Datensätzen).

ANHANG A: TERMINOLOGIE

Mit der Einführung von objektorientierten Sprachen tauchten eine ganze Reihe von neuen Begriffen in der Programmierung auf, die zunächst einmal nur dazu geeignet scheinen, jeden zu verwirren, der sich die Mühe machen möchte, die Sprache zu erlernen.

Da sich hinter keinem der Begriffe irgend etwas neues oder geheimnisvolles versteckt, sondern lediglich neue Begriffe für altbekannte Konzepte gewählt wurden, um Verwechslungen mit prozeduralen Ansätzen zu vermeiden oder bestimmte Erweiterungen mit einzuschließen, sind hier einmal die "neuen" Begriffe ihren "alten" gegenübergestellt (diese kurze Übersicht über die Terminologie von C++ setzt gute Kenntnisse in C voraus).

A

abgeleitete Klasse — Klasse, die aus einer bereits bestehenden Klasse, durch Erweiterung oder Veränderung, gebildet wird (ðVererbung), auch Childclass, Kindklasse, derived Class oder Tochterklasse genannt.

Adresse — Nummer des Speicherplatzes, an der eine Variable, ein ðObjekt oder eine Funktion im Speicher des Rechners zu finden ist. Wird bei der dynamischen ðAllokation von Speicher zur Laufzeit benötigt und in einem ðPointer gespeichert.

Algorithmus — Verfahrensvorschrift. Derb Algorithmus beschreibt die Art und Weise, wie ein Problem zu lösen ist. Im Gegensatz zur ðSemantik, welche die konkrete Umsetzung in einer Programmiersprache bezeichnet, ist das im Algorithmus beschriebene Verfahren unabhängig von der Art und Weise, sowie den Mitteln der Umsetzung.

Allokation — Anfordern von Speicherplatz (auch Speicherplatzreservierung genannt). im Gegensatz zur ðDeallokation. Speicher wird

immer angefordert (alloziert), wenn eine Variable oder eine Funktion definiert werden (ðDefinition), nicht wenn sie deklariert werden (ðDeklaration). Die Allokation erfolgt ðimplizit bei lokalen und globalen Variablen, bzw. ðexplizit (mit dem ðOperator *new* in C++ oder den Funktionen *calloc* und *malloc* in C) z.B. bei dynamischen ðArrays, deren Größe erst zur Laufzeit ermittelt werden kann.

Array — Feld. Bezeichnet die ðDefinition einer Anzahl gleichartiger Daten bzw. ðRecords (Datensätze) unter dem gleichen Namen. Innerhalb dieses Namens können die einzelnen Daten bzw. Datensätze durch eine fortlaufende Nummer (Index) angesprochen werden.

B

Baseclass — Klasse, von der abgeleitet wird (ðVererbung), auch Basisklasse oder Vaterklasse genannt.

Basisklasse — Klasse, von der abgeleitet wird (ðVererbung), auch Baseclass oder Vaterklasse genannt.

C

Childclass — Klasse, die aus einer bereits bestehenden Klasse, durch Erweiterung oder Veränderung, gebildet wird (ðVererbung), auch abgeleitete Klasse, Kindklasse, derived Class oder Tochterklasse genannt.

Constructor — Spezielle ðMethode, die in jeder ðKlasse vorhanden ist. Der Constructor wird automatisch aufgerufen, wenn eine neue Variable (ðInstanz) dieser Klasse gebildet wird. Auch der Constructor kann bei Bedarf überladen werden.

D

Data Encapsulation Verbergen von Implementationsdetails.

ðDatenkapselung.

Datenkapselung Unter Datenkapselung versteht man das zusammenfügen von Variablen (ðMember) und Funktionen (ðMethoden) zu einem neuen Datentyp (ðKlasse). Die Einkapselung besteht darin, daß der Entwickler bestimmen kann, mit welchen Methoden auf die Daten des ðObjekts zugegriffen werden kann (ðScope, ð*private*, ð*public*, ð*protected*). Durch diesen Mechanismus wird in C++ indirekt das Prinzip der lokalen Funktion eingeführt. Durch die Datenkapselung soll verhindert werden, daß in ein Objekt ungültige Daten geschrieben werden können.

Deallokation Freigabe von dynamisch belegtem Speicherplatz. Jede Form von reserviertem Speicher muß spätestens am Ende des Programms wieder freigegeben (dealloziert) werden. Die Deallokation erfolgt ðimplizit bei lokalen und globalen Variablen, bzw. ðexplizit für alle reservierten Speicherbereiche, die mit *new*, *calloc* oder *malloc* belegt worden sind. (dazu dient der ðOperator *delete* in C++, bzw. die Funktion *free* in C). Wird z.B. bei dynamischen ðArrays, deren Größe erst zur Laufzeit ermittelt werden kann, verwendet.

