3 Objektorientierte Erweiterungen in C++ und UML-Grundlagen

Alle Dinge geschehen aus Notwendigkeit.
Es gibt in der Natur kein Gutes und kein Schlechtes.
Spinoza

In diesem Kapitel werden nicht nur die wesentlichen neuen Sprachelemente vorgestellt, die C++ zu einer objektorientierten Sprache machen, sondern es werden auch bereits erste wichtige Designaspekte behandelt, die beim Entwurf eines objektorientierten Programms zu berücksichtigen sind. Dabei werden die entsprechenden Elemente der Unified Modeling Language (UML) vorgestellt.

3.1 Was ist UML und warum UML?

Die *Unified Modeling Language* ist eine Sprache und Notation, um objektorientierte Systeme zu beschreiben. So legt die *UML* z.B. Symbole für Klassen, Objekte und deren Beziehungen untereinander fest.

Objektorientierte Softwareentwicklung gibt es bereits seit 30 Jahren. Am Anfang stand die Programmiersprache Smalltalk, die das Klassenkonzept von der Programmiersprache Simula-67 übernahm und weiterentwickelte. Mit Beginn der 90er Jahre hat sich C++ als dominierende objektorientierte Sprache durchgesetzt. Seit 1996 gewinnt neben C++ noch Java an Bedeutung, während Smalltalk mehr und mehr zurückgedrängt wird.

Während die Objektorientierte Programmierung (OOP) bereits seit 30 Jahren existiert, wurden die ersten Bücher über objektorientierte Analyse (OOA) und objektorientiertes Design (OOD)[1] erst Ende der 80er/Anfang der 90er Jahre publiziert.

Es entstanden mehrere Methoden und Notationen, u.a. Booch, Coad/Yourdon, Shlaer/Mellor, Jacobson, Rumbaugh.

---

1 Analyse und Design: In der Analysephase wird zunächst geklärt, **was** zu tun ist. Erst in der Designphase wird festgelegt, **wie** dies zu tun ist.

Im Herbst 1994 haben sich Grady Booch und Jim Rumbaugh bei der Rational Software Corporation zusammengeschlossen, um ihre erfolgreichen Methoden in Form einer gemeinsamen Notation zusammenzuführen. Ein Jahr später stieß noch Ivar Jacobson dazu. Es entstand die Unified Modeling Language (UML), die 1997 von der OMG[2] als Standard akzeptiert wurde.

Die UML kann mittlerweile als Industriestandard angesehen werden. Beinahe alle CASE[3]-Tool Hersteller und Autoren unterstützen die UML.

Die UML unterstützt als Notation die OOA und OOD. Sie zielt nicht auf eine spezielle Programmiersprache ab.

Der Einsatz der UML hat damit folgende Vorteile:

- Der objektorientierte Sachverhalt kann in einer einfachen und normierten Darstellungsform beschrieben werden.

  Damit können Missverständnisse vermieden werden, Analyse- und Entwurfsfehler können bereits in einer früheren Phase (noch vor der Implementierung) entdeckt werden.
- Die durchgeführte Analyse und das Design können (theoretisch) in jeder x-beliebigen Programmiersprache umgesetzt werden.

Allerdings darf nicht übersehen werden, dass ein wirklich effektives Design letzten Endes nur dann erzielt werden kann, wenn die verwendete Programmiersprache mit ihren Sprachmittel, die sie zur Verfügung stellt, berücksichtigt wird (vgl. Zitat von Booch im Vorwort auf Seite 15).

Ferner darf nicht übersehen werden, dass die UML eine Notation ist, jedoch keine Methodik, d.h. man kann mit ihr die eigene OOA und OOD darstellen. Sie trifft jedoch keine Aussagen darüber, wie man dazu kommt.

---

2 *OMG: Object Management Group*
Systemanbieter und Anwender objektorientierter Techniken haben sich 1989 zur OMG zusammengeschlossen. Die OMG verfolgt das Ziel, Standards und Spezifikationen für verteilte objektorientierte Anwendungen zu schaffen.

3 *CASE: Computer Aided Software Engineering*
Mit Hilfe von Programmen wird der Software-Entwicklungsprozess unterstützt, z.B. Erstellen von UML-Diagrammen und deren Umsetzung in Source-Code.

Die UML macht außerdem auch keine Aussagen darüber, wie detailliert ein Sachverhalt darzustellen ist, so kann z.B. eine Klasse im Symbol nur durch ihren Namen beschrieben werden oder mit all ihren Elementen (Daten und Funktionen oder objektorientiert ausgedrückt: Attributen und Methoden).

Wir werden in diesem Buch nur die elementaren Symboliken erklären; d.h. dass es zu dem einen oder anderen Symbol oder der einen oder anderen Diagrammart noch weitere, genauere Beschreibungsmittel geben kann. Es ist aber nicht Ziel des Buches, UML vollständig zu beschreiben, sondern der Schwerpunkt liegt ganz klar auf dem Grundverständnis für Objektorientierung. Für eine komplette UML-Beschreibung sei auf entsprechende Literatur (z.B. [3])verwiesen.

3.2 Verwendung einer Klasse

Klassen und Objekte sind »der Stoff, aus dem die objektorientierten Träume sind«.

Bevor wir auf das Zusammenwirken von Klassen und Objekten eingehen, wollen wir im ersten Schritt zunächst nur den Klassen-/Objektgedanken an sich betrachten sowie dazu die C++-Syntax und die zugehörigen UML-Notationen vorstellen. Erst wenn Sie als Leser dieses fundamentale Thema verinnerlicht haben und mit der Klassen-Syntax vertraut sind, sollten Sie die nächsten Schritte gehen.

3.2.1 Das Schlüsselwort class

Durch die Neuerungen bei Strukturen, die in Kapitel 2.12 vorgestellt wurden, war bereits der erste entscheidende Schritt getan, um aufbauend auf C eine objektorientierte Sprache C++ zu schaffen. Um einen wirklichen Schritt in Richtung Objektorientierung zu tun, führte man ein neues Schlüsselwort **class** ein, das sich weitgehend wie das Schlüsselwort **struct** verwenden lässt.

Die Unterschiede zwischen den Schlüsselwörtern **struct** und **class** sind nur sehr gering:

- Während in einer **struct**-Definition alle Elemente (Variablen und Funktionen) standardmäßig **public** sind, sind in einer **class**-Definition alle Elemente standardmäßig **private**.
- In Programmen, die objektorientiert entworfen sind, verwendet man üblicherweise **class**, während man in nicht-objektorientierten Programmen das Schlüsselwort **struct** benutzt.

Fassen wir noch einmal die zentralen Punkte zusammen, was wir bisher ausgehend von einer C++ **struct** über eine C++ **class** bereits wissen:

- Eine Klasse besitzt *Zugriffsspezifizierer* (**private, public, protected**).
- Eine Klasse enthält als *»Elemente« / »Member«: Daten und Funktionen.* Als Faustregel kann gelten:
  Die Daten sind **private**, der Zugriff auf die Daten erfolgt über **public-**Funktionen.
- Eine Klasse besitzt einen *Konstruktor* und einen *Destruktor.*
- Jedes Objekt besitzt einen (versteckten) Zeiger auf sich selbst, den **this**-Zeiger.

3.2.2 Was ist eine Klasse und was nicht?

Es ist wichtig, sich stets vor Augen zu halten, dass man von **Objekt**orientierung redet und nicht von Klassenorientierung, d.h. im Vordergrund der Betrachtungen stehen stets die Objekte.

Es sind Objekte, die wir Menschen in erster Linie wahrnehmen. Erst im zweiten Schritt abstrahieren wir, indem wir gemeinsame Verhaltensweisen und Eigenschaften von Gegenständen allgemein auffassen und dafür einen Begriff finden. Auf diese Weise vereinfachen wir unsere komplexe Welt, so dass wir sie leichter verstehen können. Das zeigt sich bereits im Kindesalter: Bello und Rex haben vier Beine und machen Wau-Wau: Es sind beides Hunde. Hat dies das Kind einmal verstanden, kann es jeden x-beliebigen Hund als Hund identifizieren, auch wenn es dieses Objekt vorher noch nie gesehen hat.

Objekte, die ein gemeinsames Verhalten aufweisen, werden in Klassen allgemein beschrieben.

Wichtig ist dabei, dass es in erster Linie um das *Verhalten* des Objektes geht, d.h. wie es sich nach außen darstellt. Erst in zweiter Linie ergeben sich die Daten, die intern notwendig sind, um die Aufgabe nach außen erfüllen zu können. Neueinsteiger in die Objektorientierung betonen häufig zu sehr die Daten (»eine class ist ja auch nur eine struct«). Je eher man sich jedoch des Unterschiedes zwischen einer C-struct und einer C++-class in der Bedeutung (nicht nur in der Syntax) bewusst ist, desto besser.

Wozu wird eine Klasse also eingesetzt?

- Um Objekte mit gleichem Verhalten allgemein gültig zu beschreiben.
- Um einen komplexeren Sachverhalt zu kapseln.

Ein Beispiel dafür, wie Klassen helfen, Komplexität zu reduzieren, sind Verwaltungsmechanismen. So ist z.B. die Verwendung eines Arrays nicht gerade trivial. Hier können viele Fehler gemacht werden (Speicherüber-/unterschreitung). Es lohnt sich also, ein Integer-Array in einer Klasse zu kapseln und den Zugriff auf die Daten geschützt erfolgen zu lassen. Ist eine derartige Klasse einmal realisiert und getestet, können davon guten Gewissens viele Objekte angelegt werden. Komplexität wird dadurch reduziert, weil der Programmierer sich auf die eigentliche Lösung seines Problemes konzentrieren kann und sich keine Gedanken mehr um gewisse Realisierungsdetails (Speichermanagement) machen muss. Ein anderes Beispiel wäre der Umgang mit Zeichenketten (Strings) oder Dateien.

Fragwürdig ist es in jedem Fall, wenn das gesamte Programm einfach in eine Klasse verpackt wird. Dies ist sicherlich nicht im Sinne der Objektorientierung.

3.2.3 OO-Begriffe in Bezug auf ein(e) Klasse/Objekt

Wir wollen an dieser Stelle noch einmal alle Begriffe im Zusammenhang von Klasse und Objekt im Überblick aufführen.

Als Beispiel wollen wir eine Klasse `Kreis` verwenden. Ein Kreis ist (bei uns) gekennzeichnet durch eine Position (x-Wert und y-Wert) und einen Radius. Der Kreis soll an einer x-beliebigen Stelle positioniert werden können. Außerdem sollen seine Werte angezeigt werden können. In `main()` legen wir zwei Kreise an.

Dies führt zu folgendem Source-Code, wie er in Listing 3.1 (`KreisSimple.cpp`) gezeigt ist:

```
#include <iostream.h>

class Kreis
{
// Da hier kein Zugriffsspezifizierer angegeben ist
// sind folgende Daten private
   int radius;
   int xPos, yPos;      // Position des Kreises

public:
   Kreis(int x=0, int y=0, int r=1) // Konstruktor
   {
      xPos=x;
      yPos=y;
      radius=r;
   }
   void Show()
   {
      cout << "\nPosition (x,y): " << xPos << ", "
                                   << yPos << endl;
      cout << "Radius: " << radius << endl;
   }
   void SetPosition(int x, int y)
   {
      xPos=x;
      yPos=y;
   }
};
```

```
int main(void)
{
   Kreis kreis1, kreis2(1,2);

   kreis1.Show();
   kreis2.SetPosition(2,3);
   kreis2.Show();

   return 0;
}
```

Listing 3.1: (KreisSimple.cpp) Klasse Kreis

Das Listing 3.1 (`KreisSimple.cpp`) führt zu folgender Ausgabe:

```
Position (x,y): 0, 0
Radius: 1

Position (x,y): 2, 3
Radius: 1
```

An Hand diverser Aussagen – bezogen auf das Listing 3.1 (`KreisSimple.cpp`) – wollen wir noch einmal alle Begriffe rund um Klasse und Objekt erklären:

- Zusammenhang *Objekt – Klasse*
  - »kreis1 und kreis2 sind *Objekte* (vom Typ) der *Klasse* Kreis.«
  - »kreis1 und kreis2 sind zwei Kreise.«
  - »kreis1 und kreis2 sind *Exemplare* der Klasse Kreis.«
  - »kreis 1 und kreis2 sind *Instanzen* der Klasse Kreis.«
- Bestandteile einer Klasse
  - Die Bestandteile einer Klasse, ihre Daten und Funktionen, werden auch als *Elemente* der Klasse bezeichnet. Man spricht daher auch von *Elementvariablen* und *Elementfunktionen*.
  - Elementvariablen
    - »Die Klasse Kreis hat folgende *Membervariablen*: `x`, `y`, `radius`.«
    - »Die Klasse Kreis hat folgende *Attribute*: `x`, `y`, `radius`.« (Die UML verwendet diesen Begriff.)

- Elementfunktionen
  - »Die Klasse Kreis hat folgende *Memberfunktionen*: `SetPosition()`, `Show()`.«
  - »Die Klasse Kreis hat folgende *Methoden*: `SetPosition()`, `Show()`.«
    (-> Ein Kreis hat die Methodik, sich zu zeigen)
  - »Die Klasse Kreis ermöglicht folgende *Operationen*: `SetPosition()`, `Show()`.«
    (-> Ein Kreis kann sich zeigen, es kann auf ihn die Operation ‚show' ausgeführt werden)
  - »Das Objekt kreis1 erhält lediglich die *Botschaft* `Show()`, das Objekt kreis2 die *Nachricht* `SetPosition()` und anschließend `Show()`« (-> In der OOP werden Objekte als aktive Elemente der Software betrachtet. Man spricht also nicht von Funktionsaufrufen, sondern von Botschaften/Nachrichten, welche die Methoden des Objektes dazu veranlassen, etwas zu tun.)

Wie aus den vorherigen Aussagen ersichtlich wird, können die folgenden Begriffe synonym verwendet werden:

- *Objekt* = *Exemplar* = *Instanz*
- *Member* = *Element*
- *Elementvariable* = *Membervariable* = *Attribut*
- *Elementfunktion* = *Memberfunktion* = *Methode* = *Operation* = *Nachricht* = *Botschaft*

3.2.4 Klassen und Objekte in der UML

Die folgende Abbildung 3.1 zeigt, wie sich unsere Klasse `Kreis` aus Listing 3.1 (`SimpleKreis.cpp`) in der UML darstellen lässt:

Neben den Attributen und Operationen wurde hier noch eine *Zusicherung* modelliert. Zusicherungen sind Bedingungen, Voraussetzungen und Regeln, die die Objekte der Klasse erfüllen müssen. Sie werden in { } (geschweifte Klammern) notiert. In unserem Fall darf der Radius nicht negativ und nicht null sein (radius >0).