Definition Vereinbarung eines Wertes für eine Variable, Struktur, Union oder Klasse, bzw. des Codes für eine Funktion. Im Gegensatz zur ðDeklaration wird bei der Definition nicht nur der Name (bzw. bei Struktur, Union und Klasse auch der Aufbau) festgelegt, sondern auch der Inhalt bzw. die Funktionalität. Dazu wird der nötige Speicherplatz reserviert. Variablen

können in C/C++ zugleich deklariert und definiert werden, Funktionen sollten immer in einer Headerdatei mit ihrem Rückgabewert und ihren Parametern deklariert und in einer Quelltextdatei definiert werden.

Deklaration Vereinbarung eines Namens für eine Variable, Struktur, Union, Klasse oder Funktion. Im Gegensatz zur ðDefinition wird bei der Deklaration nur der Name (bzw. bei Struktur, Union und Klasse auch der Ausbau) festgelegt, nicht jedoch der Inhalt. Außerdem wird kein Speicherplatz reserviert. Funktionen werden üblicherweise in einer Headerdatei mit ihrem Rückgabewert und ihren Parametern deklariert und in einer Quelltextdatei definiert.

Derived Class Klasse, die aus einer bereits bestehenden Klasse, durch Erweiterung oder Veränderung, gebildet wird (ðVererbung), auch Childclass, Kindklasse, abgeleitete Klasse oder Tochterklasse genannt.

Destructor Spezielle ðMethode, die in jeder ðKlasse vorhanden ist. Der Destructor wird automatisch aufgerufen, wenn eine Variable (ðInstanz) dieser Klasse aus dem Speicher entfernt wird.

E

Encapsulation Verbergen von Implementationsdetails.

ðDatenkapselung.

Explizit ausdrücklich, im Gegensatz zu ðimplizit. Eine Information ist in einer anderen Information nicht ausdrücklich enthalten („explizit angegeben“). So enthält der Begriff „Hund“ keine Angabe um welche Rasse (z.B. Terrier

oder Dackel) es sich handelt.

F

Friend — Die Vereinbarung einer ðKlasse als Freund einer anderen Klasse, erlaubt dieser Friend-Klasse den Zugriff auf die als ð*private* gekennzeichneten ðMember. Dies ist zwar nicht im Sinne der objektorientierten Programmierung, denn fast jeder Friend-Zugriff könnte auch über eine entsprechende ðMethode gelöst werden, kann aber bei sehr häufigen Zugriffen die Verarbeitung stark beschleunigen.

I

Implizit — beinhaltend, im Gegensatz zu ðexplizit. Eine Information ist in einer anderen Information bereits enthalten („implizit vorhanden“). So enthält z.B. das Wort „Terrier“ implizit die Information, daß es sich um a) ein Lebewesen b) ein Tier c) ein Säugetier d) ein Säugetier der Wolfsgattung und e) einen Hund handelt.

Inheritance — Vererbung. Ableiten einer neuen Klasse aus einer vorhandenen.

Inline-Funktion — Funktion innerhalb einer Zeile. Mechanismus der, die K&R- und ANSI-C Präprozessormakros ersetzt. Bei Inline-Funktionen sind die üblichen Nachteile der Makros (mehrfache Auswertung von prä- und post Dekrement bzw. Inkrement durch die Quellcode-Expandierung) ausgeschaltet. Die Inline-Funktion wird zwar wie eine echte Funktion geschrieben, es erfolgt aber kein echter Funktionsaufruf, sondern der Funktionscode wird an der entsprechenden Stelle des Programms generiert. Die Datentypunabhängigkeit, einer der großen Vorteile von Makros, kann durch ðTemplates

erreicht werden.

Instance — ðInstanz.

Instanz — Unter einer Instanz versteht man die Variable einer ðKlasse, wenn diese in einem Programm ðdeklariert (instanziiert) wird. (ðObjekt). Meist bezogen auf selbstdefinierte Klassen.

Instanziierung — ðDefinition einer Variablen. Begrifflich meist im Zusammenhang mit einer ðKlasse verwendet.

K

Kindklasse — Klasse, die aus einer bereits bestehenden Klasse, durch Erweiterung oder Veränderung, gebildet wird (ðVererbung), auch abgeleitete Klasse, Childclass, derived Class oder Tochterklasse genannt.

Klasse — Im einfachsten Fall nichts anderes als eine Struktur (Record/ Datensatz), d.h. ein zusamengesetzter Datentyp. Eine Klasse unterscheidet sich von einer Struktur durch die Tatsache, daß außer den Datensatzkomponenten (ðMember) auch Funktionen (ðMethoden) als Teil der Klasse gelten. Im Gegensatz zum Record kann der Entwickler zudem bestimmen, welche Funktionen und Variablen nach „außen“ sichtbar sind (ðScope, ð*private*, ð*public*, ð*protected*). Klassen erweitern im Grunde genommen die Liste der Datentypen. Die ðDeklaration einer Klasse ist wie die Vereinbarung einer Datenstruktur und erzeugt noch keinen Speicherplatz für eine Variable (ðInstanz).

Komponente	Teil eines zusammensetzten Datentyps (Struktur, ðKlasse bzw. ðRecord). Der Zugriff erfolgt in C/C++ über den Dezimalpunkt bzw. über den ðOperator –> (falls über Pointer zugegriffen wird).

M

Member	Eine Variable der Klasse. Betrachtet man die Klasse als Variante einer Struktur (eines Records), so handelt es sich um eine Datensatzkomponente.
Memberfunction	Mitgliedsfunktion. ðMethode.
Methode	Eine Methode ist eine lokale Funktion, die Teil eines ðObjektes (ðKlasse) ist. Der Name der Methode ist, wie eine Record-Komponente, nur zusammen mit dem Variablennamen des ðObjekts (ðInstanz) gültig und wird genauso angesprochen. Eine Methode kann exportiert werden, aber auch verborgen sein, d.h. nur von anderen Methoden der gleichen Klasse (ðScope) aufrufbar sein.
Multiple Inheritance	mehrfache ðVererbung.

O

Objekt	ðInstanz einer ðKlasse.
Overloading	Unter Overloading versteht man die mehrfache Verwendung des gleichen Funktionsnamens für unterschiedliche Funktionen oder die Erweiterung eines ðOperators. Dabei trifft der Compiler zur Übersetzungszeit die Entscheidung welche Funktion verwendet wird. Dazu muß der Entwickler dafür sorgen, daß der

Entscheidungsprozeß eindeutig ist. Die überladenen Funktionen werden anhand der Parameterliste unterschieden. Anzahl und Typ der Parameter werden zur Unterscheidung herangezogen. Bei ðabgeleiteten Klassen (ðInheritance) können auch ðMethoden überladen werden, wenn sie in der ðBasisklasse als virtuell gekennzeichnet wurden (ðvirtuelle Methode).

Operator — Verknüpfungssymbol. Bezeichnet in einer Programmiersprache sowohl die mathematischen Operatoren (+,–,*,/ etc.) als auch die strukturellen Symbole um z.B. Zugriffe über Pointer, Indices oder Adressen zu ermöglichen (*,->,[] oder &). Die meisten Operatoren sind für eigene ðKlassen überladbar (ðOverloading).

P

Pointer — Zeigervariable. Variable, in der die Adresse einer anderen Variablen, eines ðObjektes oder einer Funktion gespeichert ist.

Private — Ein Klassenmember (ðMember) oder eine Klassenmethode (ðMethode), die als *private* ðdeklariert wurde, ist von außen nicht zugänglich. *Private* Member oder Methoden werden nicht mit vererbt (ðVererbung). Es handelt sich also um lokale Funktionen bzw. Variablen, deren Name nur innerhalb der jeweiligen Klassenmethoden bekannt sind, aber deren Werte erhalten bleiben.

Protected — Ein Klassenmember (ðMember) oder eine Klassenmethode (ðMethode), die als *protected* ðdeklariert wurde, ist von außen nicht zugänglich. *Protected* Member oder Methodenwerden werden im Gegensatz zu ð*private* Membern und Methoden mit vererbt

(ðVererbung). Es handelt sich also, wie bei *private* Membern und Methoden, um lokale Funktionen bzw. Variablen, deren Name nur innerhalb der jeweiligen Klassenmethoden bekannt sind, aber deren Werte erhalten bleiben.

Public — Ein Klassenmember (ðMember) oder eine Klassenmethode (ðMethode), die als *public* ðdeklariert wurde, ist (wie die eine ðKomponente eines Datensatzes) von außen zugänglich. *Public* Member sollten immer die Ausnahme bleiben, statt dessen sollte der Entwickler ð*protected* Member verwenden, die mit *public* Methoden manipuliert werden können. Auf diese Weise kann das Hineinschreiben ungültiger Werte komplett verhindert werden (ðDatenkapselung). *Public* Member und Methoden werden immer vererbt (ðVererbung).