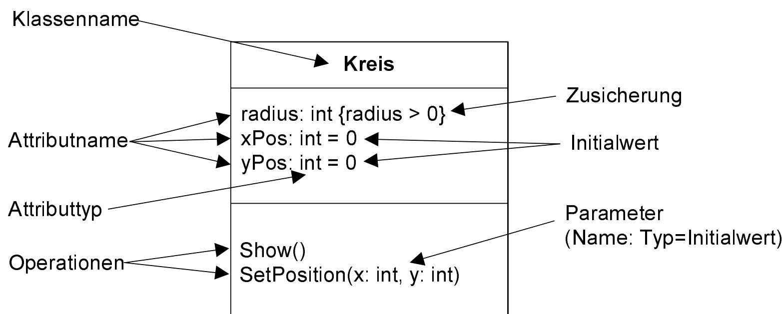

Abbildung 3.1: Klassen in der UML (am Beispiel Kreis)

Was hier nicht modelliert wurde, sind die Zugriffsspezifizierer, da in der Regel Attribute private und Methoden public sind, was hier auch der Fall ist.

Um die Zugriffsspezifizierer aus C++ in der UML zu notieren, können die Elemente mit folgenden Präfixen gekennzeichnet werden:

- private-Element
+ public-Element
# protected-Element (bei Vererbung)

Unsere Klasse Kreis würde dann wie in Abbildung 3.2 aussehen:

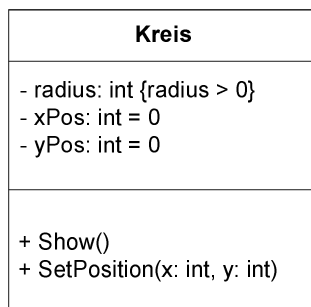

Abbildung 3.2: Klasse Kreis in UML mit Zugriffsspezifizierern

Objekte werden in UML-Diagrammen ähnlich wie Klassen dargestellt, nur dass der Objektname im Gegensatz zum Klassennamen unterstrichen wird. Für die Attribute können beispielhaft Werte eingesetzt werden, wie dies in Abbildung 3.3 gezeigt ist:

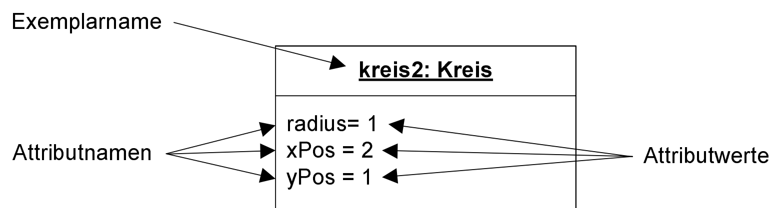

Abbildung 3.3: Objekte in der UML (am Beispiel von `kreis2`*)*

Wie detailliert eine Klasse bzw. ein Objekt beschrieben wird, hängt davon ab, was der Autor des Diagrammes ausdrücken will, so ist es z.B. auch möglich, die Elemente einer Klasse komplett auszublenden und nur den Klassennamen zu nennen. Dies geschieht vor allem dann, wenn es mehr um die Beziehungen zwischen den Klassen als um die einzelnen Klassen an sich geht.

Ebenso ist es möglich, den Zusammenhang zwischen Objekten und Klasse(n) zu modellieren. Zwischen dem Objekt und seiner Klasse wird ein gestrichelter Pfeil in Richtung Klasse gezeichnet (vgl. auch Kapitel 2.1, Objektorientierung in der realen Welt, in dem die Begriffe *Objekt* und *Klasse* am Beispiel von Samurai und Tiger erklärt wurden).

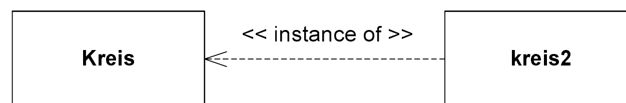

Abbildung 3.4: Klasse-Objekt-Beziehung in UML

Die Beschriftung des Pfeils lautet << *instance of*>>. Hier handelt es sich um ein so genanntes *Stereotyp*[4]. Stereotypen sind projekt-, unternehmens- oder methodenspezifische Erweiterungen vorhandener Modellelemente

4 [Duden]: Singular: das Stereotyp, Plural: die Stereotypen.

der UML. Durch Stereotypen kann die Bedeutung einer Klasse/einer Beziehung näher bestimmt werden. CASE-Werkzeuge unterscheiden zudem teilweise zwischen visueller und textueller Stereotypisierung. Visuelle Stereotypisierung definiert eigene Symbole, textuelle Stereotypen werden in doppelten Winkelklammern (<< >>) eingeschlossen, z.B. <<*instance of*>> oder <<*interface*>> (vgl. Kapitel 3.8, Schnittstellen).

Da jede Klasse über einen Konstruktor und Destruktor verfügt, werden diese im Klassendiagramm nicht modelliert, auch wenn sie überladen werden. Wie in Abbildung 3.1 gezeigt, können die Initialwerte anderweitig notiert werden.

3.2.5 Konventionen in der Namensgebung

Um die Lesbarkeit von Source-Code zu verbessern, macht es Sinn, sich an projekt- bzw. unternehmensspezifische Konventionen zu halten. Wir haben uns in diesem Buch für folgende Konventionen entschieden, die wir von nun an verwenden werden:

- Klassennamen beginnen mit dem Präfix ‚`C`' (**c**lass), um sie leicht von Objekten unterscheiden zu können.
- Membervariablen beginnen mit dem Präfix ‚`m_`'(**m**ember), um sie in Methoden leicht von lokalen Variablen unterscheiden zu können.
- Methodennamen verwenden Groß-Klein-Schreibung zur besseren Lesbarkeit. Idealerweise beginnt der Methodenname mit einem Verb, z.B. `DruckeListe()` oder `SetValue()`.

3.2.6 Beispiel: Realisierung eines Stacks

Eine sehr wichtige Datenstruktur in der Informatik ist der so genannte *Stack*. Ein Stack ist realisiert wie ein realer Stapel. Die Elemente können immer nur oben auf dem Stapel abgelegt werden. Ebenso kann auch immer nur das oberste Element vom Stapel genommen werden. Ein Stack ist also eine Datenstruktur, für die nur zwei Operationen definiert sind:

- `Push()` zum Ablegen eines Elements an oberster Stelle des Stacks und
- `Pop()` zum Entnehmen des Elements an oberster Stelle des Stacks

Abbildung 3.5 verdeutlicht die Funktionsweise eines Stacks.

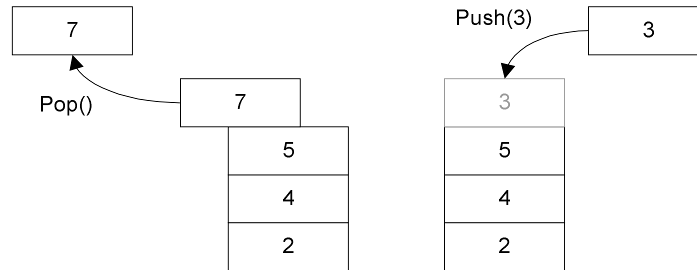

Abbildung 3.5: Funktionsweise eines Stacks

Im Folgenden wollen wir eine Klasse `CStack` entwerfen, die als Elemente Integer verwalten kann.

Klassendiagramm zum Stack: Die maximale Größe unseres Stacks soll konstant sein. Daher bietet sich als eine einfache Realisierung ein eindimensionales Array an. Beim Anlegen des Stacks soll die maximale Größe jedoch angegeben werden können. Sie wird im Attribut `m_Max` gespeichert. Der oberste Wert steht im Array an der Stelle `m_Top`.

Abbildung 3.6 zeigt das Klassendiagramm zu unserer Klasse CStack.

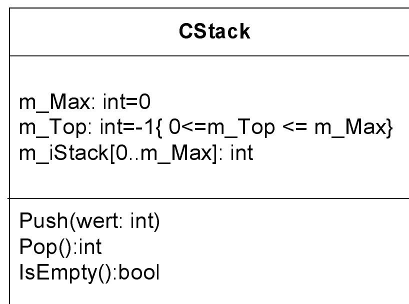

Abbildung 3.6: UML-Diagramm zur Klasse CStack

Realisierung der Stackklasse in C++: Folgendes Listing 3.2 zeigt die Realisierung der Klasse CStack in C++:

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

//................... Klasse CStack .......
class CStack
{
public:
   int  Pop();    // Oberstes Element von Stack holen
                  // Return: Wert;
                  //        wenn nicht erfolgreich: -1
   bool Push(int iValue); // Element an oberster Stelle
                          // des Stacks ablegen
                          // Return: Push erfolgreich?
   bool IsEmpty() { return m_Top < 0; }  // Stack leer ?
                                         // (inline)

   CStack(int maxEntries);           // Konstruktor
   ~CStack();                        // Destruktor

private:
   int * m_iStack;

   int m_Max;    // maximal gültiger Index
   int m_Top;    // Index, an dem oberster Wert liegt
};

//............................. Konstruktor .......
CStack::CStack(int maxEntries)
{
   m_Max = 0;
   m_Top = -1;   // Stack ist zunaechst leer

   m_iStack = new int[maxEntries];
   if (m_iStack != NULL)
      m_Max=maxEntries-1;
}
//................................ Destruktor ......
CStack::~CStack()
{
   if(m_iStack)  // Könnte ja sein, dass Anlegen im
                 // Konstruktor nicht geklappt hat
```

```
      delete [] m_iStack;
}
//..................................... Push .......
bool CStack::Push(int value)
{
   if(m_Top == m_Max)
      return false;
   m_iStack[++m_Top] = value;
   return true;
}
//...................................... Pop .......
int CStack::Pop()
{
   if (m_Top < 0)
      return -1;
   return m_iStack[m_Top--];
}

//...................................... main .......
int main(void)
{
   CStack  stack(100);
   int     zeich;

   cout << "Bitte geben Sie einen String ein: " << endl;

   while ((zeich=getchar()) != '\n'
           && stack.Push(zeich))
      ;

   cout << "Der String lautet rueckwaerts:" << endl;

   while (!stack.IsEmpty())
   {
      zeich = stack.Pop();
      cout << (char)zeich;
   }

   cout << endl;
   return 0;
}
```

Listing 3.2: (stack.cpp): Klasse CStack mit Testprogramm

Ein möglicher Ablauf des Listing 3.2 (`stack.cpp`) wäre:

```
Bitte geben Sie einen String ein:
Ein Neger mit Gazelle zagt im Regen nie ↵
Der String lautet rueckwaerts:
ein negeR mi tgaz ellezaG tim regeN niE
```

Für den Rückgabetyp bei der Methode `Push()` wurde bool gewählt, um dem Aufrufer mitzuteilen, ob der Wert im Stapel gespeichert werden konnte oder nicht. Bei der Methode `Pop()` wurde der Datentyp int gewählt, um den gelesenen Wert direkt über den Rückgabewert verfügbar zu haben. Dies kann unter Umständen problematisch sein, wenn der Wert –1 im Stack gespeichert wird, weil dies gleichzeitig auch das Signal für den Fehlerfall ist. Beim Wert –1 müsste der Aufrufer also zusätzlich mit `IsEmpty()` überprüfen, ob der Stack leer ist. Solche Besonderheiten sollten gut dokumentiert sein.

Aufgrund der Problematik, dass Fehlerwert und gespeicherter Wert unter Umständen nicht unterschieden werden können, wird in Bibliotheksklassen bei `Get()`-Methoden für die Wertrückgabe häufig Call by Reference verwendet statt direkt der Returnwert.

3.2.7 Das Leben eines Objektes

Durch die Definition einer Klasse ist noch keinerlei Speicher angelegt. Die Klasse ist lediglich der »Bauplan« für Objekte. Erst wenn ein Objekt angelegt wird, existiert etwas Konkretes und es kommt Leben in die Software.

Ebenso wie Menschen werden auch Objekte irgendwann geboren und sterben auch wieder. Während ihres Lebens sind Objekte gekennzeichnet durch ihr *Verhalten*, ihre *Identität* und ihren *Status*.

Verhalten, Identität und Status eines Objektes

Das Verhalten eines Objektes, d.h. wie ein Objekt auf eine Nachricht (Methodenaufruf) reagiert, wird prinzipiell durch seine Klasse beschrieben. Meist arbeiten die Methoden dabei mit den internen Daten. Zwei Objekte derselben Klasse haben beide ihren eigenen Satz an Daten, haben also beide eine eigene Identität. Je nachdem welche Werte die Daten besitzen, kann das Objekt in einem unterschiedlichen Zustand sein. Der Status eines Objektes wird also charaktierisiert durch den momentanen Zustand seiner Daten.

Ein Objekt unsere Klasse `CStack` aus Listing 3.2 kann folgende Zustände einnehmen:

- Der Stack ist leer (`m_Top=-1`).
- Der Stack ist voll (`m_Top == m_Max`).
- Der Stack ist im Füllzustand (`0 <= m_Top < m_Max`).

Wenn wir nun zwei Stacks anlegen, stapel1 und stapel2, hat jedes Objekt seine eigene Identität und seinen eigenen Status. Je nach Status kann es auch sein, dass die Objekte auf die gleiche Botschaft unterschiedlich reagieren, obwohl sie prinzipiell das gleiche Verhalten aufweisen. Besitzen die Objekte stapel1 und stapel2 den aktuellen Zustand, wie in Abbildung 3.7 dargestellt, so wird stapel1 auf die Botschaft `Push(3)` den Wert 3 im Stapel speichern, während stapel2 die Botschaft zurückweist (`return false`).

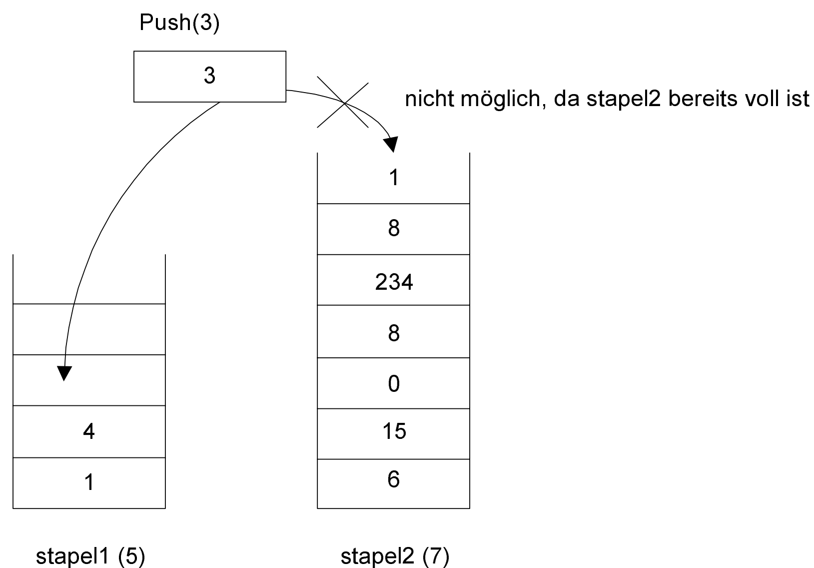

Abbildung 3.7: Identität und Zustand von stapel1 und stapel2

Zustandsdiagramm in der UML

Die UML stellt ein Diagramm zur Verfügung, um die Zustände eines Objektes (oder auch eines Systems) zu modellieren: das *Zustandsdiagramm* (engl. *state diagram*).

Im Zustandsdiagramm werden neben den Zuständen auch die Zustandsübergänge modelliert, d.h. wie das Objekt von einem Zustand in den nächsten gelangt.

Verwandte Begriffe für ein Zustandsdiagramm sind daher auch *Zustandsübergangsdiagramm* (engl. *state transition diagram*) oder *Endlicher Automat*, da das Diagramm eine hypothetische Maschine beschreibt, die sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet.

Im Zustandsdiagramm werden die Zustände durch abgerundete Rechtecke dargestellt. Die einzelnen Zustände werden mit Pfeilen verbunden, die die Zustandsübergänge beschreiben. Der Zustandsübergang ist gekennzeichnet durch ein Ereignis, das den Zustandsübergang auslöst, und eventuell einer Aktion, die dabei stattfindet. Sofern die Namen des Ereignisses und der Aktionsoperation übereinstimmen, ist nur die Operation aufgeführt. Ist der Zustandsübergang an eine Bedingung geknüpft, so kann diese in `[ ]` (eckigen Klammern) angegeben werden. Startzustand und Endzustand werden durch eigene Kreissymbole modelliert, wie dies in Abbildung 3.8 dargestellt ist.

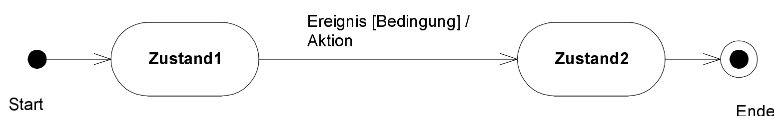

Abbildung 3.8: Allgemeiner Zustandsübergang in UML

Abbildung 3.9 zeigt das Zustandsdiagramm für unsere Klasse `CStack`.

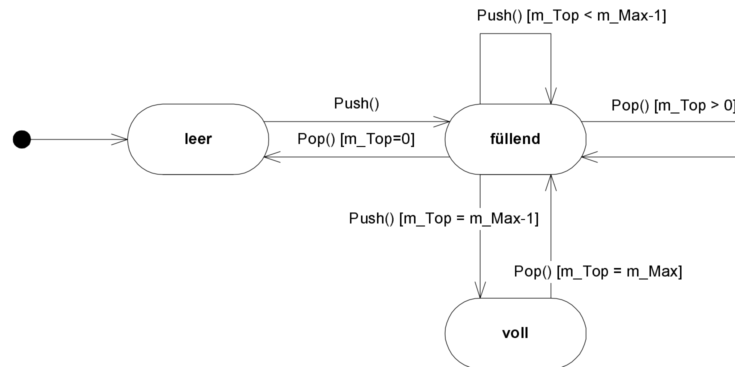

Abbildung 3.9: Zustandsdiagramm zu CStack

Ein Zustandsdiagramm ist ein gutes Mittel, um zu überprüfen, ob sich ein Objekt stabil verhält und nicht irgendwann in einen undefinierten Zustand gerät, aus dem es im schlimmsten Fall auch nicht mehr herauskommt. Potentielle Fehler können so im Vorfeld vermieden werden. So hätten bei unserer Klasse `CStack` die Fälle ‚voll' und ‚leer' leicht übersehen werden können, was fatale Folgen gehabt hätte (Speicher-über-/unterschreitung).

Auch für den Tester einer Klasse ist ein Zustandsdiagramm eine hilfreiche Grundlage, um eine Klasse auf Vollständigkeit zu überprüfen.

Eine Klasse sollte prinzipiell so entworfen werden, dass jede Methode zu einem x-beliebigen Zeitpunkt aufgerufen werden kann, ohne dass das Objekt in einen undefinierten Zustand gerät. Es ist nicht sehr anwenderfreundlich (für den Anwender einer Klasse), wenn bei den Methodenaufrufen eine bestimmte Reihenfolge eingehalten werden muss. Auch dies lässt sich am Zustandsdiagramm überprüfen.

Zustandsdiagramme sind also sehr hilfreich, um das Verhalten eines Objektes in seiner Gesamtheit zu beleuchten. Je nach Klasse lohnt sich der Aufwand, ein solches Diagramm zu zeichnen, mehr oder weniger. Man sollte sich jedoch in jedem Fall Gedanken darüber machen, welche Zustände die Objekte während ihres Lebens einnehmen können.

In Zustandsdiagrammen können auch noch mehr Aspekte modelliert werden, z.B.

- Unterzustände
- Aktionen, die während des Zustandes ausgeführt werden bzw. beim Eintritt oder Austritt

Wir wollen an dieser Stelle aus Übersichtsgründen hierauf nicht näher eingehen und stattdessen auf entsprechende Literatur verweisen.

Konstruktor und Destruktor

Wird ein Objekt angelegt /geboren, so wird in jedem Fall der zugehörige Konstruktor durchlaufen. Hier sollten die Attribute des Objektes initialisiert werden und das Objekt somit in einen definierten (Ausgangs-)zustand gebracht werden (vgl. Kapitel 2.12.8, Konstruktoren für Initialisierungen, auf Seite 84).

Beachtet werden sollte in diesem Zusammenhang, dass der Konstruktor nicht explizit aufgerufen werden kann. Es macht daher unter Umständen Sinn, eine Methode `Init()` einzuführen, mit Hilfe derer das Objekt jederzeit wieder in den Ausgangszustand versetzt werden kann. Diese Methode `Init()` könnte dann auch im Konstruktor aufgerufen werden.

Beim Tod eines Objektes wird automatisch der zugehörige Destruktor aufgerufen (vgl. Kapitel 2.12.9, Destruktoren für Aufräumarbeiten, auf Seite 88). Im Konstruktor vorgenommene Aktionen, die über das Initialisieren von Attributen hinausgehen (z.B. **new** oder Öffnen einer Datei), sollten im Destruktor auch wieder rückgängig gemacht werden.

Objektkopien

Die Attributwerte eines Objektes können standardmäßig in ein anderes Objekt derselben Klasse kopiert werden. Dies kann sowohl beim Anlegen eines Objektes geschehen als auch während der Lebensdauer eines Objektes. Je nachdem, ob bei der Zuweisung gleichzeitig ein neues Objekt angelegt wird oder nicht, kommt der so genannte *Kopierkonstruktor* oder *der Zuweisungsoperator* zum Einsatz. Das folgende Beispiel verdeutlicht dies:

```
CStack s1;
CStack s2=s1;  // Kopierkonstruktor, weil während der
               // Konstruktion des Objektes eine Zuweisung
               // durchgeführt wird
               // ist gleichbedeutend mit der Syntax:
               // CStack s2(s1)

s1=s2;        // Zuweisungsoperator:
              // Hier wird kein neues Objekt konstruiert,
              // sondern ein bestehendes Objekt einem
              // anderem zugewiesen
```

Dass dies möglich ist, liegt daran, dass der Compiler für den Kopierkonstruktor und den Zuweisungsoperator – wie auch für Konstruktor und Destruktor – eine Standardimplementierung zur Verfügung stellt.

Diese Standardimplementierung kopiert jeden Attributwert des einen Objektes in das entsprechende Attribut des anderen Objektes. Dieses Vorgehen wird als *flache Kopie* bezeichnet.

Problematisch wird eine flache Objektkopie, wenn es sich bei den Attributen um Zeiger handelt. Denn es werden stets nur die Elementvariablen an sich (also die Zeiger) kopiert, nicht jedoch der Speicherbereich, auf den die Zeiger zeigen. Dies führt also dazu, dass anschließend beide Objekte auf den gleichen Speicherbereich zeigen. Will man dies vermeiden, müssen der Kopierkonstruktor und der Zuweisungsoperator überladen werden und eine so genannte *tiefe Objektkopie* implementiert werden. Von einer tiefen Objektkopie spricht man, wenn der gesamte Speicher kopiert wird, den das Objekt benötigt, d.h. bei einem Zeiger der Bereich, auf den der Zeiger verweist. Anschließend wird der Zeiger dann entsprechend auf den neuen kopierten Bereich gestellt, statt nur den Zeiger an sich zu kopieren (flache Kopie).

Die Auswirkungen einer fehlenden tiefen Objektkopie werden anhand des Kopierkonstruktors im folgenden Unterkapitel verdeutlicht.

Kopierkonstruktor

Der *Kopierkonstruktor* (engl. *copy constructor*) wird statt des Konstruktors aufgerufen, und zwar immer dann, wenn beim Anlegen eines Objektes die Werte für dessen Attribute von einem bestehenden Objekt derselben Klasse übernommen (»kopiert«) werden.

Dies ist in folgenden Fällen der Fall:

- Explizit beim Anlegen eines Objektes, z.B:

  ```
  CKreis kreis1(1,2);
  CKreis kreis2=kreis1;// Kurzschreibweise für
                       // CKreis kreis2(kreis1)
  ```

- Implizit, wenn als Parameter einer Methode ein Objekt Call by Value übergeben wird, z.B.

  ```
  CAdresse::SetName(CString name){...}
  ```

  (angelegt wird hier das lokale Objekt name, in das die Werte des übergebenen Objektes kopiert werden)

- Implizit, wenn als Returnwert ein Objekt zurückgeliefert wird. (Man kann sich den Returnwert als ein unbenanntes Objekt auf dem Stack einer Funktion vorstellen, das nur kurzzeitig während der Returnanweisung angelegt wird und nach Zuweisung an den Aufrufer wieder zerstört wird; sozusagen am Ende der return-Anweisung.)

Wie bereits erwähnt, erstellt der vom Compiler standardmäßig zur Verfügung gestellte Kopierkonstruktor lediglich eine flache Kopie, was jedoch für Zeiger-Attribute nicht ausreicht.

Wir wollen diese Problematik anhand einer Klasse `CStr` für Strings verdeutlichen. Diese Klasse kapselt eine Elementvariable vom Typ char*, um immer genau so viel Speicherplatz in Anspruch zu nehmen, wie wirklich benötigt wird.

Listing 3.3(`string.cpp`) zeigt die Klasse `CStr` ohne überladenen Kopierkonstruktor.

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

class CStr
{
   char* m_str;
public:
```

```
   CStr();          // Standardkonstruktor
   CStr(char* str); // Konstruktor
   ~CStr();         // Destruktor

   bool SetChar(unsigned pos, char b);
   void ShowLine();
};

//=======================================================
CStr::CStr()           // Standardkonstruktor
{
   m_str=NULL;
}
//-------------------------------------------------------
CStr::CStr(char* str)      // Konstruktor
{
   m_str=new char[strlen(str)+1];
   if(m_str)
      strcpy(m_str,str);
}
//-------------------------------------------------------
CStr::~CStr()            // Destruktor
{
   if(m_str) // Speicherplatz für m_str wurde besorgt
      delete[] m_str;
}
//-------------------------------------------------------
bool CStr::SetChar(unsigned pos, char b)
{
   if(pos >= strlen(m_str)) // Ungültige Position
      return false;

   m_str[pos]=b;
   return true;
}
//-------------------------------------------------------
void CStr::ShowLine()
{
   cout << m_str << endl;
}
//=======================================================
int main(void)
{
   CStr string1("Anna");
   CStr string2=string1;    // Kopierkonstruktor,
```

```
                               // da Zuweisung beim Anlegen
                               // des Objektes

   string2.SetChar(3,'e');
   string1.ShowLine();
   string2.ShowLine();

   return 0;    // Aufruf von Destruktor für
                // string1 und string2
}
```

Listing 3.3: (string.cpp) CStr ohne Kopierkonstruktor

In diesem Beispiel verweisen die Objekte `string1` und `string2` auf den gleichen Speicher, da standardmäßig lediglich eine flache Kopie erstellt wird. Damit führt das Programm zu folgender (sicherlich nicht gewünschten) Ausgabe und anschließend zum Absturz.

```
Anne
Anne
```

Abbildung 3.10 verdeutlicht die Situation.

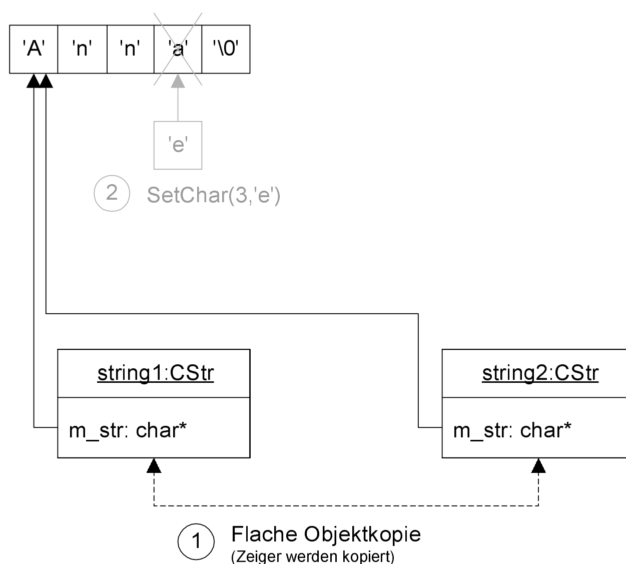

Abbildung 3.10: Flache Objektkopie bei CStr

Der Absturz erfolgt am Ende des Programmes, bei der **return**-Anweisung von `main()`. Hier wird erst der Destruktor von `string1` aufgerufen, in dem der Speicher, auf den `m_str` zeigt, mit **delete** freigegeben wird. Anschließend wird der Destruktor von `string2` aufgerufen und noch einmal versucht, den gleichen Speicher freizugeben, da auch `m_str` von `string2` auf diesen Speicher zeigt.

Hier muss also dafür gesorgt werden, dass eine tiefe Objektkopie erstellt wird, wie es in Abbildung 3.11 dargestellt ist.

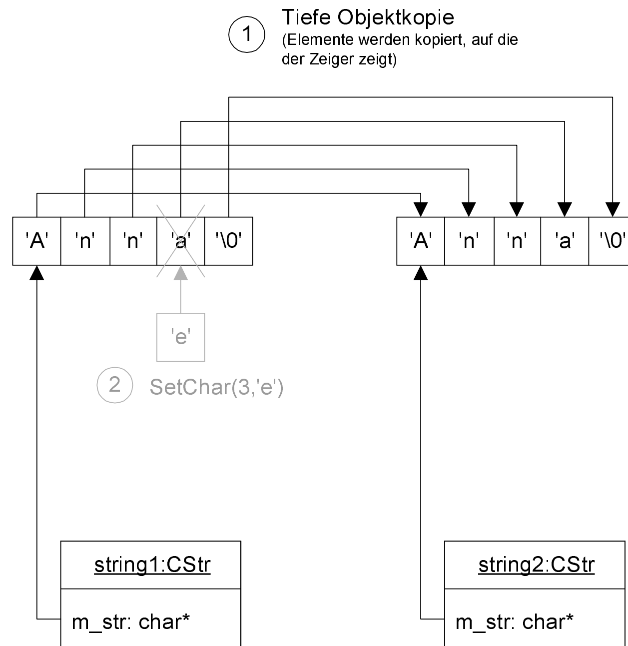

Abbildung 3.11: Tiefe Objektkopie bei CStr

Die tiefe Objektkopie erreichen wir durch Überladen des Kopierkonstruktors. Der Kopierkonstruktor besitzt als Parameter stets eine Referenz auf ein Objekt der Klasse, zu der er selbst gehört. Dieser Parameter wird häufig mit **const** angegeben (muss aber nicht).

Für unsere Klasse CStr könnte der Kopierkonstruktor wie folgt implementiert werden:

```
CStr::CStr(const CStr& str)   // Kopierkonstruktor
{
   if(str.m_str)   // str ist nicht leer
   {
      // Speicher gemäß str besorgen
      m_str=new char[strlen(str.m_str)+1];

      if(m_str)   // Allokation OK
         strcpy(m_str,str.m_str);   // String kopieren
      else
      {
         cerr << "Kein Speicher!!!";
         exit(1);   // Programmende
      }
   }
   else   // str ist leeres Stringobjekt
      m_str=NULL;
}
```

Nach dieser Definition würde unser Programm nun zu folgender Ausgabe (ohne Absturz) führen:

```
Anna
Anne
```

Der Zuweisungsoperator

Die Problematik mit flacher und tiefer Objektkopie betrifft nicht nur die Konstruktion von Objekten, sondern ebenso die Zuweisungen von Objekten an bereits bestehende. In diesem Fall wird der Zuweisungsoperator der Klasse aufgerufen, der standardmäßig eine flache Objektkopie erstellt.

Der Zuweisungsoperator lässt sich jedoch – wie auch der Kopierkonstruktor – überladen. Wie dies geschehen kann, wollen wir am Beispiel unserer Klasse CStr zeigen. Die Klasse CStr muss um eine Methode mit dem Namen **operator=** erweitert werden. Diese erhält sowohl als Parameter als auch als Rückgabewert eine Referenz auf ein Objekt der eigenen Klasse. Der Parameter entspricht dem rechten Operanden der Zuweisung, das Objekt selbst dem linken Operanden. Der Rückgabewert ist der Wert der Operation. Bei einer Zuweisung wird hier meist der Wert des linken Operanden verwendet, also das Objekt selbst.

```
CStr& CStr::operator=(const CStr& str) // Zuweisungsoperator
{
   // bisher in Anspruch genommenen Speicher freigeben
   if(m_str)
      delete[] m_str;

   if(str.m_str)   // str ist nicht leer
   {
      // Neuen Speicher gemäß str besorgen und str kopieren
      m_str=new char[strlen(str.m_str)+1];

      if(m_str)   // Allokation OK
         strcpy(m_str,str.m_str);   // String kopieren
      else
      {
         cerr << "Kein Speicher!!!";
         exit(1);        // Programmende
      }
   }
   else   // str ist leeres Stringobjekt
      m_str=NULL;

   return (*this);   // Ergebnis entspricht Wert des
                     // linken Operanden,der durch das
                     // aktuelle Objekt repräsentiert wird.
}
```

Wie sich an diesem Beispiel bereits vermuten lässt, lassen sich auch andere C++ - Operatoren überladen und auf eigene Klassen anwenden. Was dabei zu beachten ist, wird im Kapitel 4.2, Operatoren überladen, näher beschrieben.

Wichtig an dieser Stelle ist, dass bei der Definition einer eigenen Klasse prinzipiell folgende vom Compiler standardmäßig vorhandene Funktionen überdacht werden sollten und gegebenfalls überladen werden müssen:

- Der Standard-Konstruktor
  -> meist der Fall wegen Initialisierung von (nicht statischen) Membervariablen
- Der Standard-Destruktor
  -> für Aufräumarbeiten, z.B. delete, Schließen von Dateien

- Der Standard-Kopierkonstruktor
  -> bei Zeigerelementen
- Der Standard-Zuweisungsoperator
  -> bei Zeigerelementen

Als Faustregel kann gelten, dass der Zuweisungsoperator immer dann überladen werden sollte, wenn auch der Standard-Kopierkonstruktor überladen wird und umgekehrt.

Vergleicht man den Code der beiden Funktionen, so stellt man fest, dass diese nahezu identisch sind, was nicht verwunderlich ist, da sie eine ähnliche Aufgabe haben, nämlich das Kopieren eines bestehenden Objektes. Der Unterschied ist jedoch, dass der Kopierkonstrukor für ein Objekt immer nur einmal (nämlich automatisch beim Anlegen des Objektes) aufgerufen wird, wohingegen der Zuweisungsoperator für ein Objekt im Programm mehrmals aufgerufen werden kann (nämlich explizit bei jeder Zuweisung).

Um Code einzusparen und Copy-Paste-Aktionen zu vermeiden, können die beiden kombiniert werden, indem der Kopierkonstruktor den Zuweisungsoperator aufruft:

```
CStr::CStr(const CStr& str)   // Kopierkonstruktor
{
   // Membervariablen initialisieren (ist ja Aufgabe eines
   // Konstruktors)
   m_str=NULL;

   // Aufruf des Zuweisungsoperators:
   *this=str;
}
```

Diese Variante hat jedoch auch einen Nachteil: Ein Aufruf des Kopierkonstruktors dauert dann schon allein durch den zusätzlichen Methodenaufruf des Zuweisungsoperators länger und ist daher weniger performant.

Objekt-Attribute

Eine Klasse kann als Attribut auch wiederum eine Klasse enthalten. Um zu entscheiden, ob Kopierkonstruktor und Zuweisungsoperator überladen werden müssen, genügt jedoch lediglich ein Blick auf die

eigenen Membervariablen. Nur wenn diese Zeigertypen sind, muss eine Überladung stattfinden. Denn eine flache Objektkopie bedeutet genau genommen, dass für jedes Attribut des Objektes der Zuweisungsoperator aufgerufen wird; d.h., bei Verschachtelungen von Objekten wird also für jedes Objekt wieder der entsprechende Zuweisungsoperator aufgerufen, z.B.:

```
class CAdresse
{
   CStr m_Vorname;
   ...
};

int main(void)
{
   CAdresse adr1,adr2;
   adr1=adr2;   // Hier wird der Standard-Zuweisungsoperator
                // von CAdresse aufgerufen, dieser ruft
                // jedoch intern den Zuweisungsoperator von
                // CStr auf, um folgende Anweisung
                // auszuführen:
                // adr1.m_Vorname=adr2.m_Vorname;
   ...
   return 0;
}
```

Würde CAdresse für `m_Vorname` einen char* verwenden, müsste sie den Kopierkonstruktor und Zuweisungsoperator überladen. Da sie aber `CStr` verwendet, ist es nunmehr das ‚Problem' der Klasse `CStr`, die dieses Problem ja bereits löst.

3.2.8 C++-Zugriffsspezifizierer sind klassenbezogen

Die Zugriffsspezifizierer in C++ (private, protected, public) beziehen sich auf die Klasse, nicht auf die Objekte; d.h. existieren zwei Objekte derselben Klasse, so kann das eine Objekt auf die private-Elemente des anderen Objektes zugreifen.

Als Beispiel wollen wir unsere Klasse `CStr` um eine Methode `Append()` erweitern. Hier kann das aktuelle String-Objekt auf das Attribut `m_str` des Parameter-Objektes `str` direkt zugreifen.

```
CStr CStr::Append(CStr str)
{
   char * str_new;
   int myLen;

   // Ist aktueller String ein Leer-String?
   myLen= m_str? strlen(m_str):0;

   str_new = new char[myLen + strlen(str.m_str) +1];
   if(!str_new)  // Allokation nicht möglich
      return *this;   // -> alles bleibt beim Alten

   if(myLen)     // war bisher schon String enthalten
   {                 // -> neuen String anhängen
      strcpy(str_new,m_str);
      strcat(str_new,str.m_str);
      delete[] m_str;
   }
   else         // bisher Leer-String
      strcpy(str_new,str.m_str);  // -> neuen String
                                  //    kopieren

   m_str=str_new;
   return *this;
}
...
int main(void)
{
 ...
   CStr s1("Gretchen"),s2;

   s2=s1.Append(CStr(" Mueller"));  // Aufruf von Append
   s1.ShowLine();         // Verändert wird s1...
   s2.ShowLine();         // ... und s2

   return 0;
}
```

Listing 3.4: string.cpp ergänzt durch CStr: : Append

Listing 3.4 würde zu folgender Ausgabe führen:

```
Gretchen Mueller
Gretchen Mueller
```

Wir wollen an dieser Stelle explizit darauf hinweisen, dass obiger Source-Code in C++ zwar erlaubt ist, jedoch streng genommen dem Objektgedanken widerspricht. Im übertragenen Sinn würde dies bedeuten, dass ich in die Tasche meines Nachbarn greifen darf (eigentlich Privatsphäre), weil wir beide – mein Nachbar und ich – vom Typ Mensch sind. Objektorientierter wäre es also, wenn die Klasse `CStr` Methoden zur Verfügung stellen würde, mit denen die privaten Daten erfragt werden können. Dies hat jedoch folgende Nebenwirkungen:

- Erfolgt der Zugriff auf die privaten Elemente über Methoden statt direkt, so ist dies unter Umständen weniger performant, weil ein Methodenaufruf erfolgen muss. Die Methoden sollten daher auf jeden Fall inline definiert werden.
- Gibt es public-Methoden, um die privaten Daten zu erfragen, so stehen diese allen Objekten, egal von welcher Klasse, zur Verfügung.

Es hat also durchaus seine Berechtigung, warum C++ dies so regelt. Außerdem wäre sonst das Überladen des Kopierkonstruktors bzw. des Zuweisungsoperators nicht so einfach. Die Klassen müssten dann immer erst für jedes Attribut eine entsprechende `Get()`-Methode zur Verfügung stellen.

3.2.9 Statische Klassenelemente

Es gibt Fälle, in denen Daten nicht *objektbezogen*, sondern *klassenbezogen* benötigt werden, d.h. dass die Daten nur einmal pro Klasse exisitieren und nicht für jedes Objekt in einer eigenen Ausfertigung. Ein solches Datenelement wird als *statisch* bezeichnet und als **static** spezifiziert. Auf ein statisches Datenelement können alle Objekte der Klasse gemeinsam zugreifen. Auch Elementfunktionen können als static vereinbart werden und sind damit klassenbezogen anzusehen. Statische Elementfunktionen haben nur direkten Zugriff auf die statischen Elemente einer Klasse und benötigen für ihren Aufruf von außerhalb der Klasse kein Objekt.

Als kleines Demonstrationsbeispiel wollen wir eine Klasse `CRandom` verwenden, die mitzählt, wie viele Zufallszahlen bisher angelegt wurden.

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

class CRandom
{
   int m_value;
   static int m_cnt;   // Dies ist lediglich die
                       // DEKLARATION des statischen
                       // Datenelements m_cnt. Es wird
                       // noch kein Speicher belegt.
                       // Das Schlüsselwort static
                       // ist erforderlich.

public:
   CRandom()
   {
      if(m_cnt==0)
         srand(time(NULL));

      m_value=rand();
      m_cnt++;
   }
   ~CRandom()
   {
      m_cnt--;
   }
   int static GetCnt()   // statische Elementfunktion
   {
      return m_cnt;
   }
   int GetValue()
   {
      return m_value;
   }
};

int CRandom::m_cnt=0;   // DEFINITION des statischen Elements.
                        // Das Schlüsselwort static
                        // darf hier nicht verwendet werden.

int main(void)
{
   CRandom* werte[5];
```

```
   cout <<"5 Werte im Array verfuellen: \n"
       <<"===========================" << endl;

   for(int i=0; i<5; i++)
   {
      werte[i]=new CRandom;
      cout << "aktuelle Anzahl der Werte: "
           << CRandom::GetCnt() << endl;
      cout << "akutell generierter Wert: "
           << werte[i]->GetValue() << endl;
   }
   cout << endl;


   {
      CRandom ZusatzZahl;
      cout << "Anzahl Werte im Block: "
           << ZusatzZahl.GetCnt() << endl;
   }

   cout << "Anzahl Werte nach Block: "
        << CRandom::GetCnt() << endl;
   return 0;
}
```

Listing 3.5: (random.cpp): Die Klasse CRandom mit statischen Elementen

Listing 3.5 führt zu folgender Ausgabe:

```
5 Werte im Array verfuellen:
===========================
aktuelle Anzahl der Werte: 1
akutell generierter Wert: 27775
aktuelle Anzahl der Werte: 2
akutell generierter Wert: 23606
aktuelle Anzahl der Werte: 3
akutell generierter Wert: 20555
aktuelle Anzahl der Werte: 4
akutell generierter Wert: 26767
aktuelle Anzahl der Werte: 5
akutell generierter Wert: 18346

Anzahl Werte im Block: 6
Anzahl Werte nach Block: 5
```

Statische Datenelemente werden also innnerhalb der Klasse nur deklariert. Die Definition der Datenelemente erfolgt außerhalb der Klasse. Hier darf das Schlüsselwort static nicht verwendet werden. Werden statische Elementfunktionen außerhalb der Klasse definiert, entfällt auch hier das Schlüsselwort static.

Wird bei statischen Datenelementen der Zugriffsspezifizierer **public** verwendet, so sind sie wie globale Variablen anzusehen, die allerdings im Namensraum der Klasse liegen. Der Zugriff auf die Daten kann dann jederzeit analog zu statischen Elementfunktionen auch ohne Objekt erfolgen, z.B.

```
cout << CRandom::m_cnt;
```

Sind statische Datenelemente **private** (was für Attribute ja üblich ist), so muss die Initialisierung bei der Definition erfolgen. Es macht keinen Sinn, die statischen Datenelemente im Konstruktor zu initialisieren, weil die Wertzuweisung sonst mit jedem Anlegen eines neuen Objektes erfolgt, was sicherlich nicht erwünscht ist.

Statische Elementfunktionen haben keinen versteckten **this**-Pointer. Daher sind sie geringfügig performanter und können – wie bereits erwähnt – logischerweise nur auf die statischen Attribute zugreifen und nicht auf die jeweiligen Attribute des Objektes.

Anwendungsfälle für statische Datenelemente (und damit verbunden statische Elementfunktionen) sind z.B.:

- Beim ersten angelegten Objekt einer Klasse sind zusätzlich Aktionen erforderlich, die bei allen weiteren Objekten jedoch nicht mehr nötig sind, z.B. Öffnen einer Datei
- Es muss sichergestellt werden, dass es im System nur ein Objekt dieser Klasse gibt.

Da statische Elemente sich auf die Klasse beziehen, werden sie auch als *Klassenattribute* bezeichnet. In der UML werden Klassenattribute unterstrichen. Unsere Klasse `CRandom` würde damit wie in Abbildung 3.12 dargestellt werden:

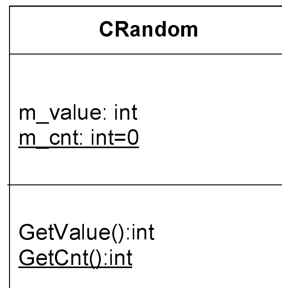

Abbildung 3.12: Klassendiagramm zur Klasse CRandom

3.2.10 Bibliotheksklassen

Es gibt mittlerweile etliche Klassenbibliotheken, die fertige Klassen zur Verfügung stellen, nicht zuletzt die Standard Template Library (STL), die zum Sprachumfang von C++ zählt (vgl. Kapitel 4.6).

Klassen, die von allgemeinem Interesse sind, wie z.B. eine Klasse für Strings oder eine Stack-Klasse, finden sich dort wieder. Natürlich sollte man erst vorhandene Bibliotheken konsultieren, bevor man das Rad zweimal erfindet. Allerdings sollte man sich dann mit der Funktionsweise der Klasse genau auseinander setzen und überprüfen, inwieweit diese den Anforderungen im vorliegenden Projekt entspricht.

Auch wenn es für die eine oder andere Problemstellung bereits eine fertige Klasse in einer Bibliothek gibt, sollte man als Neueinsteiger in die Objektorientierung und C++ zu Übungszwecken ruhig einmal eine eigene Implementierung einer solchen Klasse versuchen. Es hat sicherlich bisher auch noch keinem C-Einsteiger geschadet, wenn er zu Übungszwecken `strcpy()` einmal selbst programmiert hat.

3.2.11 ÜBUNG: Fragen zur Lernkontrolle

Bei dieser Übung handelt es sich nicht um Fragen, die explizit in diesem Kapitel geklärt wurden. Vielmehr wiederholen sie die Syntax, die im Laufe von Kapitel 2, Nicht objektorientierte Erweiterungen in C++, erklärt wurde. Die Fragenstellungen sind hier jedoch bezogen auf Klassen und Objekte formuliert und nicht bezogen auf Strukturen, anhand derer die Syntax erklärt wurde.

Welche Voraussetzungen müssen erfüllt sein, damit die Objekte punkt1 und punkt2 im folgenden Codeausschnitt angelegt werden können?

```
#include "Punkt.h"

int main(void)
{
   CPunkt    punkt1(10,20),
             punkt2;
}
```

1. Alle Konstruktoren der Klasse `CPunkt` müssen **public** sein.
2. Der Standardkonstruktor muss explizit angelegt werden.
3. Es muss ein Konstruktor definiert sein, der zwei Parameter vom Typ **int** besitzt.
4. Keine der obigen Voraussetzungen muss unbedingt erfüllt sein.

Welche der folgenden Aussagen ist/sind richtig?

1. Klassennamen müssen immer mit C anfangen, sonst meldet der Compiler einen Fehler.
2. Wenn Funktionen innerhalb der Klassendefinition definiert werden, sind sie automatisch **inline**.
3. Inline-Funktionen sind automatisch **public**.
4. Eine Klasse ist ein komplexer, selbst definierter Datentyp, der nur aus einfachen C++ Datentypen wie **int**, **char** usw. bestehen darf.
5. Alle Methoden einer Klasse können von allen anderen Klassen aufgerufen werden.
6. Keine der Antworten ist richtig.

Wo initialisiert man die (nicht statischen) Membervariablen eines Objektes?

1. In der Klasse.
2. Im Destruktor.
3. Im Konstruktor.

4. Wahlweise in der Klasse oder im Konstruktor.
5. Keine der Antworten ist richtig.

Welche der folgenden Aussagen trifft für Methoden einer Klasse zu?

1. Alle haben den gleichen Rückgabedatentyp.
2. Alle haben den gleichen Rückgabewert.
3. Sie können auf alle Membervariablen der Klasse zugreifen.
4. Sie können nur auf die **private**-Membervariablen der Klasse zugreifen.
5. Steht ihre Definition außerhalb der Klassendefinition, muss vor dem Funktionsnamen der Klassennamen gefolgt von einem doppelten Doppelpunkt :: angegeben werden.
6. Keine der Antworten ist richtig.

3.2.12 ÜBUNG: Lesen von Prototypen in Online-Hilfe

Auch diese Übung zielt in erster Linie auf die in Kapitel 2, Nicht objektorientierte Erweiterungen in C++, erklärte Syntax ab und wird nun aber im Hinblick auf Klassen/Objekte gestellt.

In der Online-Hilfe finden Sie zu der Funktion `cin.ignore()` folgende Angaben:

---

istream::ignore

istream& ignore(int nCount = 1, int delim = EOF);

Parameters

nCount

The maximum number of characters to extract.

delim

The delimiter character (defaults to **EOF**).

Remarks

Extracts and discards up to *nCount* characters. Extraction stops if the delimiter *delim* is extracted or the end of file is reached. If *delim* = **EOF** (the default), then only the end of file condition causes termination. The delimiter character is extracted.

Was sagt Ihnen der Prototyp? Welche Möglichkeiten gibt es prinzipiell, `cin.ignore()` *aufzurufen?*

Was bewirken folgende Aufrufe:

- `cin.ignore();`
- `cin.ignore(10);`
- `cin.ignore(100, ':');`

Ist auch der Aufruf `cin.ignore(':')` *möglich? Wenn ja, was bewirkt er?*

Handelt es sich bei der Funktion `ignore()` *um eine globale Funktion, oder handelt es sich um eine Methode? Was bedeutet das für den Aufruf der Funktion? Was schließen Sie daraus für cin und von welchem Typ ist cin?*

Sagen Sie mit Ihren eigenen Worten, welchen Rückgabetyp die Funktion besitzt.

3.2.13 ÜBUNG: Realisierung einer Queue

Eine weitere grundlegende Datenstruktur in der Informatik neben dem *Stack* ist die so genannte *Queue*, welche eine Schlange vor einem Postschalter oder einem Eintritt in eine Veranstaltung simuliert. Es handelt sich dabei wieder um eine Datenstruktur, für die nur zwei Operationen definiert sind:

- `Put()` zum Einfügen eines Elements am Ende der Queue und
- `Get()` zum Entfernen des Elements am Anfang der Queue.

Abbildung 3.13 verdeutlicht die Funktionsweise einer Queue.

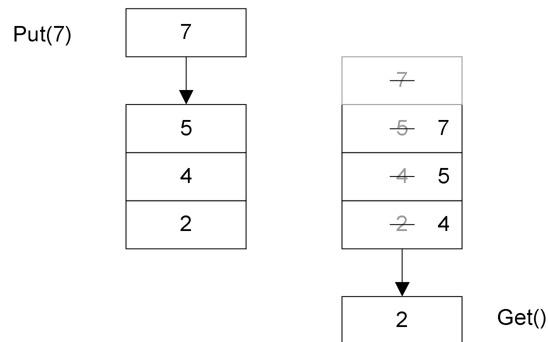

Abbildung 3.13: Funktionsweise einer Queue

Während ein Stack nach dem *LIFO*-Prinzip (*last in, first out*) arbeitet, arbeitet eine Queue nach dem *FIFO*-Prinzip (*first in, first out*). Ist die maximale Größe einer Queue konstant, bietet sich als eine einfache Realisierung ein eindimensionales Array an. Erstellen Sie nun ein Programm `queue.cpp`, das die Verwaltung einer Queue wie z.B. vor einem Postschalter oder in einem Ärztezimmer übernimmt. Dem Bediener des Programms sollen dabei folgende Möglichkeiten angeboten werden:

- Ankunft eines Kunden mit Eingabe des Namens, welcher der Einfachheit halber nur aus einem Zeichen besteht. (entspricht Einordnen am Ende der Queue)
- Bedienung eines Kunden (entspricht Entfernen aus der Queue)
- Auflisten der aktuellen Warteschlange entsprechend der Reihenfolge

Dieses Programm `queue.cpp` soll eine Klasse `CQueue` definieren, die für die Verwaltung einer Queue von Namen (hier nur Buchstaben!) verantwortlich ist. Dazu soll sie die folgenden Methoden anbieten:

```
bool    IsEmpty() { return m_count==0;} // Queue leer ?
bool    IsFull()  { return m_count==m_max; } // Queue voll ?
char    Get();
bool    Put(char name);
```

```
void    Contents();  // Auflisten der Buchstaben in der
                     // Queue entsprechend der Reihenfolge
```

Zur internen Verwaltung dieser Queue sollten Sie vier Variablen benutzen:

- m_Max enthält die maximale Anzahl von Personen, die die Warteschlange aufnehmen kann.
- m_Count enthält die aktuelle Anzahl der Personen, die sich in der Warteschlange befinden.
- m_First ist der Index der Person im Array, die sich an erster Stelle in der Warteschlange befindet und als Nächstes zu bedienen ist.
- m_Last ist der Index der Person im Array, die sich an letzter Stelle in der Warteschlange befindet.

Ist das Ende des Arrays erreicht, müssen wieder die vorderen freien Plätze benutzt werden. Das Aussehen der Funktion main() ist in Listing 3.6 (queue.cpp) gezeigt.

```
int main(void)
{
   const int MaxQueue = 10;
   CQueue    queue(MaxQueue);
   char      eingabe;
   char      name;

   while (1)
   {
      cout << endl << endl
           << "A   Ankunft eines neuen Patienten\n"
              "B   Bedienung des naechsten Patienten\n"
              "Q   Aktuellen Queue-Inhalt anzeigen\n"
              "E   Ende des Programms\n\n"
              "               Deine Wahl: ";
      cin >> eingabe;
      cout << endl;
```

```
      if (eingabe == 'e')
         break;
      else if (eingabe == 'a')
      {
         if (!queue.IsFull())
         {
            cout << "... Name des Patienten
                          (als Buchstabe): ";
            cin >> name;
            queue.Put(name);
         }
         else
            cout << "...... Wartezimmer ist voll ...."
                 << endl;
      }
      else if (eingabe == 'b')
      {
         if (queue.IsEmpty())
            cout << "...... Wartezimmer ist leer ...."
                 << endl;
         else
            cout << "...... '" << queue.Get()
                 << "' wird nun bedient" << endl;
      }
      else if (eingabe == 'q')
      {
         queue.Contents();
      }
   }
   return 0;
}
```

Listing 3.6: (queue.cpp): main()-Funktion, die Klasse CQueue verwendet

Ihre Aufgabe ist es nun, das Listing 3.6 (`queue.cpp`) um die fehlenden Konstrukte zu ergänzen, so dass es z.B. folgenden Ablauf zeigt. Bevor Sie mit der Implementierung beginnen, sollten Sie ein Klassen- und ein Zustandsdiagramm zu dieser Klasse zeichnen.

```
A   Ankunft eines neuen Patienten
B   Bedienung des naechsten Patienten
Q   Aktuellen Queue-Inhalt anzeigen
E   Ende des Programms
               Deine Wahl: a↵
```

```
... Name des Patienten (als Buchstabe): h↵

A   Ankunft eines neuen Patienten
......
               Deine Wahl: a↵
... Name des Patienten (als Buchstabe): x↵
.......
               Deine Wahl: a↵
... Name des Patienten (als Buchstabe): m↵
.......
               Deine Wahl: q↵
 1. h
 2. x
 3. m
.......
               Deine Wahl: b↵
...... 'h' wird nun bedient
.......
               Deine Wahl: b↵
...... 'x' wird nun bedient
.......
               Deine Wahl: a↵
... Name des Patienten (als Buchstabe): c↵
.......
               Deine Wahl: a↵
... Name des Patienten (als Buchstabe): y↵
.......
               Deine Wahl: q↵
 1. m
 2. c
 3. y
.......
               Deine Wahl: b↵
...... 'm' wird nun bedient
.......
               Deine Wahl: b↵
...... 'c' wird nun bedient
.......
               Deine Wahl: b↵
...... 'y' wird nun bedient
.......
               Deine Wahl: b↵
...... Wartezimmer ist leer ....
.......
               Deine Wahl: e↵
```

3.2.14 ÜBUNG: Aufrufe durch Compiler

Legen Sie eine Klasse Object an, die folgende »Methoden« überlädt:

- Standard-Konstruktor
- Destruktor
- Kopier-Konstruktor
- Zuweisungsoperator

Diese »Methoden« enthalten lediglich eine entsprechende Ausgabeanweisung, z.B. `cout << "Destruktor von Object" << endl;`

Denken Sie sich eine `main()`-Funktion aus, in der jede »Methode« mindestens einmal aufgerufen wird.

3.2.15 ÜBUNG: Realisierung einer Klasse CIntArray

Erstellen Sie eine Klasse `CIntArray`, die ein Integer-Array kapselt. Die Klasse soll sicherstellen, dass es zu keinerlei Speicherüber- oder unterschreitung kommt. Die Größe des Arrays legt der Nutzer der Klasse beim Anlegen eines `CIntArray`-Objektes fest. Die Klasse verfügt außerdem über eine Methode, um die Größe des Arrays erfragen zu können.

Erstellen Sie zunächst ein Klassendiagramm, bevor Sie mit der Implementierung beginnen. Testen Sie Ihre Klasse mit einer geeigneten `main()`-Funktion, die natürlich typische Fälle abprüfen sollte, wie z.B. einen bewussten Versuch einer Speicherüberschreitung.

Übrigens:

Es wird hier und auch an anderen Stellen oft bewusst kein festes `main()` oder ein möglicher Programmablauf vorgegeben. Denn man kann es nicht früh genug üben, sich vernünftige Testfälle – sprich ein geeignetes `main()` – auszudenken, um die Funktionsweise seiner Klasse (später Klassen) zu gewährleisten.

3.2.16 ÜBUNG: Realisierung einer Klasse CErrorLog

Erstellen Sie eine Klasse `CErrorLog`, deren Aufgabe es ist, Fehlermeldungen in einer Datei festzuhalten. Ein Objekt dieser Klasse ist fest mit der Datei verbunden, die beim Anlegen des Objektes angegeben wird. Die Datei wird im Konstruktor zum Anhängen geöffnet und im Destruktor geschlossen. Interessant ist die Klasse vor allem auch dann, wenn sie automatisch weitere Informationen zum aufgetretenen Fehler mitprotokolliert, z.B. die Zeit des Protokoll-Eintrages.

Folgende `main()`-Funktion zeigt eine mögliche Verwendung der Klasse:

```
#include <iostream.h>
#include <time.h>
#include <stdlib.h>     // f. srand

// Hier wäre die Klasse CErrorLog definiert

void delay(int sec);
//--------------------------------------------------
int main(void)
{
   CErrorLog liste17("fehler17.txt"),
             liste31("fehler31.txt");

   int nr;          // zufällige Fehlernummer

   // Zufallszahlengenerator initialisieren
   srand(time(NULL));

   // zufällig Fehler simulieren und protokollieren
   for(int i=0; i < 50 ; i++)
   {
      nr=rand();

      if(!(nr % 17))   // Fehler der Kategorie 17
         liste17.Log(nr);
      if(!(nr % 31))   // Fehler der Kategorie 13
         liste31.Log(nr);
```

```
      // etwas Zeit vertrödeln...
      cout << "Runde Nr: " << i << endl; // Ausgabe zu
                                         // Testzwecken
      delay(nr%3);
   }

   return 0;
}

//------------ delay() ----------------------------
// Hilfsfunktion wegen Portabilität. In der Praxis
// würde man hier auf Betriebssystem-Funktionen wie
// sleep() o.Ä. zurückgreifen.

void delay(int sec)
{
   time_t t_soll,   // Zeit, die erreicht werden soll
          t_ist;    // jeweils aktuelle Zeit

   // In Schleife warten, bis aktuelle Zeit Sollzeit
   // erreicht hat. Soll= Zeit_vor_Schleife + Wartezeit
   time(&t_soll);
   t_soll+=sec;
   while(time(&t_ist) != t_soll);
}
```

Nach Ablauf des Programms gibt es im aktuellen Verzeichnis die zwei Dateien `liste17.txt` und `liste31.txt`. Die Datei `liste17.txt` hat beispielsweise folgenden Inhalt:

```
Fehlernr:      9911   Tue Apr 17 18:48:24 2001
Fehlernr:      2737   Tue Apr 17 18:48:29 2001
Fehlernr:     13617   Tue Apr 17 18:48:36 2001
```

Um Ihnen mühseliges Suchen in der Online-Hilfe zu ersparen, hier die Aufrufe, die Sie dazu aus der C-Standardbibliothek (`#include <stdio.h>`) benötigen:

- **FILE* fopen(const char*** ***filename*****, const char*** ***mode*****);**
  Um die Datei zum Anhängen zu öffnen, verwenden Sie für `mode` den String »a«. Existiert die Datei nicht, so wird sie in diesem Modus erstellt. `FILE*` ist der Pointer auf die Datei, der im Fehlerfall `NULL` ist.

- **int fprintf(FILE* *file*, const char* *format [, argument]...*);**
  Diese Funktion arbeitet analog zu `printf()`; es wird lediglich als erster Parameter der File-Pointer für die Datei angegeben, in die die Ausgabe erfolgen soll. Im Fehlerfall ist der Rückgabewert ein negativer Wert.
- **int fclose(FILE* *file*);**
  Mit dieser Funktion wird die angegebene Datei wieder geschlossen. Im Erfolgsfall liefert die Funktion den Wert 0 zurück; im Fehlerfall `EOF`.

Im Beispiel enthält die Protokoll-Datei die jeweils aktuelle Zeit; diese kann prinzipiell wie folgt erfragt und ausgegeben werden:

```
#include <time.h>
#include <stdio.h>

int main( void )
{
   time_t t;
   struct tm *lt;    // Lokale Zeit in Form von struct tm

   time( &t );           // Sekunden seit 1.1.1970
   lt = localtime( &t );  // Konvertierung in eine struct tm

   // Ausgabe der local time als String
   printf( "Aktuelles Datum und Uhrzeit ist: %s\n",
           asctime( lt ) );

   return 0;
}
```

3.2.17 ÜBUNG: Kundenidentifikation

Erstellen Sie eine Klasse `CKunde`. Jeder Kunde soll über eine eindeutige Kundennummer identifiziert werden können, die automatisch beim Anlegen des Kunden generiert wird.

Vorsicht:

Sehen Sie für das Anlegen eines neuen Kunden eine eigene Methode `Anlegen()` vor und verwenden Sie nicht den Konstruktor, da dieser z.B. auch dann verwendet wird, wenn ein Kundenobjekt nur temporär angelegt wird, z.B. für eine Ringvertauschung innerhalb einer

Methode. Analoges gilt für das Löschen eines Kunden, genauer gesagt für das Freigeben einer Kundennummer; verwenden Sie hier eine eigene Methode `Loeschen()`.

3.3 Mehrere Klassen (Bsp. Personen-Queue)

Anspruchsvollere C++-Programme bestehen nicht nur aus einer, sondern aus mehreren Klassen. Hier stellt sich nun die Frage, welche Klassen benötigt werden. Eine erste einfache Grundregel, an die man sich bei der Analyse und dem Design von einfachen Programmen halten kann, ist die folgende Aussage:

Hinter jedem Substantiv, das in der Aufgabenbeschreibung vorkommt, verbirgt sich eine potentielle Klasse.

Hierzu soll nun zunächst ein Beispiel gezeigt werden:

Es ist ein Programm zu erstellen, bei dem das Warten von **Personen** in einer **Warteschlange** simuliert wird.

Ist die Person, die über einen Namen identifiziert wird, an der Reihe, so trägt sie ihr Anliegen vor.

Bei dieser Aufgabenstellung handelt es sich um eine Erweiterung der Übung »Realisierung einer Queue« (Kapitel 3.2.13), bei der lediglich der Name einer Person (in Form eines Buchstabens) verwaltet wurde. Die Person wird hier jedoch nicht nur über ihren Namen repräsentiert, sondern kann außerdem ihr Anliegen vortragen. Es lohnt sich daher, für die Person eine eigene Klasse einzuführen.

3.3.1 Klassendiagramm (statisches Design)

Stellen wir zunächst ein Klassendiagramm zu dieser Aufgabenstellung auf. Ein Klassendiagramm für mehrere Klassen zeigt die statische Beziehung zwischen den einzelnen Klassen, wie dies in Abbildung 3.14 gezeigt ist. Für das Personen-Queue-Beispiel haben wir zwei Klassen vorgesehen:

- `CQueue`
  Diese Klasse repräsentiert die Warteschlange und ist für die Verwaltung der Personen nach dem FIFO-Prinzip zuständig.

- `CPerson`
  Diese Klasse repräsentiert eine Person mit all ihrem Verhalten (Methoden) und Attributen.

Die Einführung der eigenen Klasse `CPerson` hat den Vorteil, dass wir nur diese Klasse bzw. die Methoden dieser Klasse ändern müssen, wenn wir das Verhalten der Person anders gestalten möchten, wie z.B. durch eine graphische oder akustische Ausgabe oder die Person stellt sich vorher vor mit Namen und Adresse (dann wäre sogar eine weitere Eigenschaft nötig). Die restlichen Klassen des Programms (hier nur eine) können dabei unverändert übernommen werden. In unserem Fall hat das Verhalten der Person ja nichts mit der Verwaltung an sich (`CQueue`) zu tun. Dies entspricht der Realität: Die Abhandlung der Patienten geschieht nach demselben Schema, trotzdem kann das Verhalten der Patienten variieren.

In einer ersten Grundregel gaben wir an, dass sich hinter jedem Substantiv eine potentielle Klasse verbirgt. Welche Substantive jedoch in der Aufgabenstellung vorkommen, hängt stark von der Formulierung derselben ab:

- Es gibt auch Substantive, die keine Klassen sind,
  z.B. Alter (der Person); hier handelt es sich um eine Eigenschaft von `CPerson`.
- Manchmal macht es auch Sinn, Klassen einzuführen, die auf den ersten Blick in der Aufgabenstellung – je nach Formulierung – als Substantive gar nicht erscheinen, z.B. eine eigene Klasse zum Ausgeben von Daten.

Man tut daher gut daran, sich auch einmal von der Formulierung der Aufgabenstellung zu lösen und mit Hilfe des gesunden Menschenverstandes nach Objekten zu suchen. Dies ist eigentlich auch gar nicht schwer, da es im Grunde unserem menschlichen Denken entspricht (vgl. Kapitel 3.2.2, Was ist eine Klasse und was nicht?, auf Seite 104). Wir haben es nur verlernt, uns dieser Denkweise bewusst zu werden (besonders die Programmierer mit langjähriger Erfahrung in prozeduraler Software-Entwicklung). Die stabilsten Software-Entwürfe sind oft die, die sich an der Realität orientieren. Freilich sieht die Software-Realität nicht immer wie unsere alltägliche Realität aus, und man muss

erst einen Blick dafür gewinnen. Man kann sich glücklich schätzen, wenn man Tiere, Autos, Personen, Bücher oder Ähnliches aus der alltäglichen Welt in der Aufgabenstellung findet; diese eignen sich auch hervorragend zu didaktischen Zwecken in Lehrbüchern. Allerdings hat auch die Software-Welt bei genauerem Hinsehen einiges an Objekten zu bieten, z.B. Verwaltungsmechanismen wie Stack, Queue, verkettete Liste und deren Elemente; man denke nur an die UNDO-Fähigkeit von Aktionen in Programmen oder an die Abarbeitung von Aufgaben/Jobs zwischen mehreren Threads. Außerdem darf nicht übersehen werden, dass bei komplexeren Programmen ein Unterschied besteht zwischen den Klassen der Analyse (OOA) und den Klassen des Designs (OOD). Denn das Design darf nicht nur die Realität von heute widerspiegeln, sondern muss auch die von morgen berücksichtigen. In Kapitel 6, Entwurfsprinzipien, wird daher auf designtechnische Aspekte näher eingegangen.

An dieser Stelle wollen wir den designtechnischen Aspekt auf folgende Regel reduzieren, auf die alle im ersten Schritt gefundenen Klassen im zweiten Schritt überprüft werden sollten:

Eine Klasse – eine Aufgabe,
d.h. jede Klasse sollte eine klar umrissene, überschaubare Aufgabe haben.

Die Aufgaben sollten dabei so gewählt sein, dass

- der Grad der Komplexität erniedrigt wird,
- der Grad der Wiederverwendung erhöht wird.

Beides ist in unserem Fall gegeben: Die Queue kümmert sich nur um die Verwaltung; das Verhalten der verwalteten Personen braucht die Queue nicht zu interessieren. Im Gegenzug ist es für eine Person irrelevant, wie die Verwaltung vonstatten geht, d.h. ob die Queue intern als Array oder z.B. als verkettete Liste realisiert ist. Eine andere Verwaltungsform würde die Klasse `CPerson` nicht tangieren. Durch Trennung der Problemstellung in die beiden Klassen wird also sowohl die Komplexität erniedrigt als auch die Wiederverwendbarkeit erhöht.

Diese Ausführungen sollen an dieser Stelle genügen, um erste Schritte in der objektorientierten Denkweise und Programmierung beschreiten zu können.

Wenden wir uns nun dem Klassendiagramm zu unserer Aufgabenstellung in Abbildung 3.14 zu. Dieses enthält einige neue Notationen:

Die schattierten Rechtecke mit einem Knick in der rechten oberen Ecke sind so genannte *Notizen*, welche eine Erklärung zur jeweiligen Klasse enthalten können.

Bei der Linie zwischen der Klasse CQueue und der Klasse CPerson handelt es sich um eine so genannte *Aggregation,* ein Sonderfall einer so genannten *Assoziation*. Eine Assoziation ist eine (ganz allgemeine) Beziehung zwischen verschiedenen Objekten einer oder mehrerer Klassen. Beim Sonderfall der Aggregation stehen die Objekte zueinander in Beziehung wie ein Ganzes zu seinen Teilen. In unserem Fall besteht die Warteschlange (CQueue) aus ein oder mehreren Personen (CPerson). Um eine Beziehung als Aggregation zu kennzeichnen, wird auf der Seite des Aggregats (des Ganzen) eine Raute gezeichnet. Die beiden Zahlen an den Enden der Linie geben an, wie viele Objekte auf der einen Seite der Aggregation (bzw. auch ganz allgemein bei Assoziationen) mit wie vielen Objekten auf der anderen Seite der Aggregation verbunden sind. In Abbildung 3.14 können wir somit erkennen, dass ein CQueue-Objekt aus kein, ein oder mehreren (*) Objekten der Klasse CPerson besteht. Wäre die Warteschlange von vornherein auf z.B. 10 Plätze begrenzt, würde statt des ‚*' dort als Menge ‚0..10' stehen.

Das Kapitel 3.3.7 geht auf die in der UML möglichen Beziehungen zwischen Objekten näher ein und erläutert diese im Überblick.

Macht man sich über die Klasse CPerson mehr Gedanken, so stellt man fest, dass diese ein Attribut m_Anliegen vom Typ eines Strings benötigt. Würde m_Anliegen als char* realisiert werden, so müsste für CPerson der Kopierkonstruktor und der Zuweisungsoperator überladen werden. Um dies zu vermeiden und die Komplexität damit zu verringern, aggregiert CPerson für m_Anliegen ein Objekt des Typs CStr. Die Klasse CStr wird jedoch im folgenden Klassendiagramm nicht explizit dargestellt, da sie eher als Datentyp fungiert.

Assoziationen bestehen streng genommen zwischen Objekten, nicht zwischen Klassen. Es ist jedoch üblich von einer Assoziation zwischen Klassen zu sprechen, obwohl streng genommen die Objekte dieser Klassen gemeint sind.

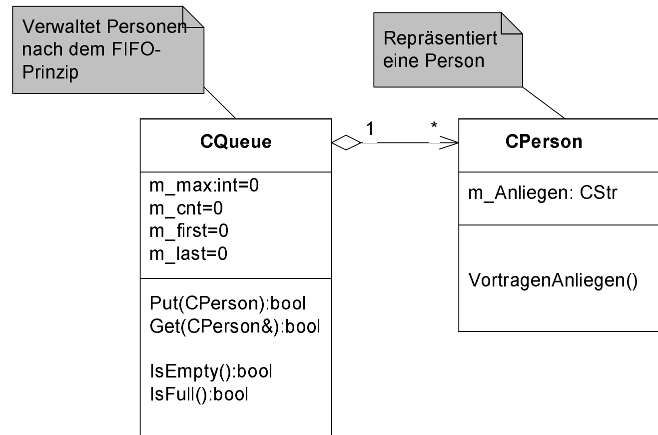

Abbildung 3.14: Klassendiagramm zum Personen-Queue-Beispiel

3.3.2 Sequenz- und Kollaborationsdiagramm (dyn. Design)

Anhand dieses Personen-Queue-Beispiels sollen weitere UML-Notationen vorgestellt werden: das *Sequenzdiagramm* und das *Kollaborationsdiagramm*. Während ein Klassendiagramm die Beziehungen der Klassen untereinander in der Art eines Bauplanes darstellt (statisch), werden Sequenz- und Kollaborationsdiagramme dazu verwendet, einen bestimmten Ablauf bzw. eine bestimmte Situation darzustellen (dynamisch). Ein Sequenz- bzw. Kollaborationsdiagramm gibt ein Szenario wider und zeigt die einzelnen Botschaften der Objekte untereinander, die nötig sind, um den gewählten Ablauf zu erreichen. Während Sequenzdiagramme den zeitlichen Ablauf in den Vordergrund stellen, betonen Kollaborationsdiagramme die prinzipielle Zusammenarbeit der beteiligten Objekte. Die dargestellten Sachverhalte sind ansonsten identisch. Wir wollen dies anhand unseres Personen-Queue-Beispiels verdeutlichen, indem wir in Abbildung 3.15 zuerst ein Sequenzdiagramm zeigen und anschließend in Abbildung 3.16 ein entsprechendes Kollaborationsdiagramm.

Im Sequenzdiagramm werden die Objekte durch gestrichelte senkrechte Linien notiert, an denen oben über der Linie der Name bzw. das Objektsymbol steht. Die Nachrichten werden als waagrechte Pfeile zwischen den Objekt-Linien gezeichnet. Die Antwort auf eine Nachricht kann entweder in Textform (`antwort := nachricht()`) notiert werden oder als eige-

ner, dann aber gestrichelter Pfeil mit offener Pfeilspitze. Wie im Zustandsdiagramm können die Nachrichten zusätzlich mit Bedingungen in der Schreibweise `[Bedingung] nachricht()` versehen werden. Die grauen senkrechten Balken geben an, welches Objekt gerade die Programmkontrolle besitzt, d.h. welches Objekt gerade aktiv ist. Am Rand des Diagramms können Beschreibung und Kommentierung des Ablaufs erfolgen.

In Abbildung 3.15 haben wir mehrere Szenarien zusammengefasst: Das Anlegen von Wartezimmer und Personen, das Eintragen von Personen in die Queue (`Put()`) und das Abholen der Personen aus der Queue (`Get()`) mit anschließendem Fragen nach dem Anliegen. Die Klasse `CStr` haben wir hier mit aufgenommen, um zu zeigen, wann diese mit einer Nachricht angesprochen wird.

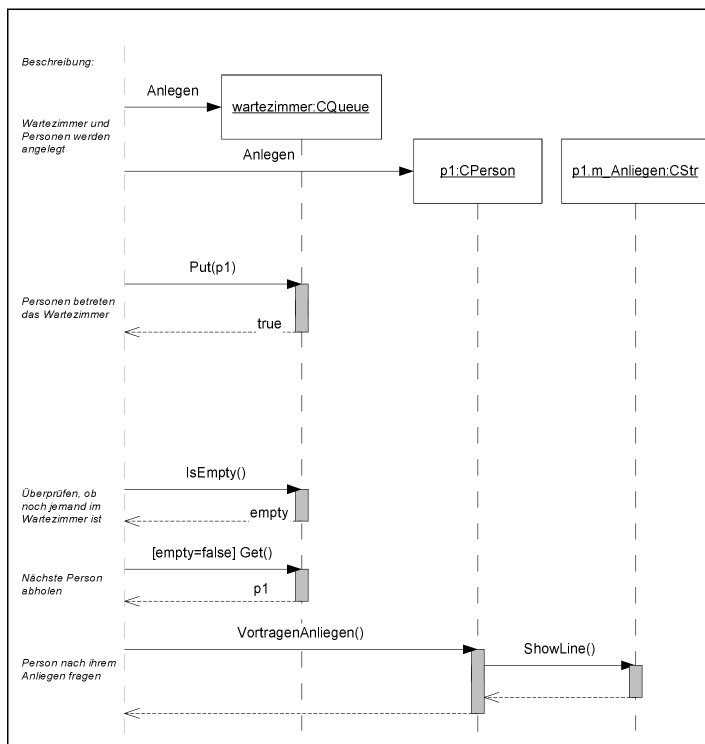

Abbildung 3.15: Sequenzdiagramm zum Personen-Queue-Beispiel

Das letzte Szenario (Abholen einer Person aus dem Wartezimmer) wird in Abbildung 3.16 nun in Form eines Kollaborationsdiagrammes gezeigt. Die dargestellten Sachverhalte sind dabei identisch.

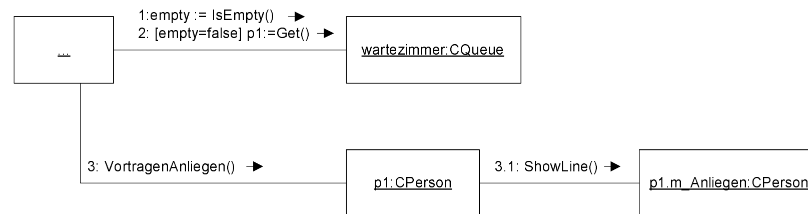

Abbildung 3.16: Kollaborationsdiagramm zum Personen-Queue-Beispiel

Im Kollaborationsdiagramm werden die Objekte mit Assoziationslinien (Beziehungslinien) verbunden, auf denen die Nachrichten notiert werden. Ein kleiner Pfeil zeigt jeweils die Richtung der Nachricht vom Sender zum Empfänger. Mögliche Antworten können in Textform wie beim Sequenzdiagramm in der Form `antwort:=nachricht` angegeben werden. Die zeitliche Abfolge der Nachrichten wird durch Sequenznummern verdeutlicht. Die erste Nachricht beginnt mit der Nummer 1. Weitere Nachrichten werden aufsteigend nummeriert. Werden innerhalb einer Operation wiederum neue Nachrichten abgesendet, so erhalten diese eine Untersequenz-Nummer in der Form `NrOperation.NrNeueNachricht`, z.B. Während der Operation `3` verschickt diese zwei weitere Nachrichten: `3.1` und `3.2`. Wird in `3.2` wiederum eine Methode aufgerufen, so erhält diese Nachricht dann die Nummer `3.2.1`.

An diesem Kollaborationsdiagramm wird sofort deutlich, dass das Wartezimmer nicht direkt mit der Person kommuniziert. Wir haben die Aufgabe der Queue ja auch so gewählt, dass sie Personen lediglich verwaltet.

Allerdings muss dabei beachtet werden, dass ein Kollaborationsdiagramm – wie auch ein Sequenzdiagramm – immer nur einen speziellen Ablauf darstellt (hier, dass jemand eine Person aus dem Wartezimmer abholt und diese nach ihrem Anliegen fragt). Das Kollaborationsdiagramm aus Abbildung 3.16 wäre damit noch kein Beweis dafür, dass ein Objekt der Klasse `CQueue` nie mit einem Objekt der Klasse `CPerson` kommuniziert, auch wenn dies der Fall ist.

Der Fokus von Kollaborations- und Sequenzdiagrammen liegt also auf der Darstellung einzelner Ablaufvarianten. Sie sind jedoch weniger dazu geeignet, ein Verhalten präzise oder vollständig zu beschreiben. Eine bessere Wahl sind dazu Zustandsdiagramme (vgl. Kapitel 3.2.7) oder Aktivitätsdiagramme, die Ablaufmöglichkeiten (eines Systems) beschreiben. Letztere finden jedoch mehr in der Analysephase Anwendung (vgl. Kapitel 7.2, Weitere UML-Konstrukte (Schwerpunkt Analyse)).

Der Haupteinsatz von Kollaborations- und Sequenzdiagrammen liegt darin, sich einzelner spezieller Ablaufsituationen bewusst zu werden, sie zu erklären oder zu dokumentieren. Sie lassen sich am Flip-Chart oder auf einer Tafel schnell skizzieren und diskutieren.

Wie wir gesehen haben, lässt sich an diesen Diagrammen auch sehr schön erkennen, wie intensiv die Kommunikation zwischen den Objekten für den betrachteten Fall ist.

Nachdem wir uns nun Gedanken über den Aufbau unserer Klassen und den Beziehungen unter den Objekten gemacht haben, müssen wir nun dieses Design noch in C++-Code umsetzen.

3.3.3 Implementierung in C++

Es hat sich eingebürgert, pro Klasse ein eigenes Modul zu benutzen; besonders bei integrierten Entwicklungsumgebungen ist dies häufig der Fall.

Listing 3.7 (`Queue.h`) und Listing 3.8 (`Queue.cpp`) zeigen die Deklaration und die Implementierung der Klasse `CQueue`.

```
// Datei: Queue.h

#ifndef QUEUE_H
#define QUEUE_H

#include "Person.h"

//=====================================================
// Aufgabe von CQueue:
// Verwaltet Personen (CPerson) nach dem FIFO-Prinzip
```

```
//----------------------------------------------------
class CQueue
{
   CPerson* m_entries; // Einträge

   int m_max,          // Max. mögliche Anzahl Elemente
      m_cnt,           // Aktuelle Anzahl Elemente
      m_first,         // Index erster Eintrag
      m_last;          // Index letzter Eintrag
public:
   bool Get(CPerson& person);
   bool Put(CPerson person);

   bool IsEmpty() {return m_cnt==0;}      // Queue leer?
   bool IsFull() {return m_cnt==m_max;}   // Queue voll
   CQueue(int max);
   ~CQueue();
};
#endif // v. ifndef (QUEUE_H)
```

Listing 3.7: (Queue.h): Deklaration der Klasse CQueue

```
// Datei: Queue.cpp

#include "Queue.h"
#include "Person.h"
#include <stdlib.h>

//-----------------------------------------------------
CQueue::CQueue(int max)
{
   m_last=m_first=m_max=m_cnt=0;

   m_entries= new CPerson[max+1];
   if(m_entries)
      m_max=max;
}
//-----------------------------------------------------
CQueue::~CQueue()
{
   if(m_entries)
      delete[] m_entries;
}
```

```
//----------------------------------------------------
bool CQueue::Put(CPerson person)
{
   if(!m_entries || IsFull())
      return false;

   m_entries[m_last]=person;

   if(m_last < m_max)
      m_last++;
   else if(m_last == m_max)
      m_last=0;

   m_cnt++;
   return true;
}
//----------------------------------------------------
bool CQueue::Get(CPerson& person)
{
   if(IsEmpty() || !m_entries)
      return false;

   person=m_entries[m_first];

   if(m_first < m_max)
      m_first++;
   else if(m_first == m_max)
      m_first=0;

   m_cnt--;
   return true;
}
```

Listing 3.8: (Queue.cpp): Implementierung der Klasse CQueue

Listing 3.9 (`Person.h`) und Listing 3.10 (`Person.cpp`) zeigen die Deklaration und die Implementierung der Klasse `CPerson`.

```
// Datei: Person.h

#ifndef PERSON_H
#define PERSON_H

#include <stdlib.h>
#include "Str.h"

//====================================================
```

```
// Aufgabe von CPerson:
// Repräsentation einer Person, die ein Anliegen
// vortragen kann
//-------------------------------------------------------

class CPerson
{
   CStr m_Anliegen;

public:
   void VortragenAnliegen();

   CPerson();
   CPerson(CStr Anliegen);
};

#endif // v. ifndef (PERSON_H)
```

Listing 3.9: (Person.h): Deklaration für die Klasse CPerson

```
// Datei: Person.cpp

#include "Person.h"
#include <stdlib.h>
//------------------------------------------------
CPerson::CPerson()
{
   m_Anliegen="Wollte nur mal guten Tag sagen";
}
//------------------------------------------------
CPerson::CPerson(CStr Anliegen)
{
   m_Anliegen=Anliegen;
}
//------------------------------------------------
void CPerson::VortragenAnliegen()
{
   m_Anliegen.ShowLine();
}
```

Listing 3.10: (Person.cpp): Implementierung der Klasse CPerson

Listing 3.11 (`Str.h`) und Listing 3.12 (`Str.cpp`) zeigen die Deklaration und die Implementierung der Klasse `CStr`. Diese Klasse wurde im Kapitel 3.2, Verwendung einer Klasse, schrittweise entwickelt. Hier wird die

Klasse einmal komplett mit den Methoden gezeigt, die für dieses Beispiel nötig sind.

```
// Datei: Str.h

#ifndef CSTR_H
#define CSTR_H
//=====================================================
// Aufgabe von CStr:
// Verwaltung eines Strings
//-----------------------------------------------------
class CStr
{
   char* m_str;
public:
   CStr();                       // Standardkonstruktor
   CStr(char* str);              // Konstruktor
   ~CStr();                      // Destruktor
   CStr(const CStr& str);        // Kopierkonstruktor
   CStr& operator= (const CStr& str); // Zuweisungsop.

   void ShowLine();
   // ...
};
//====================================================
#endif // v. ifndef(CSTR_H)
```

Listing 3.11: (Str.h): Deklaration für die Klasse CStr

```
// Datei: Str.cpp

#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include "str.h"

//====================================================
CStr::CStr()            // Standardkonstruktor
{
   m_str=NULL;
}
//----------------------------------------------------
CStr::CStr(char* str)      // Konstruktor
{
   m_str=new char[strlen(str)+1];
```

```
   if(m_str)
      strcpy(m_str,str);
}
//---------------------------------------------------
CStr::~CStr()            // Destruktor
{
   if(m_str) // Speicherplatz für m_str wurde besorgt
      delete[] m_str;
}
//---------------------------------------------------

CStr::CStr(const CStr& str)   // Kopierkonstruktor
{
   // Membervariablen initialisieren
   // (ist ja Aufgabe eines Konstruktors)
   m_str=NULL;

   // Aufruf des Zuweisungsoperators:
   *this=str;
}
//----------------------------------------------------
CStr& CStr::operator=(const CStr& str) // Zuweisungsop.
{
   // bisher in Anspruch genommenen Speicher freigeben
   if(m_str)
      delete[] m_str;

   if(str.m_str)      // str ist nicht leer
   {
      // Neuen Speicher gemäß str besorgen und
      // str kopieren
      m_str=new char[strlen(str.m_str)+1];

       if(m_str)         // Allokation OK
          strcpy(m_str,str.m_str);   // String kopieren
       else
       {
          cerr << "Kein Speicher!!!";
          exit(1);         // Programmende
       }
   }
   else            // str ist leeres Stringobjekt
      m_str=NULL;
```

```
   return (*this); // Ergebnis entspricht Wert des
                    // linken Operanden, der durch das
                   // aktuelle Objekt repräsentiert wird.
}
//-----------------------------------------------------
void CStr::ShowLine()
{
   cout << m_str << endl;
}
```

Listing 3.12: (Str.cpp): Implementierung der Klasse CStr

Programm 3.1 (`main.cpp`) zeigt die Realisierung der `main()`-Funktion.

```
#include "Queue.h"
#include "Person.h"
#include "Str.h"
#include <iostream.h>

int main(void)
{
   CQueue wartezimmer(6);   // Kleines Wartezimmer mit
                            // nur 6 Stühlen
   CPerson patient;         // aktueller Patient

   CPerson p1("Ich bin immer so muede."),
         p2("Hust,hust, schnief...Ich glaube,
            ich bin erkaeltet."),
         p3("Ueberall tut es mir weh"),
         p4("Ich bin nervlich voellig am Ende!"),
         p5;

   // Die Patienten betreten das Wartezimmer...
   wartezimmer.Put(p2);
   wartezimmer.Put(p1);
   wartezimmer.Put(p5);
   wartezimmer.Put(p3);
   wartezimmer.Put(p4);

   // Der Arzt ruft einen nach dem anderen auf...
   int cnt=0;
   while(!wartezimmer.IsEmpty())
   {
      // "Der Nächste bitte..."
      wartezimmer.Get(patient);
```

```
      cnt++;         // fuer die Abrechnung ;-)

      cout << "\n" << cnt <<".ter Patient des Tages: "
           << endl;
      cout << "Arzt: Was kann ich fuer Sie tun?"
           << endl;
      cout << "Patient: ";
      patient.VortragenAnliegen();

      // Arzt behandelt den Patienten und Patient
      // verabschiedet sich

      // Während der Arzt beschäftigt ist,
      // kommen noch weitere Patienten
      if(cnt%2==0)   // jetzt ist wieder Platz für
                     // 2 weitere Patienten
      {
         cout << "\n-->Neuer Patient kommt..." << endl;
         wartezimmer.Put(CPerson("Mich hat es glaub ich
                         auch erwischt mit Schnupfen"));
      }
   }
   return 0;
}
```

Programm 3.1 (main.cpp): Realisierung der main()-Funktion

Dieses Programm würde folgende Ausgabe erzielen:

```
1.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Hust,hust, schnief...Ich glaube, ich bin erkaeltet.

2.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Ich bin immer so muede.

-->Neuer Patient kommt...

3.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Wollte nur mal guten Tag sagen
```

```
4.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Ueberall tut es mir weh

-->Neuer Patient kommt...

5.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Ich bin nervlich voellig am Ende!

6.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Mich hat es glaub ich auch erwischt mit Schnupfen

-->Neuer Patient kommt...

7.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Mich hat es glaub ich auch erwischt mit Schnupfen

8.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Mich hat es glaub ich auch erwischt mit Schnupfen

-->Neuer Patient kommt...

9.ter Patient des Tages:
Arzt: Was kann ich fuer Sie tun?
Patient: Mich hat es glaub ich auch erwischt mit Schnupfen
```

Obwohl das Wartezimmer nur 6 Stühle hat, können 9 Patienten behandelt werden, weil ja zwischendurch auch immer wieder Patienten gehen.

Der Einsatz einer Queue macht natürlich erst dann Sinn, wenn das Eintragen und Abholen der Personen aus der Queue zeitlich entkoppelt ist. Das gewählte `main()` dient hier also lediglich als ganz simpler Test der Klassen `CQueue` und `CPerson`.

Im gewählten Beispiel haben wir die Aufgabe der Queue so gewählt, dass sie die Personen lediglich verwaltet, sie jedoch nicht direkt anspricht. Es wäre auch denkbar, die Queue intelligenter zu machen und z.B. eine Methode `Show_All()` zur Verfügung zu stellen, die alle

Elemente der Queue anzeigt. Die Queue würde die Nachricht `Show_All()` dann so umsetzen, dass sie das Anzeigen jeweils an die enthaltenen Elemente delegiert, indem sie für alle Personen `VortragenAnliegen( )` aufruft. Dies hätte jedoch den großen Nachteil, dass die Beziehung zwischen Queue und Personen enger ist. Die Queue muss damit das Verhalten einer Person kennen und wissen, dass die Methode zum Anzeigen `VortragenAnliegen()` heißt. In unserem Fall braucht die Queue nichts über das Verhalten der Person zu wissen, da sie nicht direkt mit der Person kommuniziert. Die Queue ist damit allgemein gültiger. Die einzigste Abhängigkeit, die in unserem Fall besteht, ist, dass unsere Queue speziell Personen verwaltet und nicht jeden beliebigen Typ verwalten kann. Wir werden aber in Kapitel 5.4, Templates, noch einen Mechanismus von C++ kennen lernen, um Klassen datentypunabhängig zu gestalten. Damit lässt sich dann eine Queue realisieren, die jeden x-beliebigen Typ verwalten kann.

Es kommt also stets darauf an, wie die Klasse eingesetzt werden soll, was also die Aufgabe der Klasse ist. Dementsprechend bietet es sich an, die Klasse speziell für die vorliegende Aufgabe intelligenter zu gestalten oder sie weniger speziell allgemein gültig zu entwerfen und damit offener für einen zukünftigen Einsatz zu sein.

Zwischen den Diagrammen, die während der Designphase erstellt werden, und der Implementierung besteht folgender Zusammenhang:

- Die Kästchen aus dem Klassendiagramm resultieren in der Klassenspezifikation (Schlüsselwort class)
- Die Botschaften / Nachrichten im Sequenzdiagramm resultieren in Methodenaufrufen.
  Die angesprochene Klasse muss also über eine entsprechende Methode verfügen.

Diese Tatsachen berücksichtigen auch CASE-Tools bei der automatischen Codegenerierung aus UML-Diagrammen.

3.3.4 ÜBUNG: Gemeinsamer Termin

In Gemeinschaften (Vereinen, Abteilungen, ...) gibt es oft das Problem, dass ein gemeinsamer Termin gefunden werden muss (z.B. für die gemeinsame Weihnachtsfeier). Üblicherweise wird dazu eine Tabelle

erstellt, in der alle Kanditaten eintragen können, welcher Termin für sie möglich wäre und welcher nicht. Der Termin, an dem die meisten Personen Zeit haben, wird dann genommen.

Erstellen Sie eine Klasse `CTerminFinder`, die diese Aufgabe übernimmt. Diese Klasse soll über folgende Methoden verfügen:

- `Streichen(TerminNr);`
- `Freigeben(TerminNr);`
- `GetBestTermine();`

Der Einfachheit halber wird nicht gespeichert, welche Person ihre Stimme bereits abgegeben hat, sondern mit jedem `Freigeben()` wird ein Zähler für diesen Termin hochgezählt und für jedes `Streichen()` der Zähler für den Termin dekrementiert. Die Anzahl der Termine, die zur Auswahl stehen, wird beim Konstruktor angegeben.

Berücksichtigen Sie, dass es auch mehrere Termine geben kann, an denen die meisten Personen Zeit haben. Die Methode `GetBestTermine()` muss daher ein Array aus DateNummern zurückliefern statt nur eines Wertes. Hier bietet sich unsere Klasse `CIntArray` der Übung 3.2.15 förmlich an.

Implementieren Sie eine geeignete `main()`-Funktion, um die Klasse zu testen, z.B. könnten Sie folgende Situation fest codieren:

- Es wird jeweils ein Wochentag (5 mögliche Tage: MO-FR) gesucht, an dem man sich zum Kegeln treffen kann, und ein Tag zum Doom-Spielen.
- Hugo ist Kegler und Doomer und kann Montags und Mittwochs nie.
- Fritz ist nur Kegler und kann am Dienstag nicht.
- Anna spielt nur Doom und kann am Dienstag und Mittwoch nicht.
- Nachdem Anna aus einem Verein ausgeschieden ist, kann sie nun doch dienstags.

Geben Sie die jeweils besten Tage zum Kegeln und zum Doom-Spielen aus. Das Programm sollte dann in etwa Folgendes ausgeben:

```
Beste Termine fuer Kegeln : DO FR
Beste Termine fuer Doom : DI DO FR
```

(Da die beiden TerminFinder nicht synchronisiert sind, hätte Hugo nun ein Problem, wenn beide Termine auf Donnerstagabend festgelegt werden ... Aber so ist das eben im Leben ;-))

Es kommt immer darauf an, wie die Aufgabe einer Klasse definiert ist. Natürlich hätte hier auch die Ausgabe der besten Termine direkt in `CTerminFinder` erfolgen können, etwa in einer Methode `PrintBestTermine()`. Dann wäre der Einsatz der zweiten Klasse `CIntArray` überflüssig. Der Nachteil ist dann aber, dass die Klasse nicht so flexibel eingesetzt werden kann, etwa in GUI-Applikationen. In unserem Fall dagegen kümmert sich die Klasse lediglich um die Verwaltung der Terminliste und nimmt die Auswertung vor, ohne sie jedoch optisch aufzubereiten.

3.3.5 ÜBUNG: Vieleck

Realisieren Sie eine Klasse zur Verwaltung und Darstellung eines Vielecks. Ein Vieleck ist eine graphische Figur, bei der die einzelnen Ecken über Punkte angegeben sind und über Linien verbunden werden. Im Gegensatz zu einem Polygonzug (vgl. folgende Übung) ist bei einem Vieleck von Anfang an klar, wie viele Ecken die Figur besitzen kann. Außerdem ist ein Vieleck immer eine geschlossene Figur, während es auch offene Polygonzüge gibt.

Die Vieleck-Klasse kann daher beispielsweise wie folgt eingesetzt werden:

```
#include "Vieleck.h"
#include "Punkt.h"

int main(void)
{
   CVieleck dreieck1(3), dreieck2(3), viereck1(4),
            myFuenfeck(5);

   //...
   dreieck1.Set(0, CPunkt(2,3));   // 1. Punkt des Dreiecks
   dreieck1.Set(1, CPunkt(4,5));   // 2. Punkt des Dreiecks
```

```
   dreieck1.Set(2, CPunkt(2,8));   // 3. Punkt des Dreiecks

   //...
   // Der Anwender setzt 2.Punkt von dreieck1 um
   dreieck1.Set(1, CPunkt(4,6));

   dreieck1.Zeichnen();

   return 0;
}
```

Das Zeichnen erfolgt hier natürlich rein textuell. Das Programm könnte daher beispielsweise Folgendes ausgeben:

```
Verbinde folgende Punkte: P(2,3)  P(4,6)  P(2,8)
```

3.3.6 ÜBUNG: Polygonzug

Realisieren Sie eine Klasse zur Verwaltung und Darstellung eines Polygonzuges. Die einzelnen Punkte des Polygonzuges werden zur Laufzeit eingegeben – in einer graphischen Oberfläche etwa durch Klick auf die entsprechende Position.

Die Polygon-Klasse könnte daher beispielsweise wie folgt eingesetzt werden:

```
#include "Polygon.h"
#include "Punkt.h"

int main(void)
{
   CPolygon poly;   // Polygon anlegen; hier ist noch
                    // nicht festgelegt, wie groß es wird

   // ...
   // mit jedem hinzukommenden Punkt wächst der Polygonzug
   poly.Einfuegen( CPunkt(0,4));
   poly.Einfuegen( CPunkt(3,7));
   poly.Einfuegen( CPunkt(5,5));
   poly.Einfuegen( CPunkt(3,4));
   poly.Einfuegen( CPunkt(5,2));
   poly.Einfuegen( CPunkt(3,0));
   //...
```

```
   // Polygonzug zeichnen
   poly.Zeichnen();

   return 0;   // hier wird der Destruktor von poly
               // durchlaufen und rekursiv der gesamte
               // Polygonzug mit allen gespeicherten Punkten
               // gelöscht
}
```

Wie beim Vieleck erfolgt das Zeichnen natürlich hier auch wieder rein textuell. Damit könnte das Programm beispielsweise Folgendes ausgeben:

```
P(0,4)
P(3,7)
P(5,5)
P(3,4)
P(5,2)
P(3,0)
```

3.3.7 ÜBUNG: Objektkopien

Programmieren Sie das Personen-Queue-Beispiel ohne Verwendung der Klasse `CStr`, indem Sie das Attribut `CPerson::m_Anliegen` als char* implementieren. Merken Sie, wie dadurch die Komplexität steigt?

3.4 Beziehungen zwischen Objekten

Der Ablauf in einem objektorientierten Programm entsteht dadurch, dass Objekte miteinander kommunizieren. Kommunikation zwischen Objekten bedeutet, dass die Objekte sich gegenseitig Nachrichten schicken, d.h. Methoden des anderen Objektes aufrufen, um entsprechende Aktionen anzustoßen. Voraussetzung für eine Kommunikation ist, dass sich die Objekte untereinander kennen.

Welche Beziehungen zwischen Objekten denkbar sind, wird im Folgenden erläutert.