R

Record — Datensatz, zusammengesetzter Datentyp. Wird in C/C++ mit dem Schlüsselwort *struct* ðdeklariert.

S

Scope — Sichtbarkeit einer Klassenvariable oder -funktion (ðKlasse, ðMember, ðMethode). Die Sichtbarkeit kann durch die Angabe der Schlüsselworte ð*public*, ð*private* oder ð*protected* bestimmt werden. Bei Uneindeutigkeiten kann der Scope über den ðScopedelimiter angegeben werden.

Scope resolution — Sichtauflösung. ðScopedelimiter.

Scopedelimiter — Sichtbegrenzer. Operator „::“, mit dem

eindeutig auf Membervariablen (ðMember) einer Vaterklasse (ðVererbung) zugegriffen werden kann.

Semantik[1] — Inhalt eines Programms, bezogen auf die zugrundeliegende Syntax (im Gegensatz zum ðAlgorithmus). Ein semantisch korrektes Programm setzt einen Algorithmus korrekt (d.h. ohne Laufzeitfehler) in einer bestimmten Programmiersprache um.

Stream — Streams ersetzten in C++ die Dateizugriffe über *FILE*. Streamzugriffe haben den Vorteil, daß man sie in selbstdefinierten ðKlassen überladen kann (ðOverloading), so daß die Handhabung eigener ðObjekte (ðInstanz) wesentlich vereinfacht wird.

Syntax[2] — Die „Rechtschreibung“ und „Grammatik“ eines Programms. So erzwingt die Syntax von C/C++ z.B. den Abschluß von Anweisungen durch ein Semikolon und die paarige Verwendung von Klammern (für jede öffnende Klammer muß es eine schließende geben) etc. Ein syntaktisch korrektes Programm ist übersetzbar, aber nicht zwangsläufig frei von Fehlern, die zur Laufzeit auftreten ðSemantik.

T

Template — Schablone. Mechanismus, mit dem eine Funktion definiert werden kann, ohne daß die Datentypen der Parameter und/oder Rückgabewerte vom Programmierer festgelegt werden müssen. Statt dessen werden für die Datentypen Platzhalter vergeben und der Compiler erzeugt die notwendigen Funktionen

[1] aus dem Griechischen: Bedeutungslehre

[2] aus dem Griechischen: Aufbau

(soweit ableitbar und benötigt) selbst. Templates erweitern die ðInline-Funktionen um die Typabstraktion des ANSI-C „*#define*“-Makros, um dieses überflüssig zu machen.

Tochterklasse — Klasse, die aus einer bereits bestehenden Klasse, durch Erweiterung oder Veränderung, gebildet wird (ðVererbung), auch abgeleitete Klasse, Childclass, derived Class oder Kindklasse genannt.

V

Vaterklasse — Klasse, von der abgeleitet wird (ðVererbung), auch Baseclass oder Basisklasse genannt.

Vererbung — Vererbung ist das Ableiten einer neuen ðKlasse aus einer bereits bestehenden. Die neue Klasse (auch derived class, abgeleitete Klasse, Tochterklasse, Kindklasse oder Childclass genannt) enthält alle ðMember und ðMethoden – auch ðvirtuelle Methoden, die in der ursprünglichen Klasse (bzw. Baseclass, Basisklasse oder Vaterklasse genannt) als ð*public* und ð*protected* enthalten sind. ð*private* Member oder Methoden und Freundschaftsbezüge (ð*friend*) werden nicht mit vererbt. Die abgeleitete Klasse kann man sich als neuen Datensatz (Struktur, Record) vorstellen, der (ggf. nur Teile) einer bereits bekannten *struct* (Basisklasse) enthält – also eine verschachtelten Record darstellt. die abgeleitete Klasse darf den ererbten Methoden und Membern natürlich neue hinzufügen. Bei der mehrfachen Vererbung (ðmultiple Inheritance) erbt eine abgeleitete Klasse von mehreren Basisklassen gleichzeitig.

Virtuelle Methode — Eine ðMethode, die in einer ðBasisklasse ðdeklariert und deren Name an alle ðabgeleiteten Klassen vererbt (ðVererbung)

wird, wo dieser Methodenname überladen (ðOverloading) werden kann (oder muß).

Z

Zeiger ðPointer.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS