Objektorientierte Programmierung mit C++

(Entwurf)

WS 07

Sven Eric Panitz

FH Wiesbaden

Version 20. Juni 2007

Die vorliegende Skriptversion baut auf das Skript zum Modul Programmieren 1 im ersten Semester des WS06/07 auf. Das Skript entsteht im Verlauf der Vorlesung und wächst mit dem zu behandelnden Stoff. Es basiert in Teilen auf das Skript, das zur entsprechenden Vorlesung im Diplomstudiengang entstanden ist

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Das übliche Programm: hello world

Wie in den meisten Programmierkursen, wollen wir uns auch nicht um das übliche *hello world* Programm drücken und geben hier die C++-Version davon an. Tatsächlich unterscheidet es sich schon fundamental vom emtsprechenden Programm in C:

HelloWorld.cpp

```
1 #include <iostream>
2
3 int main(){
4   std::cout << "hello" << " world " << 42 << std::endl;
5   return 0;
6 }
```

Wir sehen, dass zunächst einmal die Bibliothek für Ein- und Ausgabeströme zu inkludieren ist: `#include <iostream>`. Wir benutzen nicht mehr die Standardbibliothek `stdio.h` aus C. Zusätzlich wird in C++ nicht mehr die Dateiendung `.h` beim Inkludieren einer Kopfdatei einer im System installierten Bibliothek angegeben.

Ein- und Ausgabe wird in C++ über Operatoren ausgedrückt. Zur Ausgabe gibt es den Operator `<<`. `std::cout` ist der Ausgabestrom für die Konsole. Der Operator `<<` hat als linken Operanden einen Ausgabestrom und als rechten Operanden ein Argument, das auf dem Strom ausgegeben werden soll.

Der Operator `<<` liefert als Ergebnis einen Ausgabestrom, so dass elegant eine Folge von Ausgaben durch eine Kette von `<<` Anwendungen ausgedrückt werden kann. Auf der rechten Seite des Operators `<<` können recht unterschiedliche Operanden stehen. Zeichenketten, aber auch Zahlen. Der Operator funktioniert tatsächlich für unterschiedliche Argumenttypen.

`std::endl` steht für das Zeilenende. Zusätzlich bewirkt es, dass ein *flush* auf einen Zwischenpuffer durchgeführt wird.

An diesem Programm lassen sich bereits ein paar Eigenschaften von C++ erkennen. Offensichtlich sind Operatoren überladbar, denn der Operator `<<` ist in einer Bibliothek definiert. Desweiteren scheint es eine Form qualifizierter Namen zu geben. `std::cout` hat ein Präfix `std`, der mit dem Operator `::` vom eigentlichen Namen `cout` einer Variablen getrennt wird.

Die Quelltextdatei haben wir in eine Datei mit der Endung `.cpp` gespeichert. Dieses ist nicht einheitlich. Es ist ebenso üblich C++-Dateien mit der Endung `.cc` oder gar `c++` zu markieren.

Mit dem `g++`-Compiler läßt sich das Programm in gewohnter Weise übersetzen und dann schließlich ausführen:

```
sep@pc216-5:~/fh/cpp/student/src> g++ -o helloworld HelloWorld.cpp
sep@pc216-5:~/fh/cpp/student/src> ls -l helloworld
-rwxr-xr-x  1 sep users 10781 2005-03-11 09:46 helloworld
sep@pc216-5:~/fh/cpp/student/src> ./helloworld
hello world
sep@pc216-5:~/fh/cpp/student/src> strip helloworld
sep@pc216-5:~/fh/cpp/student/src> ls -l helloworld
-rwxr-xr-x  1 sep users 4336 2005-03-11 09:46 helloworld
sep@pc216-5:~/fh/cpp/student/src> ./helloworld
hello world 42
sep@pc216-5:~/fh/cpp/student/src>
```

Mit dem Programm `strip` können aus ausführbaren und Objektdateien Symboltabellen entfernt werden, die zur eigentlichen Ausführung nicht mehr benötigt werden. Das Programm wird damit nocheinmal wesentlich kleiner, wie am obigen Beispiel zu sehen ist. Das Programm `strip` entfernt insbesondere dabei auch alle Information, die zum Debuggen notwendig sind.

Kapitel 2

Objektorientierte Programmierung

2.1 Grundkonzepte der Objektorientierung

2.1.1 Objekte und Klassen

Die Grundidee der objektorientierten Programmierung ist, Daten, die zusammen ein größeres zusammenhängendes Objekt beschreiben, zusammenzufassen. Zusätzlich fassen wir mit diesen Daten noch die Programmteile zusammen, die diese Daten manipulieren. Ein Objekt enthält also nicht nur die reinen Daten, die es repräsentiert, sondern auch Programmteile, die Operationen auf diesen Daten durchführen. Insofern wäre vielleicht *subjektorientierte Programmierung* ein passenderer Ausdruck, denn die Objekte sind nicht passive Daten, die von außen manipuliert werden, sondern enthalten selbst als integralen Bestandteil Methoden, die ihre Daten manipulieren können.

Objektorientierte Modellierung

Bevor wir etwas in Code gießen, wollen wir ersteinmal eine informelle Modellierung der Welt, für die ein Programm geschrieben werden soll, vornehmen. Hierzu empfiehlt es sich durchaus, in einem Team zusammenzusitzen und auf Karteikarten aufzuschreiben, was es denn für Objekte in der Welt gibt, die wir modellieren wollen.

Stellen wir uns hierzu einmal vor, wir sollen ein Programm zur Bibliotheksverwaltung schreiben. Jetzt überlegen wir einmal, was gibt es denn für Objektarten, die alle zu den Vorgängen in einer Bibliothek gehören. Hierzu fällt uns vielleicht folgende Liste ein:

- Personen, die Bücher ausleihen wollen.
- Bücher, die ausgeliehen werden können.
- Tatsächliche Ausleihvorgänge, die ausdrücken, dass ein Buch bis zu einem bestimmten Zeitraum von jemanden ausgeliehen wurde.
- Termine, also Objekte, die ein bestimmtes Datum kennzeichnen.

Nachdem wir uns auf diese vier für unsere Anwendung wichtigen Objektarten geeinigt haben, nehmen wir vier Karteikarten und schreiben jeweils eine der Objektarten als Überschrift auf diese Karteikarten.

Jetzt haben wir also Objektarten identifiziert. Im nächsten Schritt ist zu überlegen, was für Eigenschaften diese Objekte haben. Beginnen wir für die Karteikarte, auf der wir als Überschrift *Person* geschrieben haben. Was interessiert uns an Eigenschaften einer Person? Wahrscheinlich ihr Name mit Vornamen, Straße und Ort sowie Postleitzahl. Das sollten die Eigenschaften einer Person sein, die für ein Bibliotheksprogramm notwendig sind. Andere mögliche Eigenschaften wie Geschlecht, Alter, Beruf oder ähnliches interessieren uns in diesem Kontext nicht. Jetzt schreiben wir die Eigenschaften, die uns von einer Person interessieren, auf die Karteikarte mit der Überschrift Person.

Schließlich müssen wir uns Gedanken darüber machen, was diese Eigenschaften eigentlich für Daten sind. Name, Vorname, Straße und Wohnort sind sicherlich als Texte abzuspeichern oder, wie der Informatiker gerne sagt, als Zeichenketten. Wir haben in C bereits viel mit Zeichenketten gearbeitet, indem wir sie als Reihungen von Buchstaben betrachtet haben, wobei eine Reihung wieder nicht mehr als ein Zeiger auf das erste Element war. Somit hatten wir den Typ `char*` für Zeichenketten benutzt. In C++ gibt es einen Standardtypen, der für Zeichenketten benutzt werden kann, der Typ `std::string`.

Die Postleitzahl ist hingegen als eine Zahl abzuspeichern. Diese Art, von der die einzelnen Eigenschaften sind, nennen wir ihren Typ. Wir schreiben auf die Karteikarte für die Objektart *Person* vor jede der Eigenschaften noch den Typ, den diese Eigenschaft hat.[1] Damit erhalten wir für die Objektart *Person* die in Abbildung 2.1 gezeigte Karteikarte.

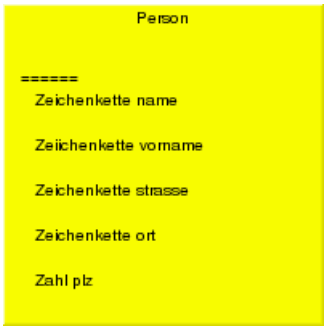

Abbildung 2.1: Modellierung einer Person.

Gleiches können wir für die Objektart *Buch* und für die Objektart *Datum* machen. Wir erhalten dann eventuell die Karteikarten aus Abbildung 2.2 und 2.3 .

Wir müssen uns schließlich nur noch um die Objektart einer Buchausleihe kümmern. Hier sind drei Eigenschaften interessant: wer hat das Buch geliehen, welches Buch wurde verliehen und wann muss es zurückgegeben werden? Wir können also drei Eigenschaften auf die Karteikarte schreiben. Was sind die Typen dieser drei Eigenschaften? Diesmal sind es keine Zahlen oder Zeichenketten, sondern Objekte der anderen drei bereits modellierten Objektarten. Wenn wir nämlich eine Karteikarte schreiben, dann erfinden wir gerade einen neuen Typ, den wir für die Eigenschaften anderer Karteikarten benutzen können.

[1] Es mag vielleicht verwundern, warum wir den Typ vor die Eigenschaft und nicht etwa hinter sie schreiben. Dieses ist eine sehr alte Tradition in der Informatik und kennen wir ja auch bereits aus C.

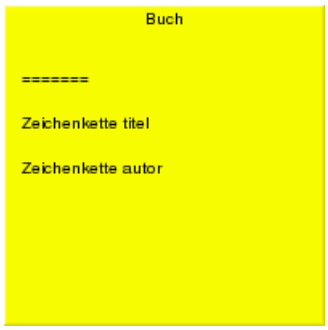

Abbildung 2.2: Modellierung eines Buches.

Abbildung 2.3: Modellierung eines Datums.

Somit erstellen wir eine Karteikarte für den Objekttyp *Ausleihe*, wie sie in Abbildung 2.4 zu sehen ist.

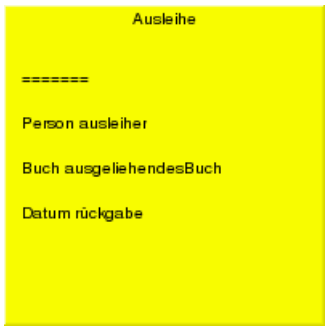

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

Umsetzung als Datenstrukturen

Die oben entworfenden Daten sollen jetzt in eine Datenstruktur umgesetzt werden. Bevor wir dieses in der Programmiersprache C++ tun, sei es in der Sprache C, die wir schon einiegermaßen gut kennen, demonstriert.

Umsetzung als Strukturen in C Um die Daten, wie wir sie in unseren Beispielentwurf modelliert haben, in C darzustellen, gibt es bereits ein recht praktisches und naheliegendes Konstrukt in C in Form der Strukturen. Strukturen bündeln Daten, die zusammen ein inhaltliches Ganzes ergeben als eigene Typen. In der Regel haben wir schon gesehen, ist es innvoll solche strukturierten Daten immer über einen Zeiger anzusprechen, so dass nicht Kopien von ihnen bei der Parameterübergabe oder Zuweisung vorgenommen werden.

Wir erhalten folgende Definitionen in C.

Bibliothek.h

```
1  typedef struct {
2    char* name;
3    char* vorname;
4    char* strasse;
5    char* ort;
6    unsigned int plz;
7  } Person;
8
9  typedef struct {
10   char* titel;
11   char* autor;
12 } Buch;
13
14 typedef struct {
15   short tag;
16   short monat;
17   unsigned int jahr;
18 } Datum;
19
20 typedef struct {
21   Person* ausleiher;
22   Buch* ausgeliehenesBuch;
23   Datum* rueckgabe;
24 } Ausleihe;
25
```

Umsetzung als Klassen in C++ Die Aufgabe der Strukturen aus C sind in C++ in die Klassen übergegangen. Eine Klasse in C++ kann ebenso wie eine Struktur in C definiert werden. Im Prinzip ist lediglich das Schlüsselwort `struct` durch das Wort `class` zu ersetzen. So ergibt sich zunächst einmal beim Übergang zu C++ nicht wirklich viel Neues:

Bibliothek.hpp

```
1  #include <string>
2
3  class Person {
4   public:
5    std::string name;
6    std::string vorname;
7    std::string strasse;
8    std::string ort;
9    unsigned int plz;
```

```
10 };
11
12 class Buch{
13  public:
14   std::string titel;
15   std::string autor;
16 };
17
18 class Datum{
19  public:
20   short tag;
21   short monat;
22   unsigned int jahr;
23 } ;
24
25 class Ausleihe {
26  public:
27   Person* ausleiher;
28   Buch* ausgeliehenesBuch;
29   Datum* rueckgabe;
30 };
```

Neu ist, dass wir nicht mehr `char*` als Typ zur Darstellung von Zeichenketten benutzen, sondern den Standardtyp `std::string` aus C++. Außerdem ist, wenn wir die eigentlichen Felder der Klasse auch außerhalb der Klasse benutzen wollne, diesem das Attribut `publbic:` voranzustellen.

Die Kopfdatei ist in diesem Fall in einer Datei mit der Endung `.hpp` gespeichert. Das ist nicht zwingend notwendig, vielfach wird auch in C++ nur die Endung `.h` verwendet.

Erzeugen von Objekten

Wir haben jetzt in C respektive C++ definiert, welcher Art die Daten sind, die Objekte für unsere Anwendung bilden. Als erstes, um mit solchen Objekten zu arbeiten, sind die Objekte im Speicher zu erzeugen. Hierzu sind spezielle Funktionen vorzusehen, die sogenannten Konstruktoren. Ein Konstruktor wird dazu benutzt, um aus Einzeldaten ein Objekt zu erzeugen, das zu einer bestimmten Klasse von Objekten gehört.

Konstruktorfunktionen in C In C können wir Funktionen schreiben, die Objekte erzeugen. Um namentlich deutlich zu machen, dass diese Funktionen ein neues Objekt konstruieren, geben wir ihnen einen Namen, der mit `new` beginnt und mit dem Namen der zu Struktur, für die das Objekt erzeugt wird, endet. Für die Struktur `Person` schreiben wir also eine Konstruktorfunktion `newPerson`. Sie soll als Ergebnis einen Zeiger auf ein neues Personenobjekt haben. Als Parameter soll die Konstruktorfunktion die Daten für die einzelnen Bestandteile der Struktur haben.

Wir erhalten also folgende Signatur, die wir in einer Kopfdatei definieren:

BibliothekKonstruktorFunktionen.h

```
#include "Bibliothek.h"
Person* newPerson
 (char* name,char* vorname,char* strasse,char* ort,unsigned int plz);
```

Das Erzeugen eines neuen Objektes werden wir in zwei Schritten vornehmen. Im ersten Schritt wird Platz im Speicher für das neuen Objekt allokiert, um im zweiten Schritt die Daten in diesem neu allokierten Speicherbereich zu initialisieren. Zum Erzeugen eines neuen Objektes sei die Funktion `createPerson`, zum Initialisieren die Funktion `initPerson` definiert:

BibliothekKonstruktorFunktionen.h

```
Person* createPerson();
void initPerson
    (Person* this,char* name,char* vorname,char* strasse
    ,char* ort,unsigned int plz);
```

In gleicher Weise seien für die drei übrigen Strukturen die Konstruktorfunktionen definiert:

BibliothekKonstruktorFunktionen.h

```
Buch* newBuch(char* titel,char* autor);
Buch* createBuch();
void initBuch(Buch* this,char* titel,char* autor);

Datum* newDatum
  (short tag,short monat,unsigned int jahr);
Datum* createDatum();
void initDatum(Datum* this,short tag,short monat
               ,unsigned int jahr);

Ausleihe* newAusleihe
  (Person* ausleiher,Buch* ausgeliehenesBuch,Datum* rueckgabe);
Ausleihe* createAusleihe();
void initAusleihe
  (Ausleihe* this,Person* ausleiher,Buch* ausgeliehenesBuch
  ,Datum* rueckgabe);

```

Es folgt die Implementierung dieser Konstruktorfunktionen. Der eigentliche Konstruktor `newPerson` erzeugt mit dem Aufruf von `createPerson` das neue Objekte und initialisiert dieses mit dem Aufruf der Funktion `initPerson`.

BibliothekKonstruktorFunktionen.c

```
#include "BibliothekKonstruktorFunktionen.h"
#include <stdlib.h>

Person* newPerson(char* name,char* vorname,char* strasse
                 ,char* ort,unsigned int plz){
  Person* this = createPerson();
  initPerson(this,name,vorname,strasse,ort, plz);
  return this;
}
```

Zum Erzeugen eines neuen Objektes benutzen wir einen einfachen Aufruf an die Funktion `malloc` der Standardbibliothek:

BibliothekKonstruktorFunktionen.c

```
10 Person* createPerson(){return (Person*)malloc(sizeof(Person));}
```

Zum Initialisieren wird das als erstes Argument übergebene Objekt genommen und den einzelnen Felder hierin die als Parameter übergebenen Daten zugewiesen:

BibliothekKonstruktorFunktionen.c

```
11 void initPerson
12     (Person* this,char* name,char* vorname,char* strasse
13     ,char* ort,unsigned int plz){
14   this->name=name;
15   this->vorname=vorname;
16   this->strasse=strasse;
17   this->ort=ort;
18   this->plz=plz;
19 }
```

Analog erfolgt die Implementierung für die übrigen drei Klassen:

BibliothekKonstruktorFunktionen.c

```
20 Buch* createBuch(){return (Buch*)malloc(sizeof(Buch));}
21 void initBuch(Buch* this,char* titel,char* autor){
22   this->titel=titel;
23   this->autor=autor;
24 }
25 Buch* newBuch(char* titel,char* autor){
26   Buch* this=createBuch();
27   initBuch(this,titel,autor);
28   return this;
29 }
30
31 Datum* createDatum(){return (Datum*)malloc(sizeof(Datum));}
32 void initDatum(Datum* this,short tag,short monat
33                 ,unsigned int jahr){
34   this->tag=tag;
35   this->monat=monat;
36   this->jahr=jahr;
37 }
38 Datum* newDatum
39      (short tag,short monat,unsigned int jahr){
40   Datum* this = createDatum();
41   initDatum(this,tag,monat,jahr);
42   return this;
43 }
44
45 Ausleihe* createAusleihe(){
46   return (Ausleihe*)malloc(sizeof(Ausleihe));}
```

```
47
48 void initAusleihe(Ausleihe* this,Person* ausleiher
49                  ,Buch* ausgeliehenesBuch,Datum* rueckgabe){
50   this->ausleiher=ausleiher;
51   this->ausgeliehenesBuch=ausgeliehenesBuch;
52   this->rueckgabe=rueckgabe;
53 }
54
55 Ausleihe* newAusleihe
56      (Person* ausleiher,Buch* ausgeliehenesBuch,Datum* rueckgabe){
57   Ausleihe* this=createAusleihe();
58   initAusleihe(this,ausleiher,ausgeliehenesBuch,rueckgabe);
59   return this;
60 }
61
```

Zeit, um eine erste Hauptfunktion zum Testen der Objekte zu schreiben. Zum Erzeugen von Objekten werden nur noch die Konstruktorfuktionen benutzt. Mit der bereits bekannten Notation über den Operator `->` kann auf die einzelnen Felder zugegriffen werden:

TestBibliothekKonstruktorFunktionen.c

```
1  #include "BibliothekKonstruktorFunktionen.h"
2  #include <stdio.h>
3
4  int main(){
5    Person* p
6     = newPerson("Heck","Dieter Thomas","Schlagerstr.","Koeln",4000);
7    Buch* b    = newBuch("Meine besten Sprüche","Dieter Bohlen");
8    Datum* d   = newDatum(17,4,2007);
9    Ausleihe* a= newAusleihe(p,b,d);
10   printf("%s\n",a->ausleiher->name);
11   printf("%s\n",a->ausleiher->ort);
12   return 0;
13 }
```

Konstruktoren in C++ Jetzt wollen wir das Gleiche in C++ implementieren. Die grundlegende Idee der Objektorientierung ist, möglichst alle Funktionalität, die die Objekte einer Klasse betreffen, auch direkt in dieser Klasse zu definieren. Daher gibt es eine Notation, die es erlaubt Konstruktorfunktionen direkt in der Klasse zu definieren. Dieses geschieht syntaktisch dadurch, dass innerhalb der Rumpfes der Klasse die Konstruktordefinition steht. Ein Konstruktor besteht aus dem Namen der Klasse gefolgt von seinen Parameterdefinitionen in runden Klammern, so wie man es von Funktionen gewohnt ist.

Damit sieht die Klasse `Person` nun mit Konstruktor wie folgt aus:

BibliothekKonstruktoren.hpp

```
1  #include <string>
2
3  class Person {
4   public:
```

```
5     std::string name;
6     std::string vorname;
7     std::string strasse;
8     std::string ort;
9     unsigned int plz;
10
11    Person
12      (std::string name,std::string vorname
13      ,std::string strasse,std::string ort,unsigned int plz);
14  };
```

Analog seien für die anderen Klassen auch Konstruktoren definiert:

BibliothekKonstruktoren.hpp

```
15  class Buch{
16   public:
17    std::string titel;
18    std::string autor;
19
20    Buch(std::string titel,std::string autor);
21  };
22
23  class Datum{
24   public:
25    short tag;
26    short monat;
27    unsigned int jahr;
28
29    Datum(short tag,short monat,unsigned int jahr);
30  };
31
32  class Ausleihe {
33   public:
34    Person* ausleiher;
35    Buch* ausgeliehenesBuch;
36    Datum* rueckgabe;
37    Ausleihe(Person* ausleiher
38             ,Buch* ausgeliehenesBuch,Datum* rueckgabe);
39  };
```

In der Implementierungsdatei sind die Konstruktoren umzusetzen. Hierzu ist dem Konstruktornamen, der ja gleich dem Namen der Klasse ist, nocheinmal der Klassenname voranzusetzen, mit einem doppelten Doppelpunkt getrennt. Der Konstruktor für die Klasse `Person` hat also in der Implementierungsdatei den Namen `Person::Person`. Die Angabe vor dem doppelten Doppelpunkt gibt an, dass nun eine Eigenschaft aus der Klasse `Person` implementiert wird. Die Angabe nach dem dopppelten Doppelpunkt, dass es sich bei dieser Eigenschaft um einen Konstruktor handelt.

Der Rumpf der Implementierung ist eins-zu-eins die, die wir in der Implementierung der Initialisierung in unserer C Umsetzung benutzt haben. Der fundamentale Unterschied ist, dass

wir uns nicht um das Erzeugen des Zeigers `this` kümmern müssen. `this` ist tatsächlich ein Schlüsselwort in C++ und bezeichnet den Zeiger auf das Objekt für das etwas implementiert wird. Im Fall von Konstruktoren gerade ein Zeiger auf das Objekt, das gerade erzeugt wird.

Damit sieht die Implementierung des Konstruktors der Klasse `Person` wie folgt aus:

BibliothekKonstruktoren.cpp

```
1  #include "BibliothekKonstruktoren.hpp"
2  #include <iostream>
3
4  Person::Person
5      (std::string name,std::string vorname
6      ,std::string strasse,std::string ort,unsigned int plz){
7    this->name=name;
8    this->vorname=vorname;
9    this->strasse=strasse;
10   this->ort=ort;
11   this->plz=plz;
12 }
```

Ganz analog wird für die drei übrigen Klassen verfahren:

BibliothekKonstruktoren.cpp

```
13 Buch::Buch(std::string titel,std::string autor){
14   this->titel=titel;
15   this->autor=autor;
16 }
17
18 Datum::Datum
19     (short tag,short monat,unsigned int jahr){
20   this->tag=tag;
21   this->monat=monat;
22   this->jahr=jahr;
23 }
24
25 Ausleihe::Ausleihe
26     (Person* ausleiher,Buch* ausgeliehenesBuch,Datum* rueckgabe){
27   this->ausleiher=ausleiher;
28   this->ausgeliehenesBuch=ausgeliehenesBuch;
29   this->rueckgabe=rueckgabe;
30 }
```

Jetzt ist interessant, wie ein Konstruktor in C++ aufgerufen wird. Hierzu stellt C++ ein neues Schlüsselwort zur Verfügung, das Wort `new`. Dem Wort `new` hat der Namen einer Klassen zu folgen[2]. Damit wird angezeigt, dass ein Konstruktor für diese Klasse aufgerufen werden soll. In runden Klammern, ebenso wie bei Funktionsaufrufen, folgen die Argumente für diesen Konstruktor. Damit sieht ein erster Test mit unseren vier Klassen wie folgt aus:

[2]Tatsächlich kann dem Wort `new` auch ein anderer Typname folgen, z.B. `int`.

TestBibliothekKonstruktoren.cpp

```
#include "BibliothekKonstruktoren.hpp"
#include <iostream>

int main(){
 Person* p
  =new Person("Heck","Dieter Thomas","Schlagerstr.42","Koeln",4000);
 Buch* b   = new Buch("Meine besten Sprüche","Dieter Bohlen");
 Datum* d  = new Datum(17,4,2007);
 Ausleihe* a=new Ausleihe(p,b,d);

 std::cout << a->ausleiher->vorname << std::endl;
 std::cout << a->ausleiher->ort<< std::endl;
 return 0;
}
```

Konstruktoren und `new` scheinen nur eine leichte Abkürzung dessen zu sein, was wir im vorigen Abschnitt in C implementiert haben.

Operationen auf Objekten

Nachdem wir nun Objekte erzeugen können, gibt es sicher eine ganze Reihe unterschiedlicher Dinge, die mit diesen Objekten gemacht werden sollen.

Funktionen in C In C schreiben wir gängiger Weise Funktionen, um mit bestimmten Daten zu operieren. Bei einem möglichst objektorientierten Stil in C, ist es empfehlenswert als ersten Parameter diesen Funktionen, einen Zeiger auf das Objekt zu geben, mit dem operiert werden soll. Schon im ersten Semester haben wir uns angewöhnt diesen Zeiger mit dem Namen `this` zu bezeichnen. Wollen wir Funktionen bereitstellen, die die einzelnen Objekte auf der Kommandozeile ausgeben können, so stellen wir sinnvoller Weise Funktionen mit folgenden Signaturen bereit:

BibliothekFunktionen.h

```
#include "BibliothekKonstruktorFunktionen.h"

void personPrint(Person* this);
void buchPrint(Buch* this);
void datumPrint(Datum* this);
void ausleihePrint(Ausleihe* this);
```

Vielleicht wollen wir desweiteren eine Funktion haben, die ein Datumsobjekt so verändert, dass der Termin um einen Monat erhöht wird. Diese Funktionalität könnte dann benutzt werden, um eine Ausleihe um einen Monat zu verlängern:

BibliothekFunktionen.h

```
void datumUmEinenMonatErhoehen(Datum* this);
void ausleiheVerlaengere(Ausleihe* this);
```

Vielleicht ist für eine Ausleihe interessant, ob sie nicht schon überfällig ist. Hierzu muss zusätzlich das aktuelle Datum als Parameter vorgesehen werden:

BibliothekFunktionen.h

```
9  typedef enum {false,true} bool;
10 bool ausleiheUeberfaellig(Ausleihe* this,Datum* heute);
```

Wie man sieht, haben wir die Funktionen so definiert, dass sie immer mit dem Namen der Struktur, für die die Funktion gedacht ist, beginnen lassen. Desweiteren hat die Funktion immer als ersten Parameter einen Zeiger auf ein Objekt der Struktur, für die die Funktion gedacht ist. Diesen Parameter haben wir immer mit `this` bezeichnet.

Die Implementierung der Funktionen ist einfacher, wenig interessanter C-Code:

BibliothekFunktionen.c

```
1  #include "BibliothekFunktionen.h"
2  #include <stdio.h>
3
4  void personPrint(Person* this){
5    printf("Person(%s,%s,%s,%s,%i)"
6           ,this->name
7           ,this->vorname
8           ,this->strasse
9           ,this->ort
10          ,this->plz);
11 }
12 void buchPrint(Buch* this){
13   printf("Buch(%s,%s)",this->titel,this->autor);
14 }
15 void datumPrint(Datum* this){
16   printf("%i.%i.%i",this->tag,this->monat,this->jahr);
17 }
18 void ausleihePrint(Ausleihe* this){
19   printf("Ausleihe(");
20   personPrint(this->ausleiher);
21   printf(",");
22   buchPrint(this->ausgeliehenesBuch);
23   printf(",");
24   datumPrint(this->rueckgabe);
25   printf(")");
26 }
27
28 void datumUmEinenMonatErhoehen(Datum* this){
29   if (this->monat==12)
30     this->jahr = this->jahr+1;
31
32   this->monat=(this->monat%12)+1;
33
34   if (this->tag>28 && this->monat==2)this->tag=28;
35
36   if (this->tag==31)
```

```
37      switch (this->monat){
38        case 4:
39        case 6:
40        case 9:
41        case 11: this->tag=30;
42      }
43  }
44
45  void ausleiheVerlaengere(Ausleihe* this){
46    datumUmEinenMonatErhoehen(this->rueckgabe);
47  }
48
49  bool ausleiheUeberfaellig(Ausleihe* this,Datum* heute){
50    Datum* d=this->rueckgabe;
51    return
52        heute->jahr>d->jahr
53     || (heute->jahr==d->jahr && heute->monat>d->monat)
54     || (heute->jahr==d->jahr && heute->monat==d->monat
55                              && heute->tag>d->tag);
56  }
57
```

In gewohnter Weise können wir nun diese Funktionen aufrufen. Hierzu sind ihnen die entsprechenden Argumente zu übergeben. Diese sind die Objekte, wie sie zunächst mit Konstruktoren zu erzeugen sind.

TestBibliothekFunktionen.c

```
1  #include "BibliothekFunktionen.h"
2
3  #include <stdio.h>
4
5  int main(){
6   Person* p
7    =newPerson("Heck","Dieter Thomas","Schlagerstr.42","Koeln",4000);
8   Buch* b   = newBuch("Meine besten Sprüche","Dieter Bohlen");
9   Datum* d  = newDatum(17,4,2007);
10  Ausleihe* a=newAusleihe(p,b,d);
11
12  ausleihePrint(a);
13  ausleiheVerlaengere(a);
14  ausleihePrint(a);
15  return 0;
16 }
```

Methoden in C++ Jetzt soll die gleiche Funktionalität in C++ realisiert werden. Hier gilt gleiches, wie für die Konstruktoren. Funktionen, die sich primär auf Objekte einer bestimmten Klasse beziehen, werden direkt in dieser Klasse definiert. Solche Funktionen nennt man dann *Methoden*. Ebenso wie bei Konstruktoren wird dann auf das erste Argument, der Zeiger auf das

Objekt, für das die Methode geschrieben wird, verzichtet. Dieses ist implizit da und wird durch das Schlüsselwort `this` bezeichnet.

Da in C++ Methoden direkt innerhalb einer Klasse stehen, ist es nicht mehr nötig, im Namen zu signalisieren, zu welcher Klasse die Methode gehört. Statt also eine Funktion mit Namen `personPrint` zu definieren, reicht es jetzt aus, die methode `print` in der Klasse `Person` zu definieren.

Ergänzen wir die Klasse `Person` nun um eine Methode `print`, so erhalten wir folgende Klassendefinition:

BibliothekMethoden.hpp

```
1  #include <string>
2
3  class Person {
4   public:
5    std::string name;
6    std::string vorname;
7    std::string strasse;
8    std::string ort;
9    unsigned int plz;
10
11   Person
12     (std::string name,std::string vorname
13     ,std::string strasse,std::string ort,unsigned int plz);
14
15   virtual void print();
16 };
```

Die Methode `print` benötigt keinen Parameter, denn sie braucht nur einen Zeiger auf das Objekt, für die die Methode ausgeführt wird. Dieser Zeiger ist bei Methoden implizit vorhanden.

Vor der Signatur der Methode `print` wurde oben noch das Attribut `virtual` eingefügt. Dieses Attribut werden wir später noch genauer hinterleuchten. Will man alle Vorteile, die die Objektorientierung bietet, vollständig ausnutzen, ist es sinnvoll immer vor Methoden noch das Attribut `virtual` in der Kopfdatei zu stellen.

Auf gleiche Weise seien die übrigen Klassen um ein Paar Methoden ergänzt:

BibliothekMethoden.hpp

```
1  class Buch{
2   public:
3    std::string titel;
4    std::string autor;
5
6    Buch(std::string titel,std::string autor);
7    virtual void print();
8  };
9
10 class Datum{
11  public:
12   short tag;
13   short monat;
```

```
14    unsigned int jahr;
15
16    Datum(short tag,short monat,unsigned int jahr);
17    virtual void print();
18    virtual void umEinenMonatErhoehen();
19  };
20
21  class Ausleihe {
22   public:
23    Person* ausleiher;
24    Buch* ausgeliehenesBuch;
25    Datum* rueckgabe;
26
27    Ausleihe
28       (Person* ausleiher,Buch* ausgeliehenesBuch,Datum* rueckgabe);
29    void print();
30    void verlaengere();
31    bool ueberfaellig(Datum* heute);
32  };
```

Tatsächlich ist die Implementierung dieser Methoden ganz geradeheraus, wie es auch bei den Funktionen in C bereits der Fall war. Entscheidener Unterschied ist, dass wir mit dem Schlüsselwort `this` arbeiten können, und damit auf das Objekt zeigen, für das die Methode ausgeführt wird.

Ebenso wie bei Konstruktoren ist in der Implementierungsdatei vor dem Methodennamen mit doppelten Doppelpunkt der Klassenname zu setzen, für die Klasse, für die die entsprechende Methode implementiert wird. Nur so läßt sich in der Implementierungsdatei unterscheiden, welche der vier Methoden, die alle `print` heißen gerade implementiert wird.

Damit können die ersten drei Methoden `print`, wie folgt implementiert werden:

BibliothekMethoden.cpp

```
1   #include "BibliothekMethoden.hpp"
2   #include <iostream>
3
4   void Person::print(){
5     std::cout
6       << "Person "
7       << "(" << this->name
8       << "," << this->vorname
9       << "," << this->strasse
10      << "," << this->ort
11      << "," << this->plz
12      << ")";
13  }
14  void Buch::print(){
15    std::cout
16      << "Buch"
17      << "(" << this->titel
18      << "," << this->autor
```

```
19      << ")";
20  }
21  void Datum::print(){
22    std::cout
23      << "Datum"
24      << "(" << this->tag
25      << "," << this->monat
26      << "," << this->jahr
27      << ")";
28  }
```

In der Methode `print` für die Objekte der Klasse `Ausleihe` werden wir sicher den Ausleiher, der ja ein Objekt der Klasse `Person` ist, sowie das ausgeliehene Buch, und schließlich das Rückgabedatum ausdrucken wollen. Hierzu gibt es in den entsprechenden Klassen bereits eine Methode `print`, die wir gerne nutzen wollen. Ebenso, wie in der C Umsetzung in der Funktion `ausleihePrint` Aufrufe an die Funktionen: `personPrint`, `buchPrint` und `datumPrint` standen. In C++ werden Methoden aufgerufen, indem zunächst der Zeiger auf das Objekt steht, für das die Methode aufzurufen ist, und dann mit dem bereits für Felder bekannten Pfeiloperator `->` der eigentliche Methodenaufruf folgt. Dieses kann man lesen als:
Du Objekt, auf den dieser Zeiger verweist, du hast doch in Deiner Klasse eine Methode mit folgenden Namen. Führe diese doch bitte mit den hier spezifizierten Argumenten aus.

Somit läßt sie die Methode `print` der Klasse `Ausleihe` wie folgt implementieren:

BibliothekMethoden.cpp

```
29  void Ausleihe::print(){
30    std::cout<< "Ausleihe(";
31    this->ausleiher->print();
32    std::cout<< ",";
33    this->ausgeliehenesBuch->print();
34    std::cout<< ",";
35    this->rueckgabe->print();
36    std::cout<< ")";
37  }
```

Hier werden drei Aufrufe an Methoden `print` gemacht. Der Zeiger auf dem diese Methoden aufgerufen werden bezeichnet jeweils ein Objekt. Und die Klasse dieser Objekte bestimmt jeweils, welche Methode `print` aufgerufen wird.

Nun ist es ein Leichtes auch die übrigen Methoden zu implementieren:

BibliothekMethoden.cpp

```
38  void Datum::umEinenMonatErhoehen(){
39    if (this->monat==12) this->jahr = this->jahr+1;
40
41    this->monat=(this->monat%12)+1;
42
43    if (this->tag>28 && this->monat==2)this->tag=28;
44
45    if (this->tag==31)
46      switch (this->monat){
```

```
47        case 4:
48        case 6:
49        case 9:
50        case 11: this->tag=30;
51      }
52  }
53
54  void Ausleihe::verlaengere(){
55    this->rueckgabe->umEinenMonatErhoehen();
56  }
57
58  bool Ausleihe::ueberfaellig(Datum* heute){
59    Datum* d=this->rueckgabe;
60    return
61        heute->jahr>d->jahr
62     || (heute->jahr==d->jahr && heute->monat>d->monat)
63     || (heute->jahr==d->jahr && heute->monat==d->monat
64                              && heute->tag>d->tag);
65  }
```

Natürlich übernehmen wir ebenso auch die Implementierungen der Konstruktoren, wie sie im vorhergehenden Abschnitt bereits zu sehen waren.

BibliothekMethoden.cpp

```
66  Person::Person
67      (std::string name,std::string vorname
68      ,std::string strasse,std::string ort,unsigned int plz){
69    this->name=name;
70    this->vorname=vorname;
71    this->strasse=strasse;
72    this->ort=ort;
73    this->plz=plz;
74  }
75
76  Buch::Buch(std::string titel,std::string autor){
77    this->titel=titel;
78    this->autor=autor;
79  }
80
81  Datum::Datum
82      (short tag,short monat,unsigned int jahr){
83    this->tag=tag;
84    this->monat=monat;
85    this->jahr=jahr;
86  }
87
88  Ausleihe::Ausleihe
89      (Person* ausleiher,Buch* ausgeliehenesBuch,Datum* rueckgabe){
90    this->ausleiher=ausleiher;
91    this->ausgeliehenesBuch=ausgeliehenesBuch;
92    this->rueckgabe=rueckgabe;
93  }
```

Nun können wir erstmals Methoden testen. Hierbei ist immer nur zu bedenken, dass vor einem Methodenaufruf mit dem Pfeiloperator getrennt, das Objekt stehen muss, für welches die Methode aufzurufen ist.

TestBibliothekMethoden.cpp

```
1  #include "BibliothekMethoden.hpp"
2  #include <iostream>
3
4  int main(){
5   Person* p
6   =new Person("Heck","Dieter Thomas","Schlagerstr.","Koeln",4000);
7   Buch* b   = new Buch("Meine besten Sprüche","Dieter Bohlen");
8   Datum* d  = new Datum(17,4,2007);
9   Ausleihe* a=new Ausleihe(p,b,d);
10
11  a->print();         std::cout<<std::endl;
12  a->verlaengere();
13  a->print();         std::cout<<std::endl;
14  return 0;
15 }
```

implizites this

Dem einem oder anderen Leser mag das inflationär in unserem Code auftauchende Schlüsselwort `this` schon etwas enervierend aufgestoßen sein. Und tatsächlich kann fast immer auf die Referenz zum Objekt, für das der Code ausgeführt wird, verzichtet werden. Variablennamen werden in C++ so aufgelöst, dass zunächst geschaut wird, ob es eine lokale Variable oder aber einen Parameter diesen Namens gibt. Ist dieses nicht der Fall, so wird geschaut, ob es in der Klasse, in der der Code steht, ein Feld mit diesem Namen gibt. Ebenso wird mit Methoden verfahren. Bei einen Methodenaufruf ohne vorangestellten Objekt, wird implizit das `this`-Objekt berücksichtigt.

Beispiel 2.1.1 *Auch hierzu soll es ein kleines Beispiel geben. Wir definieren eine Klasse mit ein paar willkürlichen Eigenschaften.*

WithoutThis.hpp

```
1  class WithoutThis{
2  public:
3    int x;
4    int y;
5    int f(int z);
6    void print();
7    WithoutThis(int pX,int y);
8  };
```

Bei der Implementierung des Konstruktors kann beim Initialisieren des Feldes `x` *auf die explizite Referenzierung des Feldes* `x` *über den Zeiger* `this` *verzichtet werden. Es gibt sonst keine lokale Variable oder Parameter gleichen Namens. Damit löst der Compiler den Bezeichner als das Feld der Klasse* `WithoutThis` *auf. Bei der Initialisierung des Feldes kann auf die Referenzierung über* `this` *nicht verzichtet werden. Hier ist das Feld* `this->y` *vom Parameter* `y` *zu unterscheiden.*

WithoutThis.cpp

```
1  #include "WithoutThis.hpp"
2  #include <iostream>
3
4  WithoutThis::WithoutThis(int pX,int y){
5    x=pX;
6    this->y=y;
7  }
```

Bei der Implemengtierung der Methode `f` *kann ohne die Referenz des* `this`*-Zeigers auf die Felder* `x` *und* `y` *des Objekts, für das die Methode ausgeführt wird, zugegriffen werden.*

WithoutThis.cpp

```
8  int WithoutThis::f(int z){
9    return (x+y)*z;
10 }
```

Ebenso können ohne die `this`*-Referenz andere Methoden der Klasse aufgerufen werden. So läßt sich die Methode* `f` *im Rumpf der Methode* `print` *aufrufen.*

WithoutThis.cpp

```
11 void WithoutThis::print(){
12   std::cout<<f(2)<<std::endl;
13 }
```

Ein abschließender minimaler Aufruf der Klasse:

WithoutThis.cpp

```
14 int main(){
15   (new WithoutThis(17,4))->print();
16   return 0;
17 }
```

Aufgabe 1 In dieser Aufgabe sollen Sie Klassen zur Beschreibung geometrischer Objekte im 2-dimensionalen Raum entwickeln. Schreiben Sie für jede Klasse geeignete Konstruktoren und eine Methode `print`.

Lösen Sie die Aufgaben jeweils einmal, indem Sie sie in C programmieren und geeignete Strukturen objektorrientiert verwenden, und einmal, indem Sie mit Klassen in C++ arbeiten.

a) Schreiben Sie eine Klasse (Struktur) `Vertex`, die Punkte im 2-dimensionalen Raum darstellt. Betrachten Sie diese Punkte als Vektoren und definieren Sie Methoden zur Addition und Subtraktion von Punkten, sowie zur Multiplikation mit einem Skalar und zur Betragsberechnung. Diese Methoden sollen das Objekt unverändert lassen, und ein Ergebnis zurückliefern.
Testen Sie die Klasse `Vertex` in einer `main`-Methode.

b) Ergänzen Sie jetzt Ihre Klasse `Vertex` um Varianten der Addition und Subtraktion, die kein Ergebnis liefern, aber das `this`-Objekt verändern.

c) Schreiben Sie eine Klasse `GeometricObject`. Ein geometrisches Objekt soll eine Weite und Höhe, sowie eine Position der oberen rechte Ecke im 2-dimensionalen Raum enthalten.

d) Implementieren Sie die folgende Methoden für Ihre Klasse `GeometricObject`:

- `bool hasWithin(Vertex* p)`, die wahr ist, wenn der Punkt `p` innerhalb der geometrischen Figur liegt.
- `bool touches(GeometricObject* that)` sei wahr, wenn es mindestens einen Punkt gibt, der in beiden Objekten liegt.

Testen Sie auch diese Methoden.

2.1.2 Vererbung

Bisher haben wir, außer jede Menge Syntax, neue Schlüsselwörter und Notationen, nicht sehr viel gewonnen, gegenüber der Implementierungen in C. Das wird sich mit zunehmenden Maße nun ändern. Eine der Schlüsselideen der Objektorientierung ist es, Hierarchien von Objekten zu definieren. Die Klasse `Person` in unserem Beispiel ist relativ allgemein. Es gibt viele verschiedene Arten von Personen. In einer Hochschule wie der unsrigen sind das vielleicht Studenten, Professoren, Lehrbeauftragte oder Laboringenieure. Dieses sind mit Sicherheit alles Personen, allerdings Personen, über die es speziellere Informationen gibt, als die, die in der Klasse `Person` gespeichert werden können. Vielleicht haben diese Personen nicht nur speziellere Informationen, sondern verhalten sich auch anders. So ist vorstellbar, dass Professoren eine längere Ausleihfrist in der Bibliothek haben, als Studenten.

Wenn wir die oben genannten speziellen Formen von Personen modellieren wollen, so wollen wir sicher nicht noch einmal die Eigenschaften die alle Personen haben komplett aufzählen, sondern sagen können: Du bist eine spezielle Person, hast damit aber automatisch alle Eigenschaften, die allgemeine Personen auch haben.

Strukturen erweitern in C

Es geht uns also darum, bestehende Klassen von Objekten zu spezialisieren. Diese also um weitere Eigenschaften zu erweitern. Ein Student sei z.B. um eine Matrikelnummer erweitert, ein Professor um die Information seiner Büronummer. In C können wir neue Strukturen definieren, die als erstes Feld die Struktur bezeichnen, die um weitere Eigenschaften erweitert werden soll. So läßt sich die Struktur für Studenten wie folgt definieren:

Personen.h

```
1 #include "BibliothekFunktionen.h"
2 typedef struct {
3   Person super;
4   char* matrNr;
5 } Student;
```

Hier sagen wir, ein Student hat als Obereigenschaft die Eigenschaften einer Person und zusätzlich eine Matrikelnummer. So wie wir es bereits gewohnt sind, definieren wir einen Konstruktor und eine Initialisierungsfunktion für die Struktur `Student`:

Personen.h

```
1  Student* newStudent
2    (char* name,char* vorname,char* strasse
3    ,char* ort,unsigned int plz,char* matrNr);
4
5  void initStudent(Student* this,char* matrNr);
```

Der Konstruktor enthält dabei auch alle Parameter, die man benötigt, um eine neue Person zu erzeugen. Die Initialisierungsfunktion enthält hingegen nur noch die Eigenschaften, die ein Student zusätzlich zu einer Person hat.

Um mehr als ein Beispiel zu haben, sei auch noch eine Struktur `Professor` definiert:

Personen.h

```
6  typedef struct  {
7    Person super;
8    char* bueroNummer;
9  } Professor;
10
11 Professor* newProfessor
12   (char* name,char* vorname,char* strasse
13   ,char* ort,unsigned int plz,char* bueroNummer);
14
15 void initProfessor(Professor* this,char* bueroNummer);
```

Wir wollen auch für diese beiden Strukturen eigene Funktionen `print` vorsehen:

Personen.h

```
16 void studentPrint(Student* this);
17 void professorPrint(Professor* this);
```

Schreiten wir zur Implementierung dieser Funktionen. Die beiden Funktionen zum Erzeugen neuer Objekte sind ganz nach Schema wie zuvor bei den anderen vier Strukturen zu implementieren:

Personen.c

```
1  #include "Personen.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  Student* createStudent(){return (Student*)malloc(sizeof(Student));}
6  Professor* createProfessor(){
7    return (Professor*)malloc(sizeof(Professor));
8  }
```

Die beiden Initialisierungsfunktionen brauchen nur die jeweils neuen Eigenschaften zu initialisieren:

Personen.c
```
9  void initStudent(Student* this,char* matrNr){
10   this->matrNr = matrNr;
11 }
12 void initProfessor(Professor* this,char* bueroNummer){
13   this->bueroNummer = bueroNummer;
14 }
```

Die eigentlichen Konstruktoren, erzeugen im ersten Schritt das neue Objekt im Speicher. Anschließend initialisieren sie die Eigenschaften der Struktur, die erweitert wurde. In diesem Fall also alle Eigenschaften der Struktur `Person`. Dieses geschieht über den Aufruf der Initialisierungsfunktion für Personen. Im dritten Schritt sind dann die neuen Eigenschaften zu initialisieren.

Für die Klasse `Person` erhalten wir also folgende Implementierung der Konstruktorfunktion:

Personen.c
```
15 Student* newStudent(char* name,char* vorname,char* strasse
16                    ,char* ort,unsigned int plz,char* matrNr){
17   Student* this = createStudent();
18   initPerson((Person*)this,name,vorname,strasse,ort,plz);
19   initStudent(this,matrNr);
20   return this;
21 }
```

Entsprechend analog die Konstruktorimplementierung für Professoren:

Personen.c
```
22 Professor* newProfessor
23     (char* name,char* vorname,char* strasse
24     ,char* ort,unsigned int plz,char* bueroNummer){
25   Professor* this = createProfessor();
26   initPerson((Person*)this,name,vorname,strasse,ort,plz);
27   initProfessor(this,bueroNummer);
28   return this;
29 }
```

Für die beiden Funktionen `print` können wir durchaus die Funktion `print` für die Struktur `Person` benutzen:

Personen.c
```
30 void studentPrint(Student* this){
31   printf("Student(");
32   personPrint((Person*)this);
33   printf(",%s)",this->matrNr) ;
34 }
35 void professorPrint(Professor* this){
36   printf("Professor(");
37   personPrint((Person*)this);
38   printf(",%s)",this->bueroNummer) ;
39 }
```

Tatsächlich können wir jeden Zeiger auf einen Studenten auch als einen Zeiger auf eine Person, und jeden Zeiger auf einen Professor ebenso als einen Zeiger auf eine Person benutzen. Damit ist zum Beispiel sichergestellt, dass nicht nur allgemeine Personen Bücher ausleihen können, sondern auch Objekte der spezielleren Klassen `Student` und `Professor`.

Hiervon überzeuge uns folgender Test:

TestPersonen.c

```
1  #include "Personen.h"
2  #include <stdio.h>
3
4  int main(){
5    Student* s= newStudent("Wiseguy","Martin"
6                          ,"Nerdstreet.42","Offenbach",6000,"674656");
7    Person* p = (Person*)s;
8    Buch* b   = newBuch("Meine besten Sprüche","Dieter Bohlen");
9    Datum* d  = newDatum(17,4,2007);
10   Ausleihe* a=newAusleihe(p,b,d);
11
12   ausleihePrint(a);       printf("\n");
13   ausleiheVerlaengere(a);
14   ausleihePrint(a);       printf("\n");
15
16   personPrint(p);         printf("\n");
17   studentPrint(s);        printf("\n");
18   return 0;
19 }
```

Wie man sieht, muss man sich entscheiden, ob man die Funktion `print` für Personen, oder die Funktion `print` für Studenten auf ein Objekt der Struktur `Student` anwenden möchte. Beides ist möglich, denn ein Studentenobjekt ist ein vollgültiges Personenobjekt.

```
sep@pc305-3:~/www/fh/cpp/student> bin/TestPersonen
Ausleihe(Person(Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),Buch(Meine besten Sprüche,Dieter Bohlen),17.4.2007
Ausleihe(Person(Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),Buch(Meine besten Sprüche,Dieter Bohlen),17.5.2007
Person(Wiseguy,Martin,Nerdstreet.42,Offenbach,6000)
Student(Person(Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),674656)
sep@pc305-3:~/www/fh/cpp/student>
```

Man vergegenwärtige sich schon einmal, dass beim Ausdrucken eines Ausleiheobjektes die speziellere Information +ber die Person, nämlich dass es sich um einen Studenten handelt, verloren geht.

Unterklassen in C++

Auch das Erweitern von Klassen um weitere Eigenschaften ist in der Objektorientierung kultiviert worden und hat eine eigene Syntax erhalten. Anstatt ein Feld der Struktur, die um weitere Eigenschaften erweitert werden soll, vorzusehen, kann man dem Klassennamen mit einem Doppelpunkt getrennt den Namen der Klasse folgen lassen, die erweitert werden soll. Zusätzlich sollte man im Allgemeinen dieser sogenannten Oberklasse das Schlüsselwort `public` voranstellen. Soll also die Klasse `Student` die Klasse `Person` erweitern, so sieht der Kopf der neuen Klassendefinition wie folgt aus:

CppPersonen.hpp
```
1 #include "BibliothekMethoden.hpp"
2 class Student:public Person{
```

Damit sagen wir: jedes Objekt der Klasse `Student` sei auch ein Objekt der Klasse `Person`. `Student` wird dabei als Unterklasse von der Oberklasse `Person` bezeichnet. Die Unterklasse erweitert die Oberklasse. Im Rumpf der Klassendefinition werden jetzt nur noch die Eigenschaften aufgelistet, die die Unterklasse zusätzlich hat, oder die sie neu definiert. In unserem Fall soll die Klasse `Student` zusätzlich die Eigenschaft `matrNr` und einen eigenen Konstruktor haben. Die Methoden `print` hat zwar schon jede Person, wir allerdings wollen für Studenten eine neue von der Implementierung in der Klasse `Person` abweichende Implementierung haben. Daher hat die Klasse `Student` drei eigene Eigenschaften:

CppPersonen.hpp
```
3   public:
4     std::string matrNr;
5     Student(std::string name,std::string vorname,std::string strasse
6           ,std::string ort,unsigned int plz,std::string matrNr);
7     virtual void print();
8 };
```

Auf die gleiche Weise kann mit der zweiten Unterklasse verfahren werden:

CppPersonen.hpp
```
9  class Professor:public Person{
10   public:
11     std::string bueroNr;
12     Professor
13       (std::string name,std::string vorname,std::string strasse
14       ,std::string ort,unsigned int plz,std::string bueroNr);
15     virtual void print();
16 };
```

Zum Implementieren der Konstruktoren einer Unterklasse gibt es in C++ eine eigene Syntax, die sich aufs Initialisieren der Eigenschaften aus der Oberklasse bezieht. In der Umsetzung der Unterklassen in der Programmiersprache C haben wir den Initialisierungsfunktion für die Oberklasse aufgerufen. Das müssen wir auch in C++ machen. Dieses ist auch recht logisch: bevor wir eine Person erweitern können zu einen Studenten, müssen wir diese Person erst einmal vollständig initialisiert haben. Syntaktisch geschieht dieses, indem vor der öffnenden Klammer des Konstruktorrumpfes der Aufruf des Konstruktors der Oberklasse erfolgt. Dieser Aufruf wird von der Parameterliste mit einem Doppenpunkt getrennt. Man nennt dieses den Aufruf des Superkonstruktors.

CppPersonen.cpp
```
1 #include "CppPersonen.hpp"
2 #include <iostream>
3
4 Student::Student
5    (std::string name,std::string vorname,std::string strasse
```

```
6     ,std::string ort,unsigned int plz,std::string matrNr)
7   :Person(name,vorname,strasse,ort,plz){
8    this->matrNr = matrNr;
9   }
10
11  Professor::Professor
12     (std::string name,std::string vorname,std::string strasse
13     ,std::string ort,unsigned int plz,std::string bueroNr)
14   :Person(name,vorname,strasse,ort,plz){
15    this->bueroNr = bueroNr;
16  }
```

In den beiden Beispiel ist der Aufruf des Superkonstruktors farblich markiert. Man merke sich: im Konstruktor muss immer ein Aufruf an den Superkonstruktor als erstes geschehen.

Schreiten wir jetzt zur Implementierung der beiden Methoden `print`. Die Klassen `Student` und `Professor` haben beiden bereits eine Methode namens `Print`, nämlich die der Klasse `Person`. Wir wollen nun eine eingene spezialisierte Methode `print` schreiben. Schreiben wir in einer Unterklasse eine Methode neu, die es in einer Oberklasse bereits gibt, so spricht man vom *Überschreiben* der Methode. Dieses Überschreiben wird im Englische mit dem Begriff *override* bezeichnet.

Tatsächlich brauchen wir nirgendwo anzugeben, dass wir eine bestehende geerbte Methode überschreiben. Es gibt objektorientierte Sprachen, in denen dieses notwendig ist.

In unserer C Umsetzung der Unterklassen wurde in der Methode `studentPrint` ein Aufruf auf die Methode `personPrint` getätigt. Dieses ist auch in C++ möglich. Nun heißen alle Methoden allerdings in unserem Beispiel nur `print`. Wie kann man dem C++-Compiler sagen, dass eine ganz bestimmte Methode `print` gemeint ist? Hierzu besinne man sich auf den Namen, den die Methode in der Implementierungsdatei hatte. Hier ist dem Namen `print` mit doppelten Doppelpunkt vorangestellt, in welcher Klasse sich diese Methode befindet. Der um diese Information zusätzlich spezifizierte Name wird auch gerne als voll qualifizierter Name bezeichnet. Wollen wir innerhalb der Klasse `Professor` die Mehtode `print` der Klasse `Person` auf einem `Professor`-Objekt aufrufen, so müssen wir den Methodennamen voll-qualifiziert als `Person::print` angeben.

Damit können wir zu folgender Implementierung der Methode `print` in den beiden Unterklassen gelangen:

CppPersonen.cpp

```
17  void Professor::print(){
18    std::cout << "Professor(";
19    this->Person::print();
20    std::cout << "," << this->bueroNr <<")" ;
21  }
22
23  void Student::print(){
24    std::cout << "Student(";
25    this->Person::print();
26    std::cout << "," << this->matrNr <<")" ;
27  }
```

Es soll jetzt einmal der gleiche Test wie in der zuvor gemachten C-Umsetzung durchgeführt werden:

TestCppPersonen.cpp

```
1  #include "CppPersonen.hpp"
2  #include <iostream>
3
4  int main(){
5    Student* s= new Student("Wiseguy","Martin"
6                            ,"Nerdstreet.42","Offenbach",6000,"674656");
7    Person* p = (Person*)s;
8    Buch* b   = new Buch("Meine besten Sprüche","Dieter Bohlen");
9    Datum* d  = new Datum(17,4,2007);
10   Ausleihe* a=new Ausleihe(p,b,d);
11
12   a->print();           std::cout << std::endl;
13   a->verlaengere();
14   a->print();           std::cout << std::endl;
15
16   p->print();           std::cout << std::endl;
17   s->print();           std::cout << std::endl;
18   s->Person::print();   std::cout << std::endl;
19   return 0;
20 }
```

Man sieht, dass die Methoden mit der Pfeilnotation für ein Objekt ausgeführt werden. Man sieht auch, dass man unqualifiziert die Mehtode auf ein Objekt ausführen kann wie bei `s->print()` und voll-qualifiziert wie bei `s->Person::print()`.

Schauen wir uns jetzt aber einmal genau die Ausgabe an und vergleichen diese mit der Ausgabe in der C-Implmentierung der gleichen Klassen:

```
sep@pc305-3:~/www/fh/cpp/student> bin/TestCppPersonen
Ausleihe(Student(Person (Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),674656),Buch(Meine besten Sprüche,Dieter
Ausleihe(Student(Person (Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),674656),Buch(Meine besten Sprüche,Dieter
Student(Person (Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),674656)
Student(Person (Wiseguy,Martin,Nerdstreet.42,Offenbach,6000),674656)
Person (Wiseguy,Martin,Nerdstreet.42,Offenbach,6000)
sep@pc305-3:~/www/fh/cpp/student>
```

Es gibt einen fundamentalen Unterschied zur entsprechenden Umsetzung in C: die `ausleihePrint`-Funktion für die Struktur Ausleihe hat in der C-Implementierung fest verdrahtet die Funktion `personPrint` aufgerufen. Daher war dort in der Ausgabe nicht zu erkennen, dass es sich bei dem Ausleiher um eine spezielle Person, nämlich ein Objekt der Klasse `Student` handelt.

In der C++-Implementierung ist diese Information der Ausgabe zu entnehmen. Offensichtlich wird in der Methode `Ausleihe::print` nicht unbedingt die Methode `Person::print` aufgerufen, sondern mitunter die überschreibende Methode `print` in diesem Fall `Student::print`. Der Aufruf `ausleiher->print()` aus der Methode `Ausleihe::print` ist zu lesen als:

Ich weiß, dass das Objekt, das im Feld `ausleiher` *gespeichert ist eine Person darstellt. Daher weiß ich, dass dieses Objekt eine Methode* `print` *hat. Es könnte sein, dass dieses Objekt ein*

Objekt einer Unterklasse von `Person` *ist, die die Methode* `print` *überschreibt. Wenn dem so ist, dann soll das Objekt jetzt seine überschriebene Version der Methode* `print` *benutzen.*

Dieses Phänomen wird als späte Bindung (*late binding*) bezeichnet.

Späte Bindung

Das im letzten Abschnitt kennengelernte Phänomen der späten Bindung verdient noch ein wenig einer eingehenderen und isolierteren Betrachtung.

Nutzen später Bindung für Bibliotheken Wir werden hierzu zunächst einen weiten Ausflug in hintere Kapitel des Skriptes machen. Auf Seite 4.2.3 findet sich die Klasse `Dialogue`. Sie ist eine kleine Klasse zum Erzeugen eines Ein-/Ausgabedialogs in der graphischen Benutzeroberfläche. Die Klasse `Dialogue` hat eine Funktion `run`. Diese erwartet ein Objekt einer Klasse `DialogueLogic` als Argument. Die Klasse `DialogueLogic` hat dabei zwei Methoden:

- die Methode `description`, die einen String als Ergebnis zurückgibt. Mit diesem String soll der Knopf des GUI-Dialogs beschriftet sein.
- eine Methode `eval`. Diese enthält einen String als Argument und berechnet einen String als Ergebnis. Diese Methode soll im GUI-Dialog aus dem String des Eingabefeldes, den String errechnen, der nach Drücken des Knopfes im Ausgabefeld angezeigt werden soll.

DialogueLogic.hpp

```
1 #ifndef DIALOGUE_LOGIC_H_
2 #define DIALOGUE_LOGIC_H_
3 #include <string>
4 class DialogueLogic{
5  public:
6   virtual std::string eval(std::string str);
7   virtual std::string description();
8 };
9 #endif
```

Diese beiden Methoden sind denkbar einfach implementiert. Die Methode `eval` gibt gerade das Argument unverändert als Ergebnis zurück, und die Methode `description` gibt gerade einen String zurück, der dazu auffordert, den Knopf zu drücken.

DialogueLogic.cpp

```
1 #include "DialogueLogic.hpp"
2 std::string DialogueLogic::eval(std::string str){return str;}
3 std::string DialogueLogic::description(){return "push this button";}
```

Mit dieser Klasse läßt sich jetzt mit Hilfe der Funktion `run` die kleine GUI-Anwendung erzeugen, wie sie in Abbildung 2.5 zu sehen ist.

Hierzu wird ein Objekt der Klasse `DialogueLogic` erzeugt und dieses zusammen mit den Kommandozeilenparametern an die Funktion `Dialogue::run` übergeben.

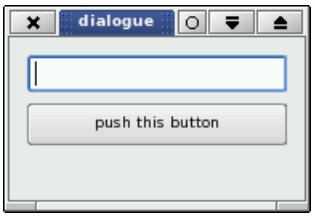

Abbildung 2.5: Eine kleine GUI-Anwendung.

TestDialogue.cpp

```
1 #include "Dialogue.hpp"
2 int main(int argc,char** argv){
3   return Dialogue::run(new DialogueLogic(), argc, argv);
4 }
```

Diese Anwendung zu Übersetzen benötigt leider ein wenig Know-How bezüglich der benutzen Bibliothek zur GUI-Programmierung. Die Klasse `Dialogue` ist mit Hilfe der GUI-Bibliothek *Qt* geschrieben. Die Übersetzung von *Qt*-Programmen beinhaltet, die Generierung von Klassen durch zur Entwicklungsumgebung zu *Qt* gehörende Werkzeuge. Um dem Entwickler diesen Prozess zu erleichtern enthält die Entwicklungsumgebung von *Qt* das Programm `qmake`. Mit diesem Programm kann ein Projekt angelegt werden und ein `Makefile` für dieses Projekt generiert werden.

Für unser Beispiel gehen wir in den Ordner: `name/panitz/gui/dialogue` in dem sich die Dateien: `Dialogue.hpp`, `Dialogue.cpp`, `DialogueLogic.hpp` , `DialogueLogic.hpp` und `TestDialogue.hpp` befinden. Dort rufen wir das Programm `qmake` mit der Option `-project` auf. Damit wird ein *Qt*-Projekt generiert. Durch den parameterlosen Aufruf des Programms `qmake` wird ein `Makefile` erzeugt, welches dann durch den Aufruf des Programms `make` dafür sorgt, dass das komplette Projekt übersetzt und gebaut wird. Wir erhalten also die folgende Session auf der Kommandozeile:

```
sep@pc305-3:~/fh/cpp/student/src> cd name/panitz/gui/dialogue
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> ls
Dialogue.cpp  DialogueLogic.cpp  TestDialogue.cpp
Dialogue.hpp  DialogueLogic.hpp
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> qmake -project
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> ls
Dialogue.cpp  DialogueLogic.cpp  dialogue.pro
Dialogue.hpp  DialogueLogic.hpp  TestDialogue.cpp
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> qmake
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> ls
Dialogue.cpp  DialogueLogic.cpp  dialogue.pro  TestDialogue.cpp
Dialogue.hpp  DialogueLogic.hpp  Makefile
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue> make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT  -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQ
T_SHARED -I/usr/local/Trolltech/Qt-4.2.2/mkspecs/linux-g++ -I. -I/usr/local/Trolltec
h/Qt-4.2.2/include/QtCore -I/usr/local/Trolltech/Qt-4.2.2/include/QtCore -I/usr/loca
l/Trolltech/Qt-4.2.2/include/QtGui -I/usr/local/Trolltech/Qt-4.2.2/include/QtGui -I/
usr/local/Trolltech/Qt-4.2.2/include -I. -I. -I. -o Dialogue.o Dialogue.cpp
g++ -c -pipe -O2 -Wall -W -D_REENTRANT  -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQ
T_SHARED -I/usr/local/Trolltech/Qt-4.2.2/mkspecs/linux-g++ -I. -I/usr/local/Trolltec
h/Qt-4.2.2/include/QtCore -I/usr/local/Trolltech/Qt-4.2.2/include/QtCore -I/usr/loca
l/Trolltech/Qt-4.2.2/include/QtGui -I/usr/local/Trolltech/Qt-4.2.2/include/QtGui -I/
usr/local/Trolltech/Qt-4.2.2/include -I. -I. -I. -o DialogueLogic.o DialogueLogic.cp
```

```
p
g++ -c -pipe -O2 -Wall -W -D_REENTRANT  -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQ
T_SHARED -I/usr/local/Trolltech/Qt-4.2.2/mkspecs/linux-g++ -I. -I/usr/local/Trolltec
 h/Qt-4.2.2/include/QtCore -I/usr/local/Trolltech/Qt-4.2.2/include/QtCore -I/usr/loc
al/Trolltech/Qt-4.2.2/include/QtGui -I/usr/local/Trolltech/Qt-4.2.2/include/QtGui -I
/usr/local/Trolltech/Qt-4.2.2/include -I. -I. -I. -o TestDialogue.o TestDialogue.cpp
/usr/local/Trolltech/Qt-4.2.2/bin/moc -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQT_
SHARED -I/usr/local/Trolltech/Qt-4.2.2/mkspecs/linux-g++ -I. -I/usr/local/Trolltech/
Qt-4.2.2/include/QtCore -I/usr/local/Trolltech/Qt-4.2.2/include/QtCore -I/usr/local/
Trolltech/Qt-4.2.2/include/QtGui -I/usr/local/Trolltech/Qt-4.2.2/include/QtGui -I/us
r/local/Trolltech/Qt-4.2.2/include -I. -I. -I. Dialogue.hpp -o moc_Dialogue.cpp
g++ -c -pipe -O2 -Wall -W -D_REENTRANT  -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQ
T_SHARED -I/usr/local/Trolltech/Qt-4.2.2/mkspecs/linux-g++ -I. -I/usr/local/Trolltec
h/Qt-4.2.2/include/QtCore -I/usr/local/Trolltech/Qt-4.2.2/include/QtCore -I/usr/loca
l/Trolltech/Qt-4.2.2/include/QtGui -I/usr/local/Trolltech/Qt-4.2.2/include/QtGui -I/
usr/local/Trolltech/Qt-4.2.2/include -I. -I. -I. -o moc_Dialogue.o moc_Dialogue.cpp
g++ -Wl,-rpath,/usr/local/Trolltech/Qt-4.2.2/lib -o dialogue Dialogue.o DialogueLogi
c.o TestDialogue.o moc_Dialogue.o    -L/usr/local/Trolltech/Qt-4.2.2/lib -lQtGui -L/
home/sep/software/qt-x11-opensource-src-4.2.2/lib -L/usr/X11R6/lib -lpng -lSM -lICE
-L/usr/local/lib -lXi -lXrender -lXrandr -lXcursor -lXinerama -lfreetype -lfontconfi
g -lXext -lX11 -lQtCore -lz -lm -lglib-2.0 -ldl -lpthread
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue>ls
dialogue           DialogueLogic.hpp  Makefile          TestDialogue.o
Dialogue.cpp       DialogueLogic.o    moc_Dialogue.cpp
Dialogue.hpp       Dialogue.o         moc_Dialogue.o
DialogueLogic.cpp  dialogue.pro       TestDialogue.cpp
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue>
```

Wie man sieht sind recht komplexe Aufrufe an den Compiler notwendig, um ein *Qt*-Programm übersetzt zu bekommen und man kann froh sein, dass diese Augfabe komplett vom Programm `qmake` übernommen wird.

Doch unser eigentliches Thema ist die späte Bindung. Was hat jetzt die späte Bindung mit dieser kleinen GUI-Anwendung zu tun?

Hierzu überlegen wir uns, dass wir jetzt eine GUI-Anwendung entwickeln möchten, die einen anderen String auf den Knopf hat und auf dem Ausgabefeld unterhalb des Knopfes nicht wieder genau den eingegebenen String anzeigt, sondern einen neuen String, der zum Beispiel aus dem Eingabestring entsteht, indem die Reihenfolge der Buchstaben gerade umgedreht werden. Hierzu schreiben wir jetzt eine Unterklasse der Klasse `DialogueLogic`:

MyDialogueLogic.hpp

```
1 #include "../dialogue/DialogueLogic.hpp"
2 class MyDialogueLogic:public DialogueLogic{
3  public:
4   virtual std::string eval(std::string str);
5   virtual std::string description();
6 };
```

Die Methode `description` sei in dieser Unterklasse so überschrieben, dass sie einen anderen String als Ergebnis zurück gibt:

MyDialogueLogic.cpp

```
1 #include "MyDialogueLogic.hpp"
2 #include <iostream>
3 std::string MyDialogueLogic::description(){return "reverse input";}
```

Die Methode `eval` soll so implementiert werden, dass der übergebene String gerade umgedreht wird. Tatsächlich hat die Standard-String-Klasse eine entsprechende Methode, da wir aber noch wenig über diese wissen, werden wir diese auch nicht benutzen, sondern quasi von Hand den String umdrehen. Hierzu benutzen wir den String wie eine Reihung, also genauso wie wir auch die Strings in C, die als `char*` dargestellt waren, benutzt haben. Eine zusätzlich angenehme Eigenschaft ist, dass wir einen String nach seiner Länge fragen können.

Diese Überlegungen münden in die folgende (hoffentlich) nachvollziehbare Methode, zum umdrehen der Buchstaben in einem String.

MyDialogueLogic.cpp

```
4  std::string MyDialogueLogic::eval(std::string str){
5    int strLength=str.length();
6    for (int i=0;i<strLength/2;i++){
7      char tmp=str[i];
8      str[i]=str[strLength-i-1];
9      str[strLength-i-1]=tmp;
10   }
11   return str;
12 }
```

Mit der Klasse `MyDialogueLogic` haben wir jetzt eine Unterklasse der Klasse `DialogueLogic` geschrieben. Jedes Objekt der Klasse `MyDialogueLogic` ist auch ein Objekt der Klasse `DialogueLogic`. Überall, wo ein Objekt der Klasse `DialogueLogic` erwartet wird, können wir jetzt auch ein Objekt der Klasse `MyDialogueLogic` benutzen. Damit können wir jetzt einen Dialog mit einem Objekt der neuen Unterklasse starten:

TestDialogue2.cpp

```
1  #include "../dialogue/Dialogue.hpp"
2  #include "MyDialogueLogic.hpp"
3  int main(int argc,char** argv){
4    return Dialogue::run(new MyDialogueLogic(), argc, argv);
5  }
```

Und jetzt schlägt tatsächlich die späte Bindung zu. Wird das neue Programm gestartet, dann erscheint nach Drücken des Knopfes unterhalb des Knopfes der String der oberhalb des Knopfes eingegeben hat in umgekehrter Reihenfolge. Das ist zunächst einmal doch erstaunlich, denn als die Klasse `Dialogue` entwickelt wurde, die ja immerhin spezifiziert, was passieren soll, wenn der Knopf gedrückt wird, existierte die Klasse `MyDialogueLogic` noch gar nicht. Der Programmierer der Klasse `Dialogue` konnte somit keinen direkten Aufruf von Methoden aus der Klasse `MyDialogueLogic` programmieren. Trotzdem wird diese in der Anwendung aufgerufen; und genau das ist durch die späte Bindung geschehen. In der Klasse `Dialogue` steht der Aufruf der Methode `eval`. Dort ist aber nicht fest verdrahtet, dass die Methode `eval` der Klasse `DialogueLogic` auszuführen ist. Vielmehr ist der dortige Aufruf zu lesen als:

Du bist doch ein Objekt, dass alles kann, was ein Objekt der Klasse `DialogueLogic` *kann. Daher hast du doch sicher eine Methode* `eval`*. Diese Methode führe doch bitte aus.*

Man spricht von später Bindung, weil eigentlich erst zur Laufzeit das Objekt gefragt wird, welche Methode `eval` es denn genau enthält und nicht bereits während der Übersetzungszeit direkt ein Sprung zur entsprechenden Methode vorgenommen wird.

Das Beispiel illustriert, wie über die späte Bindung unabhängige Komponenten einer Software entstehen können: der Programmierer der Unterklassen von `DialogueLogic` braucht nichts über die Programmierung von graphischen Oberflächen zu wissen. Er oder sie kann sich vollkommen auf die Logik, wie die Ausgabe abhängig von der Eingabe programmiert wird, konzentrieren. Der Kollege, der den GUI-Part entwickelt, braucht sich keine Gedanken zu machen, wie genau die Methode `eval` überschrieben wurde.

Beide Entwickler haben sich lediglich über die Klasse `DialogueLogic` als gemeinsame Schnittstelle verständigt.

Da der GUI-Part der Anwenung vollkommen getrennt von der Anwendungslogik ist, läßt sich der GUI-Part auch durch eine andere Implementierung austauschen, auch durch eine Implementierung, die eine vollkommen andere GUI-Bibliothek benutzt.

Deaktivieren der späten Bindung Die Programmiersprachen C und C++ haben als ein wichtiges Entwurfskriterium, dem Programmierer möglichst genaue Kontrolle über das Verhalten der Programme zu geben und vor allen Dingen ihm alle denkbare Einflußnahme auf die Performanz seiner Programme zu geben. Späte Bindung kostet ein wenig Zeit während der Laufzeit, nämlich die Zeit, in der das Objekt gefragt wird, wo denn der Code zu seiner speziellen Variante der aufgerufenen Methode zu finden ist. Um diesen kleinen Aufwand zu sparen, kann der Programmierer die späte Bindung für Mehtoden ausschalten. Dieses geschieht dadurch, dass die Methode in der in Frage kommenden Klasse nicht mit dem Attribut `virtual` markiert ist.

Überzeugen wir uns anhand eines minimalen Beispiels. Gegeben sei eine kleine Hierarchie aus zwei Klassen, der Oberklasse `Upper` und der Unterklasse `Lower`:

NonVirtual.hpp

```
1  #include <string>
2  class Upper{
3  public:
4    virtual std::string m1();
5    std::string m2();
6  };
7
8  class Lower:public Upper{
9  public:
10   virtual std::string m1();
11   std::string m2();
12 };
```

Eine der beiden Methoden ist *virtual*, also soll für sie die späte Bindung funktionieren, für die andere hingegen nicht.

Die Methoden sind so implementiert, dass sie jeweils einen String zurückgeben, der angibt, welche Methode aus welcher Klasse aufgerufen wurde:

NonVirtual.cpp

```
1  #include "NonVirtual.hpp"
2  #include <iostream>
3  std::string Upper::m1(){return "Upper::m1()";}
4  std::string Upper::m2(){return "Upper::m2()";}
5  std::string Lower::m1(){return "Lower::m1()";}
6  std::string Lower::m2(){return "Lower::m2()";}
```

Nun seien die Methoden einmal aufgerufen, und zwar zunächst einmal nur, indem sie als Objekte der Klasse `Upper` betrachtet werden. Und dann auch einmal ein Objekt der Klasse `Lower`, das auch über eine Variablen diesen Typs angesprochen wird.

NonVirtual.cpp

```
7  int main(){
8    Upper* u = new Upper();
9    Upper* l = new Lower();
10   std::cout << "u->m1() " << u->m1() << std::endl;
11   std::cout << "u->m2() " << u->m2() << std::endl;
12   std::cout << "l->m1() " << l->m1() << std::endl;
13   std::cout << "l->m2() " << l->m2() << std::endl;
14
15   Lower* ll = new Lower();
16   std::cout << "ll->m1() " << ll->m1() << std::endl;
17   std::cout << "ll->m2() " << ll->m2() << std::endl;
18   return 0;
19 }
```

Die Ausgabe gibt sehr genau darüber Aufschluß, welche Methoden in welcher Konstellation welche Implementierungen aufgerufen werden:

```
sep@pc305-3:~/fh/cpp/student> bin/NonVirtual
u->m1() Upper::m1()
u->m2() Upper::m2()
l->m1() Lower::m1()
l->m2() Upper::m2()
ll->m1() Lower::m1()
ll->m2() Lower::m2()
sep@pc305-3:~/fh/cpp/student>
```

Interessant sind dabei die dritte im Vergleich zur vierten Ausgabe. In der dritten Ausgabe sieht man, dass, obwohl wir die Methode `m1` auf ein Objekt, das wir über eine Variable des Typs `Upper` ansprechen, trotzdem die Methode `m1` der Klasse `Lower` ausgeführt wird. Dieses ist gerade das Prinzip, der späten Bindung der als virtuell markierten Methoden. In Falle der nicht virtuellen Methode `m2` funktioniert dieses nicht. Hier wird nach dem Typ der Variablen entschieden, welche Methode auszuführen ist. Dieses kann schon früh festgelegt werden, nämlich bereits statisch vom Compiler beim Generieren des Maschinencodes.

2.1.3 Standardkonstruktor und Standardaufruf der Super-Initialisierung

Aufmerksame Leser werden erstaunt festgestellt haben, dass in den Klassen `DialogueLogic` und `MyDialogueLogic` gar keine Konstruktoren definiert wurden. Trotzdem ließen sich mit `new` Objekte dieser Klassen erzeugen.

Da von einer Klasse ohne Konstruktor niemals ein Objekt erzeugt werden kann, und damit eine solche Klasse also unbrauchbar wäre, hat C++ folgende pragmatische Regel: für Klassen, in denen kein Konstruktor definiert wird, generiert der Compiler automatisch einen Konstruktor ohne Parameter. Dieser parameterlose Konstruktor wird als Standardkonstruktor bezeichnet.

Aber der C++-Compiler generiert noch mehr in Bezug auf Konstruktoren.

Beispiel 2.1.2 *Hierzu betrachte man folgendes kleines fehlerhaftes Programm:*

UpperLowerError.hpp

```
1  class Upper{
2  public:
3    int i;
4    Upper(int i);
5  };
6
7  class Lower:public Upper{};
```

Es gibt zwei Klassen, eine Oberklasse und eine Unterklasse, für die keine weiteren Eigenschaften definiert wurden. Lediglich der Konstruktor der Oberklasse ist zu implementieren:

UpperLowerError.cpp

```
1  #include"UpperLowerError.hpp"
2  Upper::Upper(int i){this->i=i;}
```

Die Unterklasse, da für sie kein Konstruktor entwickelt wurde, hat automatisch den Standardkonstruktor ohne Parameter. Die Übersetzung dieses Programms führt allerdings zu einen Fehler:

```
sep@pc305-3:~/fh/cpp/student/src> g++ UpperLowerError.cpp
In file included from UpperLowerError.cpp:1:
UpperLowerError.hpp:7: error: base ‘Upper’ with only non-default constructor in
   class without a constructor
sep@pc305-3:~/fh/cpp/student/src>
```

Die Fehlermeldung ist schon relativ brauchbar. Er berschwert sich über eine Unterklasse ohne Konstruktor im Falle einer Oberklasse, die nicht den Standardkonstruktor hat.

Beispiel 2.1.3 *Versuchen wir also den Konstruktor der Unterklase zu implementieren und erhalten das folgende immer noch fehlerhafte Programm:*

UpperLowerError2.cpp

```
1  class Upper{
2  public:
3    int i;
4    Upper(int i);
5  };
6
7  class Lower:public Upper{
8  public:
9    Lower();
10 };
11
12 Upper::Upper(int i){this->i=i;}
13 Lower::Lower(){}
```

Jetzt erhalten wir folgende Fehlermeldung:

```
ep@pc305-3:~/fh/cpp/student/src> g++ UpperLowerError2.cpp
UpperLowerError2.cpp: In constructor 'Lower::Lower()':
UpperLowerError2.cpp:13: error: no matching function for call to 'Upper::Upper(
   )'
UpperLowerError2.cpp:1: error: candidates are: Upper::Upper(const Upper&)
UpperLowerError2.cpp:12: error:                Upper::Upper(int)
sep@pc305-3:~/fh/cpp/student/src>
```

Hier beschwert sich der Compiler darüber, dass innerhalb des Konstruktors der Klasse `Lower` der parameterlose Konstruktor der Klasse `Upper` aufgerufen wird, und ein solcher nicht existiert. Skeptiker mögen einwerfen, dass wir einen Aufruf eines parameterlosen Konstruktors der Klasse `Upper` gar nicht programmiert haben. Nein, wir vielleicht nicht, aber jemand anderes, nämlich der Compiler. Man erinnere sich daran, dass als allererstes in einem Konstruktor die Konstruktoren der Oberklasse aufgerufen werden müssen. Das haben wir aber im obigen Beispiel unterlassen. Wenn im Programmtext der Aufruf eines Konstruktors der Oberklasse nicht vorhanden ist, so generiert der Compiler einen solchen in den Code hinein. Und da er es besser nicht wissen kann, wird dieser Aufruf ohne Parameter vorgenommen. Anschließend beschwert sich der Kompiler, dass er hier den Aufruf eines Konstruktors hineingeneriert hat, der gar nicht existiert.

Wenn es den Standardkonstruktor der Oberklasse nicht gibt, so müssen wir unbedingt den Aufruf des Konstruktors der Oberklasse explizit programmieren.

Beispiel 2.1.4 *Unser fehlerhaftes Beispiel kann somit korrigiert werden zu:*

UpperLowerOK.cpp

```
1  class Upper{
2  public:
3    int i;
4    Upper(int i);
5  };
6
7  class Lower:public Upper{
8  public:
9  Lower();
10 };
11
12 Upper::Upper(int i){this->i=i;}
13 Lower::Lower():Upper(42){}
```

2.1.4 Abstrakte Methoden

Im Beispiel der Klasse `Dialogue`, in dem wir einen der Vorteile des Prinzips der späten Bindung kennengelernt haben, da war die Klasse `DialogueLogic` etwas künstlich. Tatsächlich hat die Methode `eavl` rein gar nichts berechnet, sondern gerade den Parameter wieder zurück gegeben. Tatsächlich wird niemand Interesse an einer Anwendung haben, in der ein Dialog für ein Objekt der Klasse `DalogueLogic` zu bauen, sondern sind lediglich Applikationen mit interessanten Unterklassen, in denen die Methode `eval` mit einer komplexen Berechnung überschrieben ist und die Methode `description` eine aussagekräftige Beschriftung für den Knopf zurückgibt. Der einzige Sinn der Klasse `DialogueLogic` bestand darin, zu spezifizieren, welche beiden Methoden

eine sinnvolle Unterklasse implementieren soll, damit die Klasse `Dialogue` sinnvoll mit dieser arbeiten kann.

Für diese Situation gibt es die Möglichkeit eine Methode als abstrakt zu definieren. Eine abstrakte Methode hat keine Implementierung, es gibt sie eigentlich noch gar nicht. Sie ist lediglich das Versprechen, dass sie in einer Unterklasse implementiert werden wird. So ließe sich sinnvoller Weise die Methoden `eval` und `description` erst einmal als abstrakte Methoden spezifizieren. In C++ geschieht dieses, indem in der Kopfdatei nach der Methodensignatur folgende Zeichen stehen: `=0`.

Somit sähe eine Version mit abstrakten Methoden der Klasse `DialogueLogic` wie folgt aus:

AbstractDialogueLogic.hpp

```
1 #ifndef ABSTRACT_DIALOGUE_LOGIC_H_
2 #define ABSTRACT_DIALOGUE_LOGIC_H_
3 #include <string>
4 class AbstractDialogueLogic{
5  public:
6   virtual std::string eval(std::string str)=0;
7   virtual std::string description()=0;
8 };
9 #endif
```

Da abstrakte Methoden keine Implementierung haben und die Klasse `AbstractDialogueLogic` nur abstrakte Methoden enthält, braucht es keine Implementierungsdatei `AbstractDialogueLogic.cpp`. Klassen, die nur abstrakte Methoden enthalten, werden auch als Schnittstelle oder englisch als `interface` bezeichnet.

Da abstrakten Klassen noch die Implementierungen der abstrakten Methoden fehlen, können keine Objekte abstrakter Klassen erzeugt werden.

Beispiel 2.1.5 *Folgendes Beispiel zeigt, dass wir dabei eine Fehlermeldung vom Compiler enthalten.*

AbstractDialogueLogicError.cpp

```
1 #include "AbstractDialogueLogic.hpp"
2 int main(){
3   AbstractDialogueLogic* logic=new AbstractDialogueLogic();
4   return 0;
5 }
```

Der Compiler gibt folgende Fehlermeldung:

```
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue3> g++ -c AbstractDialogueLogicError.cpp
AbstractDialogueLogicError.cpp: In function ‘int main()’:
AbstractDialogueLogicError.cpp:3: error: cannot allocate an object of type ‘
   AbstractDialogueLogic’
AbstractDialogueLogicError.cpp:3: error:   because the following virtual
   functions are abstract:
AbstractDialogueLogic.hpp:7: error:     virtual std::string
   AbstractDialogueLogic::description()
AbstractDialogueLogic.hpp:6: error:     virtual std::string
   AbstractDialogueLogic::eval(std::basic_string<char, std::char_traits<char>,
   std::allocator<char> >)
sep@pc305-3:~/fh/cpp/student/src/name/panitz/gui/dialogue3>
```

Wir können nur Objekte von Klassen erzeugen, die keine abstrakten Methoden mehr enthalten. Also z.B. von einer Unterklasse der Klasse `AbstractDialogueLogic`, in der für alle geerbten abstrakten Methoden, konkrete Implementierungen existieren.

MyLogic.hpp

```
1  #include "AbstractDialogueLogic.hpp"
2  class MyLogic:public AbstractDialogueLogic{
3   public:
4    virtual std::string eval(std::string str);
5    virtual std::string description();
6  };
```

Wir benutzen für diese Implementierung die gleiche Umsetzung, wie schon in der Klasse `MyDialogueLogic`:

MyLogic.cpp

```
1  #include "MyLogic.hpp"
2  #include <iostream>
3  std::string MyLogic::description(){return "reverse input";}
4  std::string MyLogic::eval(std::string str){
5    int strLength=str.length();
6    for (int i=0;i<strLength/2;i++){
7      char tmp=str[i];
8      str[i]=str[strLength-i-1];
9      str[strLength-i-1]=tmp;
10   }
11   return str;
12 }
```

Von dieser Klasse läßt sich jetzt ein Objekt erzeugen und die Methoden der Klasse auf diesem Objekt ausführen.

MyLogic.cpp

```
13 int main(){
14   AbstractDialogueLogic* logic = new MyLogic();
15   std::cout << logic->description() << std::endl;
16   std::cout << logic->eval("123456789") << std::endl;
17   return 0;
18 }
```

Abstrakte Methoden müssen immer virtuell sein, d.h. die späte Bindung muss für sie immer wirksam sein. Das ergibt sich logisch aus der Idee der abstrakten Methoden. Sie sollen in einer Unterklasse erst wirksam implementiert werden. Es muss daher sichergestellt werden, dass ein Objekt durch späte Bindung nach seiner konkreten Implementierung einer abstrakten Methode gefragt wird. Tatsächlich weist der Compiler bereits Klassen zurück, in denen abstrakte Methoden nicht mit dem Attribut `virtual` markiert sind.

Beispiel 2.1.6 *Provozieren wir mit einem kleinen Beispiel einmal die Fehlermeldung des Compilers, für nicht als virtuell markierte abstrakte Methoden.*

AbstractError.hpp

```
1 class AbstractError{
2   public:  void f()=0;
3 };
```

Bereits das Inkludieren dieser Klasse führt zu einer Fehlermeldung:

AbstractError.cpp

```
1 #include "AbstractError.hpp"
2 int main(){
3   return 0;
4 };
```

Der Compiler gibt die folgende Fehlermeldung aus:

```
sep@pc305-3:~/fh/cpp/student/src> g++ -c AbstractError.cpp
In file included from AbstractError.cpp:1:
AbstractError.hpp:3: error: initializer specified for non-virtual method ‘void
   AbstractError::f()’
sep@pc305-3:~/fh/cpp/student/src>
```

Konstruktoren abstrakter Klassen

Im Hinblick auf Konstruktoren gibt es keinen Unterschied zwischen abstrakten und konkreten Klassen. Das mag vielleicht zunächst verwundern, weil ja von abstrakten Klassen direkt kein neues Objekt erzeugt werden kann, sondern nur von Unterklassen, die alle abstrakten Eigenschaften implementieren. Warum also haben abstrakte Klassen auch einen Konstruktor?

Die Antwort liegt in dem Aufruf des Konstruktors der Superklasse. Der Konstruktor einer abstrakten Klasse wird nie durch Erzeugen eines Objekts der abstrakten Klasse mit `new` aufgerufen, denn das ist sinnvoller Weise verboten, aber in den Konstruktoren der konkreten Unterklassen einer abstrakten Klasse, wird der Konstruktor der abstrakten Klasse aufgerufen.

Beispiel 2.1.7 *Auch hierzu ein kleines Beispiel. Wir definieren eine abstrakte Klasse mit einem eigenen Konstruktor:*

AbstractUpper.hpp

```
1 class AbstractUpper {
2 public:
3   int i;
4   AbstractUpper(int i);
5   virtual void print()=0;
6 };
```

Desweiteren eine konkrete Klasse, die von dieser abstrakten Klasse ableitet:

AbstractUpper.hpp

```
7 class Lower:public AbstractUpper{
8 public:
9   int i2;
```

```
10    Lower(int i, int i2);
11    virtual void print();
12  };
```

Die Implementierung des Konstruktors der abstrakten Klasse sorgt dafür, dass das Feld `i` *initialisiert wird:*

AbstractUpper.cpp

```
1  #include "AbstractUpper.hpp"
2  #include <iostream>
3
4  AbstractUpper::AbstractUpper(int i){this->i=i;}
```

Im Konstruktor der konkreten Klasse wird zunächst, so wie es Pflicht ist, der Konstrukor der abstrakten Oberklasse aufgerufen, bevor weitere Initialisierungen vorgenommen werden:

AbstractUpper.cpp

```
5  Lower::Lower(int i,int i2):AbstractUpper(i){this->i2=i2;}
```

Schließlich vervollständigen wir das Programm mit der Implementierung der Methode `print` *und einer kleinen Hauptfunktion.*

AbstractUpper.cpp

```
6  void Lower::print(){std::cout<<this->i<<" "<<this->i2<<std::endl;}
7
8  int main(){
9    AbstractUpper* au=new Lower(17,42);
10   au->print();
11   return 0;
12 }
```

abstrakte und konkrete Methoden

Bisher haben wir in abstrakten Klassen immer nur abstrakte Methoden gehabt. in abstrakten Klassen können aber beides sein, abstrakte und konkrete Methoden. Das Spannende ist, dass in den konkreten Methoden die abstrakten Methoden aufgerufen werden können. Das mag vielleicht zunächst widersprüchlich klingen, denn die abstrakten Methoden haben in der abstrakten Klasse noch keine Implementierung, wie können sie dann auch in der abstrakten Klasse bereits aufgerufen werden? Das Phänomen erklärt sich, wenn man sich vergegenwärtigt, dass auch die konkreten Methoden nur für ein Objekt aufgerufen werden können, das von einer konkreten Unterklasse der abstrakten Klasse ist. Und dieses Objekt hat dann auch für alle abstrakten Methoden Implementierungen.

Beispiel 2.1.8 *Auch hierzu ein minimales Beispiel:*

CallAbstractMethode.hpp

```
1  #include <string>
2  class CallAbstractMethode{
3  public:
```

```
4   virtual std::string description()=0;
5   virtual void print();
6 };
7
8 class Lower:public CallAbstractMethode{
9 public:
10  virtual std::string description();
11 };
```

In der Oberklasse ist die Methode `description` *abstrakt. Trotzdem wollen wir sie bereits in der Methode* `print` *der abstrakten Klasse aufrufen:*

CallAbstractMethode.cpp

```
1 #include "CallAbstractMethode.hpp"
2 #include <iostream>
3 void CallAbstractMethode::print(){
4   std::cout<<"Description: "<<this->description()<<std::endl;
5 }
```

Erst in der konkreten Unterklasse, wird die Methode `description` *implementiert. Diese Implementierung wird letztendlich in der Methode* `print` *über die späte Bindung aufgerufen.*

CallAbstractMethode.cpp

```
6 std::string Lower::description(){return "Lower()";}
7
8 int main(){
9   CallAbstractMethode* am = new Lower();
10  am->print();
11  return 0;
12 }
```

Aufgabe 2 Sie sollen auf diesem Übungsblatt die Aufgaben des ersten Übungsblattes weiterführen und die dort programmierten Klassen benutzen und gegebenenfalls verändern. Allerdings soll die C-Version der Lösung des ersten Übungsblattes nicht weiter verfolgt werden.

a) Machen Sie `GeometricObject` zu einer abstrakten Klassen und fügen ihr eine abstrakte Methode zur Berechnung des Flächeninhalts zu. Benutzen Sie diese abstrakte Methoden in der Methode `print`.

b) Implementieren Sie eine konkrete Unterklasse `Rectangle` der Klasse `GeometricObject`, die Rechtecke darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie die Methoden `print`.

c) Implementieren Sie eine Unterklasse `Square` der Klasse `Rectangle`, die Quadrate darstellt. Schreiben Sie geeignete Konstruktoren.

d) Implementieren Sie eine konkrete Unterklasse `Oval` der Klasse `GeometricObject`, die Ellipsen darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie die Methoden `print` und `hasWithin`.

e) Implementieren Sie eine Unterklasse `Circle` der Klasse `Oval`, die Kreise darstellt. Schreiben Sie geeignete Konstruktoren.

f) Implementieren Sie eine konkrete Unterklasse `EquilateralTriangle` der Klasse `GeometricObject`, die gleichseitige Dreiecke darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie die Methoden `print`.

g) Schreiben Sie Tests, in denen von jeder Klasse Objekte erzeugt und benutzt werden.

2.1.5 Löschen von Objekten

In C++ sind wir leider genausowenig wie in C davon befreit, einmal angeforderten Speicherplatz auch wieder freizugeben. Anders als in C, wo wir Speicherplatz mit der Funktion `malloc` vom Betriebssystem angefordert haben, wird in C++ Speicherplatz über das Schlüsselwort `new` direkt für ein Objekt einer Klasse angefordert und gleich auch initialisiert. Ebenso wie in C gilt, dass jeder mit `malloc` angeforderte Speicherplatz mit einem `free` nach Benutzung wieder freizusetzen ist, so ist jeder in C++ mit `new` angeforderte Speicherplatz wieder freizugeben. Dieses geschieht in C++ mit dem Befehl `delete`.

Beispiel 2.1.9 *Überzeugen wir uns einmal von der Wirkungsweise der* `delete`*-Operation. Hierzu nehmen wir eine minimale Klasse ohne Eigenschaften.*

DeleteMe.hpp

```
1 class DeleteMe{};
```

In einem ersten Programm wird eine große Anzahl Objekte dieser Klasse erzeugt:

NotDeleted.cpp

```
1 #include "DeleteMe.hpp"
2 #include <iostream>
3 int main(){
4   for (long i=0;i<10000000;i++){
5     DeleteMe* dm=new DeleteMe();
6     std::cout<<"*";
7   }
8   return 0;
9 }
```

In einem zweiten Programm werden die Objekte sofort wieder gelöscht:

Deleted.cpp

```
1  #include "DeleteMe.hpp"
2  #include <iostream>
3  int main(){
4    for (long i=0;i<10000000;i++){
5      DeleteMe* dm=new DeleteMe();
6      std::cout<<"*";
7      delete dm;
8    }
9    return 0;
10 }
```

Betrachtet man den Resourcenverbrauch der beiden Programme während der Laufzeit, so sieht man, wie das erste Programm mehrere hundert Megabyte Speicher beanspruchen, während das zweite Programm mit einem konstanten Speicherbedarf von wenigen Bytes abläuft.

Destruktoren

Auch im ersten Semester haben wir schon ein Beispiel gesehen, dass manchmal wenn ein Objekt aus dem Speicher gelöscht werden soll, gleich weitere Objekte au dem Speicher zu löschen sind. So geschehen bei den einfach verketteten Listen. Wir haben eine Funktion geschrieben, die dafür sorgt, dass beim Löschen einer Listenzelle auch alle von dieser Listenzelle erreichbaren weiteren Listenzellen gelöscht werden. Auch hierfür ist in C++ eine eigene Notation vorgesehen. In einer Klasse kann auch ein sogenannter Destruktor definiert werden. Der Destruktor hat als Bezeichner den Klassennamen mit vorangestellten Tildesymbol `~`.

Beispiel 2.1.10 *Beispiel einer minimalen Klasse mit Destruktor und Standardkonstruktor.*

Destructor.hpp

```
1  class Destructor{
2  public:
3    Destructor();
4    virtual ~Destructor();
5  };
```

Konstruktor und Destruktor seien beide so definiert, dass sie beide eine Meldung auf der Kommandozeile vornehmen:

Destructor.cpp

```
1  #include "Destructor.hpp"
2  #include <iostream>
3  Destructor::Destructor(){
4    std::cout<<"objekt erzeugt"<<std::endl;
5  }
6  Destructor::~Destructor(){
7    std::cout<<"objekt geloescht"<<std::endl;
8  }
```

Mit Hilfe folgender testfunktion kann man sich davon vergewissern, dass Konstrukor und Destruktor entsprechend ihrer Implementierung ausgeführt werden.

Destructor.cpp

```
9  int main(){
10   for (long i=0;i<10;i++){
11     Destructor* dm=new Destructor();
12     delete dm;
13   }
14   return 0;
15 }
```

Es läßt sich also über den Destruktor Code programmieren, der ausgeführt wird, wenn ein Objekt aus dem Speicher gelöscht wird. Häufig sollen in diesem Fall auch andere Objekte im Speicher entfernt werden. Betrachten wir wieder eine minimale Klasse, mit der einfach verkettete Listen dargestellt werden können, ähnlich wie sie schon im ersten Semester programmiert wurden:

Li1.hpp

```
1  #define NULL 0
2  class Li1{
3  public:
4    int head;
5    Li1* tail;
6    Li1(int head,Li1* tail);
7    virtual bool isEmpty();
8    virtual void print();
9    virtual ~Li1();
10 };
```

Nebenbei sehen wir einen kleinen Unterschied zu C. In C war `NULL` in der Bibliothek `stdlib.h` definiert als ein Wert vom Typ `void*`. In C++ ist dies aufgrund des Typsystems nicht ohne weiteres möglich. Hier wird tatsächlich `NULL` als sie einfache Zahl `0` verwendet. Die Adresse `0` ist keine gültige Adresse um Speicherbereich.

Doch nun zu unserer altbekannten Implementierung der einfach verketteten Listen. Zunächst sei der Konstruktor implementiert, der die Felder `head` und `tail` initialisiert.

Li1.cpp

```
1  #include "Li1.hpp"
2  #include "iostream"
3  Li1::Li1(int head,Li1* tail){
4    this->head=head;
5    this->tail=tail;
6  }
```

Eine Liste wird als leer angesehen, wenn ihr Feld `tail` den Nullzeiger hat.

Li1.cpp

```
7  bool Li1::isEmpty(){
8    return this->tail==NULL;
9  }
```

Schließlich gelangen wir zum eigentlichen Destruktor. In ihm wird der Code implementiert, der aufgerufen werden soll, bevor das Objekt entgültig aus dem Speicher entfernt wird. In unserem sorgen wir dafür, dass, sofern es sich nicht um die leere Listenzelle, die das Ende einer Liste anzeigt handelt, der `tail` der Liste aus dem Speicher entfernt wird. Zusätzlich machen wir noch eine Meldung auf die Kommandozeile.

Li1.cpp

```
10 Li1::~Li1(){
11   if (!this->isEmpty()) delete this->tail;
12   std::cout<<"tail entfernt"<<std::endl;
13 }
```

Auch für die Klasse `Li1` haben wir eine Methode `print` vorgesehen. Sie ist in ähnlicher Weise implementiert, wie wir es aus dem ersten Semester kennen.

Li1.cpp

```
14 void Li1::print(){
15   if (this->isEmpty()) std::cout<<"[]"<<std::endl;
16   else{
17     std::cout<<"["<<this->head;
18     for (Li1* tmp=this->tail;!tmp->isEmpty();tmp=tmp->tail){
19       std::cout<<","<<tmp->head;
20     }
21     std::cout<<"]"<<std::endl;
22   }
23 }
```

Und schließlich ein minimaler Test, in dem eine Liste erzeugt, ausgegeben und aus dem Speicher entfernt wird.

Li1.cpp

```
24 int main(){
25   Li1* xs
26    = new Li1(17,new Li1(4,new Li1(42,new Li1(0,NULL))));
27
28   xs->print();
29   delete xs;
30   return 0;
31 }
```

Die Ausgabe bezeugt, dass tatsächlich durch einen einzigen Aufruf der `delete`-Operation alle vier Listenzellen gelöscht wurden.

```
sep@pc305-3:~/fh/cpp/student> bin/Li1
[17,4,42]
tail entfernt
tail entfernt
tail entfernt
tail entfernt
sep@pc305-3:~/fh/cpp/student>
```

Einen Destruktor sollte man immer als virtuell markieren.

2.1.6 statische Klasseneigenschaften

statische Felder

Es gibt die Situation, dass Informationen nicht an ein bestimmtes Objekt gebunden ist, sondern global für alle Objekte einer Klasse gelten. In diesem ließe sich eine solche Information in einer ganz herkömmlichen C Variablen speichern. Wie wir schon im ersten Semester betont haben, sind globale Variablen mit Vorsicht zu genießen.

In C++ gibt es die Möglichkeit innerhalb einer Klasse ein Feld zu deklarieren, so dass in diesem Feld nicht Informationen gespeichert werden, die jedes Objekt der Klasse für sich eigenständig hat, sondern eine Information, die alle Objekte der Klasse gemeinsam haben. Ein solches Feld wird mit dem Attribut `static` versehen. Es wird als statisches Feld bezeichnet.

Hierzu eine kleine Beispielklasse, in der es ein statisches Feld `sum` und ein Objektfeld `myNumber` gibt. Beide sind vom Typ `int`.

StaticField.hpp

```
1  class StaticField{
2  public:
3    static int sum;
4    int myNumber;
5    StaticField();
6    ~StaticField();
7    virtual void print();
8  };
```

In der entsprechenden Implementierungsdatei ist zunächst für ein statisches Fled eine Initialisierung vorzunehmen. In unserem Beispiel soll das Feld `sum` die Gesamtanzahl der Objekte der Klasse beschreiben. Wir initialisieren es mit `0`.

StaticField.cpp

```
9  #include "StaticField.hpp"
10 #include <iostream>
11 int StaticField::sum=0;
```

Das Feld `sum` beinhaltet als statisches Feld eine für alle Objekte der Klasse `StaticField` globale Information. Da es die Gesamtzahl aller erzeuten Objekte der Klasse ist, wird diese Anzahl im Konstruktor für ein konkretes Objekt um eins erhöht. Desweiteren nehmen wir das Ergebnis, um es als Nummer für das neu erzeugte Objekt zu benutzen.

StaticField.cpp

```
12 StaticField::StaticField(){
13   sum=sum+1;
14   this->myNumber=sum;
15 }
```

Folgerichtig erniedriegen wir das globale Feld im Destruktor wieder um eins.

StaticField.cpp

```
16 StaticField::~StaticField(){
17   sum=sum-1;
18 }
```

Für eine einfache Handhabung der Klasse, sei eine Methode `print` geschrieben.

StaticField.cpp

```
19 void StaticField::print(){
20   std::cout<<"StaticField("<<myNumber<<") total = "<<sum<<std::endl;
21 }
```

Ein einfacher Test zeigt, wie sich statische und Objektfelder unterschiedliche Verhalten. Egal, welches Objekt der Klasse ausgegeben wird, es ist dabei zu einem bestimmten Zeitpunkt immer der gleiche Wert für das statische Feld zu sehen.

StaticField.cpp

```
22 int main(){
23   StaticField* x1 = new StaticField();
24   x1->print();
25   StaticField* x2 = new StaticField();
26   x1->print();
27   x2->print();
28   StaticField* x3 = new StaticField();
29   x1->print();
30   x2->print();
31   x3->print();
32   delete x2;
33   x1->print();
34   x3->print();
35   delete x3;
36   x1->print();
37   delete x1;
38
39   return 0;
40 }
```

Zugriff auf statische Felder Tatsächlich ist die Information eines statischen Feldes nicht an ein bestimmtes Objekt gebunden, sondern existiert global für eine bestimmte Klasse. Daher sollte ein statisches Feld in der Regel auch nicht über ein Objekt zugegriffen werden, sondern über die Klasse. Dieses geschieht, indem mit einem dopppelten Doppelpunkt getrennt der name der Klasse vorangeschrieben wird. So sollte auf das statische Feld im letzten Beispiel über `StaticField::sum` zugegriffen werden.

statische Methoden

In C++ können natürlich in gleicher Weise wie in C globale Funktionen geschrieben werden, die von überall im Programm aufgerufen werden können. Eine solche C Funktion hat hat kein Objekt, auf die sie sich bezieht und erhält im besten Fall alle ihre Informationen über die Parameter.

C++ bietet eine elegantere Lösung solche Funktionen zumindest organisatorisch an eine Klasse anzugliedern. Hierzu ist die Funktion als eine statische Methode einer Klasse zu deklarieren. Man fügt solche Funktionen sinnvoller Weise einer Klasse als statische Methode hinzu, mit der die Funktion eine gewissen inhaltlichen Zusammenhang hat. Auch wie bei statischen Feldern ist eine statische Methode durch das Attribut `static` gekennzeichnet. Auch wie bei statischen Feldern erfolgt der Zugriff auf eine statische Methode über den mit einem doppelten Doppelpunkt vorangestellten Klassennamen.

StaticMethod.hpp

```
1 class StaticMethod{
2  public:
```

```
3   int n;
4   StaticMethod(int n);
5   void addOne();
6   static StaticMethod* addition(int x,int y);
7 };
```

StaticMethod.cpp

```
1  #include "StaticMethod.hpp"
2  StaticMethod::StaticMethod(int n){this->n=n;}
3  void StaticMethod::addOne(){n=n+1;}
4  static StaticMethod* addition(int x,int y){
5    return new StaticMethod(x+y);
6  }
7
8  int main(){
9    StatStaticMethod sm = StaticMethod::addition(17,4);
10   std::cout<<sm->n<<std::endl;
11   return 0;
12 }
```

Vier Einschränkungen liegen für statische Methoden vor, die sich aus Ihrer Natur, dass sie nicht an ein bestimmtes Objekt gebunden sind, ergibt.

statische Methoden kennen nur statische Eigenschaften Daten die in nicht-statischen Feldern einer Klasse liegen, können in einer statischen Methode nicht benutzt werden. Somit führt folgende Klasse zu einem Compilerfehler:

StaticMethodError1.hpp

```
1  class StaticMethodError1{
2   public:
3    int n;
4    StaticMethodError1(int n);
5    void addOne();
6    static StaticMethodError1* addition(int x,int y);
7  };
```

StaticMethodError1.cpp

```
1  #include "StaticMethodError1.hpp"
2  StaticMethodError1::StaticMethodError1(int n){this->n=n;}
3  void StaticMethodError1::addOne(){n=n+1;}
4  static StaticMethodError1* addition(int x,int y){
5    return new StaticMethodError1(x+y+n);
6  }
```

Der Compiler gibt die folgende Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/student/src> g++ -c StaticMethodError1.cpp
StaticMethodError1.cpp: In function ‘StaticMethodError1* addition(int, int)’:
StaticMethodError1.cpp:5: error: ‘n’ undeclared (first use this function)
StaticMethodError1.cpp:5: error: (Each undeclared identifier is reported only
   once for each function it appears in.)
sep@pc305-3:~/fh/cpp/student/src>
```

statische Methoden kennen kein this Aus diesem Grund gibt es auch keinen `this`-Zeiger in statischen Methoden: Somit führt auch folgende Klasse zu einem Compilerfehler:

StaticMethodError2.hpp

```
1 class StaticMethodError2{
2  public:
3   int n;
4   StaticMethodError2(int n);
5   void addOne();
6   static StaticMethodError2* addition(int x,int y);
7 };
```

StaticMethodError2.cpp

```
1 #include "StaticMethodError2.hpp"
2 StaticMethodError2::StaticMethodError2(int n){this->n=n;}
3 void StaticMethodError2::addOne(){n=n+1;}
4 static StaticMethodError2* addition(int x,int y){
5   return new StaticMethodError2(x+y+this->n);
6 }
```

Der Compiler gibt die folgende Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/student/src> g++ -c StaticMethodError2.cpp
StaticMethodError2.cpp: In function ‘StaticMethodError2* addition(int, int)’:
StaticMethodError2.cpp:5: error: invalid use of ‘this’ in non-member function
sep@pc305-3:~/fh/cpp/student/src>
```

statische Methoden sind nie virtuell Statische Methoden haben nichts mit Objektorientierung oder Vererbung zu tun. Sie können insbesondere nicht in Unterklassen überschrieben werden. Daher hat es auch keinen Sinn statische Methoden virtuell zu deklarieren.

Somit führt auch folgende Klasse zu einem Compilerfehler:

StaticMethodError3.hpp

```
1 class StaticMethodError3{
2  public:
3   int n;
4   StaticMethodError3(int n);
5   void addOne();
6   virtual static StaticMethodError3* addition(int x,int y);
7 };
```

StaticMethodError3.cpp

```
1 #include "StaticMethodError3.hpp"
2 StaticMethodError3::StaticMethodError3(int n){this->n=n;}
3 void StaticMethodError3::addOne(){n=n+1;}
4 static StaticMethodError3* addition(int x,int y){
5   return new StaticMethodError3(x+y);
6 }
```

Der Compiler gibt die folgende Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/student/src> g++ -c StaticMethodError3.cpp
In file included from StaticMethodError3.cpp:1:
StaticMethodError3.hpp:6: error: member ‘addition’ cannot be declared both
   virtual and static
sep@pc305-3:~/fh/cpp/student/src>
```

statische Methoden sind nie abstrakt Aus dem gleichen Grund ist es auch nicht möglich statische Methoden als abstrakt zu deklarieren. Statische Methoden stehen außerhalb der Vererbungshierarchie und können damit auch nicht in Unterklassen implementiert werden.

Somit führt auch folgende Klasse zu einem Compilerfehler:

StaticMethodError4.hpp

```
1 class StaticMethodError4{
2  public:
3   int n;
4   StaticMethodError4(int n);
5   void addOne();
6   virtual static StaticMethodError4* addition(int x,int y);
7 };
```

StaticMethodError4.cpp

```
1 #include "StaticMethodError4.hpp"
2 StaticMethodError4::StaticMethodError4(int n){this->n=n;}
3 void StaticMethodError4::addOne(){n=n+1;}
4 static StaticMethodError4* addition(int x,int y){
5   return new StaticMethodError4(x+y);
6 }
```

Der Compiler gibt die folgende Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/student/src> g++ -c StaticMethodError4.cpp
In file included from StaticMethodError4.cpp:1:
StaticMethodError4.hpp:6: error: member ‘addition’ cannot be declared both
   virtual and static
sep@pc305-3:~/fh/cpp/student/src>
```

2.1.7 Zusammenfassung

2.2 Spezielle Konzepte der Objektorientierung in C++

Im vorherigen Kapitel wurden die Grundzüge der objektorientierten Programmierung aufgezeigt, wie sie im wesentlichen in fast allen objektorientierten Programmiersprachen vorzufinden sind. Mit den vorgestellten Konzepten sollte es leicht möglich sein, auf andere objektorientierte Sprachen, z.B. Java, umzusteigen. In diesem Kapitel werden weitere Konzepte der objektorientierten Prorammierung in C++ vorgestellt, die eher als C++-spezifisch anzusehen sind.

2.2.1 Arbeiten mit Strings

Zum Arbeiten mit Zeichenketten bieten alle gängigen Programmiersprachen eine Standardbibliothek an. Auch in C gab es rudimentäre Unterstützung für Strings, auch wenn diese nur als Zeiger auf einen zusammenhängender Speicherbereich dargestellt waren.

In C++ gibt es sie Standardbibliothek `string`, die den Typ `std::string`, den wir bereits benutzt haben, zur Verfügung stellt. Allerdings ist diese Bibliothek in vielen Bereichen nicht sehr konfortabel. So gibt es viele Bibliohteken, die mit einer eigenen Implementierung zur Verarbeitung mit Zeichenketten aufwarten. So gibt es in der GUI-Bibliothek *Qt* eine eigene Implementierung für Zeichenketten, ebenso in der Bibliothek *xerces* zur Verarbeitung von XML-Dateien, oder mit *Gtkmm* in einer weiteren GUI-Bibliothek. Benutzt man verschiedene Bibliotheken in einer Applikation, so kann es eine sehr grundlegende Entscheidung sein, auf welche Stringrepräsentation letztendlich zurückgegriffen wird.

Vorerst sollen hier die wichtigsten Eigenschaften der Standardimplementierung für Zeichenketten vorgestellt werden.

Tatsächlich braucht die Bibliothek `string` nicht explizit inkludiert zu werden, wenn auch mit der Bibliothek `iostream` gearbeitet wird. Die Bibliothek `iostream` inkludiert ihrerseits `string`.

StringTest.cpp
```
1 #include <iostream>
```

Wie schon in vielen Beispielen gesehen, lassen sich Objekte des Typs `std::string` durch in Anführungszeichen eingeschlossene Zeichenketten erzeugen:

StringTest.cpp
```
2 int main(){
3   std::string hallo = "hallo";
```

Derartige Stringobjekte lassen sich mit dem Operator `<<` auf die Kommandozeile ausgeben:

StringTest.cpp
```
4   std::cout<<hallo<<std::endl;
```

Zur Verknüpfung zweier Stringobjekte kann der Operator `+` benutzt werden. Allerdings verändert diese Operation nicht das Stringobjekt, sondern erzeugt einen neuen String:

StringTest.cpp
```
5  hallo+" welt";
6  std::cout<<hallo<<std::endl;
```

Erst wenn dieser neue String wieder in einer Variablen gespeichert wird, kann das ergebnis dieser Stringaddation auch effektiv beobachtet werden:

StringTest.cpp
```
7  hallo=hallo+" welt";
8  std::cout<<hallo<<std::endl;
```

Stringobjekte verhalten sich ebenso wie Reihungen. Mit der eckigen Klammernotation kann ein Index im String angegeben werden. Damit lassen sich einzelne Zeichen einer Zeichenkette ansprechen und verändern.

StringTest.cpp
```
9  hallo[1]='e';
10 hallo[7]='o';
11 hallo[8]='r';
12 hallo[9]='l';
13 hallo=hallo+'d';
14 std::cout<<hallo<<std::endl;
```

Die +-Operation hat einen String nicht verändert, sondern einen neuen String erzeugt. Es existiert eine Methode `append`, die an einen bestehenden String einen weiteren String anhängt. Anders als die +-Operation verändert die Methode `append` das Stringobjekt, für das sie aufgerufen wird.

StringTest.cpp
```
15 hallo.append("!");
16 std::cout<<hallo<<std::endl;
```

Anders als in C, können Stringobjekte nach ihrer Länge befragt werden. Tatsächlich gibt es hierzu gleich zwei Methoden: die Methode `size` und die Methode `length`. Beide Methoden sind funktionsgleich.

StringTest.cpp
```
17 std::cout<<hallo.length()<<std::endl;
18 std::cout<<hallo.size()<<std::endl;
```

Für ein Stringobjekt existiert die Möglichkeit einen aus C bekannten nullterminierten Speicherbereich zu bekommen. Dieser ist mit dem Schlüsselwort `const` als unveränderbar zu kennzeichnen:

StringTest.cpp
```
19 const char* cHallo = hallo.c_str();
20 for (int i=0;cHallo[i]!='\0';i++)
21   std::cout<<cHallo[i];
22 std::cout<<std::endl;
```

Soweit die Grundfunktionalität die für Stringobjekte zur Verfügung steht.

StringTest.cpp

```
23   return 0;
24 }
```

primitive Typen in Stringdarstellung

Leider bietet die Standardimplementierung der Zeichenketten in C++ keine praktische Funktionalität an, um Zahlen als Stringobjekte darzustellen. Hierzu können aber recht einfach ein paar kleine Hilfsfunktionen geschrieben werden.

Hierzu seien vier recht nützliche Funktionen definiert:

PrimitiveAsString.hpp

```
1  #ifndef PRIMITIVE_AS_STRING_H
2  #define PRIMITIVE_AS_STRING_H
3
4  #include <string>
5  std::string intToString(int i);
6  std::string doubleToString(double i);
7
8  std::string addInt(std::string str,int i);
9  std::string addDouble(std::string str,double i);
10 #endif
```

Die benötigte Funktionalität befindet sich in der Bibliothek `sstream`, die den Typ `std::stringstream` bereitstellt. Dieser bietet in gleicher Weise den Operator `<<`, den wir schon eifrig benutzt haben, um Daten primitiver Typen auf die Konsole auszugeben. Die Vier Funktionen lassen sich somit wie folgt implementieren:

PrimitiveAsString.cpp

```
1  #include "PrimitiveAsString.hpp"
2  #include <sstream>
3
4  std::string intToString(int i){
5    std::stringstream s;
6    s << i;
7    return s.str();
8  }
9
10 std::string doubleToString(double i){
11   std::stringstream s;
12   s << i;
13   return s.str();
14 }
15
16 std::string addInt(std::string str,int i){
17   return str+intToString(i);
18 }
19
```

```
20 std::string addDouble(std::string str,double i){
21   return str+doubleToString(i);
22 }
```

Im folgenden Test ist zu sehen, wie nun Stringobjekte auch um Zahlendarstellungen erweitert werden können.

TestPrimitiveAsString.cpp

```
1  #include "PrimitiveAsString.hpp"
2  #include <iostream>
3  int main(){
4    std::string hallo="hallo";
5    std::cout<<hallo<<std::endl;
6    hallo=addInt(hallo,42);
7    std::cout<<hallo<<std::endl;
8    hallo=addDouble(hallo,42.4242);
9    std::cout<<hallo<<std::endl;
10
11   return 0;
12 }
```

eine Klasse ToString

ToString.hpp

```
1  #ifndef TO_STRING_H
2  #define TO_STRING_H
3  #include <string>
4  #include "PrimitiveAsString.hpp"
5
6  class ToString{
7  public:
8    virtual std::string toString()=0;
9    virtual void print();
10   virtual void println();
11 };
12 #endif
```

ToString.cpp

```
1  #include <iostream>
2  #include "ToString.hpp"
3  void ToString::print(){
4    std::cout<<toString();
5  }
6  void ToString::println(){
7    print();
8    std::cout<<std::endl;
9  }
```

2.2.2 Mehrfaches Erben

Bisher haben wir nur immer genau eine Oberklasse gehabt. Darauf sind wir in C++ nicht beschränkt. Wir können durchaus mehrere Oberklassen haben. Dann vereinigen wir Eigenschaften aus mehreren Klassen in einer Klasse.

MultipleParents.cpp

```
#include <iostream>
#include "name/panitz/gui/dialogue3/AbstractDialogueLogic.hpp"
#include "ToString.hpp"

class MyToStringButtonLogic:public AbstractDialogueLogic,public ToString{
 public:
  std::string description(){return "nun nicht mehr abstrakt";}
  std::string eval(std::string x){return x.replace (3,0," HUCH");}
  std::string toString(){return "Blödsinnige GUI Anwendung";}
};

int main(){
  MyToStringButtonLogic* bL = new MyToStringButtonLogic();
  std::cout<<bL->description()<<" "
           <<bL->eval("Na, was ist das")<<" "
           <<bL->toString()<<" "<<std::endl;
}
```

Besonders in Zusammenhang mit Klassen, die nur abstrakte Methoden bereitstellen, hat mehrfaches Erben[3] Sinn. Wir können allerdings ein Problem mit mehrfachen Erben bekommen. Wenn in beiden Oberklassen ein und dieselbe Methode konkret existiert, dann weiß man nicht, welche dieser zwei Methoden mit gleicher Signatur denn zu erben ist:

VaterMuterKindError.cpp

```
#include <string>
#include <iostream>
class Vater{public:std::string tuDas(){return "Vater sagt dies!";}};
class Mutter{public:std::string tuDas(){return"Mutter sagt das!";}};
class Kind:public Vater, public Mutter{};

int main(){
 Kind* kind = new Kind();
 std::cout<<kind->tuDas()<<std::endl;
}
```

Bei der Übersetzung kann nicht aufgelöst, ab `tuDas` des Vaters oder der Mutter für `kind` auszuführen ist.

```
VaterMuterKind.cpp: In function ‘int main()’:
VaterMuterKind.cpp:9: error: request for member ‘tuDas’ is ambiguous
```

[3] In der Literatur wird meistens von mehrfacher Vererbung gesprochen. Dass eine Klasse mehrere Unterklassen hat, also seine Eigenschaften mehrfach vererbt, ist keine Besonderheit, hingegen, dass eine Klasse mehrere Oberklassen, aus denen sie mehrfach erbt, schon.

```
VaterMuterKind.cpp:4: error: candidates are: std::string Mutter::tuDas()
VaterMuterKind.cpp:3: error:                 std::string Vater::tuDas()
VaterMuterKind.cpp: In function ‘int main()’:
VaterMuterKind.cpp:9: error: request for member ‘tuDas’ is ambiguous
VaterMuterKind.cpp:4: error: candidates are: std::string Mutter::tuDas()
VaterMuterKind.cpp:3: error:                 std::string Vater::tuDas()
```

Da es diese Methode zweimal gibt, müssen wir explizit angeben, welche der beiden geerbten Methoden denn auszuführen ist:

VaterMuterKind.cpp

```
1  #include <string>
2  #include <iostream>
3  class Vater{public:std::string tuDas(){return "Vater sagt dies!";}};
4  class Mutter{public:std::string tuDas(){return"Mutter sagt das!";}};
5  class Kind:public Vater, public Mutter{};
6
7  int main(){
8   Kind* kind = new Kind();
9   std::cout<<kind->Vater::tuDas()<<std::endl;
10  std::cout<<kind->Mutter::tuDas()<<std::endl;
11 }
```

So lassen sich beide Methoden aufrufen:

```
sep@pc216-5:~/fh/cpp/student> bin/VaterMuterKind
Vater sagt dies!
Mutter sagt das!
sep@pc216-5:~/fh/cpp/student>
```

Aus der verschiedenen Gründen hat man in Java darauf verzichtet, mehrfache Erbung zuzulassen. Dafür werden allerdings Klassen, die nur abstrakte Methoden enthalten gesondert behandelt. Von ihnen darf auch in Java mehrfach geerbt werden, denn sie Vererben ja keine Implementierunng, sondern nur Schnittstellenbeschreibungen.

Aufgabe 3 Machen Sie jetzt Ihre Klasse `GeometricObject` zu einer Unterklasse der Klasse `ToString` aus dem Skript. Implementieren Sie möglichst aussagekräftig die Methode `toString` in den einzelnen Unterklassen.

Aufgabe 4 Fügen Sie jetzt folgende Schnittstelle Ihrem Projekt hinzu:

Paintable.hpp

```
1  #ifndef PAINTABLE_H_
2  #define PAINTABLE_H_
3  #include <QPainter>
4
5  class Paintable{
6  public:
7    virtual void paintMe(QPainter* p)=0;
8  };
```

```
9
10 #endif
```

Diese Klasse, wie die übrigen dieser Aufgabe, ist auch im Netz herunterzuladen:
`http://panitz.name/cpp/student/src/name/panitz/paintable/Paintable.hpp`

Ihre Klasse `GeometricObject` soll jetzt zusätzlich die Schnittstelle `Paintable` erweitern. Hierzu muss die Methode `paintMe` in den entsprechenden Unterklassen implementiert werden. Hier soll sich mit dem `QPainter`-Objekt die geometrische Figur zeichnen. Eine Dokumentation zu `QPainter` finden Sie auf:
`http://doc.trolltech.com/4.2/qpainter.html`

Jetzt sollten Sie in der Lage sein, einzelne geometrische Figuren in einem Fenster graphisch mit der nachfolgenden Klasse in einem Fenster anzuzeigen.

Board.hpp

```
1  #ifndef BOARD´ #define BOARD´
2  #include <QWidget>
3  #include "Paintable.hpp"
4
5  class Board:public QWidget {
6   public:
7    Board(Paintable* pt);
8    Paintable* pt;
9    QSize minimumSizeHint();
10   QSize sizeHint() ;
11   void paintEvent(QPaintEvent *event);
12   static int showPaintable(Paintable* pt,int argc,char **argv);
13 };
14 #endif /*BOARD´/
```

Board.cpp

```
1  #include "Board.hpp"
2  #include <QtGui>
3
4  Board::Board(Paintable* pt): QWidget(0){
5    this->pt=pt;
6    setBackgroundRole(QPalette::Base);
7  }
8
9  QSize Board::minimumSizeHint(){return QSize(100, 100);}
10 QSize Board::sizeHint(){return QSize(400, 300);}
11
12 void Board::paintEvent(QPaintEvent *){
13   QPen pen;
14   QPainter painter(this);
15   painter.setPen(pen);
16   pt->paintMe(&painter);
17 }
18
```

```
19 int Board::showPaintable(Paintable* pt,int argc,char **argv){
20   QApplication application(argc,argv);
21   Board* b = new Board(pt);
22   b->show();
23   return application.exec();
24 }
```

Hierzu brauchen Sie eine `main`-Funktion der folgenden Art nur die statische Methde `showPaintable` Aufrufen:

```
1 int main(int argc,char **argv){
2   return Board::showPaintable
3    (new EquilateralTriangle(new Vertex(300,100),50),argc,argv);
4 }
```

Übersetzen Sie das so entstandene Qt-Programm am besten über die Kommandozeile mit `qmake -project`, `qmake` und `make`.

Es ist allerdings mit ein wenig Geschick auch möglich mit einem *standard-make-Projekt* in Eclipse Qt-Anwendungen zu kompilieren.

Aufgabe 5 Nehmen Sie jetzt die Klasse `Li1` aus dem Skript, die eine einfache verkettete Liste ganzer Zahlen modelliert.

a) Ändern Sie die Klasse so ab, dass Sie nicht `int`-Werte als Elemente hat, sondern Zeiger auf `Paintable`-Objekte. Nennen Sie diese geänderte Klasse `PaintableList`.

b) Dann ändern Sie die Klasse Board so ab, dass jetzt nicht ein einzelnes `Paintable`-Objekt darin abgespeichert ist, sondern ein `PaintableList`-Objekt.

c) In der Methode `Board::paintEvent` soll jetzt nicht nur eine geometrische Figur gezeichnet werden, sondern alle in der Liste gespeicherten Objekte.

d) Fügen Sie in `Board` eine Methode `addPaintable(Paintable*)` hinzu.

e) Schreiben Sie einen Destruktor für die Klasse `Board`, die auch die Liste komplett aus dem Speicher löscht.

f) Stellen Sie in einem Fenster verschiedene geometrische Figuren dar.

2.2.3 Initialisierungslisten

C++ kennt eine spezielle Syntax, die beim Schreiben von Konstruktoren zum Initialisieren von Objekten benutzt werden kann, die sogenannten Initialisierungslisten. Initialisierungslisten stehen genau da, wo auch die Aufrufe der Konstruktoren der Oberklassen stehen: zwischen Parameterliste und Konstruktorrumpf. Sie bestehen aus den Namen des Feldes, dem in runden Klammern der Wert, mit dem das Feld initialisiert werden soll folgt:

Initialize.cpp

```
#include <iostream>
class Initialize{
 public:
  int x;
  int y;
  int z;
  Initialize(int x,int yP):x(x),y(yP),z(2*(x+yP)){}
};

int main(){
  Initialize init(17,4);
  std::cout<<init.x<<" "<<init.y<<" "<<init.z<<std::endl;
  return 0;
}
```

Bei der Benutzung von Initialisierungslisten kommt es sehr schnell dazu, dass der Rumpf eines Konstruktors leer bleibt. Die meisten Konstruktoren initialisieren lediglich die Felder einer Klasse.

Initialsierungslisten sind nicht eine einfache syntaktische Variante, einen Konstruktor zu schreiben, sondern in bestimmten Fällen die einzige Möglichkeit ein Feld zu initialisieren. Dieses ist der Fall, wenn das Feld mit dem Attribut `const` markiert ist. Das Attribut `const` besagt, dass dem Feld nur zur Initialisierung ein Wert zugewiesen werden darf. Weitere Zuweisungsoperationen auf das Feld sind nicht erlaubt. Das bedeutet, sogar im Konstruktor darf diesem Feld kein Wert mehr zugewiesen werden:

ConstInitError.cpp

```
class ConstInitError{
  public:
   const int x;
   ConstInitError(int x){this->x=x;}
};
```

Dieses Programm kann nicht erfolgreich übersetzt werden. Der Compiler beschwert sich mit folgender Fehlermeldung über die Zuweisung zu einem Feld mit dem Attribut `const`.

```
sep@pc305-3:~/fh/cpp/student> g++ src/ConstInitError.cpp
src/ConstInitError.cpp: In constructor ‘ConstInitError::ConstInitError(int)’:
src/ConstInitError.cpp:4: error: assignment of read-only data-member ‘
   ConstInitError::x’
sep@pc305-3:~/fh/cpp/student>
```

Unter Benutzung der Initialisierungsliste kann dieses Feld hingegen im Konstruktor initialisiert werden:

ConstInitOk.cpp

```
class ConstInitOk{
  public:
   const int x;
   ConstInitOk(int x):x(x){}
};
```

2.2.4 Objekte als Werte

Bisher wurden Objekte in unseren Programmen nur über einen Zeiger angesprochen. Die Objekte wurden mit dem Operator `new` erzeugt. Damit wurde direkt Speicher für das Objekt allokiert und der Zeiger auf diesen Speicherbereich als Referenz auf das neue Objekt zurückgegeben. Nur so können die eigentlichen Konzepte der Ojektorientierung, insbesondere die Späte-Bindung, aber auch abstrakte Methoden, benutzt werden.

Klassen in C++ sind im Prinzip eine Erweiterung der Strukte in C. Und so bietet C++ auch die Möglichkeit Objekte direkt als Werte zu benutzen. Damit verliert man allerdings die meisten Konzeote der Objektorientierung. In vielen objektorientierte Sprache, wie z.B. in Java gibt es keine Möglichkeit Objekte direkt als Werte zu benutzen.

Auch in den bisherigen Beispielen haben wir schon Objekte einer Klasse als Werte ohne Zeiger benutzt. Das waren die Objekte der Klasse `std::string`.

Schreiben wir nun eine einfache Klasse, deren Objekte direkt als Werte benutzt werden sollen. Hierzu bedienen wir uns wieder einer Klasse, die Punkte im zweidimensionalen Raum darstellt:

P2D.hpp

```
1 class P2D{
2 public:
3   double x;
4   double y;
5   P2D(double x, double y);
6 };
```

Die klasse ist bewusst klein gehalten. Es braucht nur der Konstruktor implementiert zu werden

P2D.cpp

```
1 #include "P2D.hpp"
2 P2D::P2D(double x, double y):x(x),y(y){}
```

Default-Konstruktor

Versuchen wir zunächst einmal nun eine lokale Variable vom Typ `P2D` zu deklarieren, ohne dass das Objekt über einen Zeiger angesprochen wird:

P2DUseError.cpp

```
1 #include "P2D.hpp"
2 int main(){
3   P2D p;
4   return 0;
5 }
```

Tatsächlich führt dieses Programm zu einem Fehler:

```
sep@pc305-3:~/fh/cpp/student> g++ src/P2DUseError.cpp
src/P2DUseError.cpp: In function ‘int main()’:
src/P2DUseError.cpp:3: error: no matching function for call to ‘P2D::P2D()’
src/P2D.hpp:1: error: candidates are: P2D::P2D(const P2D&)
src/P2D.hpp:5: error:                 P2D::P2D(double, double)
sep@pc305-3:~/fh/cpp/student>
```

Der Grund für diesen Fehler liegt darin, dass sobald eine Variable eines Werttyps für eine Klasse deklariert wird, ein solches Objekt zu erzeugen ist. Im obigen Beispiel ist also mit der Deklaration der Variablen `p` auch ein Objekt der Klasse `P2D` zu konstruieren. Hierzu benötigt der Compiler einen Konstruktor der Klasse. Da keine Argumente angegeben wurden, mit denen das Objekt zu initialisieren ist, wird der default-Konstruktor aufgerufen, der keine Patameter aut. Dieser existiert in der Klasse `P2D` nicht. Es existiert lediglich ein Konstruktor mit zwei Parametern. Für diese Parameter sind konkrete Argumente direkt bei der Deklaration der Variablen anzugeben:

P2DUse.cpp

```
1  #include "P2D.hpp"
2  #include <iostream>
3  int main(){
4    P2D p(17,4);
5    std::cout<<p.x<<std::endl;
6    return 0;
7  }
```

Jetzt übersetzt das Programm ohne Fehler und es wird ein Objekt mit den Werten `17` und `4` für die Felder `x` und `y` erzeugt. In schon aus C gewohnter Weise kann über den Punktoperator direkt auf die Felder des Objekts zugegriffen werden.

Kopier-Konstruktor

Objekte als Werte verhalten sich in C++ genauso, wie Strukturobjekte als Wert in C. Bei jeder Parameterübergabe, Ergebnisrückgabe und Initialisierung einer neuen Variablen werden diese Objekte kopiert. Dieses läßt sich am folgenden Beispiel analog zum Beispiel aus der C-Vorlesung illustrieren:

P2DCopyArgument.cpp

```
1  #include "P2D.hpp"
2  #include <iostream>
3
4  void skalarMult(P2D p,double s){
5    p.x=s*p.x;
6    p.y=s*p.y;
7  }
8
9  int main(){
10   P2D p(17,4);
11   std::cout<<p.x<<std::endl;
12   skalarMult(p,4);
13   std::cout<<p.x<<std::endl;
14   return 0;
15 }
```

Der Aufruf der Funktion `skalarMult` verändert das übergeben Objekt nicht. Es wird beim Aufruf eine Kopie des Objekts angefertigt. Anschließen wird diese Kopie verändert und verworfen. Das Originalobjekt verworfen.

Anders als in C, gibt es eine Möglichkeit auf diesen Kopiervorgang Einfluß zu nehmen. Zum Kopieren des Objektes benutzt C++ einen speziellen Konstruktor. Dieser Konstruktor wird aufgerufen, wenn bei der Parameterübergabe per Wert, die Kopie des Objekts anzufertigen ist. Bisher haben wir diesen speziellen Kopierkonstruktor noch nicht geschrieben. Trotzdem gab es diesen bereits, wie wir auch schon aus einigen Fehlermeldungen ablesen konnten. In der Fehlermedlung der Übersetzung des Programms `P2DUseError` werden zwei Kandidaten für Konstruktoren angegeben. Der eine Kandidat `P2D::P2D(double, double` ist der Konstruktor, den wir explizit in der Klasse definiert haben. Der andere Kandidat `P2D::P2D(const P2D&)` ist der Kopierkonstruktor.

Wenn explizit kein Kopierkonstruktor geschrieben wird, so generiert der C++-Kompiler einen Kopierkonstruktor. Dieser hat in unserem Beispiel die Signatur: `P2D(const P2D&)`. Das Symbol `&` ist uns bisher noch nicht bei Parametern begegnet. Wir nehmen es vorerst als absolut notwendig beim Kopierkonstruktor hin.

Wir können diesen Kopierkonstruktor aber auch selbst schreiben:

P2DC.hpp

```
1  class P2DC{
2  public:
3    double x;
4    double y;
5    P2DC(double x, double y);
6    P2DC(const P2DC& copyMe);
7  };
```

Jetzt implementieren wir beide Konstruktoren, wobei wir den Kopierkonstruktor so implementieren, dass er nicht eine eins zu eins Kopie des Objekts anfertigt, sondern die Koordinaten in der Kopie verdoppelt.

P2DC.cpp

```
1  #include "P2DC.hpp"
2  P2DC::P2DC(double x, double y):x(x),y(y){}
3  P2DC::P2DC(const P2DC& copyMe):x(2*copyMe.x),y(2*copyMe.y){}
```

Das ist natürlich eine irreführende und exotische Art, eine Kopie anzufertigen. untersuchen wir jetzt einmal, wann Kopien mit diesem Konstruktor angefertigt werden.

Hierzu sei eine Funktion vorgesehen, die einen Objektwert als Parameter hat und eine weitere Funktion, die einen Objektwert als Rückgabe hat:

P2DCUse.cpp

```
1  #include "P2DC.hpp"
2  #include <iostream>
3
4  void print(P2DC p ){
5    std::cout<<"("<<p.x<<","<<p.y<<")"<<std::endl;
6  }
7
8  P2DC oneOne(){
9    P2DC result(1,1);
10   return result;
11 }
```

Als erster Test sei ein Punkt einmal direkt aufsgegeben und einmal, indem er der Funktion `print` als Parameter übergeben wird.

P2DCUse.cpp

```
12 int main(){
13   P2DC p(17,4);
14   std::cout<<"("<<p.x<<","<<p.y<<")"<<std::endl;
15   print(p);
```

Als nächstes sei einmal der Wert der Rückgabe der Funktion `oneOne` unter die Lupe genommen.

P2DCUse.cpp

```
16   std::cout<<"("<<oneOne().x<<","<<oneOne().y<<")"<<std::endl;
```

Schließlich interessiert, wie bei der Initialisierung einer neuen Variablen kopiert wird.

P2DCUse.cpp

```
17   P2DC p2(p);
18   std::cout<<"("<<p2.x<<","<<p2.y<<")"<<std::endl;
```

Und es interessiert, wie bei einer Zuweisung kopiert wird:

P2DCUse.cpp

```
19  p2=p;
20   std::cout<<"("<<p2.x<<","<<p2.y<<")"<<std::endl;
21
22   return 0;
23 }
```

Das Programm erzeugt folgende Aufgabe:

```
sep@pc305-3:~/fh/cpp/student/src> ./a.out
(17,4)
(34,8)
(1,1)
(34,8)
(17,4)
sep@pc305-3:~/fh/cpp/student/src>
```

Wie man sieht, wird bei der Parameterübergabe eine Kopie angefertigt. Bei der Rückgabe hingegen wird der Kopierkonstruktor nicht benutzt, hingegen wieder beim Initialisieren einer neuen Variablen. Nicht hingegen wird der Kopierkonstruktor bei der Zuweisung benutzt.

Werte und abstrakte Klassen

Von abstrakten Klassen können keine Objekte erzeugt werden. Daher läßt sich ein Konstruktor einer abstrakten Klasse nicht mit `new` aufrufen. Für Objektwerte hat das eine weitreichende Konsequenz: es kann gar kein Wertvariablen oder Parameter einer abstrakten Klasse geben. Diese sind immer über einen Zeiger anzusprechen:

NoAbstractObject.cpp

```
1  #include <iostream>
2  class A{
3   public:
4    virtual void doNothing()=0;
5  };
6
7  class B:public A{
8   public:
9    virtual void doNothing(){
10     std::cout<<"nothing"<<std::endl;
11   }
12 };
13
14 void f1(B b){b.doNothing();}
15 void f2(A a){b.doNothing();}
```

Die Kompilierung dieses Programms führt zu der folgnden Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/student/src> g++ NoAbstractObject.cpp
NoAbstractObject.cpp:15: error: cannot declare parameter ‘a’ to be of type ‘A’
NoAbstractObject.cpp:15: error:   because the following virtual functions are
   abstract:
NoAbstractObject.cpp:4: error:  virtual void A::doNothing()
sep@pc305-3:~/fh/cpp/student/src>
```

Objekte abstrakter Klassen können nur über Zeiger und Referenzen angesprochen werden.

Werte und Vererbung

Die zentralen Konzepte der objektorientierten Programmierung funktionieren nicht mehr, oder zumindest nur noch eingeschränkt, wenn Objekte als Wert betrachtet werden. Insbesondere die späte Bindung funktioniert nihct mehr.

Hierzu seien zwei Klassen definiert. Eine Oberklasse und eine Unterklasse, die eine virtuelle Methode überschreibt und ein zusätzliches Feld besitzt.

ValueAndInheritance.hpp

```
1  class Upper{
2  public:
3    int x;
4    Upper(int x);
5    virtual void doSomething();
6  };
7
8  class Lower:public Upper{
9  public:
10   int y;
11   Lower(int x,int y);
12   virtual void doSomething();
13 };
```

Die Methode soll kurz Auskunft darüber geben, aus welcher Klasse sie aufgerufen wurde. Dabei ruft die überschreibende Version die überschriebene version auf.

ValueAndInheritance.cpp

```
1  #include "ValueAndInheritance.hpp"
2  #include <iostream>
3
4  Upper::Upper(int x):x(x){};
5  Lower::Lower(int x,int y):Upper(x),y(y){};
6
7  void Upper::doSomething(){std::cout<<"upper class "<<x<<std::endl;}
8  void Lower::doSomething(){
9    Upper::doSomething();
10   std::cout<<"working class "<<y<<std::endl;
11 }
```

Jetzt seien je ein Wert dieser Klassen angelegt:

ValueAndInheritance.cpp

```
12 int main(){
13   Upper up(42);
14   Lower lo(17,4);
```

Für beide Werte wird die Methode `doSomething` aufgerufen.

ValueAndInheritance.cpp

```
15   up.doSomething();
16   lo.doSomething();
```

Jetzt werden die Werte der Unterklasse der Variablen vom Typ der Oberklasse einmal zugewiesen. Dabei erwarten wir natürlich eine Kopie der relevanten Werte:

ValueAndInheritance.cpp

```
17   up=lo;
```

Und schließlich sei geschaut, welche der beiden Versionen der Methode `doSomething` letztendlich aufgerufen wird.

ValueAndInheritance.cpp

```
18   up.doSomething();
19   return 0;
20 }
```

Das Programm erzeugt die folgende Ausgabe:

```
sep@pc305-3:~/fh/cpp> ./student/bin/ValueAndInheritance
upper class 42
upper class 17
working class 4
upper class 17
sep@pc305-3:~/fh/cpp>
```

Wie an der letzten Ausgabe zu sehen ist, findet keine späte Bindung statt. Die Zuweisung `up=lo` kopiert nur die Teile des Objektes `lo` der Unterklasse, die für ein Objekt der Klasse `Upper` notwendig sind. Das bedeutet nur den Wert des Feldes `x`. Der Wert des Feldes `y` wird bei dieser Kopie vernachlässigt. Im Speicherbereich für die Variable `up` wäre schließlich auch gar kein Platz, für diese zusätzliche nicht zur Klasse `Upper` gehörenden Information.

Explizite-Konstruktoren

Wir haben gesehen, dass C++ implizit den Kopierkonstruktor aufruft, wenn ein Feld initialisiert wird, oder ein Wertparameter übergeben wird. C++ ist dazu noch in der Lage ganz andere Konstruktoren implizit aufzurufen. Hierzu schreiben wir eine Klasse, mit zwei Konstruktoren.

Implicit.cpp

```
1  #include <iostream>
2  class Implicit{
3    public:
4     std::string s;
5     Implicit(std::string s):s(s){
6       std::cout<< s <<" war Konstruktorargument"<<std::endl;
7     };
8     Implicit(int i):s("hallo"){
9       std::cout<< i <<" war Konstruktorargument"<<std::endl;
10    };
11 };
```

Wir können in regulärer Weise ein Objekt dieser Klasse deklarieren und dabei den Konstruktor mit dem Stringargument benutzen. Wir können erstaunlicher Weise Werten dieser Klasse einen `int`-Wert zuweisen. In diesem Fall wird implizit der Konstruktor mit einem `int`-Parameter aufgerufen, um das Objekt neu zu initialisieren.

Implicit.cpp

```
12 int main(){
13   Implicit imp("witzelbritz");
14   imp = 42;
15   return 0;
16 }
```

Folgende Programmausgabe illustriert dieses:

```
sep@pc305-3:~/fh/cpp/student> bin/Implicit
witzelbritz war Konstruktorargument
42 war Konstruktorargument
sep@pc305-3:~/fh/cpp/student>
```

Wem das zuviel Gefuddel unter der Hand ist und als potentielle Fehlerquelle erscheint, kann die implizite Benutzung eines Konstruktors verbieten, indem dieser Konstruktor mit dem Attribut `explicit` gekennzeichnet wird.

ExplicitError.cpp

```
1  #include <iostream>
2  class Explicit{
3    public:
4     std::string s;
5     Explicit(std::string s):s(s){};
6     explicit Explicit(int i):s("hallo"){
7       std::cout<< i <<" war Konstruktorargument"<<std::endl;
8
9     };
10 };
```

Von nnun an wird die zuweiseung zwischen den zwei sehr unterschiedlichen Typen vom Compiler zurückgewiesen:

ExplicitError.cpp

```
11 int main(){
12   Explicit exp = 42;
13   return 0;
14 }
```

Die Übersetzung des Programms bricht mit folgender Fehlermeldung ab.

```
sep@pc305-3:~/fh/cpp/student> g++ src/ExplicitError.cpp
src/ExplicitError.cpp: In function ‘int main()’:
src/ExplicitError.cpp:13: error: conversion from ‘int’ to non-scalar type ‘
   Explicit’ requested
sep@pc305-3:~/fh/cpp/student>
```

Wie zu sehen war, gibt es zwei fundamental unterschiedliche Arten, wie mit Objekten in C++ gearbeitet werden kann. Einmal über verzeigerte Objekte, und einmal mit Objekten als Werten. Ersteres ist zu empfehlen, wenn eine komplexe objektorientiert entworfene Bibliothek entwickelt wird, in der Vererbung und späte Bindung eine große Rolle spielt. Letzteres ist zu empfehlen für kleine einfache Klassen, der Datenhaltung, wie Punkte im geometrischen Raum, oder auch Strings. Die Arbeit mit Werten hat auch einen Vorteil: es ist keine explizite Speicherverwaltung vom Programmierer notwendig. Es muss nicht explizit ein Destruktor aufgerufen werden.

Aufgabe 6 Scheiben Sie die Klasse `Star`, eine weitere Unterklasse von `GeometricObject`. Es sollen Sterne in dieser Klasse dargestellt werden. Ein Stern hat dabei einen äußeren Radius für die Spitzen der Strahlen und einen inneren Radius für die Zacken. Implementieren Sie für die Klasse `Star` die Methoden `toString`, `paintMe` und `area`.

Testen Sie die Darstellung von Sternen. Zeichnen Sie dabei auch Sterne, die zu einem Viereck und zu einem Kreis werden. Testen Sie, ob der Flächeninhalt eines Sterns, der einen Kreis annähert, dem Flächeninhalt aus der Klasse `Circle`, gleicht.

2.2.5 Referenztypen

Wir haben nun zwei sehr unterschiedliche Arten mit Objekten unmzugehen kennengelernt: als Werte und Verzeigert. Beides hat Vor- und Nachteile.

Zeiger sind ein sehr mächtiges aber auch sehr gefährliches Programmkonstrukt. Leicht kann man sich bei Zeigern vertun, und plötzlich in undefinierte Speicherbereiche hineinschauen. Zeiger sind also etwas, von dem man lieber die Finger lassen würde. Ein C++ Programm kommt aber kaum ohne Zeiger aus. Um dynamisch wachsende Strukturen, wie z.B. Listen zu modellieren, braucht man Zeiger. Um nicht Werte sondern Referenzen an Funktionen zu übergeben, benötigt man Zeiger.

Beim Arbeiten mit Objektwerten auf der anderen Seite muss auf die wichtigsten Konzeote der Objektorientierung verzichtet werden. Zusätzlich werden Daten permanent kopiert.

Daher hat man sich in C++ entschlossen einen weiteren Typ einzuführen, der die Vorteile von Zeigern hat, aber weniger gefährlich in seiner Anwendung ist und nicht zu Kopien der Werte führt: den Referenztyp.

Deklaration von Referenztypen

Für einen Typ, der kein Referenztyp ist, wird der Referenztyp gebildet, indem dem Typnamen das Ampersandzeichen `&` nachgestellt wird. Achtung, in diesen Fall ist das Ampersandzeichen kein Operator, sondern ein Bestandteil des Typnamens[4]. So gibt es z.B. für den Typ `int` den Referenztyp `int &` oder für den Typ `string` den Referenztyp `string &`. Es gibt keine Referenztypen von Referenztypen: `int &&` ist nicht erlaubt. Anders als bei Zeigern, bei denen Zeiger auf Zeigervariablen erlaubt waren.

Initialisierung

Eine Referenzvariable benötigt bei der Deklaration einen initialen Wert. Andernfalls kommt es zu einen Übersetzungsfehler:

NoInitRef.cpp

```
1 int &i;
```

Der Übersetzer gibt folgende Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ -c NoInitRef.cpp
NoInitRef.cpp:1: error: ‘i’ declared as reference but not initialized
sep@linux:~/fh/prog3/examples/src>
```

Referenzvariablen lassen sich mit Variablen des Typs, auf den referenziert werden soll, initialisieren:

InitRef.cpp

```
1 int i;
2 int &j=i;
```

Referenzvariablen lassen sich nicht mit Konstanten oder dem Ergebnis eines beliebigen Ausdrucks initialisieren:

[4]Ebenso wie das Zeichen `*` sowohl der Dereferenzierungsoperator für Zeiger als auch Bestandteil des Typnamens für Zeiger war.

WrongInitRef.cpp

```
1 int i;
2 int &j1=i+2;
3 int &j2=2;
```

Auch das führt zu Übersetzungsfehlern:

```
sep@linux:~/fh/prog3/examples/src> g++ -c WrongInitRef.cpp
WrongInitRef.cpp:2: error: could not convert ‘(i + 2)’ to ‘int&’
WrongInitRef.cpp:3: error: could not convert ‘2’ to ‘int&’
sep@linux:~/fh/prog3/examples/src>
```

Benutzung

Ist eine Referenzvariable einmal initialisiert, so kann sie als Synonym für die Variable auf die die Referenz geht, betrachtet werden. Jede Änderung der Referenzvariablen ist in der Originalvariablen sichtbar und umgekehrt.

Im folgenden Programm kann man sehen, dass Modifizierungen an der Originalvariablen über die Referenzvariablen sichtbar werden.

ModifyOrg.cpp

```
1  #include <iostream>
2  int  main(){
3      int i=42;
4      int &j=i;
5      std::cout << j << std::endl;
6      i=i+1;
7      std::cout << j << std::endl;
8      i=i-1;
9      std::cout << j << std::endl;
10     int k=j;
11     std::cout << k << std::endl;
12     i=i+1;
13     std::cout << k << std::endl;
14 }
```

Das Programm hat folgende Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> ./ModifyOrg
42
43
42
42
42
sep@linux:~/fh/prog3/examples/src>
```

Wie man sieht, wird anders als bei Zeigern, für die Referenzvariablen nicht eine Speicheradresse ausgegeben, sondern der Wert, der in der Originalvariablen gespeichert ist. Eine Referenzvariable braucht nicht wie ein Zeiger dereferenziert zu werden. Dieses wird bereits automatisch

gemacht. Referenzen werden automatisch dereferenziert, wenn der Kontext den Originaltyp erwartet. Daher kann auch eine Referenzvariable wieder direkt einer Variablen des Originaltyps zugewiesen werden. So wird in der Zuweisung der Referenz `j` auf die Variable `k` des Originaltyps `int` oben, der in der referenzierten Variablen gespeicherte Wert in `k` gespeichert. Ändert sich dieser, so ist das in `k` nicht sichtbar.

Genauso findet eine automatische Dereferenzierung statt, wenn einer Referenzvariablen ein neuer Wert des Originaltyps zugewiesen wird:

ModifyRef.cpp

```
1  #include <iostream>
2
3  int main(){
4    int i = 21;
5    int &j=i;
6
7    j=2*j;
8    std::cout << i << ", "<< j << std::endl;
9  }
```

Das Programm führt zu folgender Ausgabe:

```
sep@linux:~/fh/prog3/examples/bin> ./ModifyRef
42, 42
sep@linux:~/fh/prog3/examples/bin>
```

Die Änderung an der Referenzvariabel wird auch in der Originalvariabel sichtbar.

Besonders vertrackt wird es, wenn wir einer Referenzvariablen eine andere Referenzvariable zuweisen.

RefToRef.cpp

```
1  #include <iostream>
2
3  int main(){
4    int i1 = 21;
5    int &j1=i1;
6
7    int i2 = 42;
8    int &j2=i2;
9    std::cout << i2 <<", "<< j2 << std::endl;
10
11   j2 = j1;
12   std::cout << i2 <<", "<< j2 << std::endl;
13
14   j2 = 15;
15   std::cout << i1 <<", "<< j1 << std::endl;
16   std::cout << i2 <<", "<< j2 << std::endl;
17 }
```

Die erzeugte Ausgabe ist:

```
sep@linux:~/fh/prog3/examples/src> ./RefToRef
42, 42
21, 21
21, 21
15, 15
sep@linux:~/fh/prog3/examples/src>
```

Obwohl wir eine Zuweisung `j2=j1` haben, sind beides weiterhin Referenzen auf unterschiedliche Variablen! Die Zeile bedeutet nicht, dass die Referenz `j2` jetzt die gleiche Referenz wie `j1` ist; sondern dass lediglich der mit der Referenz `j2` gefundene Wert in die durch `j1` referenzierte Variable abzuspeichern ist. In dieser Zeile werden beide Referenzvariablen also implizit dereferenziert. Eine einmal für eine Variable initialisierte Referenz läßt sich also nicht auf eine andere Variable umbiegen.

Referenzen als Funktionsparameter

Im Prinzip gilt für Funktionsparameter von einem Referenztyp dasselbe wie für Variablen von einem Referenztyp. Der Funktion wird keine Kopie der Daten übergeben, sondern eine Referenz auf diese Daten:

RefArg.cpp

```
1  #include <iostream>
2
3  void f(int &x){
4    x=x+1;
5  }
6
7  int main(){
8    int i = 41;
9    f(i);
10   std::cout << i << std::endl;
11 }
```

Die erwartete Ausgabe ist:

```
sep@linux:~/fh/prog3/examples/bin> ./RefArg
42
sep@linux:~/fh/prog3/examples/bin>
```

Es gelten dieselben Einschränkungen, wie wir sie schon für die Zuweisung auf Referenzvariablen kennengelernt haben. Es dürfen Referenzfunktionsparameter nicht mit beliebigen Ausdrücken aufgerufen werden:

WrongCall.cpp

```
1  #include <iostream>
2
3  void f(int &x){}
4
5  int main(){
6    int i = 41;
7    f(i+1);
```

```
8   f(1);
9 }
```

Wir bekommen wieder die schon bekannten Fehlermeldungen:

```
sep@linux:~/fh/prog3/examples/src> g++ -c WrongCall.cpp
WrongCall.cpp: In function ‘int main()’:
WrongCall.cpp:8: error: could not convert ‘(i + 1)’ to ‘int&’
WrongCall.cpp:3: error: in passing argument 1 of ‘void f(int&)’
WrongCall.cpp:9: error: could not convert ‘1’ to ‘int&’
WrongCall.cpp:3: error: in passing argument 1 of ‘void f(int&)’
sep@linux:~/fh/prog3/examples/src>
```

2.2.6 Zusammenfassung

Kapitel 3

Weitere Programmierkonzepte in C++

Die in diesem Kapitel vorgestellten Konzepten von C++ sind keine Konzepte der Objektorientierung, sondern zusätzliche Konzepte, die in vielen modernen auch nicht objektorientierten Programmiersprachen zu finden sind.

3.1 Überladen

Es ist in C++ möglich zwei Methoden oder Funktionen mit gleichen Namen zu haben.

3.1.1 Überladen von Funktionen

Wir werden Funktionen überladen, so dass wir mehrere Funktionsdefinitionen für ein und dieselbe Funktion erhalten.

Beispiel 3.1.1 *Wir schreiben eine* `trace` *Funktion durch Überladung.*

OverloadedTrace.cpp

```
1  #include <string>
2  #include <iostream>
3
4  struct Person
5    {std::string name
6    ;std::string vorname
7    ;};
8
9  Person trace(Person p){
10   std::cout << p.vorname << " " << p.name<< std::endl;
11   return p;
12 }
13
14 int trace (int i){
```

```
15   std::cout << i << std::endl;return i;
16 }
17
18 float trace (float i){
19   std::cout << i << std::endl;return i;
20 }
21
22 std::string trace (std::string i){
23   std::cout << i << std::endl;return i;
24 }
25
26 int main(){
27   Person p1 = {"Zabel", "Walther"};
28   Person p2 = trace(p1);
29   trace (456456456);
30   trace (425456455.0f);
31   std::string s1 = "hallo";
32   std::string s2 = trace (s1);
33 }
```

Das Programm führt zu der erwarteten Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> ./OverloadedTrace
Walther Zabel
456456456
4.25456e+08
hallo
sep@linux:~/fh/prog3/examples/src>
```

3.1.2 Standardarameter

In C++ ist es möglich, in der Funktionssignatur bestimmte Parameter mit einem Standardwert zu belegen. Dann kann die Funktion auch ohne Übergabe dieses Parameters aufgerufen werden. Der in dem Aufruf fehlende Parameter wird dann durch den Standardwert der Signatur ersetzt.

Beispiel 3.1.2 *Im folgenden definieren wir eine Funktion, die zwei ganze Zahlen als Parameter erhält. Der zweite Parameter hat einen Standardwert von 40. Bei Aufruf der Funktion mit nur einem Parameter wird für den Parameter* `y` *der Standardwert benutzt.*

DefaultParameter.cpp

```
1 #include <iostream>
2
3 int f(int x,int y=40){return x+y;}
4
5 int main(){
6     std::cout << "f(1,2) = "<< f(1,2) << std::endl;
7     std::cout << "f(2)   = "<< f(2) << std::endl;
8 }
```

An der Ausgabe können wir uns davon überzeugen, dass für das fehlende `y` *der Wert* `40` *benutzt wird.*

```
sep@linux:~/fh/prog3/examples/src> g++ -o DefaultParameter DefaultParameter.cpp
sep@linux:~/fh/prog3/examples/src> ./DefaultParameter
f(1,2) = 3
f(2)   = 42
sep@linux:~/fh/prog3/examples/src>
```

Bei der Benutzung von Standardwerten für Parameter gibt es eine Einschränkung in C++. Sobald es für einen Parameter einen Standardwert gibt, muss es auch einen Standardwert für alle folgenden Parameter geben. Es geht also nicht, dass für den n-ten Parameter ein Standardwert definiert wurde, nicht aber für den m-ten Parameter mit $n < m$.

Standardwerte sind eine bequeme Notation, können aber leicht mit überladenen Funktionen simuliert werden:

Beispiel 3.1.3 *Jetzt simulieren wir, Standardparameter durch eine überladene Funktionsdefinition.*

OverloadedInsteadOfDefaults.cpp

```
1 #include <iostream>
2
3 int f(int x,int y){return x+y;}
4 int f(int x){return f(x,40);}
5
6 int main(){
7     std::cout << "f(1,2) = "<< f(1,2) << std::endl;
8     std::cout << "f(2)   = "<< f(2) << std::endl;
9 }
```

Das Programm funktioniert entsprechende dem vorherigen Beispiel:

```
sep@linux:~> g++ -o OverloadedInsteadOfDefaults OverloadedInsteadOfDefaults.cpp
sep@linux:~> ./OverloadedInsteadOfDefaults
f(1,2) = 3
f(2)   = 42
sep@linux:~>
```

3.1.3 Überladen von Operatoren

C++ erlaubt es, alle in C++ bekannten Operatoren für jede beliebige Klasse zu überladen. Das gilt für die gängigen Operatoren wie `+`, `-`, `*`, `/` aber auch tatsächlich für jeden irgendwie in C++ legalen Operator, inklusive solcher Dinge wie der Funktionsanwendung, ausgedrückt durch das runde Klammernpaar.

Zum Überladen eines Operators $\oplus$ ist in einer Klasse eine Funktion `operator`$\oplus$ zu schreiben.

Beispiel 3.1.4 *Wir definieren eine Klasse zur Darstellung komplexer Zahlen. Auf komplexen Zahlen definieren wir Addition, Multiplikation und die Norm, die durch Funktionsanwendungsklammern ausgedrückt werden soll.*

Complex.hpp

```
#include <string>
#include <iostream>

class Complex {
  public:
    float im;
    float re;

    Complex(float re,float im);

    Complex operator+(const Complex& other)const;
    Complex operator*(const Complex& other)const;
    float operator()()const;

    std::string toString()const;
};
```

Die Operatoren lassen sich wie reguläre Funktionen implementieren:

Complex.cpp

```
#include "Complex.hpp"
#include <sstream>

Complex::Complex(float re,float im){
  this->re=re;this->im=im;
}

Complex Complex::operator+(const Complex& other)const{
  return Complex(re+other.re,im+other.im);
}

Complex Complex::operator*(const Complex& other)const{
  return
   Complex(re*other.re-im*other.im,re*other.im+im*other.re);
}

float Complex::operator()()const{
  return re*re+im*im;
}

std::string Complex::toString()const{
  std::string result="";
  std::stringstream ss;
  ss<<re;
  ss>>result;
  result=result+"+";

  std::string resulti="";
  std::stringstream sx;
```

```
30    sx<<im;
31    sx>>resulti;
32    result=result+resulti+"i";
33    return result;
34  }
```

Jetzt können wir mit komplexen Zahlen ebenso rechnen, wie mit eingebauten Zahlentypen:

TestComplex.cpp

```
1  #include "Complex.cpp"
2
3  int main(){
4    Complex c1(31,40);
5    Complex c2(0.5,2);
6    std::cout << (c1+c2).toString() << std::endl;
7    std::cout << (c1*c2).toString() << std::endl;
8    std::cout << c2() << std::endl;
9  }
```

Nicht nur auf klassischen Zahlentypen können wir Operatoren definieren. Auch für unsere immer mal wieder beliebte Listenklasse, können wir Operatoren definieren. Zur Konkatenation von Listen ist z.B. der Operator + eine natürliche Wahl, so dass wir folgende Kopfdatei für Listen erhalten:

StringLi.hpp

```
1  #include <string>
2
3  class StringLi {
4    private:
5      std::string hd;
6      StringLi* tl;
7      bool empty;
8
9    public:
10     StringLi(std::string hd,StringLi* tl);
11     StringLi();
12     virtual ~StringLi();
13
14     bool isEmpty();
15     std::string head();
16     StringLi* tail();
17
18     StringLi* clone();
19
20     StringLi* operator+(StringLi* other);
21
22     std::string toString();
23 };
```

Im folgenden die naheliegende Implementierung dieser Listen.

StringLi.cpp

```
#include <string>
#include "StringLi.hpp"

StringLi::StringLi(std::string h,StringLi* t):empty(false){
  this->hd=h;
  this->tl=t;
}

StringLi::StringLi():empty(true){}

StringLi::~StringLi(){
  if (!isEmpty()) delete tl;
}

bool StringLi::isEmpty(){return empty;}
std::string StringLi::head(){return hd;}
StringLi* StringLi::tail(){return tl;}

StringLi* StringLi::clone(){
  if (isEmpty()) return new StringLi();
  return new StringLi(head(),tail()->clone());
}

StringLi* StringLi::operator+(StringLi* other){
  if (isEmpty()) return other->clone();
  return new StringLi(head(),(*tail())+other);
}

std::string StringLi::toString(){
  if (isEmpty()) return "()";
  std::string result="["+head();
  for (StringLi* it=this->tail()
      ; !it->isEmpty();it=it->tail()){
    result=result+","+it->head();
  }
  return result+"]";
}
```

Listen lassen sich jetzt durch die Addition konkatenieren:

StringLiTest.cpp

```
#include <iostream>
#include "StringLi.hpp"

int main(){
  StringLi* xs
    = new StringLi("friends",
      new StringLi("romans",
      new StringLi("contrymen",
```

```
9       new StringLi())));
10
11
12    StringLi* ys
13     = new StringLi("lend",
14       new StringLi("me",
15       new StringLi("your",
16       new StringLi("ears",
17       new StringLi()))));
18
19    StringLi* zs = *xs+ys;
20    delete xs;
21    delete ys;
22
23    std::cout << zs->toString() << std::endl;
24  }
```

Wie man der Ausgabe entnehmen kann, führt der Operator + tatsächlich zur Konkatenation.

```
sep@linux:~/fh/prog3/examples/src> g++ -o StringLiTest StringLi.cpp StringLiTest.cpp
sep@linux:~/fh/prog3/examples/src> ./StringLiTest
[friends,romans,contrymen,lend,me,your,ears]
sep@linux:~/fh/prog3/examples/src>
```

Eine vollständige Tabelle aller in C++ überladbaren Operatoren findet sich in Abbildung 3.1.

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/*	%=	^=	&=	|=
<<	>>	<<=	>>=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	delete	new[]	delete[]

Abbildung 3.1: C++ Operatoren, die überladen werden können.

Überladen des Ausgabestroms

PrintAsString.hpp

```
1  #ifndef PRINT_AS_STRING_H
2  #define PRINT_AS_STRING_H
3  #include "ToString.hpp"
4  #include <iostream>
5
6  std::ostream& operator<<(std::ostream& stream,ToString* ob);
7  #endif
```

PrintAsString.cpp

```
1  #include "PrintAsString.hpp"
2
```

```
3 std::ostream& operator<<(std::ostream& stream,ToString* ob){
4   stream << ob->toString();
5   return stream;
6 }
```

3.2 Generische Typen

3.2.1 Generische Klassen

Wir werden in der Programmierung häufig in die Situation kommen, dass sich zwei Klassen lediglich im Typ eines ihrer Felder unterscheiden. Wir hatten schon ein prominentes Beispiel im laufe dieser Vorlesung. Die Klasse `Li1` hatte als Typ des Feldes `head int`. Aufgabe war es, eine Klasse zu schreiben, die genauso funktioniert, wie die Klasse `Li1` nur im `head` eines jeden Objekts sollte kein `int` Wert gespeichert sein, sondern ein Zeiger auf ein `Paintable`-Objekt. In der entsprechenden Aufgabe war tatsächlich lediglich die Typinformation `int` durch die Information `Paintable*` zu ersetzen, um zu einer neuen Klasse zu gelangen. Der Rest des Codes wurde kopiert.

Um solchen verdoppelten Code zu verhindern, bietet C++ an, Klassen generisch zu schreiben. Eine generische Klasse läßt für bestimmte Felder den Typen einfach offen. Erst wenn ein Objekt dieser Klasse angelegt wird, ist zu spezifizieren, welcher Typ für den zunächst offen gelassenen Typen einzusetzen ist.

Betrachten wir einmal zwei Klassen, die einfache Container sind. Einmal einen Container für die Stringwerte:

BoxString.cpp

```
1 #include <string>
2
3 class BoxString{
4   public:
5     std::string contents;
6     BoxString(std::string c) {contents = c;}
7 };
```

Zum anderen eine analoge Klasse hierzu, zum Speichern von ganzen Zahlen:

BoxInt.cpp

```
1 class BoxInt{
2   public:
3     int contents;
4     BoxInt(int c) {contents = c;}
5 };
```

Über den Schablonenmechanismus von C++ können wir statt dieser beiden Klassen eine einzige Klassenschablone, eine generische Klasse schreiben. Hierzu ersetzen wir den konkreten Typ der variabel gehalten werden soll durch eine frei zu wählende Typvariable und deklarieren vor der Klasse, dass es sich um eine Klassenschablone mit genau dieser Typvariablen für die Klasse handeln soll:

Box.hpp

```
#include <string>

template <typename elementType>
class Box{
  public:
    elementType contents;
    Box(elementType c) {contents = c;}
};
```

Diese generische Klasse kann jetzt jeweils für konkrete Typen instanziiert werden. Hierzu dient eine Notation mit spitzen Klammern. Dem Typ `Box` ist stets mit anzugeben, mit welchem konkreten Typ für die Typvariable er benutzt werden soll. So gibt es jetzt den Typ `Box<std::string>` ebenso wie den Typ `Box<int>` und sogar den in sich verschachtelten Typ `Box<Box<std::string>*>`:

TestGenBox.cpp

```
#include <string>
#include <iostream>
#include "Box.hpp"

int main(){
  const Box<std::string>* bs = new Box<std::string>("hallo");
  const Box<int>* bi = new Box<int>(42);
  const Box<Box<std::string>*>* bbs
    = new Box<Box<std::string>*>(new Box<std::string>("hallo"));

  std::string s = bs -> contents;
  int i = bi -> contents;
  std::string s2= bbs -> contents->contents;

  std::cout << bs -> contents << std::endl;
  std::cout << bi -> contents << std::endl;
  std::cout << bbs -> contents->contents << std::endl;
}
```

Generische Typen sind ein zur Objektorientierung vollkommen orthogonales Konzept; es verträgt sich vollkommen mit den objektorientierten Konzepten, wie der Vererbung. Es ist möglich Unterklassen generischer Klassen zu bilden. Diese Unterklassen können selbst generisch sein oder auch nicht.

Wir können z.B. die obige Klasse `Box` erweitern, so dass wir in der Lage sind Paare von Objekten zu speichern. Zusätzlich erlauben wir uns den Luxus, dass die Klasse `Pair` auch noch die Klasse `ToString` implementiert:

Pair.hpp

```
#ifndef PAIR_H
#define PAIR_H

#include <string>
```

```
5  #include <sstream>
6  #include "Box.hpp"
7  #include "ToString.hpp"
8
9  template <class fstType,class sndType>
10 class Pair: public Box<fstType>,public ToString{
11   public:
12     sndType snd;
13     Pair(fstType c,sndType snd):Box<fstType>(c),snd(snd) {}
14
15     std::string toString(){
16       std::stringstream s;
17       s<<"("<<contents<<","<<snd<<")";
18       return s.str();
19     }
20 };
21
22 #endif
```

Diese Klasse läßt sich ebenso wie die Klasse `Box` für konkrete Typen für die zwei Typvariablen instanziieren.

TestPair.cpp

```
1  #include "Pair.hpp"
2  #include <iostream>
3  #include <string>
4
5  int main(){
6    Pair<std::string,int> p("hallo",42);
7    std::cout << p.contents << std::endl;
8    std::cout << p.snd << std::endl;
9  }
```

Wir können z.B. eine Paarklasse definieren, die nur noch eine Typvariable hat. Für solche Paare können die beiden im Paar gespeicherten Elemente vertauscht werden, was in der zusätzlich Methode `swap` implemengtiert ist:

UniPair.hpp

```
1  #ifndef UNI_PAIR_H
2  #define UNI_PAIR_H
3  #include "Pair.hpp"
4
5  template <typename fstType>
6  class UniPair: public Pair<fstType,fstType>{
7   public:
8    UniPair(fstType c,fstType snd):Pair<fstType,fstType>(c,snd) {}
9    void swap(){fstType tmp=contents;contents=snd;snd=tmp;}
10 };
11 #endif
```

Jetzt muss für diese Klasse im konkreten Fall auch nur noch ein einziger Typparameter instanziiert werden:

TestUniPair.cpp

```
1  #include "UniPair.hpp"
2  #include <iostream>
3  #include <string>
4
5  int main(){
6    UniPair<std::string> p("welt","hallo");
7    p.swap();
8    std::cout << p.contents << std::endl;
9    std::cout << p.snd << std::endl;
10 }
```

Generische Listen

Eine Paradeanwendung für generische Typen sind natürlich sämtliche Sammlungsklassen. Anstatt jetzt einmal Listen für Zeichenketten und einmal Listen für ganze Zahlen zu programmieren, können wir eine Schablone schreiben, in der der Elementtyp der Listenelemente variabel gehalten wird.

Im folgenden also eine weitere Version underer Listenklasse. Dieses Mal als generische Klasse. Die Typvariable haben wir mit dem Symbol `A` bezeichnet.

Li.hpp

```
1  #include <string>
2  #include <sstream>
3  #include "UniPair.hpp"
```

Beginnen wir mit den gängigen Definitionen für Konstruktoren, Selektoren und Testmethode:

Li.hpp

```
4  #include "ToString.hpp"
5  template <typename A>
6  class Li:public ToString {
7    private:
8      A hd;
9      Li<A>* tl;
10     bool empty;
11
12   public:
13     Li(A hd, Li<A>* tl):empty(false){
14       this->hd=hd;
15       this->tl=tl;
16     }
17
18     Li():empty(true){}
19
20     bool isEmpty(){return empty;}
```

```
21      A head(){return hd;}
22      Li<A>* tail(){return tl;}
```

Im Destruktor wird die Listenstruktur aus dem Speicher geräumt. Achtung, die Elemente werden nicht gelöscht. Dieses ist extern vorzunehmen.

Li.hpp

```
23      virtual ~Li(){
24        if (!isEmpty()) delete tl;
25      }
```

Es folgen ein paar gängige Methoden, die wir schon mehrfach implementiert haben.

Li.hpp

```
26      Li<A>* clone(){
27        if (isEmpty()) return new Li<A>();
28        return new Li<A>(head(),tail()->clone());
29      }
30
31      Li<A>* operator+(Li<A>* other) {
32        if (isEmpty()) return other->clone();
33        return new Li<A>(head(),(*tail())+other);
34      }
35
36      int size()  {
37        if (isEmpty()) return 0;
38        return 1+tail()->size();
39      }
```

Die Methode `contains`, die testen soll, ob ein Element in der Liste enthalten ist. Als Gleicheit wird der Operator `==` verwendet.

Li.hpp

```
40      bool contains(A el){
41        if (isEmpty())return false;
42        if (head()==el) return true;
43        return tail()->contains(el);
44      }
45
```

Als nächstes folgt eine typische Methode höherer Ordnung. Es wird eine neue Liste erzeugt, indem auf jedes Listenelement eine Funktion angewendet wird. Hier kommt eine weitere Typvariable ins Spiel. Wir lassen in dieser Methode offen, welchen Typ die Elemente der Ergebnisliste haben soll.

Li.hpp

```
46        void forEach(A f(A)){
47        for (Li<A>* it=this;!it->isEmpty();it=it->tail())
48          it->hd=f(it->head());
49      }
```

Und schließlich die Methode, um eine Stringdarstellung der Liste zu erhalten. Diese Methode erwartet vom Elementtyp `A`, dass für diesen der Operator `<<` zur Ausgabe auf einen Ausgabestrom definiert ist. Diese haben wir bereits für Zeiger auf Klassen, die eine Methode `toString` haben, definiert.

Li.hpp

```
50      std::string toString(){
51        if (isEmpty()) return "[]";
52        std::stringstream ss;
53        ss<<"["<<this->head();
54        for ( Li<A>* it=this->tail()
55            ; !it->isEmpty();it=it->tail()){
56          ss<<","<<it->head();
57        }
58        ss<<"]";
59        return ss.str();
60      }
61  };
```

Zeit einen Test für die generischen Listen zu schreiben. Wir legen Listen von Zeichenketten, aber auch Listen von Paaren an. Einige Methoden werden ausprobiert, insbesondere auch die Methode `forEach`. Auch die Methode `toString` läßt sich für die verschiedenen Elementtypen ausführen.

TestLi.cpp

```
1   #include "Li.hpp"
2   #include "UniPair.hpp"
3   #include "PrintAsString.hpp"
4   #include <iostream>
5   #include <string>
6
7   UniPair<std::string>*  swap(UniPair<std::string>* p){p->swap();return p;}
8
9   std::string reverse(std::string str){
10    int strLength=str.length();
11    for (int i=0;i<strLength/2;i++){
12      char tmp=str[i];
13      str[i]=str[strLength-i-1];
14      str[strLength-i-1]=tmp;
15    }
16    return str;
17  }
18
19  int main(){
20    Li<std::string>* xs
21      = new Li<std::string>("Schimmelpfennig",
22        new Li<std::string>("Moliere",
23        new Li<std::string>("Shakespeare",
24        new Li<std::string>("Bernhard",
25        new Li<std::string>("Brecht",
26        new Li<std::string>("Salhi",
```

```
27        new Li<std::string>("Achternbusch",
28        new Li<std::string>("Horvath",
29        new Li<std::string>("Sophokles",
30        new Li<std::string>("Calderon",
31        new Li<std::string>()))))))))));
32
33    std::cout << xs << std::endl;
34
35    std::cout << xs->contains("Brecht") << std::endl;
36    std::cout << xs->contains("Zabel") << std::endl;
37
38    xs->forEach(reverse);
39    std::cout<<xs<<std::endl;
40    delete xs;
41
42    Li<UniPair<std::string>* >* ys =
43    new Li<UniPair<std::string>* >(new UniPair<std::string>("A","1")
44   ,new Li<UniPair<std::string>* >(new UniPair<std::string>("B","2")
45   ,new Li<UniPair<std::string>* >(new UniPair<std::string>("C","3")
46   ,new Li<UniPair<std::string>* >()));
47
48    std::cout << ys << std::endl;
49
50    ys->forEach(swap);
51    std::cout << ys << std::endl;
52  }
```

Der Test führt zu folgenden Ausgaben:

```
sep@pc305-3:~/fh/cpp/student> bin/TestLi
[Schimmelpfennig,Moliere,Shakespeare,Bernhard,Brecht,Salhi,Achternbusch,Horvath,Sophokles,Calderon]
1
0
[ginnefplemmihcS,ereiloM,eraepsekahS,drahnreB,thcerB,ihlaS,hcsubnrethcA,htavroH,selkohpoS,noredlaC]
[(A,1),(B,2),(C,3)]
[(1,A),(2,B),(3,C)]
sep@pc305-3:~/fh/cpp/student>
```

Aufgabe 7 Ergänzen Sie die obige Listenklasse um weitere Listenoperatoren.

- `A operator[](int i);` soll das i-te Element zurückgeben.
- `Li<A>* operator-(int i);` soll eine neue Liste erzeugen, indem von der Liste die ersten i Elemente weggelassen werden.
- `bool operator<(Li<A>* that);` soll die beiden Listen bezüglich ihrer Länge vergleichen.
- `bool operator==(Li<A>* that);` soll die Gleichheit auf Listen implementieren.

Testen Sie ihre Implementierung an einigen Beispielen.

3.2.2 generische Funktionen

Im vorangegangenen Abschnitt haben wir die generische Programmierung eng mit der Objektorientierung zusammen eingeführt, indem Klassen generisch gemacht wurden. Einfache normale Funktionen können auch generisch geschrieben werden.

Die Idee einer generischen Funktion ist, eine Funktionsdefinition zu schreiben, die für mehrere verschiedene Parametertypen funktioniert. In funktionalen Sprachen werden generische Funktionen auch als polymorphe Funktionen bezeichnet; dieser Begriff ist in der objektorientierten Programmierung aber anderweitig belegt. Oft wird auch der Begriff der parameterisierten Typen verwendet.

Die Grundidee ist, eine Funktion zu schreiben, die für mehrere Typen gleich funktioniert. Eine typische und einfache derartige Funktion ist eine *trace*-Funktion, die ihr Argument unverändert als Ergebnis wieder ausgibt, mathematisch also die Identität darstellt. Als Seiteneffekt gibt die Funktion das Argument auf den Bildschirm aus. Diese Funktion läßt sich generisch schreiben. Hierzu definiert man vor der Funktionssignatur, dass ein Typ variabel gehalten wird. Es wird wieder eine Typvariabel eingeführt. Dazu bedient man sich des Schlüsselworts `template` und setzt in spitze Klammern den Namen der eingeführte Typvariablen. Dieser ist das Schlüsselwort `typename` voranzustellen. Für die generische Funktion `trace` ergibt sich folgende Signatur.

Trace.hpp

```
1 template <typename a>
2 a trace(a x);
```

Man kann Typvariablen als allquantifiziert ansehen. Dann sagt obige Signatur aus: für alle Typen `a`, funktioniert die Funktion `trace`, indem sie ein Parameter dieses Typs nimmt, und diesen Typ auch als Ergebnis hat.

Implementieren wir nun die Funktion. Hierzu schreiben wir wieder die entsprechende Signatur mit der Variablen. Der Parameter soll auf der Konsole aufgegeben werden und wieder zurückgegeben werden.

Trace.cpp

```
1  #include <iostream>
2  #include "Trace.hpp"
3
4  template <typename a>
5  a trace(a x){
6    std::cout << ">>>trace: " << x << std::endl;
7    return x;
8  }
```

Jetzt können wir die Funktion für einfache Beispiele einmal ausprobieren:

Trace.cpp

```
9  int main(){
10   int x = 5 + trace(21*2);
11   char *str = "hallo";
12   trace(str);
13   x = trace(6 + trace(18*2));
14 }
```

Wir rufen die Funktion mehrmals für ganze Zahlen und einmal für Strings auf.

```
sep@linux:~/fh/prog3/examples/src> g++ -o Trace Trace.cpp
sep@linux:~/fh/prog3/examples/src> ./Trace
>>>trace: 42
>>>trace: hallo
>>>trace: 36
>>>trace: 42
sep@linux:~/fh/prog3/examples/src>
```

Wir haben oben behauptet, dass die Typvariable `a` für die Funktion `trace` als allquantifiziert betrachtet werden kann. das ist nicht ganz wahr.[1] Für welche Typen die Formularmethode `trace` anwendbar ist, geht aus der Signatur nicht hervor. Dieses wird von C++ erst dann geprüft wenn tatsächlich die Funktion aufgerufen wird. Dann wird das Formular genommen und die Typvariabel ersetzt durch den Typ, auf den die Funktion angewendet wird. Das Formular wird benutzt, der konkrete Typ eingestzt und dann diese Instanz des Formulars vom Compiler auf Typkorrektheit geprüft.

Unsere Funktion `trace` kann nur übersetzt werden für Typen, auf denen der Ausgabeoperator `<<` definiert ist. Dieses ist für ganze Zahlen und Strings der Fall. Versuchen wir jetzt einmal die Funktion für eine Struktur aufzurufen. Die Struktur kann nicht mit dem Operator `<<` ausgegeben werden und daher kann auch die Funktion `trace` nicht für diese Struktur aufgerufen werden. Versuchen wir dieses:

TraceWrongUse.cpp

```
1  #include <iostream>
2  #include "Trace.h"
3
4  template <typename a>
5  a trace(a x){
6    std::cout <<">>>trace: "<< x << std::endl;
7    return x;
8  }
9
10 struct S {int x;char* s;};
11
12 int main(){
13   S u = {42,"hallo"};
14   u = trace(u);
15 }
```

Wir erhalten folgende wortreiche Fehlermeldung (von der wir hier nur den Anfang wiedergeben):

```
sep@linux:~/fh/prog3/examples/src> g++ -c TraceWrongUse.cpp
TraceWrongUse.cpp: In function ‘a trace(a) [with a = S]’:
TraceWrongUse.cpp:13:   instantiated from here
TraceWrongUse.cpp:6: error: no match for ’operator<<’ in ’
   std::operator<<(std::basic_ostream<char, _Traits>&, const char*) [with
   _Traits = std::char_traits<char>]((&std::cout), ">>>trace: ") << x’
/usr/include/g++/bits/ostream.tcc:63: error: candidates are:
   std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
   _Traits>::operator<<(std::basic_ostream<_CharT,
```

[1] In Javas generischen Typen wäre dieses auch der Fall, in C++ kann man dieses nicht sagen.

```
   _Traits>&(*)(std::basic_ostream<_CharT, _Traits>&)) [with _CharT = char,
   _Traits = std::char_traits<char>]
/usr/include/g++/bits/ostream.tcc:85: error:
   std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
   _Traits>::operator<<(std::basic_ios<_CharT,
   _Traits>&(*)(std::basic_ios<_
         .
         .
         .

sep@linux:~/fh/prog3/examples/src>
```

So wortreich die Fehlermeldung ist, so gibt sie uns doch in den ersten paar Zeilen sehr gut Auskunft darüber, was das Problem ist. Wir instanziieren die Typvariable in der Methode `trace` mit der Struktur `S`. Dann muss der Operator für den Typ `S` angewendet werden, wofür dieser aber nicht definiert ist.

In C++ kann man aus der Signatur einer generischen Funktion nicht erkennen, für welche Typen die Funktion aufgerufen werden kann.

Aufgabe 8 Schreiben Sie eine Funktion zur Funktionskomposition mit folgender Signatur:

```
1 template <typename a, typename b, typename c>
2 c composition(c f2(b),b f1(a),a x);
```

Sie sei spezifiziert durch die Gelichung: $composition(f_2, f_1, x) = f_2(f_1(x))$. Testen Sie Ihre Implementierung für verschiedene Typen.

3.2.3 Heterogene und homogene Umsetzung

Wir haben bereits gesehen, dass die Signatur einer generischen Funktion allein nicht ausreicht, um zu bestimmen, auf welche Parametertypen die Funktion anwendbar ist. Das ist ein neues Phänomen, welches uns bisher noch nicht untergekommen ist und offenbahrt eine große Schwäche der Umsetzung der Genrizität in C++. Diese Eigenschaft hat ein weitreichende Konsequenz für die Arbeit mit Kopfdateien. Versuchen wir jetzt einmal unser Beispiel der Funktion `trace` in der gleichen Weise mit einer Kopfdatei zu bearbeiten, wie wir es bisher gemacht haben. Wir schreiben also eine Kopfdatei mit lediglich der Signatur:

T.hpp

```
1 template <typename a>
2 a trace(a x);
```

Und die entsprechende Implementierungsdatei:

T.cpp

```
1 #include <iostream>
2 #include "T.hpp"
3
4 template <typename a>
5 a trace(a x){
6   std::cout << ">>>trace: " << x << std::endl;
```

```
7   return x;
8 }
```

Schließlich ein Programm, dass die Funktion `trace` benutzen will und daher die Kopfdatei `T.h` einbindet:

UseT.cpp

```
1 #include "T.hpp"
2
3 int main(){
4   trace(2*21);
5 }
```

Wir übersetzen diese zwei Programme und erzeugen für sie die Objektdateien, was fehlerfrei gelingt:

```
ep@linux:~/fh/prog3/examples> cd src
sep@linux:~/fh/prog3/examples/src> g++ -c T.cpp
sep@linux:~/fh/prog3/examples/src> g++ -c UseT.cpp
sep@linux:~/fh/prog3/examples/src>
```

Wollen wir die beiden Objektdateien nun allerdings zusammenbinden, dann bekommen wir eine Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ -o UseT UseT.o T.o
UseT.o(.text+0x16): In function ‘main’:
: undefined reference to ‘int trace<int>(int)’
collect2: ld returned 1 exit status
sep@linux:~/fh/prog3/examples/src>
```

Es gibt offensichtlich keinen Code für die Anwendung der generischen Funktion `trace` auf ganze Zahlen. Hieraus können wir schließen, dass der C++-Übersetzer nicht einen Funktionscode für alle generischen Anwendungen einer generischen Funktion hat, sondern für jede Anwendung auf einen speziellen Typ, besonderen Code erzeugt. Für das Programm `T.cpp` wurde kein Code für die Anwendung der Funktion `trace` auf ganze Zahlen erzeugt, weil dieser Code nicht im Programm `T.cpp` benötigt wird. Das Formular wurde quasi nicht für den Typ `int` ausgefüllt. Etwas anderes wäre es gewesen, wenn in der Datei `T.cpp` eine Anwendung von `trace` auf einen Ausdruck des Typs `int` enthalten gewesen wär. Dann wäre auch in der Objektdatei `T.o` Code für eine Anwendung auf `int` zu finden gewesen.

Die Umsetzung generischer Funktionen, in der für jede Anwendung der Funktion auf einen bestimmten Typen eigener Code generiert wird, das Formular für einen bestimmten Typen ausgefüllt wird, nennt man die heterogene Umsetzung. Im Gegensatz dazu steht die homogene Umsetzung, wie sie in Java realisiert ist. Hier findet sich in der Klassendatei nur ein Code für eine generische Funktion.

Wie können wir aber aus dem Dilemma mit den Kopfdateien für generische Funktionen herauskommen. Da C++ zum Ausfüllen des Formulares für die Anwendung einer generischen Funktion den Funktionsrumpf benötigt, bleibt uns nichts anderes übrig, als auch den Funktionsrumpf in die Kopfdatei mit aufzunehmen.

3.2.4 Explizite Instanziierung des Typparameters

Wir haben oben die generischen Funktionen aufgerufen, ohne explizit zu sagen, für welchen Typ wir die Funktion in diesem Fall aufzurufen wünschen. Der C++ Übersetzer hat diese Information inferiert, indem er sich die Typen der Parameter angeschaut hat. In manchen Fällen ist diese Inferenz nicht genau genug. Dann haben wir die Möglichkeit explizit anzugeben, für was für einen Typ wir eine Instanz der generischen Funktion benötigen. Diese explizite Typangabe für den Typparameter wird dabei in spitzen Klammern dem Funktionsnamen nachgestellt.

Beispiel 3.2.1 *In diesem Beispiel rufen wir zweimal die Funktion* `trace` *mit einer Zahlenkonstante auf. Einmal geben wir explizit an, dass es sich bei der der Konstante um eine* `int`*-Zahl handeln soll, einmal dass es sich um eine Fließkommazahl handeln soll.*

ExplicitTypeParameter.cpp

```
1 #include "T.cpp"
2
3 int main(){
4   trace<float>(2*21656567);
5   trace<int>(2*21656567);
6 }
```

Die zwei Aufrufe der Funktion `trace` *führen entsprechend zu zwei verschieden formatierten Ausgaben.*

```
sep@linux:~/fh/prog3/examples/src> ./ExplicitTypeParameter
>>>trace: 4.33131e+07
>>>trace: 43313134
sep@linux:~/fh/prog3/examples/src>
```

3.2.5 Generizität für Reihungen

Besonders attraktiv sind generische Funktionen für Reihungen und wie wir später auch sehen werden für Sammlungsklassen. Man denke z.B. daran, dass, um die Länge einer Liste zu berechnen, der Typ der Listenelemente vollkommen gleichgültig ist.

Eine interessante Funktionalität ist es, eine Funktion auf alle Elemente einer Reihung anzuwenden. Auch dieses kann generisch über den Typ der Elemente der Reihung gehalten werden. Hierzu führen wir zwei Typvariablen ein: eine erste die den Elementtyp der ursprünglichen Reihung angibt, den zweiten, der den Elementtyp der Ergebnisreihung angibt. Damit lät sich eine generische Funkton `map` auf Reihungen wie folgt definieren:

GenMap.cpp

```
1 #include <iostream>
2
3 template <typename a, typename b>
4 void map(a xs [],b ys [],int l,b f(a)){
5   for (int i=0;i<l;i++){
6     ys[i]=f(xs[i]);
7   }
8 }
```

Die an `map` übergebene Funktion, die in der ursprünglichen Funktion `map` den Typ `int f(int)` hatte, ist nun in ihrer Typisierung variabel gehalten. Sie hat einen Typ `a` als Parametertyp und einen beliebigen Typ `b` als Ergebnistyp. Jetzt übergeben wir zwei Reihungen: eine Ursprungsreihung von Elementen des variabel gehaltenen Typs `a` und eine Ergebnisreihung von dem ebenfalls variabel gehaltenen Typ `b`. Die Funktion `map` wendet ihren Funktionsparameter `f` auf die Elemente der Ursprungsreihung an und speichert die Ergebnisse in der Ergebnisreihung ab.

Sofern es sich bei den Elementtypen einer Reihung um Typen handelt, für die der Operator `<<` definiert ist, läßt sich auch eine generische Ausgabefunktion für Reihungen schreiben:

GenMap.cpp

```
9  template <typename a>
10 void printArray(a xs [],int l){
11   std::cout<<"{";
12   for (int i=0;i<l-1;i++){
13     std::cout << xs[i]<<", ";
14   }
15   std::cout << xs[l-1] <<"}"<<std::endl;
16 }
```

Nun können wir die beiden generischen Reihungsfunktionen einmal testen. Hierzu schreiben wir drei kleine Funktionen, die wir der Funktion `map` übergeben werden:

GenMap.cpp

```
17 int add5(int x){return x+5;}
18 char charAt2(std::string xs){return xs[2];}
19 std::string doubleString(std::string ys){return ys+ys;}
```

Und tatsächlich läßt sich die Funktion `map` für ganz unterschiedliche Typen anwenden:

GenMap.cpp

```
20 int main(){
21   int xs [] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
22   int ys [15];
23   map(xs,ys,15,add5);
24   printArray(ys,15);
25
26   std::string us  []= {"hallo","jetzt","wirds","cool"};
27   char vs [4];
28   map(us,vs,4,charAt2);
29   printArray(vs,4);
30
31   std::string ws [4];
32   map(us,ws,4,doubleString);
33   printArray(ws,4);
34 }
```

Hier die Ausgabe dieses Tests:

```
sep@linux:~/fh/prog3/examples/bin> ./GenMap
{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
{l, t, r, o}
{hallohallo, jetztjetzt, wirdswirds, coolcool}
sep@linux:~/fh/prog3/examples/bin>
```

Aufgabe 9 Schreiben Sie eine Funktion `fold` mit folgender Signatur:

Fold.h

```
1 template <typename a,typename b>
2 b fold(a xs [],int length,b op(b,a),b startV);
```

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

$$\text{fold(xs,l,op,st)} = \text{op}(\ldots\text{op}(\text{op}(\text{op}(\text{st,xs[0]}),\text{xs[1]}),\text{xs[2]})\ldots,\text{xs[l-1]})$$

Oder bei Infixschreibweise der Funktion `op` gleichbedeutend über folgende Gleichung:

$$\text{fold}(\{x_0, x_1,,\ldots,x_{l-1}\},\text{l,op,st}) = st \text{ op } x_0 \text{ op } x_1 \text{ op } x_2 \text{ op } \ldots \text{op } x_{l-1}$$

Testen Sie Ihre Methode `fold` mit folgenden Programm:

FoldTest.cpp

```
1  #include <iostream>
2  #include <string>
3  #include "Fold.h"
4
5  int add(int x,int y){return x+y;}
6  int mult(int x,int y){return x*y;}
7  int gr(int x,int y){return x>y?x:y;}
8  int strLaenger(int x,std::string y){
9    return x>y.length()?x:y.length();}
10
11 int main(){
12   int xs [] = {1,2,3,4,5,4,3,2,1};
13   std::cout << fold(xs,9,add,0) << std::endl;
14   std::cout << fold(xs,9,mult,1)<< std::endl;
15   std::cout << fold(xs,9,gr,0)  << std::endl;
16   std::string ys [] = {"john","paul","george","ringo","stu"};
17   std::cout << fold(ys,5,strLaenger,0)  << std::endl;
18 }
```

Schreiben Sie ein paar zusätzliche Beispielaufrufe von `fold`.

Aufgabe 10 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester und in ADS vorgestellt wurde, möglichst generisch umsetzen:

a) Implementieren Sie eine Funktion
`void bubbleSort(int xs [],int length )`,
die Elemente mit dem Bubblesort-Algorithmus der Größe nach sortiert. Schreiben Sie ein paar Tests für Ihre Funktion.

b) Schreiben Sie jetzt eine verallgemeinerte Sortierfunktion, in der die Sortierrelation als Parameter mitgegeben wird. Die Sortierfunktion soll also eine Funktion höherer Ordnung mit folgender Signatur sein:
`void bubbleSortBy(int xs [],int length, bool smaller(int,int) )`

c) Verallgemeinern Sie jetzt die Funktion `bubbleSortBy`, dass Sie generisch für Reihungen mit verschiedenen Elementtypen aufgerufen werden kann.

3.3 Namensräume

Mit der objektorientierten Programmierung geht auch ein stark komponentenweises Denken der Softwareentwicklung einher. Eigene unabhängige Komponenten in Form von Bibliotheken werden entwickelt. Bei der Entwicklung von einer komplexen Software werden verschiedenste Bibliotheken mit unterschiedlichsten Klassen benutzt.

Hier stellt sich das Problem, wie verhindert man, dass in zwei unterschiedlichen Bibliotheken zwei Klassen oder Funktionen mit denselben Namen zu finden sind. Vielleicht hat jede Bibliothek eine Funktion Test. Klassennamen wie *Node* sind sehr beliebt und die wahrscheinlichkeiten, dass zwei unterschiedliche Bibliotheken eine Klasse *Node* beinhalten, ist sehr groß.

In C gab es daher die Konvention, dass alle toplevel Bezeichner einer Bibliothek mit demselben Präfix anfangen. So fangen alle Bezeichner der Bibliothek *openGL* mit dem Buchstaben `gl` an und alle Bezeichner der Bibliothek *glut* mit `glut`. Damit werden die Namen natürlich auf dauer immer länger und unhandlicher.

Um hier koordinierter die Klassen und Funktionen einer Bibliothek einen eindeutigen Namen zuzuordnen, gibt es in C++ Namensräume. Es können Namensräume definiert werden und alle Definitionen einer Bibliothek in diesen Namensraum vorgenommen werden. So läßt sich z.B. ein Namenraum `meinPaket` definieren und innerhalb dieses Namensraums eine Klasse mit den Namen `C`.

Die Qualifizierung eines Bezeichners über seinen Namensraum wird in C++ mit einem zweifachen Doppelpunkt vorgenommen. So ist die Klasse `C` mit vollständigen Namen als `meinPaket::C` anzusprechen. Der vollständige Name der Klasse ist also `meinPaket::C`, man spricht auch vom vollständig qualifizierten Namen, wobei der Namensraum `meinPaket` als Präfix des Namens anzusehen ist. Wir haben schon sehr oft vollqualifizierte Namen benutzt, wenn die Standardströme der Ein- und Ausgabebibliothek verwendet wurden, wie z.B. `std::cout`. Oder auch die Klassen der STL wie z.B. `std::vector`.

Oft benötigt man ganz viele Klassen und Funktionen einer Bibliothek. Dann möchte man gerne auf die umständliche vollqualifizierte Schreibweise verzichten. Zur unqualifizierten Benutzung von Bezeichnern aus einem Namensraum kann die Klausel `using namespace` benutzt werden. Die `using namespace` Deklaration gilt ähnlich wie Variablendeklarationen nur in dem Block, in dem sie getätigt wird, also nicht außerhalb des Blockes.

Beispiel 3.3.1 *Im folgenden Programm können einige der Konzepte der Namensräume in C++ nachvollzogen werden:*

Zunächst definieren wir drei Namensräume, in denen sich jeweils eine Funktion `drucke` *befindet. Einer der Namensräume ist ein Unternamensraum.*

NamespaceTest.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  namespace paket1 {
6    void drucke() { cout << "paket1" << endl;
7    }
8  }
9
10 namespace paket2 {
11   void drucke() { cout << "paket2" << endl;
12   }
13   namespace paket3{
14     void drucke() { cout << "paket3" << endl;
15     }
16   }
17 }
```

In der Hauptmethode gibt es Tests mit drei `using` *Blöcken. Entsprechend bezieht sich der unqualifizierte Name stets auf eine andere Funktion* `drucke`.

NamespaceTest.cpp

```
18 int main(){
19   {
20     using namespace paket2::paket3;
21     paket2::drucke();
22     drucke();
23     paket1::drucke();
24   }
25   {
26     using namespace paket2;
27     drucke();
28     paket3::drucke();
29     paket1::drucke();
30   }
31   {
32     using namespace paket1;
33     paket2::drucke();
34     paket2::paket3::drucke();
35     drucke();
36   }
37 }
```

3.4 Ausnahmen

C++ realisiert das Konzept der Ausnahmen, die geworfen und gefangen werden können, um bestimmte Sonderfälle abzufangen. Es gibt hierzu den `throw`-Befehl und `try` und `catch` Blöcke.

3.4.1 Werfen von Ausnahmen

Wann immer während eines Programmdurchlaufs eine Situation auftritt, in der aus irgendeinem Grund nicht mehr auf dem normalen Weg weiter gerechnet werden kann, hat der Programmierer die Möglichkeit eine Ausnahme zu werfen. Hierzu gibt es das Schlüsselwort `throw`. Dem `throw` kann ein beliebiger Ausdruck folgen, der die Ausnahmesituation beschreiben soll. Ein `throw` Befehl bewirkt den sofortigen Abbruch des aktuellen Programmdurchlaufs.

Betrachten wir hierzu einmal die folgende kleine Version der Fakultätsfunktion. bei Übergabe einer negativen Zahl wird eine Ausnahme geworfen. Das Ausnahmeobjekt ist in diesem Fall eine Zeichenkette.

Throw1.cpp

```
1  #include <string>
2  #include <iostream>
3
4  int fac(int x){
5    if (x<0) {
6      std::string msg = "negative Zahl in Fakultaet";
7      throw msg;
8    }
9    if (x==0) return 1;
10   return x*fac(x-1);
11 }
12
13 int main(){
14   std::cout << fac(5)   << std::endl;
15   std::cout << fac(-5)  << std::endl;
16   std::cout << "fertig" << std::endl;
17 }
```

Starten wir dieses Programm, so kommt es beim zweiten Aufruf von `fac` zum Programmabbruch. Alle nachfolgenden Befehle werden nicht mehr ausgeführt.

```
sep@pc305-3:~/fh/cpp/tutor> bin/Throw1
120
Abgebrochen
sep@pc305-3:~/fh/cpp/tutor>
```

Solange nirgends spezifiziert ist, wie mit eventuell geworfenen Ausnahmen umzugehen ist, führt das Programm zu einem Abbruch. Typische Situationen, in denen Ausnahmen geworfen werden, sind illegale Parameter, wie im obigen Beispiel negative Eingaben für die Fakultät, oder auch leere Listen, die nach einem Element gefragt werden, aber auch sehr typischer Weise IO-Operationen. Dateien, die aus irgendeinen Grund nichzt mehr lesbar sind, Netzwerke, die nicht mehr erreichbar sind, Datenbanken, die nicht antworten etc.

3.4.2 Fangen von Ausnahmen

Der Sinn von Ausnahmen liegt natürlich nicht nur darin, die Programmausführung zu unterbrechen, sondern auch auf die Ausnahmesituation zu reagieren. Hierzu gibt es den `try-catch`-Befehl in C++. Ein beliebiger Programmteil kann mit einem `try`-Block geklammert werden. Am Ende dieses `try`-Blockes läßt sich dann spezifizieren, wie im Fall einer geworfenen Ausnahme weiter zu verfahren ist.

Catch1.cpp

```
1  #include <string>
2  #include <iostream>
3
4  int fac(int x){
5    if (x<0) {
6      std::string msg = "negative Zahl in Fakultaet";
7      throw msg;
8    }
9    if (x==0) return 1;
10   return x*fac(x-1);
11 }
12
13 int main(){
14   try{
15     std::cout << fac(5) << std::endl;
16     std::cout << fac(-5) << std::endl;
17     std::cout << "fertig" << std::endl;
18   }catch (std::string s){
19     std::cout << s << std::endl;
20   }
21   std::cout << "fertig nach Ausnahme" << std::endl;
22 }
```

In diesem Programm wird versucht die Fakultät für mehrere Argumente zu berechnen. Beim zweiten Aufruf schlägt dieses Fehl. Eine Ausnahme wird in Form eines `string` geworfen. Diese wird in der Hauptmethode gefangen.

```
sep@pc305-3:~/fh/cpp/tutor> bin/Catch1
120
negative Zahl in Fakultaet
fertig nach Ausname
sep@pc305-3:~/fh/cpp/tutor>
```

Fangen verschiedener Ausnahmeobjekte

Es kann spezifiziert werden, dass unterschiedliche Typen, die geworfen wurden auch unterschiedlich bei der Ausnahmebehandlung verfahren wird.

Catch2.cpp

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  class Minimal{};
5
6  int fac(int n){
7      std::string help="negativ argument to fac";
8      if (n<0)  throw help;
9      if (n>20) throw 42;
10     if (n>15) throw Minimal();
11     if (n==0) return 1;
12     return n*fac(n-1);
13 }
14
15 int main(){
16   try{
17     cout << fac(5)<<endl;
18     cout << fac(16)<<endl;
19   }catch(string s){
20       cout << "error: "<<s<<endl;
21   }catch (int i){
22       cout << "error no: "<<i<<endl;
23   }catch (Minimal m){
24       cout << "minimal error"<<endl;
25   }
26
27   cout << fac(0)<<endl;
28 }
```

Fangen unspezifizierter Ausnahmeobjekte

Können verschiedene unterschiedliche Arten von Fehlern geworfen werden, und sollen diese aber nicht unterschiedlich behandelt werden, so läßt sich das in der `catch`-Klausel mit drei Punkten ausdrücken.

Catch3.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class Minimal{};
5
6  int fac(int n){
7      std::string help="negativ argument to fac";
8      if (n<0)  throw help;
9      if (n>20) throw 42;
10     if (n>15) throw Minimal();
11     if (n==0) return 1;
12     return n*fac(n-1);
13 }
14
15 int main(){
16   try{
17     cout << fac(5)<<endl;
```

```
18      cout << fac(16)<<endl;
19    }catch(...){
20      cout << "some error occured"<<endl;
21    }
22    cout << fac(0)<<endl;
23  }
```

3.4.3 Deklaration von Ausnahmen

Damit sich der Aufrufer einer Funktion auf eventuelle Ausnahmefälle einstellen kann, gibt es in C++ die Möglichkeit in der Signatur einer Funktion anzugeben, dass und welche Ausnahmen geworfen werden können. Dies ist insbesondere für Bibliotheken sinnvoll. So weiß der Benutzer der Bibliothek, welche Ausnahmesituationen eventuell auftreten können und kann sich darauf mit entsprechenden `catch`-Klauseln einstellen. Mögliche Ausnahmen werden in der Signatur einer Funktion auch mit dem Schlüsselwort `throw` markiert. In Klammern eingeschlossen folgt die Liste der Typen, die als Ausnahmen auftreten können. Fehlt diese Deklaration in der Signatur, kann jeder Typ eventuell als Ausnahme auftreten. Tritt keine Ausnahme auf, so ist das Klammernpaar nach dem Schlüsselwort `throw` leer zu lassen.

Declare.cpp

```
1  #include <string>
2  #include <iostream>
3
4  int fac(int x) throw(std::string){
5    if (x<0) {
6      std::string msg = "negative Zahl in Fakultaet";
7      throw msg;
8    }
9    if (x==0) return 1;
10   return x*fac(x-1);
11 }
12
13 int main(){
14   try{
15     std::cout << fac(5) << std::endl;
16     std::cout << fac(-5) << std::endl;
17   }catch (std::string s){
18     std::cout << s << std::endl;
19   }
20 }
```

Anders als z.B. in Java überprüft der Compiler nicht die Angaben der Signatur mit den tatsächlich auftretenen Ausnahmen. Allerdings werden Ausnahmen die auftreten und nicht deklariert sind auch nicht mehr zum Abfangen angeboten.

3.4.4 Verträge

Eine Funktion kann als eine Art Vertrag betrachtet werden. Der Kunde ist der Aufrufer der Funktion. Er verpflichtet sich bestimmte Argumente anzuliefern, die vertraglich vereinbahrte

Eigenschaften haben. Zum Beispiel dass das Argument der Fakultätsfunktion keine negative Zahl ist. Der Anbieter der Funktion verspricht, wenn der Kunde seine Auflagen einhält eine bestimmte Leistung zu erbringen.

Die Einhaltung von Vertragsvereinbahrungen sollte überprüft werden. Bei Vertragsverletzungen sollte die Strafe auch entsprechend hart sein. Hierzu bietet C++ eine Funktion `assert` an. In dieser können bool'sche Bedingungen zugesichert werden. Während der Laufzeit wird die Bedingung geprüft und, sollte sie nicht wahr sein, das Programm abgebrochen.

Schreiben wir die Fakultät nun also so, dass die Vorbedingung als Zusicherung geprüft wird:

FacAssert.cpp

```
1  #include <assert.h>
2  #include <iostream>
3
4  int fac(int n){
5    assert(n>=0);
6    if (n==0) return 1;
7    return n*fac(n-1);
8  }
9
10 int main(){
11   std::cout<<fac(5)<<std::endl;
12   std::cout<<fac(-5)<<std::endl;
13 }
```

Die Verletzung der Vorbeding führt während der Laufzeit zu einem Programmabbruch mit entsprechender Fehlermeldung.

```
sep@pc305-3:~/fh/cpp/tutor> ./a.out
120
a.out: src/Assert.cpp:5: int fac(int): Assertion 'n>=0' failed.
Abgebrochen
sep@pc305-3:~/fh/cpp/tutor>
```

Nun ist es sicherlich nicht wünschenswert, wenn das Programm beim Kunden wegen einer solchen Verletzung einer Zusihcerung abbricht. Beim ausgelieferten Programm sollten solche Verletzungen nicht mehr auftreten können und auch nicht mehr geprüft werden, denn schließlich kostet das ja Zeit. Während der Entwicklung hingegen sollten viele solche Bedingungen geprüft werden, damit Programmfehler frühzeitig entdeckt werden und nicht vertuscht werden.

Man kann sich des Präprozessors bedienen, um Zusicherungen nur während der Entwicklungszeit zu prüfen, sie aber beim ausgelieferten Produkt nicht mehr aktiv im Code zu haben. Hierzu kapselt man die Zusicherung in einem `if`-Block des Präprozessor, wobei man sich einen sprechenden Bezeichner ausdenkt, in unserem Fall `DEBUG`.

FacAssertDebug.cpp

```
1  #include <assert.h>
2  #include <iostream>
3
4  int fac(int n){
5  #ifdef DEBUG
6    assert(n>=0);
```

```
7  #endif
8    if (n==0) return 1;
9    return n*fac(n-1);
10 }
11
12 int main(){
13   std::cout<<fac(5)<<std::endl;
14   std::cout<<fac(-5)<<std::endl;
15 }
```

Wird obiges Programm übersetzt und der Bezeichner `DEBUG` ist im Präprozessor definiert, so wird die Zusicherung im Code eingefügt, ansonsten nicht. Der g++ kennt eine einfache Möglichkeit Bezeichner für den Präprozessor auf der Kommandozeile zu definieren: mit der Option -D. Wird obiges Programm mit der Option `-DDEBUG` übersetzt, so wird die zusicherung ausgeführt.

3.5 Konstantendeklarationen

Es lassen sich in C++ Werte als unveränderbar deklarieren.[2] Hierzu dient in C das Schlüsselwort `const`.

3.5.1 Konstante Objektmethoden

In C++ Klassen können Methoden als konstant in Bezug auf das `this`-Objekt deklariert werden. Damit wird gekennzeichnet, dass über den Aufruf der Methode für ein Objekt, dieses Objekt seine Werte nicht ändert. Hierfür wird das Schlüsselwort `const` der Parameterliste nachgestellt.

ConstThis.cpp

```
1  #include <iostream>
2
3  class Const {
4  public:
5    int i;
6
7    void printValue() const {
8      std::cout << i << std::endl;
9    }
10 };
11
12 int main(){
13   Const test;
14   test.i=42;
15   test.printValue();
16 }
```

[2] Dieser Abschnitt konnte aus Zeitgründen in der Vorlesung im SS05 nicht behandelt werden. Unglücklicher Weise wurde das Schlüsselwort `const` schon in vielen vorangehenden Beispielen benutzt.

Im obigen Programm konnte die Methode `printValue` als konstant deklariert werden. Sie gibt lediglich den Wert eines Feldes der Klasse auf der Konsole aus. Das Objekt selbst wird nicht verändert.

Das folgende Programm fügt der Klasse eine zweite Methode zu. Auch diese wird als konstant deklariert, allerdings zu unrecht, denn sie setzt den Wert eines Feldes des Objekts neu.

ConstThisError1.cpp

```
1  #include <iostream>
2
3  class Const {
4  public:
5    int i;
6    void setValue(int j) const {
7      i=j;
8    }
9
10   void printValue() const {
11     setValue(42);
12     std::cout << i << std::endl;
13   }
14 };
```

Der C++-Übersetzer weist dieses Programm zu Recht zurück und gibt die folgende Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ ConstThisError1.cpp
ConstThisError1.cpp: In member function ‘void Const::setValue(int) const’:
ConstThisError1.cpp:7: error: assignment of data-member ‘Const::i’ in read-only
   structure
sep@linux:~/fh/prog3/examples/src>
```

Der Übersetzer führt sehr genau Buch darüber, dass nicht über Umwege durch Aufruf nicht konstanter Methoden innerhalb einer konstanten Methode, das Objekt geändert wird. Auch fogendes Programm führt zu einen Übersetzungsfehler.

ConstThisError2.cpp

```
1  #include <iostream>
2
3  class Const {
4  public:
5    int i;
6    void setValue(int j) {
7      i=j;
8    }
9
10   void printValue() const {
11     setValue(42);
12     std::cout << i << std::endl;
13   }
14 };
```

Jetzt beschwert sich der Übersetzer darüber, dass durch den Aufruf nicht konstanten Methode `setValue` innerhalb der als konstant deklarierten Methode `printValue` zum modifizieren des Objektes führen könnte.

```
sep@linux:~/fh/prog3/examples/src> g++ ConstThisError2.cpp
ConstThisError2.cpp: In member function ‘void Const::printValue() const’:
ConstThisError2.cpp:11: error: passing ‘const Const’ as ‘this’ argument of ‘
   void Const::setValue(int)’ discards qualifiers
sep@linux:~/fh/prog3/examples/src>
```

3.5.2 konstante Parameter

Auch Parameter von Funktionen können in C als konstant deklariert werden. Hierzu ist das Attribut `const` dem Parameternamen voranzustellen. Wenn wir in einem Programm Variablen inklusive des `this`-Zeigers auf die ein oder andere Weise als konstant deklariert haben, dürfen wir die Referenzen nur an Funktionen übergeben, wenn der entsprechende Parameter auch als konstant deklariert wurde.

In der folgenden Klasse ist der Parameter der Funktion `doNotModify` als konstant deklariert. Daher darf diese Funktion mit dem `this`-Zeiger in der konstanten Methode `printValue` aufgerufen werden.

ConstParam.cpp

```
1  #include <iostream>
2  class Const {
3    public:
4      int i;
5      void Const::printValue() const;
6  };
7
8  void doNotModify(const Const* c){
9    std::cout << "in doNotModify" << std::endl;
10 }
11
12 void Const::printValue() const {
13   doNotModify(this);
14   std::cout << i << std::endl;
15 }
16
17 int main(){
18   Const c;
19   c.i=42;
20   c.printValue();
21 }
```

Wenn hingegen der Parameter der Funktion `doNotModify` nicht als konstant deklariert wäre, bekämen wir einen Übersetzungsfehler.

ConstParamError.cpp

```
1  #include <iostream>
2  class Const {
```

```
3  public:
4    int i;
5    void Const::printValue() const;
6  };
7
8  void doNotModify(Const* c){
9    std::cout << "in doNotModify" << std::endl;
10 }
11
12 void Const::printValue() const {
13   doNotModify(this);
14   std::cout << i << std::endl;
15 }
```

Wieder beschwert sich der Übersetzer zu Recht darüber, dass das als konstant deklarierte eventuell verändert werden könnte.

```
sep@linux:~/fh/prog3/examples/src> g++ ConstParamError1.cpp
ConstParamError1.cpp: In member function ‘void Const::printValue() const’:
ConstParamError1.cpp:13: error: invalid conversion from ‘const Const* const’ to
   ‘Const*’
sep@linux:~/fh/prog3/examples/src>
```

Das gilt natürlich nicht nur für den `this`-Zeiger, sondern für beliebig als nicht modifizierbar deklarierte Referenzen. Auch folgendes Programm wird vom Übersetzer zurückgewiesen.

ConstParamError2.cpp

```
1  #include <iostream>
2  class Const {
3  public:
4    int i;
5    void Const::printValue() const;
6  };
7
8  void doModify(Const* c){
9    std::cout << "in doModify" << std::endl;
10 }
11
12 int main(){
13   const Const c();
14   doModify(&c);
15 }
```

Wir erhalten die nun bereits bekannte Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ ConstParamError2.cpp
ConstParamError2.cpp: In function ‘int main()’:
ConstParamError2.cpp:14: error: cannot convert ‘const Const (*)()’ to ‘Const*’
   for argument ‘1’ to ‘void doModify(Const*)’
sep@linux:~/fh/prog3/examples/src>
```

In C++ wird sehr genau über `const` Attribute Buch geführt. Betrachten wir hierzu einmal Objekte der Klasse `Box` aus dem vorherigen Kapitel.

Als Test legen wir jetzt ein Objekt der Klasse `Box<string>` an und speichern einen Zeiger darauf in einer als konstant deklarierten Zeigervariablen und direkt in einer Variablen:

BoxBox.cpp

```
16 #include "Box.h"
17 #include <string>
18 using namespace std;
19
20 int main(){
21   const Box<Box<string>*>* pbbs
22    = new Box<Box<string>*>(new Box<string>("hallo"));
23   const Box<Box<string>*> bbs = *pbbs;
```

Jetzt versuchen wir über diese beiden Variablen drei verschiedenen Modifikationen. Einmal für den Inhalt der inneren Box, dann die Zuweisung eines neuen Inhalts und schließlich die Zuweisung auf die Variable selbst:

BoxBox.cpp

```
24   //allowed
25   pbbs->contents->contents = "welt";
26
27   //not allowed
28   //pbbs->contents = new Box<string>("world");
29
30   //allowed
31   pbbs = new Box<Box<string>*>(new Box<string>("hello"));
```

Für den Zugriff über die Zeigervariable wird unterbunden, einem Feld des referenzierten Objekts einen neuen Wert zuzuweisen. Hingegen wird erlaubt in einem Feld des Objekts referenzierte Objekte zu verändern. Die Konstanzeigenschaft setzt sich also nicht transitiv über alle per Zeiger erreichbare Objekte fort. Vielleicht überraschend, dass der Variablen selbst ein neuer Zeiger zugewiesen werden darf.

Jetzt können wir noch die drei obigen Tests für den direkten Zugriff auf das Objekt durchführen.

BoxBox.cpp

```
32   //allowed
33   bbs.contents->contents = "welt";
34
35   //not allowed
36   //bbs.contents = new Box<string>("world");
37
38   //not allowed
39   //bbs = *pbbs;
40 }
```

Zusätzlich wird verboten, dass der Variablen selbst ein neuer Wert zugewiesen wird.

Prinzipiell ist es zu empfehlen, möglichst wann immer möglich das Attribut `const` in C++ zu setzen, allerdings, wie man oben sieht, kann das recht kompliziert sein, sich immer genau darüber klar zu werden, was eine `const` Deklaration genau aussagt.

3.6 Formen der Typkonversion

Das Typsystem teilt die Daten in unterschiedliche Typen ein. Jede Klassendefinition führt einen neuen Typen ein, aber es gibt natürlich auch die eingebauten primitiven Typen. Zusaätzlich haben wir gesehen, dass Daten noch weitere Einschränkungen bekommen können, wie durch die Konstantendeklaration.

Manchnmal ist es notwendig Daten eines Typen zu konvertieren zu Daten eines anderen Typen. Oft passiert dieses implizit, z.B. wenn eine ganze Zahl vom Typ `int` als Fließkommazahl gebraucht wird.

Implizite Konvertierungen lassen sich, wie zu sehen war, auch implementieren, indem ein einstelliger Konstruktor für eine Klasse definiert wird. Hierzu vergegenwärtige man sich noch einmal das Beispiel auf Seite 2.2.4.

Aus C ist eine Konvertierungsnotation bekannt, indem in runden Klammern der Zieltyp den zu konvertierenden Ausdruck vorangestellt wird. Diese Notation hat auch in C++ weiterhin Gültigkeit. C++ kennt aber zusätzlich vier verschiedene Arten von Typkonvertierungen:

- `static_cast`
- `const_cast`
- `dynamic_cast`
- `reinterpret_cast`

Alle vier Konvertierungsoperationen benutzen die gleiche Syntax, die der Syntax für den Aufruf generischer Funktionen entspricht. Dem Konvertierungsoperator folgt in spitzen Klammern der Zieltyp, dem wiederum folgt in runden Klammern der zu konvertierende Ausdruck.

3.6.1 der Operator

Der Operator `static_cast` entspricht weitgehendst der Typkonversion, wie sie aus C bereits bekannt ist. Statisch soll in diesem Fall bedeuten, dass bereits der Compiler weiß, von welchen auf welchen Typ die Daten konvertiert werden sollen. Dabei kann die Konvertierung beinhalten, sofern entsprechende Umwandlungssequenz bereitgestellt wird:

- Zeiger auf ein Objekt einer Klasse können auf Zeiger einer Unterklasse dieses Objekttyps konvertiert werden
- Die üblichen Umwandlungen auf arithmetischen Typen z.B. `int` in `float` etc.
- Umwandlung von `int` auf einen Aufzählungstypen.
- Konvertierung einer Referenz vom Typ `X&` zu einem anderen Referenztypen `Y&`
- Objekte von einem Typ auf einen anderen Typen umwandeln.

Damit ist die statische Konvertierung sehr ähnlich zu der Typkonvertierung in C.

Beispiel 3.6.1 *Ein paar unterschiedliche Anwendungen statischer Typkonvertierungen zeigt dieses kleine Beispielprogramm.*

SomeCasts.cpp

```
#include <iostream>
class Upper{
 public:
  int i1;
  Upper(int i1):i1(i1){}
};

class Lower:public Upper{
 public:
  int i2;
  Lower(int i1,int i2):Upper(i1),i2(i2){}
};

int main(){
  Upper u1 = static_cast<Upper>(42);
  std::cout<<u1.i1<<std::endl;

  Upper* up = new Lower(17,4);
  Lower* lo = static_cast<Lower*>(up);

  std::cout<<static_cast<double>(2*(lo->i1+lo->i2))/5<<std::endl;
  return 0;
}
```

Die statische Typkonvertierung in C++ ist nicht ganz so frei, wie die Konvertierung in C und verbietet statisch schon einige Konvertierungen. So dürfen beliebige Zeigertypen in C aufeinander konvertiert werden. In C++ darf nur ein sogenannter *Downcast* vorgenommen werden, dh dass die Objekte nur entlang der Ableitungshierarchie der Klassen konvertiert werden dürfen.

Beispiel 3.6.2 *Dieses Beispiel zeigt eine C Konvertierung, die mit dem C++* `static_cast` *nicht gemacht werden darf.*

CCast.cpp

```
#include <iostream>
class O{};

class A{
public:
  char* i;
  A(char* i):i(i){}
  void a(){std::cout<<"A"<<i<<std::endl;}
};

class B:public O{
public: void b(){std::cout<<"B"<<std::endl;}};

int main(){
  O* o = new B();
  A* a = (A*)o;  //cannot be done as static_cast<A*>(o)
```

```
17    a->a();
18    return 0;
19  }
```

Die C++ Version würde mit folgender Fehlermeldung vom Compiler aus auch nachvollziehbaren Gründen zurückgewiesen:
`error: invalid static_cast from type ‘O*’ to type ‘A*’`

Ganz verhindert der C++ Compiler nicht immer, dass ein `static_cast` auf gefährliche Weise eingesetzt wird. Wenn mit einem `static_cast` versucht wird, ein Objekt auf eine Unterklasse zu konvertieren, dieses Objekt real aber nicht von dieser speziellen Unterklasse ist, dann kann je nach Compiler ein unterschiedliches Verhalten beobachtet werden.

Beispiel 3.6.3 *In diesem Beispiel gibt es von der Klasse* `O` *zwei Unterklassen, die Klasse* `A` *und die Klasse* `B`*. Wir erzeugen ein Objekt der Klasse* `B`*, referenzieren es als Objekt der Klasse* `O` *und konvertieren diese Referenz zu einem Objekt der Klasse* `A`*. Es ist aber nie ein Objekt der Klasse* `A` *gewesen. Daher ist nicht definiert, wie sich Operationen auf die Referenz* `a` *verhalten werden.*

StrangeCast.cpp

```
1   #include <iostream>
2   class O{};
3
4   class A:public O{
5   public:
6     char* i;
7     A(char* i):i(i){}
8     void a(){std::cout<<"A"<<i<<std::endl;}
9   };
10
11  class B:public O{
12  public: void b(){std::cout<<"B"<<std::endl;}};
13
14  int main(){
15    B* b = new B();
16    O* o = b;
17    A* a = static_cast<A*>(o);
18    b->b();
19    a->a();
20    std::cout<<"ende"<<std::endl;
21
22    return 0;
23  }
```

Tatsächlich zeigt die Ausgabe, dass das Programm irgendwie überraschend ohne fehlermeldung abbricht:

```
sep@pc305-3:~/fh/cpp> g++ StrangeCast.cpp
sep@pc305-3:~/fh/cpp> ./a.out
B
Asep@pc305-3:~/fh/cpp>
```

3.6.2 der `const_cast` Operator

Der `const_cast` dient dazu, das Konstantenattribut mehr oder weniger zu überlisten.

ConstCast.cpp

```
#include <iostream>
double f(double& d){
  d=2*d;
  return d;
}

double g (const double& d){
  return 2*f(const_cast<double&>(d));
}

int main(){
  double d=2;
  std::cout<<g(d)<<std::endl;
  std::cout<<d<<std::endl;
  return 0;
}
```

3.6.3 der `dynamic_cast` Operator

Die dynamische Typkonvertierung überprüft während der Laufzeit, ob sich ein Objekt konvertieren läßt. Hierzu ist folgende Einschränkungen zu beachten:

- die zu konvertierenden Daten müssen ein Zeiger auf ein Objekt einer Klasse sein, die mindestens eine virtuelle Methode hat.

Das interessante an der dynamischen Konvertierung ist, dass im Rückgabewert gemeldet wird, ob die Konvertierung geglückt ist. Die Rückgabe ist immer ein Zeiger auf ein Objekt. Ist dieser Zeiger 0, so ließ sich das Objekt nicht zu einer gewünschten Referenz konvertieren.

Beispiel 3.6.4 *Auch in diesem Beispiel soll es eine Oberklasse mit zwei erbenden Unterklassen geben. In der Oberklasse muss es eine virtuelle Methode geben, damit eine dynamische Typkonvertierung überhaupt angewendet werden darf. Beide Unterklassen haben eine Methode* `f` *die Oberklasse hingegen nicht.*

DynamicCast.cpp

```
#include <iostream>
class Upper{public: virtual void dummy(){}};

class Lower1: public Upper{
public:
  virtual void f(){std::cout<<"unten1"<<std::endl;}};

class Lower2: public Upper{
public:
  virtual void f(){std::cout<<"unten2"<<std::endl;}};
```

Zum Testen sei eine reihung mit Zeigern auf `Upper`*-Objekten angelegt. Diese können von unterschiedlichen Unterklassen sein. Innerhalb einer* `for`*-Schleife testen wir nun jedes Reihungselement, ob es zu einer der beiden Unterklassen konvertiert werden kann, um dann die Methode* `f` *im Erfolgsfall auf die entsprechende Referenz anzuwenden:*

DynamicCast.cpp

```
11 int main(){
12   Upper* us [ ]= {new Lower1(),new Lower2()};
13   for (int i=0;i<2;i++){
14     Lower1* lo1 = dynamic_cast<Lower1*>(us[i]);
15     if (lo1) lo1->f();
16     Lower2* lo2 = dynamic_cast<Lower2*>(us[i]);
17     if (lo2) lo2->f();
18   }
19
20   return 0;
21 }
```

Javaprogrammieren mögen erkennen, dass ihnen mit der dynamischen Konvertierung ein zum Javaoperator `instanceof` recht ähnliches Konstrukt vorliegt. So läßt sich ein ähnlicher Stil durchführen, wie er oft in der Standardgleichheitsmethode in Java verwendet wird. Prinzipiell ist es möglich beliebige Objekte miteinander zu vergleichen. In der Standardimplementierung der Gleichheit wird zunächst getestet, ob das zweite Objekt von derselben Klasse ist, wie das `this`-Objekt. Im C++ fall heisst das, dass der Parameter dynamische konvertiert werden kann. Wenn dieses der Fall ist, werden die Felder von `this` und dem Parameter `that` paarweise verglichen, ist dieses nicht der Fall, so sollen die beiden Objekte auch nicht gleich sein.

Beispiel 3.6.5 *In diesem Beispiel wird auf typische Weise eine Gleichheit auf beliebigen Objekten definiert:*

Equals.cpp

```
1  #include <iostream>
2  class Object{public:virtual bool equals(Object* that)=0;};
3
4  class Vertex:public Object{
5  public:
6    double x;
7    double y;
8    Vertex(double x,double y):x(x),y(y){}
9
10   bool equals(Object* thatObject){
11     Vertex* that = dynamic_cast<Vertex*>(thatObject);
12     if (that){
13       return this->x==that->x && this->y==that->y;
14     }
15     return false;
16   }
17 };
18
19 class MyString:public Object{
```

```
20 public:
21   std::string str;
22   MyString(std::string str):str(str){}
23   bool equals(Object* thatObject){
24     MyString* that = dynamic_cast<MyString*>(thatObject);
25     if (that) return this->str==that->str;
26     return false;
27   }
28 };
29
30 int main(){
31   MyString hallo = std::string("hallo");
32   Object* o1 = &hallo;
33   Object* o2=new Vertex(17,4);
34   std::cout<<o1->equals(o1)<<std::endl;
35   std::cout<<o2->equals(o2)<<std::endl;
36   std::cout<<o1->equals(o2)<<std::endl;
37   std::cout<<o2->equals(o1)<<std::endl;
38   return 0;
39 }
```

3.6.4 Der `reinterpret_cast` Operator

Der `reinterpret_cast` sei hier nur der Vollständigkeit halber angegeben. Er erlaubt Konvertierungen zwischen Typen, die ansonsten nicht miteinander vereinbar sind. Das Verhalten des `reinterpret_cast` kann dabei durchaus sehr Maschinen und Implementierungsabhängig sein. Insbesondere lassen sich Zahlen als Zeiger und Zeiger als Zahlen und auch Funktionszeiger untereinander konvertieren.

3.7 Zusammenfassung

Kapitel 4

Bibliotheken

4.1 Die *Standard Template Library*

Eine der in allen Anwendungen irgendwann einmal benötigte Funktionalitäten, sind Sammlungen von Objekten. Solche Sammlungen können in Formen von Listen oder auch in Formen von Mengen benötigt werden. Im Laufe des Semesters haben wir schon eine eigene Listenklasse entworfen, mit der eine beliebige Anzahl von Objekten zusammengefasst werden kann. Da Sammlungsklassen für jede Anwendung eine Rolle spielen, gibt es in so gut wie jeder Programmiersprache eine Standardbibliothek, die entsprechende Funktionalität bereit stellt. In C++ steht hierzu, die *standard template library (STL)* zur Verfügung. Wie der Name bereits andeutet, benutzt die *standard template library* nur Schablonenklassen.

Tatsächlich ist die *standard template library* nicht objektorientiert entworfen. Es gibt keinerlei Vererbung innerhalb der Bibliothek. Es gibt keine abstrakten Klasse, Schnittstellen oder virtuellen Methoden.

Vielmehr versucht die *standard template library* Sammlungsklassen möglichst ähnlich zu Reigungen (arrays) zu modellieren. Hierzu betrachten wir einmal, wie durch eine Reihung iteriert werden kann:

ArrayIteration1.cpp

```
1  #include <iostream>
2
3  int main(){
4    int xs [] = {1,2,3,4,5,6,7,8};
5    int* ende = xs + 8;
6
7    for (int* anfang=xs;anfang<ende;anfang++)
8      std::cout<< *anfang <<std::endl;
9
10   return 0;
11 }
```

Da Reihungen nur als Zeiger dargestellt werden, können wir den Zeiger auf den Anfang und auf das Ende einer Reihung leicht ermitteln. Eine for-Schleife kann über den durch Anfang bis Ende markierten Speicherbereich über alle Elemente des Arrays iterieren.

Mit Hilfe von generischen Typen kann die obige for-Schleife verallgemeinert werden.

ArrayIteration2.cpp

```
1  #include <iostream>
2
3  template <typename a>
4  void print(a* anfang,a* ende){
5    for (;anfang<ende;anfang++)
6      std::cout<< *anfang <<std::endl;
7  }
```

Jetzt kann dieselbe for-Schleife für Reihungen mit unterschiedlichen Elementtypen benutzt werden.

ArrayIteration2.cpp

```
8  int main(){
9    int xs [] = {1,2,3,4,5,6,7,8};
10   int* ende = xs + 8;
11   print(xs,ende);
12
13   std::string ys [] = {"frinds","romans","contrymen"};
14   print(ys,ys+3);
15
16   return 0;
17 }
```

Mit einem Trick, kann die Funktion `print` nun so geschrieben werden, dass noch nicht einmal mehr in der Signatur auftaucht, dass zwei Zeiger auf ein bestimmten Elementtypen übergeben werden sollen. Hierzu wird eine Typvariable eingeführt, die für einen Typ steht, der den Iterationsbereich darstellt:

PrintIteration.hpp

```
1  #include <iostream>
2
3  template <typename Iterator>
4  void print(Iterator anfang,Iterator ende){
5    for (;anfang<ende;anfang++)
6      std::cout<< *anfang <<std::endl;
7  }
```

Aus dem Code geht hervor, dass die Funktion `print` für alle Typen *Iterator* benutzt werden kann, für die gilt:

- auf dem Typ *Iterator* existiert ein Vergleich, insbesondere der Operator `<` muss definiert sein.
- auf dem Typ *Iterator* gibt es einfache arithmetische Operationen insbesondere der Operator `++` muß definiert sein.
- auf dem Typ *Iterator* muß eine Dereferenzierung mit dem Sternoperator möglich sein.

Diese drei Eigenschaften sind für Zeigertypen erfüllt. Daher läßt sich die Funktion für zwei übergebene Zeiger anwenden:

PrintIterationArray.cpp

```
#include "PrintIteration.hpp"
int main(){
  int xs [] = {1,2,3,4,5,6,7,8};
  int* ende = xs + 8;
  print(xs,ende);

  std::string ys [] = {"frinds","romans","contrymen"};
  print(ys,ys+3);

  return 0;
}
```

Die *standard template library* stellt nun Klassen bereit, für die es Iterationstypen mit genau diesen drei obigen Eigenschaften gibt. Eine solche Klasse der *standard template library* ist die Klasse `std::vector`. Diese Klasse stellt im klassischen Sinne eine Liste dar. Sie ist generisch über den Elementtypen. Objekte der Klasse `std::vector` haben eine Methode `push_back`, mit der neue Elemente ans Ende der Liste eingefügt werden können.

Mit den Methoden `begin()` und `end()` der Klasse `std::vector` kann der Anfang und Ende des Iterationsbereichs eines Vektorobjekts in Form eines Iterationstypen erfragt werden.

So läßt sich tatsächlich die oben für Reihungen geschriebene Funktion `print` auch für Vektorobjekte benutzen.

PrintIterationVector.java

```
#include <vector>
#include "PrintIteration.hpp"
int main(){
  std::vector<int> xs;
  xs.push_back(1);xs.push_back(2);xs.push_back(3);xs.push_back(4);
  xs.push_back(5);xs.push_back(6);xs.push_back(7);xs.push_back(8);

  print(xs.begin(),xs.end());

  std::vector<std::string> ys;
  ys.push_back("friends");
  ys.push_back("romans");
  ys.push_back("contrymen");

  print(ys.begin(),ys.end());

  return 0;
}
```

Tatsächlich wissen wir an dieser Stelle überhaupt nicht, um was für einen Typ es sich handelt, der uns die Iterationseigenschaft über Vektoren zur Verfügung stellt. Wir wissen vorerst nicht, was der Rückgabetyp der Methoden `begin()` und `end()` ist. Alles, was wir wissen, ist, dass er

die drei Anforderungen erfüllt, die die Methode `print` an den für die Typvariablen `Iterator` substituierten Typ stellt; sprich einen Vergleich, eine Addition und eine Dereferenzierung stehen zur Verfügung.

Es gibt in C++, und das ist eine große Schwäche der Umsetzung von generischen Typen als `template`, keine Möglichkeit explizit auszudrücken, was für Eigenschaften ein Typ erfüllen muss, der für eine bestimmte Variablen eingesetzt wird. Daher geht die *standard template library* den Weg, dieser Information in der Dokumentation eigene Seiten zu widmen. Hierzu wurde der Begriff des Konzeptes eingefügt. Typvariablen werden einem Konzept zugehörig definiert. Ein Konzept beschreibt, welche Operationen für einen Typ definiert sein müssen, der für eine Variablen eines Konzepts eingesetzt wird.

4.1.1 wichtigsten Containerklassen

Die STL stellt primär Behälterklasssen zur Verfügung. Hierzu gehören sowohl listenartige Sequenzen, als auch Mengen und Abbildungen und schließlich auch Strings.

Sequenzen

Sequenzen sind Behältertypen, die die klassischen Listenfunktionalität zur Verfügung stellen:

- sie können dynamische beliebig lang wachsen
- die gespeicherten Elemente haben eine Reihenfolge und können über einen Index angesprochen werden.

Als Implementierung für Sequenzen stehen in der STL die folgenden fünf Klassen zur Verfügung:

- `vector`: ermöglicht Elementzugriff über den Index in konstanter Zeit, Einfügen und Löschen am Ende der Sequenz in konstanter Zeit, Löschen und Einfügen an beliebigen Indexstellen in linearer Zeit.[1]
- `deque`: ähnlich zu `vector` erlaubt aber auch Einfügen und Löschen am Anfang der Sequenz mit konstanten Zeitaufwand.
- `list`: eine vorwärts und rückwärts verlinkte Liste
- `slist`: einfach verkettete Liste, ähnlich, wie wir sie selbst geschrieben haben.
- `bit_vector`: sehr spezialisierter Vektor, in dem nur ein Bit pro Element zur Verfügung steht. Damit als speichereffiziente Lösung für eine Sequenz von bool'schen Werten hervorragend geeignet.

Alle diese Klassen haben die Möglichkeit über `begin()` und `end()` den gesammten Iterationsbereich zu erhalten. Zusätzlich sind Operatoren überladen, insbesondere die eckigen Indexklammern, die von Reihungen bekannt sind.

Beispiel 4.1.1 *Folgendes kleine Programm zeigt den einfachen Umgang mit Sequenzklassen. Man beachte, dass die eigentlichen Typen der Iteratoren oft unbekannt sind. Wir können über diese abstrahieren, indem wir eine generische Funktion für die Iteration schreiben, die wir ohne konkrete Angabe der Typparameter aufrufen.*

[1] Diese Klasse entspricht am ehesten der Klasse `java.util.ArrayList` aus Java.

VectorTest.cpp

```
#include <vector>
#include <list>
#include <string>
#include <iostream>

template <typename Iterator>
void doPrint(Iterator begin,Iterator end){
  std::cout << "[";
  for (Iterator it = begin;it!=end;){
    std::cout << *it;
    it++;
    if (it != end) std::cout << ",";
  }
  std::cout << "]"<<std::endl;
}

int main (){
  std::string words [] ={ "the","world","is","my","oyster"};
  std::vector<std::string> sv;
  std::list<std::string> sl;
  std::cout << sv.capacity() << std::endl;

  for (int i=0;i<5;i++){
    sv.insert(sv.end(),words[i]);
    sl.insert(sl.end(),words[i]);
  }
  std::cout << sv.capacity() << std::endl;

  doPrint(sv.begin(),sv.end());
  sl.reverse();
  doPrint(sl.begin(),sl.end());
}
```

Interessant wäre es doch einmal, den eigentlichen Iteratortypen für eine bestimmte Sammlungsklasse kennenzulernen. Tatsächlich sind dieses Klassen, die innerhalb einer Klasse definiert sind. In der Klasse `std::vector` findet sich so z.B. eine innere Klasse `std::vector::iterator`.

VectorIterator.cpp

```
#include <vector>
#include <iostream>
int main(){
  std::vector<std::string> xs;
  xs.push_back("friends");
  xs.push_back("romans");
  xs.push_back("contrymen");

  for (std::vector<std::string>::iterator it=xs.begin()
      ;it<xs.end()
      ;it++)
```

```
12        std::cout << *it << std::endl;
13
14    return 0;
15  }
```

Wie man sieht, ist es kein schöner Typ und es ist tatsächlich von Vorteil, dass über generische Schablonen, dieser Typ sogar ganz versteckt werden kann.

Mengen und Abbildungen

Die zweite große Gruppe von Behälterklassen in der STL bilden die Abbildungen und Mengen, wobei eine Abbildung als eine Menge von Paaren dargestellt wird. Hierzu steht in ähnlicher Weise, wie wir es schon mehrfach implementiert haben die generische Klasse `pair` zur Verfügung.

Es stehen jeweils vier Klassen für Mengen und Abbildungen zur Verfügung. Für jede dieser Klassen gibt es jeweils zwei Versionen: einmal werden die Elemente in sortierter Reihenfolge gespeichert, einmal über einen Hashcode. Ersteres ist sinnvoll, wenn die Elemente oft sortiert benötigt werden, letzteres, wenn eine schnelle Suche erforderlich ist. Die vier Klassen sind:

- `set`: Mengen mit nur einfachen vorkommen von Elementen.
- `map`: Abbildungen, die für einen Schlüssel einen Werteintrag haben.
- `multiset`: Variante mit Mehrfachauftreten eines Elements.
- `multimap`: Variante mit Mehrfachauftreten eines Eintrags.

Entsprechend gibt es die gleichen Klassen mit dem Hashprinzip unter den Namen: `hash_set`, `hash_map`, `hash_multiset` und `hash_multimap`.

Beispiel 4.1.2 *Folgendes Programm zeigt einen rudimentären Umgang mit Abbildungen. Eine Abbildung von Namen nach Adressen wird definiert. Interessant ist die Überladung des Operators* `[]` *für Abbildungen.*

MapTest.cpp

```
1  #include <map>
2  #include <string>
3  #include <iostream>
4
5  class Address {
6   public:
7    std::string strasse;
8    int hausnummer;
9    Address(){}
10   Address(std::string strasse,int hasunummer)
11         :strasse(strasse),hausnummer(hausnummer){}
12 };
13
14
15 int main(){
16   std::map<std::string, Address>  addressBook;
```

```
17    addressBook["Hoffmann"]=Address("Gendarmenmarkt",42);
18    addressBook["Schmidt"]=Address("Bertholt-Brecht-Platz",1);
19
20    std::cout << addressBook["Hoffmann"].strasse << std::endl;
21
22    addressBook["Tierse"] = Address("Pariser Platz",1);
23
24    std::cout << addressBook["Tierse"].strasse << std::endl;
25    std::cout << addressBook["Andrak"].strasse << std::endl;
26  }
```

Man beachte, dass die Klasse `map` über zwei Typen parameterisiert ist:

- der Typ des Suchschlüssels
- der Typ des assoziierten Wertes

Arbeiten mit String

Auch die Klasse `string` enthält die wesentlichen Eigenschaften einer Behälterklasse. Ein String kann als eine Art Vektor mit Zeichen verstanden werden, also ähnlich dem Typ `vector<char>`. Entsprechend gibt es für einen String auch die Iteratoren, die Anfang und Ende eines Strings bezeichnen. Da die meisten Algorithmen auf Iteratoren definiert sind, lassen sich diese auch für Strings aufrufen.

Beispiel 4.1.3 *Folgendes kleine Programm demonstriert einen sehr rudimentären Umgang mit Strings.*

Pallindrom.cpp

```
1   #include <string>
2   #include <iostream>
3
4   using namespace std;
5
6   template <typename t>
7   bool isPallindrome(t w){
8     t w2 = w;
9     reverse(w2.begin(),w2.end());
10    return w==w2;
11  }
12
13  int main(){
14    cout << isPallindrome<string>("rentner")  << endl;
15    cout << isPallindrome<string>("pensioner")<< endl;
16    cout << isPallindrome<string>("koblbok")  << endl;
17    cout << isPallindrome<string>("alexander")<< endl;
18    cout << isPallindrome<string>("stets")    << endl;
19    cout << isPallindrome<string>
20                      ("ein neger mit gazellezag tim regen nie")
```

```
21              << endl;
22    return 0;
23  }
```

4.1.2 Algorithmen

Wie wir gesehen haben, enthalten die unterschiedlichen Container-Klassen der STL nur die wesentlichen Methoden, die sich mit der Iteration über der Elemente des Elements beschäftigen. Weitere Algorithmen, die auf Containerklassen anzuwenden sind, wie z.B. auch eine Sortierung, befinden sich nicht in diesen Klassen, sondern sind separat als generische Funktionen implementiert. Hierzu stellt die STL separate Algorithmen zur Verfügung, welche mit `#include <algorithm>` zu inkludieren sind.

Hier finden sich eine Vielzahl interessanter Algorithmen, insbesondere auch soche höherer Ordnung. Zum Beispiel findet sich eine Funktion `for_each`, die eine Funktion auf jedes Element eines Iterationsbereiches anwendet:

ForEachVector.cpp

```
1  #include <vector>
2  #include <algorithm>
3  #include <iostream>
4
5  void doppel(int& x){x=2*x;}
6
7  int main(){
8    std::vector<int> v = std::vector<int>(3);
9    v[0] = 17; v[1]=28; v[2] = 21;
10   std::for_each(v.begin(),v.begin()+3,doppel);
11
12   for (int i= 0;i<3;i=i+1){
13     std::cout << v[i] << std::endl;
14   }
15   return 0;
16 }
```

Viele Algorithmen für Behäterobjekte sind wie `for_each` höherer Ordnung, indem Sie eine Funktion als Parameter übergeben bekommen.

Funktionsobjekte

Da viele Algorithmen der STL höherer Ordnung sind, werden Funktionsobjekte oft benötigt. In der STL sind einige einfache oft benötigte Funktionsobjekte bereits vordefiniert. So gibt es z.B. für die arithmetischen Operatoren Klassen, die Funktionsobjekte der entsprechenden Operation darstellen. Die entsprechenden Klassen können mit der Unterbibliothek `functional` inkludiert werden.

Wir treffen in der STL einige Bekannte aus den letzten Abschnitten. So gibt es das Funktionsobjekt `compose1` (allerdings bisher nur in der Erweiterung der STL von silicon graphics (`www.sgi.com`)), das die Komposition zweier Funktionen darstellt.

Eine weitere interessante Funktion ist `bind1st` , mit der das sogenannte *currying* praktiziert werden kann.

Beispiel 4.1.4 *Folgendes Programm demonstriert Currying in der STL mit der Funktion* `bind1st`.

STLCurry.cpp

```
#include <functional>
#include <iostream>

int main(){
  std::cout <<bind1st(std::plus<int>(),17)(25) <<std::endl;
  return 0;
}
```

Beispiel 4.1.5 *Folgendes Programm demonstriert die Benutzung der Klasse* `plus`, *die ein Funktionsobjekt für die Addition darstellt in Zusammenhang mit einer Funktion höherer Ordnung. Mit der STL-Funktion* `transform` *werden die Elemente zweier Reihungen paarweise addiert.*

TransformTest.cpp

```
#include <algorithm>
#include <functional>
#include <iostream>

using namespace std;

void print (int a){cout << a <<",";}

int main(){
  int xs [] = {1,2,3,4,5};
  int ys [] = {6,7,8,9,10};
  int zs [5];
  transform(xs,xs+5,ys,zs,plus<int>());
  for_each(zs,zs+5,print);
  cout << endl;
  return 0;
}
```

Beispiel 4.1.6 *Selbstredend bietet die STL auch Funktionen zum Sortieren von Behälterobjekten an. Die Sortiereigenschaft kann dabei als Funktionsobjekt übergeben werden. Im folgenden Programm wird nach verschiedenen Eigenschaften sortiert.*

SortTest.cpp

```
#include <vector>
#include <list>
#include <string>
#include <iostream>
```

Zunächst definieren wir eine Klasse, für die kleiner Relation, die sich nur auf die Länge von Strings bezieht.

SortTest.cpp

```
5  class ShorterString{
6    public:
7      bool operator()(std::string s1,std::string s2){
8        return s1.length()<s2.length();
9      }
10 };
```

Die nächste Kleinerrelation vergleicht zwei Strings rückwärts gelesen in der lexikographischen Ordnung:

SortTest.cpp

```
11 bool reverseOrder(std::string s1,std::string s2){
12   reverse(s1.begin(),s1.end());
13   reverse(s2.begin(),s2.end());
14   return s1<s2;
15 }
```

Mit folgenden Hilfsmethoden werden wir die Behälterobjekte auf dem Bildschirm ausgeben:

SortTest.cpp

```
16 void print(std::string p){
17   std::cout << p << ",";
18 }
19
20 template <typename Iterator>
21 void printSequence(Iterator begin,Iterator end){
22   if (begin==end) {
23     std::cout << "[]" << std::endl;return;
24   }
25   std::cout << "[";
26   for_each(begin,end-1,print);
27   std::cout << *(end-1) << "]"<<std::endl;
28 }
```

Eine Reihung von Wörtern wird sortiert. Erst nach der Länge der Wörter, dann nach den rückwärts gelesenen Wörtern (sich reimende Wörter stehen so direkt untereinander) und schließlich nach der Standardordnung, der lexikographischen Ordnung:

SortTest.cpp

```
29 int main (){
30   int l = 8;
31   std::string words []
32    ={"rasiert","schweinelende","passiert","legende"
33     ,"schwestern","verlustiert","gestern","stern"};
34
35   std::sort(words,words+l,ShorterString());
36   printSequence(words,words+l);
37
```

```
38    std::sort(words,words+l,reverseOrder);
39    printSequence(words,words+l);
40
41    std::sort(words,words+l);
42    printSequence(words,words+l);
43  }
```

Und tatsächlich hat das Programm die erwartete Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> g++ -o SortTest SortTest.cpp
sep@linux:~/fh/prog3/examples/src> ./SortTest
[stern,rasiert,legende,gestern,passiert,schwestern,verlustiert,schweinelende]
[legende,schweinelende,stern,gestern,schwestern,rasiert,passiert,verlustiert]
[gestern,legende,passiert,rasiert,schweinelende,schwestern,stern,verlustiert]
sep@linux:~/fh/prog3/examples/src>
```

4.2 Gui-Programmierung

Es ist gewiss kein Zufall, dass objektorientierte Sprachen genau in dem Moment weitere Verbreitung fanden, als graphische Benutzeroberflächen mehr und mehr gefragt waren und schließlich zum Standard wurden. Hier kann die Objektorientierung ihre Stärken voll ausschöpfen: jede graphische Komponente ist ein eigenes Objekt. Eine Bibliothek läßt sich leicht um weitere Klassen, die weitere graphische Kompoenenten beschreiben, erweitern.

Eine graphische Benutzeroberfläche ist wie geschaffen für objektorientierte Programmierung. Graphische Komponenten sind leicht identifizierbare Objekte, für die es sich anbietet eigene Klassen vorzusehen. Eigenschaften, die allen graphischen Komponenten zu eigen sind, wie z.B. ihre Dimension, lassen sich gut in einer gemeinsamen Oberklasse beschreiben.

Es gibt eine Reihe verschiedener objektorientierter GUI-Bibliotheken für C++. Die zwei gebräuchlichsten weitestgehendst platformunabhängigen dürften Qt (`http://www.trolltech.com/products/qt/index.html`) und Gtkmm (`http://www.gtkmm.org/`) sein. Während die Benutzeroberfläche KDE mit Hilfe von Qt implementiert ist, ist die Benutzeroberfläche Gnome auf Gtk basiert. Gtkmm ist ein C++-Port für die in C geschriebene GUI Biblithek gtk+ (`http://www.gtk.org/`) . Eine weitere interessante GUI-Bibliothek für C++ ist wxWidget (`http://www.wxwindows.org/`) . Sie setzt auf einer etwas höheren Abstraktionsebene an. Auch `wxWidget` hat eine auf Gtk+ basierende Implementierung.

Natürlich stellt auch die Firma Microsoft eine Bibliothek zur Programmierung graphischer Oberflächen speziell für das von der Firma vertriebene Betriebssystem bereit.

Im folgenden sollen einige Grundlagen der Programmierung graphischer Oberflächen anhand der Bibliothek Qt gezeigt werden. Die Einführung wird stark Beispiel getrieben sein.

4.2.1 Widgets in Fenstern öffnen

Graphische Oberflächen bestehen zunächst einmal aus zumeist sichtbaren graphischen Komponenten, den sogenannten *Widgets*. Hierbei handelt es sich um ein Kunstwort, das aus *window gadget* entstanden ist.

Die Bibliothek Qt hält viele Klassen bereit, die jeweils ein bestimmtes Widget implementieren. Ein einfaches und gängiges Widget ist dabei sicher ein Knopf. Hierfür kennt Qt die Klasse `QPushButton`.

Jedes Widget kann in Qt in einer Anwendung in einen Fenster dargestellt werden. Hierzu ist ein Objekt der Klasse `QApplication` zu erzeugen und schließlich auf diesem die Methode `exec` aufzurufen. Damit Qt ein Widget in einem Fenster darstellt, ist für dieses Widget die Methode `show` aufzurufen

Beispiel 4.2.1 *Ein allererstes Qt-Programm. Ein Qt-Applikationsobjekt wird mit den Kommandozeilenparametern erzeugt. Dann wird ein Knopfobjekt erzeugt, auf diesen die Methode* `show` *aufgerufen und dann die Anwendung gestartet.*

knopp1.cpp

```
1  #include <QPushButton>
2  #include <QApplication>
3
4  int main(int argc,char** argv){
5    QApplication app(argc,argv);
6    QPushButton* b=new QPushButton("do not push this button");
7    b->show ();
8    return app.exec();
9  }
```

Und tatsächlich öffnet diese Applikation ein Fenster mit einen Knopf, wie es in der Abbildung 4.1 zu sehen ist.

Abbildung 4.1: Ein erstes Widget wird sichtbar.

Beispiel 4.2.2 *Dieses Spielchen läßt sich natürlich mehrfach treiben. Sollen in einer Applikation zwei Fenster mit je einen Knopf geöffnet werden, so sind halt zwei Knöpfe zu instanziieren und sichtbar zu machen.*

doppelknopp.cpp

```
1  #include <QPushButton>
2  #include <QApplication>
3
4  int main(int argc,char** argv){
5    QApplication app(argc,argv);
6    QPushButton* b1=new QPushButton("do not push this button");
7    b1->show ();
8    QPushButton* b2=new QPushButton("better push this button");
9    b2->show ();
10   return app.exec();
11 }
```

4.2.2 Widgets zu komplexeren Widgets zusammenfassen

In der Regel soll nicht jedes Widget in einem eigenen Fenster dargestellt werden, sondern mehrere Widgets in einem Fenster zusammengefasst werden. Hierzu gibt es Layoutklassen in Qt, die es erlauben mehrere Widgets zusammenzufassen. Dabei gibt es die Möglichkeit diese vertikal oder horizontal anordnen zu lassen.

Beispiel 4.2.3 *Im folgenden Programm werden wieder zwei Knöpfe definiert. Diese beiden werden dann in einem* `QVBoxLayout` *zusammengefasst. Damit werden in einem Fenster zwei Knöpfe vertikal, also untereinander, angezeigt. Hierzu ist ein neues Widget zu definieren, das das Layout mit den zwei Knöpfen erhält.*

knopp2.cpp

```
1  #include <QApplication>
2
3  #include <QPushButton>
4  #include <QVBoxLayout>
5  #include <QWidget>
6
7  int main(int argc,char** argv){
8    QApplication app(argc,argv);
9    QPushButton* b1=new QPushButton("do not push this button");
10   QPushButton* b2=new QPushButton("better push this button");
11   QVBoxLayout* mainLayout = new QVBoxLayout();
12   mainLayout->addWidget(b1);
13   mainLayout->addWidget(b2);
14
15   QWidget* mywidget=new QWidget();
16
17   mywidget->setLayout(mainLayout);
18   mywidget->show();
19   return app.exec();
20 }
```

Und tatsächlich öffnet diese Applikation ein Fenster mit zwei Knöpfen, wie es in der Abbildung 4.2 zu sehen ist.

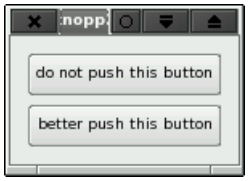

Abbildung 4.2: Ein Fenster mit zwei Widgets als Inhalt.

Eleganter ist es, eine neue Widgetklasse zu definieren, die die Widgets, die zusammengefasst werden sollen, als Felder enthält. Hierzu wird einfach eine Unterklasse der Klasse `QWidget` definiert.

Beispiel 4.2.4 *Statt in einer* `main`*-Funktion zwei Knöpfe vertikal in einem neuen Widget zusammenzufassen, wird eine neue Widgetklasse definiert. Sie enthält zwei Knöpfe und das entsprechende Layout.*

knoppwidget.hpp

```
1  #include <QPushButton>
2  #include <QVBoxLayout>
3  #include <QWidget>
4
5  class KnoppWidget:public QWidget{
6  public:
7    QPushButton* b1;
8    QPushButton* b2;
9    QVBoxLayout* mainLayout;
10   KnoppWidget();
11   virtual ~KnoppWidget();
12 };
```

Der Code aus der `main`*-Funktion wandert nun weitgehendst in den Konstruktor:*

knoppwidget.cpp

```
1  #include "knoppwidget.hpp"
2  #include <QApplication>
3  KnoppWidget::KnoppWidget(){
4    b1=new QPushButton("do not push this button");
5    b2=new QPushButton("better push this button");
6    mainLayout = new QVBoxLayout();
7    mainLayout->addWidget(b1);
8    mainLayout->addWidget(b2);
9
10   setLayout(mainLayout);
11 }
```

Zusätzlich läßt sich ein Destruktor schreiben, der die im Konstruktor erzeugten Objekte wieder löscht.

knoppwidget.cpp

```
12 KnoppWidget::~KnoppWidget(){
13   hide();
14   delete mainLayout;
15   delete b1;
16   delete b2;
17 }
```

Schließlich braucht nur noch eine Instanz des neuen Widgets erzeugt zu werden und sichtbar gemacht werden:

knoppwidget.cpp

```
18 int main(int argc,char** argv){
19   QApplication app(argc,argv);
```

```
20    KnoppWidget* kw= new KnoppWidget();
21    kw->show();
22
23    return app.exec();
24  }
```

4.2.3 Ereignisbehandlung

Das A und O einer graphischen Benutzeroberfläche ist natürlich, dass nicht nur Widgets angezeigt werden, sondern auch auf Benutzerinteraktion reagiert werden kann. Es muß also eine Möglichkeit geben, auf bestimmte Ereignisse reagieren zu können. Hierin, wie dieses zu programmieren ist, unterscheiden sich die unteschiedlichen GUI-Bibliotheken. Es gibt Bibliotheken, in denen bestimmte Methoden der Basisklasse überschrieben werden, die ausgefphrt werden, wenn ein bestimmtes Ereignis auftritt. So gibt es in Gtkmm eine Methode in dem zugehörigen Widget für Knöpfe, die ausgeführt wird, wenn der Knopf gedrückt wird. Schreibt man eine Unterklasse der entsprechenden Widgetklasse, in der diese Methode überschrieben wird, so läßt sich damit die Reaktion auf den Knopfdruck eines speziellen Knopfes spezifizieren.

Eine andere Möglichkeit sind Funktionen höherer Ordnung. Hierzu übergibt man einem Knopf eine sogenannte *callback* Funktion. Dieses ist ein Zeiger auf die Funktion, die ausgeführt wird, wenn der Knopf gedrückt wird.

Qt hat einen eigen Mechanismus jenseits von C++ entwickelt, um Ereignisse zu programmieren. Hierbei werden sogenannte Signale und Slots definiert. Signale und Slots sind keine Begriffe aus C++ und nicht Bestandteil der Sprache C++, sondern quasi eine Qt-Erweiterung. Daher ist die Übersetzung eines Qt-Projektes auch mit dem Programm `qmake` zu bewerkstelligen. Dieses ist ein eigener Präprozessor von Qt, der ganze Klasse zusätzlich generiert.

Signale sind in Qt die Ereignisse, auf die reagiert werden soll. Ein typisches Signal ist das Drücken eines Knopfes. Slots sind die Methoden, die ausgeführt werden sollen, wenn ein bestimmtes Signal aufgetreten ist. Hierzu sind Signale mit Slots zu verbinden.

Beispiel 4.2.5 *Das Widget aus dem vorherigen Beispiel ergänzen wir im zwei Slotmethoden. Diese sollen ausgeführt werden wenn einer der beiden Knöpfe gedrückt wird. Es ist dabei unbedingt notwendig zu beginn der Klassendefinition das Wort* `Q_OBJECT` *einzufügen. Dieses signalisiert Qt, dass hier eine Klasse mit Signalen und Slots arbeitet und dafür Code zu generieren ist.*

knoppwidget2.hpp

```
1   #include <QPushButton>
2   #include <QVBoxLayout>
3   #include <QWidget>
4
5   class KnoppWidget:public QWidget{
6   Q_OBJECT
7   public:
8     QPushButton* b1;
9     QPushButton* b2;
10    QVBoxLayout* mainLayout;
11    KnoppWidget();
12    virtual ~KnoppWidget();
```

```
13 public slots:
14   void execute1();
15   void execute2();
16 };
```

Zunächst seien die beiden Slotmethoden ganz simple implementiert, so dass sie eine Ausgabe auf der Kommandozeile tätigen:

knoppwidget2.cpp

```
1  #include "knoppwidget2.hpp"
2  #include <QApplication>
3  #include <iostream>
4
5  void KnoppWidget::execute1(){
6    std::cout<<"I told you not to push the button"<<std::endl;
7  }
8
9  void KnoppWidget::execute2(){
10   std::cout<<"so thats all that happens"<<std::endl;
11 }
```

Im Konstruktor werden zunächst zwei Knöpfe angelegt:

knoppwidget2.cpp

```
12 KnoppWidget::KnoppWidget(){
13   b1=new QPushButton("do not push this button");
14   b2=new QPushButton("oh yeah push this button");
```

Dann werden jeweils die in der Klasse `QPushButton` *vordefinierten Signale* `pressed` *mit den Slotmethoden verbunden. Hierzu gibt es die Qt-Funktion* `connect`*. Sie hat vier Parameter, jeweils die Objektreferenz und die Signal- bzw. Slotmethoden.*

knoppwidget2.cpp

```
15   connect(b1, SIGNAL(pressed(void)),this, SLOT(execute1(void)));
16   connect(b2, SIGNAL(pressed(void)),this, SLOT(execute2(void)));
```

Ansonsten ändert sich nichts im Vergleich zum vorherigen Beispiel:

knoppwidget2.cpp

```
17   mainLayout = new QVBoxLayout();
18   mainLayout->addWidget(b1);
19   mainLayout->addWidget(b2);
20
21   setLayout(mainLayout);
22 }
23
24 KnoppWidget::~KnoppWidget(){
25   hide();
26   delete mainLayout;
```

```
27    delete b1;
28    delete b2;
29  }
30
31  int main(int argc,char** argv){
32    QApplication app(argc,argv);
33    KnoppWidget* kw= new KnoppWidget();
34    kw->show();
35
36    return app.exec();
37  }
```

Damit reagieren jetzt die Knöpfe, indem beim Drücken der Knöpfe eine Ausgabe auf der Kommandozeile getätigt wird.

Damit sind wir jetzt in der Lage, erste GUI-Programme zu schreiben. Bereits um die Stärken der späten Bindung zu demonstrieren, wurde eine GUI-Klasse benutzt, die Klasse `Dialogue`. Ein `Dialogue` ist ein Widget, das sich aus drei Widgets zusammensetzt: einen Eingabefeld der Klasse `QLineEdit`, einen Knopf der Klasse `QPushButton` und einem Ausgabefeld der Klasse `QLabel`. Die Klasse `Dialogue` erhält zusätzlich eine Referenz auf ein `DialogueLogic`-Objekt und eine Slotmethode, die ausgeführt werden soll, wenn der Knopf gedrückt wird:

Dialogue.hpp

```
1   #ifndef DIALOGUE_H_
2   #define DIALOGUE_H_
3
4   #include <QWidget>
5   #include <QPushButton>
6   #include <QLabel>
7   #include <QLineEdit>
8
9   #include "DialogueLogic.hpp"
10
11  class Dialogue:public QWidget{
12  Q_OBJECT
13  public:
14    Dialogue(DialogueLogic* logic);
15    QLabel* output;
16    QLineEdit* input;
17    QPushButton* b;
18    DialogueLogic* logic;
19
20    static int run(DialogueLogic* logic,int argc,char** argv);
21  public slots:
22    void execute();
23  };
24  #endif
```

Im konstruktor dieser Klasse werden die einzelnen Widgets instanziiert und das Knopfsignal mit der Slotmethode verbunden:

Dialogue.cpp

```
1  #include "Dialogue.hpp"
2  #include <QApplication>
3  #include <QVBoxLayout>
4  Dialogue::Dialogue(DialogueLogic* logic):logic(logic){
5    output=new QLabel();
6    input=new QLineEdit();
7    b=new QPushButton(logic->description().c_str());
8
9    QVBoxLayout* mainLayout = new QVBoxLayout();
10   mainLayout->addWidget(input);
11   mainLayout->addWidget(b);
12   mainLayout->addWidget(output);
13   setLayout(mainLayout);
14   connect(b   , SIGNAL(pressed(void))
15           ,this, SLOT  (execute(void)));
16 }
```

Die Slotmethode liest den Text aus dem Eingabefeld, nimmt diesen als Eingabe für die Methode `eval` des `DialogueLogic`-Objektes und schreibt deren Ergebnis in das Ausgabefeld:

Dialogue.cpp

```
17 void Dialogue::execute(){
18   output->setText
19     (QString(logic->eval(input->text().toStdString()).c_str()));
20   this->repaint();
21 }
```

Eine statische Methode zum Öffnen des Dialogs mit einem entsprechenden Logikobjekt wird für die einfache Benutzung bereitgestellt.

Dialogue.cpp

```
22 int Dialogue::run(DialogueLogic* logic,int argc,char** argv){
23   QApplication app(argc,argv);
24   Dialogue* dia=new Dialogue(logic);
25   dia->resize(100, 30);
26   dia->show ();
27   return app.exec();
28 }
```

Ein Beispiel zur Benutzung der Klasse `Dialogue` war bereits auf Seite 2.1.2 zu sehen.

4.2.4 Zeichnen von Graphiken

Bisher haben wir fertige Widgetklassen aus Qt benutzt, wie `QPushButton` oder `QLabel`, oder aber fertige Widgets mit einem Layout in einer neuen Widgetklasse zusammengefügt. Manchmal ist es wünschenswert ein eigenes Widget zu kreieren, das eine eigene optische Ausprägung hat. Es soll also eine eigene Graphik im Widget gezeichnet werden.

Dieses läßt sich in Qt recht einfach realisieren. In der Klasse `QWidget` definiert eine Methode `paintEvent`, wie das Widget zu zeichnen ist. Es läßt sich diese Methode in einer Unterklasse von `QWidget` überschreiben.

Beispiel 4.2.6 *Es sei eine simple Unterklasse von* `QWidget` *definiert, in der die Methode zum Zeichnen des Widgets überschrieben ist:*

PaintArea.hpp

```
#include <QWidget>
class PaintArea:public QWidget {
  protected:
    virtual void paintEvent(QPaintEvent *event);
};
```

Für ein Widget läßt sich ein `QPainter`*-Objekt instanziieren. Diesem kann ein* `QPen` *hinzugefügt werden. Dann gibt es eine Vielzahl von Zeichenmethoden auf dem* `QPainter`*-Objekt. In diesem Beispiel wird ein einfaches Viereck gezeichnet:*

PaintArea.cpp

```
#include "PaintArea.hpp"
#include <QPainter>
#include <QPen>

void PaintArea::paintEvent(QPaintEvent *){
  QPainter painter(this);
  QPen pen;

  painter.setPen(pen);
  painter.drawRect(10,20,110,80);
}
```

In gewohnter Weise kann dieses neue Widget in einem Fenster dargestellt werden:

TestPaintArea.cpp

```
#include "PaintArea.hpp"
#include <QApplication>

int main(int argc,char **argv){
  QApplication application(argc,argv);
  PaintArea* pa = new PaintArea();
  pa->show();
  return application.exec();
}
```

4.2.5 Bewegen von Graphiken

In diesem Abschnitt sollen zwei neue Ideen realisiert werden. Zum einen sollen Bilddateien in einem Widget angezeigt werden, zum anderen sollen sich diese Bilddateien im zweidimensionalen Raum bewegen können.

Hierzu brauchen wir zunächst eine Klasse, die eine Bild-Objekt mit einer Position und einem Bewegungsvektor darstellt. Für Bildobjekte gibt es in Qt die Klasse `QImage`. Es sei eine Klasse entworfen, die fünf Felder beinhaltet:

- das darzustellende Bildobjekt
- die x- und y-Koordinate des Bildursprungs im zweidimensionalen Raum
- die Bewegungsrichtung für die beiden dimensionen im Raum

Die Methode `move` soll angeben, wie in einem Bewegungsschritt sich die Koordinaten des Bildursprungs ändern:

MovingImage.hpp

```
1  #ifndef MOVING_IMAGE_HPP
2  #define MOVING_IMAGE_HPP
3  #include <QPainter>
4  #include <QString>
5  class MovingImage{
6  public:
7    double x;
8    double y;
9    double dX;
10   double dY;
11   QImage img;
12   MovingImage(QString imageFile
13               ,double x=0,double y=0
14               ,double dX=0,double dY=0);
15   virtual ~MovingImage();
16   virtual void move();
17   virtual void paintMe(QPainter& p);
18 };
19 #endif
```

Die Implementierung dieser Klasse ist recht simpel. Ein einfacher Konstruktor, eine naheliegende Implementierung der Methode `move` und die Methode `paintMe`, die auf dem `QPainter`-Objekt die Methode `drawImage` aufruft:

MovingImage.cpp

```
1  #include "MovingImage.hpp"
2  MovingImage::MovingImage
3    (QString imageFile,double x,double y,double dX,double dY):
4       x(x),y(y),dX(dX),dY(dY),img(imageFile){};
5
6  MovingImage::~MovingImage(){};
7
8  void MovingImage::move(){
9    x+=dX;
10   y+=dY;
11 }
12
```

```
13 void MovingImage::paintMe(QPainter& p){
14   p.drawImage(QPointF(x,y),img);
15 }
```

Bewegung auf Knopfdruck

Das Bildobjekt soll nun in einem Fenster dargestllt werden. Hierzu sei eine neue Widgetklasse definiert, die ein `MovingImage`-Objekt enthält. In dieser sei die methode `paintMe` überschrieben und eine Slotmethode definiert, die auszuführen ist, wenn ein Bewegungsschritt auszuführen ist:

AnimatedPane.hpp

```
1  #include <QWidget>
2  #include "MovingImage.hpp"
3
4  class AnimatedPane:public QWidget {
5  Q_OBJECT
6  public:
7    MovingImage* mi;
8    AnimatedPane(MovingImage* mi);
9    void paintEvent(QPaintEvent *);
10
11 public slots:
12   virtual void tick();
13 };
```

Auch die Implementierung dieser Klasse ist in sich sehr einfach:

AnimatedPane.cpp

```
1  #include "AnimatedPane.hpp"
2  #include <QPaintEvent>
3  #include <QPainter>
4
5  AnimatedPane::AnimatedPane(MovingImage* mi):mi(mi){};
6
7  void AnimatedPane::paintEvent(QPaintEvent *){
8    QPainter painter(this);
9    mi->paintMe(painter);
10 }
11
12 void AnimatedPane::tick(){
13   mi->move();
14   this->repaint();
15 }
```

Um die erste Animation zu bewerkstelligen, wird in einer Applikation ein `AnimatedPane`- und ein `QPushButton`-Objekt erzeugt. Das Signal des Knopfes wird mit der Slotmethode der Animation verbunden.

TestButtonAnimatedPane.cpp

```
#include "AnimatedPane.hpp"
#include <QApplication>
#include <QPushButton>

int main(int argc,char **argv){
  QApplication application(argc,argv);
  QPushButton b("Move");
  b.show();

  QString fileName =argc<1ïestbild.png":argv[1];

  AnimatedPane* p
   =new AnimatedPane(new MovingImage(fileName,0,0,2,1));
  p->show();

  QObject::connect(&b,SIGNAL(clicked()),p,SLOT(tick()));
  return application.exec();
}
```

Wir erhalten eine Applikation, in der sich auf Knopfdruck das Bild innerhalb des Fensters bewegt.

Zeitgesteuerte Bewegung

Es ist natürlich etwas mühsam permanent auf einen Knopf drücken zu müssen, um eine Animation im Gang zu halten. Daher wäre es schön, wenn uns jemand diese Arbeit abnimmt. Am besten wäre ein Knopf, der sich selbst immer wieder drückt. Dieser Knopf bräuchte dann auch nicht sichtbar zu sein.

Qt bietet eine Klasse an, die eine Art unsichtbaren Knopf, der sich selbst in bestimmten Abständen drückt, darstellt. Dieses ist die Klasse `QTimer`. Ebenso wie ein `QPushButton`-Objekt löst ein `QTimer`-Objekt Signale aus, die mit Slotmethoden verbunden werden können. Diese Signale werden in einem vorgegebenen Zeitintervall ausgelöst. Damit läßt sich jetzt eine automatische Animation definieren. Diese braucht zusätzlich ein Feld für ein `QTimer`-Objekt:

TimeAnimatedPane.hpp

```
#include <QTimer>
#include "../buttonAnimated/AnimatedPane.hpp"

class TimeAnimatedPane:public AnimatedPane {
Q_OBJECT
public:
  TimeAnimatedPane(MovingImage* mi);
  QTimer timer;
};
```

Im Konstruktor wird das Signal des `QTimer`-Objekts vernbunden mit der Slotmethode des `AnimatedPane`-Objektes. Anschllließend wird das `QTimer`-Objekt gestartet mit einem Signalintervall von 25ms.

TimeAnimatedPane.cpp

```
#include "TimeAnimatedPane.hpp"
TimeAnimatedPane::TimeAnimatedPane(MovingImage* mi)
   :AnimatedPane(mi){
  connect(&timer,SIGNAL(timeout()),this,SLOT(tick()));
  timer.start(25);
};
```

Auch dieses Widget läßt sich nun in der gewohnten Weise in einem Fenster darstellen.

TestTimeAnimatedPane.cpp

```
#include "TimeAnimatedPane.hpp"
#include <QApplication>
#include <QPushButton>

int main(int argc,char **argv){
  QApplication application(argc,argv);
  QString fileName =argc<1?"testbild.png":argv[1];
  TimeAnimatedPane* p
   =new TimeAnimatedPane(new MovingImage(fileName,0,0,2,1));
  p->show();
  return application.exec();
}
```

Aufgabe 11 Lassen Sie jetzt zusätzlich die Klasse `GeometricObject` folgende Schnittstelle implementieren:

MoveableObject.hpp

```
#ifndef MOVEABLE_OBJECT_HPP
#define MOVEABLE_OBJECT_HPP
class MoveableObject{
public:
    virtual void move()=0;
};
#endif
```

Die Methode `move` soll für eine geometrische Figur die Position der Figur verändern, wenn durch einen äußeren Tick das Objekt aufgefordert wird, sich zu bewegen. Sehen sie hierzu Felder `double dX` und `double dY` in der Klasse `GeometricObject` vor, die angeben, um wieviel sich ein Objekt in `x`- bzw in `y`-Richtung bei einem Aufruf von `move` bewegen soll.

Aufgabe 12 Die folgende neue Version der Klasse `Board` ist jetzt in der Lage, eine Liste von geometrischen Figuren animiert darzustellen:

Board.hpp

```
#ifndef BOARD_H_
#define BOARD_H_

```

```
4  #include <QWidget>
5  #include <QTimer>
6  #include <vector>
7  #include "GeometricObject.hpp"
8
9  class Board:public QWidget {
10 Q_OBJECT
11  public:
12   QTimer timer;
13   Board(std::vector<GeometricObject*>& geos);
14   std::vector<GeometricObject*>& geos;
15   QSize minimumSizeHint();
16   QSize sizeHint() ;
17   void paintEvent(QPaintEvent *event);
18
19   static int showPaintable
20    (std::vector<GeometricObject*>& geos,int argc,char **argv);
21
22 public slots:
23   virtual void tick();
24 };
25 #endif /*BOARD_H_*/
```

Board.cpp

```
1  #include "Board.hpp"
2  #include <QtGui>
3
4  Board::Board(std::vector<GeometricObject*>& geos)
5                                          :QWidget(0),geos(geos){
6    setBackgroundRole(QPalette::Base);
7    connect(&timer,SIGNAL(timeout()),this,SLOT(tick()));
8    timer.start(25);
9  }
10
11 QSize Board::minimumSizeHint(){return QSize(100, 100);}
12 QSize Board::sizeHint(){return QSize(400, 300);}
13
14 template <typename IT>
15 void paintIt(IT begin,IT end,QPainter& painter){
16     for (;begin!=end;begin++) (*begin)->paintMe(&painter);
17 }
18 template <typename IT>
19 void moveIt(IT begin,IT end){
20   for (;begin!=end;begin++) (*begin)->move();
21 }
22
23 void Board::paintEvent(QPaintEvent *){
24   QPen pen;
25   QPainter painter(this);
26   painter.setPen(pen);
27
28   paintIt(geos.begin(),geos.end(),painter) ;
```

```
29 }
30
31 int Board::showPaintable
32     (std::vector<GeometricObject*>& geos,int argc,char **argv){
33   QApplication application(argc,argv);
34   Board* b = new Board(geos);
35   b->show();
36   return application.exec();
37 }
38
39 void Board::tick(){
40   moveIt(geos.begin(),geos.end());
41   this->repaint();
42 }
```

a) Testen Sie Sie Klasse `Board` mit Instanzen Ihrer geometrischen Figuren.

b) Sorgen Sie jetzt in Ihrer Applikation dafür, dass sich die Objekte am Fensterrand abstoßen.

c) Sorgen Sie jetzt dafür, dass Objekte, die aneinanderstoßen, sich wieder voneinander abstoßen. Benutzen Sie hierzu die Methode `touches`.

4.2.6 Ereignisbehandlungen

Tastatureingabe

AnimatedPaneWithKeyboard.hpp

```
1 #include "../timeAnimated/TimeAnimatedPane.hpp"
2
3 class AnimatedPaneWithKeyboard:public TimeAnimatedPane {
4  public:
5   AnimatedPaneWithKeyboard(MovingImage* mi);
6  protected:
7   void keyPressEvent(QKeyEvent *event);
8 };
```

AnimatedPaneWithKeyboard.cpp

```
1 #include <QKeyEvent>
2 #include "AnimatedPaneWithKeyboard.hpp"
3 AnimatedPaneWithKeyboard::AnimatedPaneWithKeyboard(MovingImage* mi)
4    :TimeAnimatedPane(mi){};
5
6 void AnimatedPaneWithKeyboard::keyPressEvent(QKeyEvent* event){
7   switch (event->key()) {
8     case Qt::Key_Left: mi->dX -= 0.5; break;
9     case Qt::Key_Right:mi->dX += 0.5; break;
10    case Qt::Key_Down: mi->dY += 0.5; break;
11    case Qt::Key_Up:   mi->dY -= 0.5; break;
12    default: QWidget::keyPressEvent(event);
13  }
14 }
```

TestAnimatedPaneWithKeyboard.cpp

```
#include "AnimatedPaneWithKeyboard.hpp"
#include <QApplication>
#include <QPushButton>

int main(int argc,char **argv){
  QApplication application(argc,argv);
  QString fileName =argc<1ẗestbild.png":argv[1];
  AnimatedPaneWithKeyboard* p
   =new AnimatedPaneWithKeyboard(new MovingImage(fileName,0,0,2,1));
  p->show();
  return application.exec();
}
```

Mausereignisse

AnimatedPaneWithMouse.hpp

```
#include "../animatedWithKeyboard/AnimatedPaneWithKeyboard.hpp"
#include <QMouseEvent>
class AnimatedPaneWithMouse:public AnimatedPaneWithKeyboard {
 public:
  AnimatedPaneWithMouse(MovingImage* mi);
 protected:
  void mousePressEvent (QMouseEvent* e );
};
```

AnimatedPaneWithMouse.cpp

```
#include "AnimatedPaneWithMouse.hpp"
AnimatedPaneWithMouse::AnimatedPaneWithMouse(MovingImage* mi)
   :AnimatedPaneWithKeyboard(mi){};

void AnimatedPaneWithMouse::mousePressEvent (QMouseEvent* e ){
 mi->x = e->x();
 mi->y = e->y();
}
```

TestAnimatedPaneWithMouse.cpp

```
#include "AnimatedPaneWithMouse.hpp"
#include <QApplication>
#include <QPushButton>

int main(int argc,char **argv){
  QApplication application(argc,argv);
  QString fileName =argc<1ẗestbild.png":argv[1];
  AnimatedPaneWithMouse* p
   =new AnimatedPaneWithMouse(new MovingImage(fileName,0,0,2,1));
  p->show();
  return application.exec();
}
```

Aufgabe 13 Sie sollen in diesem Übungsblatt eine kleines Spielprojekt mit Qt realisieren. Hierzu sollen die Programmteile aus den letzten Übungsblättern als Grundlage benutzt werden. Das Spielprojekt soll beinhalten:

- Mehrere bewegliche Figuren auf einer Spielfläche. Dieses können abstrakte geometrische Figuren sein, oder aber durch Bilddateien dargestellte Figuren.
- Mindestens eine der Figuren soll durch Maus und/oder Tastatur von einem Spieler gesteuert werden.
- Es soll Kollisionen von Figuren geben, die einen Effekt auf das Spiel haben. (z.B. die Bahn der Figuren wird abgelenkt, Figuren verschwinden vom Spielfeld, Figuren zersplittern in mehrere kleine Figuren, Punktestand wird gezählt oder ähnliches)
- Das Spiel soll eine Menüleiste haben, über die eine About-Box zu öffnen ist, das Spiel zu beenden (Quit), zu starten (oder auch zu pausieren) ist und ein Optionsdialog geöffnet werden kann.

Abzugeben ist ein Ordner mit Ihren Namen, in dem sich befinden:

- ein Ordner `src` mit den Quelltext Ihres Spiels
- ein Ordner `doc` mit einer vorzugsweise mit Doxygen generierten API-Dokumentation des Projekts
- Ein kleines PDF-Dokument (2-5 Seiten), in dem Ihr Name, Ihre Matrikelnummer verzeichnet, die Spielidee und das Spielziel erklärt sind und die zentrale objektorientierte Architektur des Spiels skizziert ist.

Dieser Ordner `myname` ist als komprimierte tar-Datei zu verpacken (auf der Kommandozeile mit `tar -cvzf myname.tgz myname` und in den Übungsgruppen die von Herrn Panitz geleitet werden an die Emailadresse `abgabecpp@panitz.name` zu verschicken. Beachten Sie, dass das Projekt keine `.exe`-Dateien enthalten darf.

In den übrigen Gruppen fragen Sie bitte die entsprechenden Lehrbeauftragten nach der Form der Abgabe.

Das Projekt ist in der letzten Praktikumsstunde oder in einem mit dem Leiter der Praktikumsstunde auszumachenden Termin innerhalb der vorlesungsfreien Zeit in einen 5-minütigen Kurzvortrag vorzuführen.

Die endgültige Abgabe hat per Mail bis zum 29. Juli um 20:15Uhr rechtzeitig zum Beginn des Tatorts zu erfolgen.

Es wird nach folgenden Kriterien bewertet:

- Einhaltung der Abgabebedingungen
- objektorientierter Klassenentwurf
- Code-Qualität
- Dokumentation
- Code Kommentierung

- Funktionalität
- Spielidee
- Optik
- Kurzpräsentation

Programmverzeichnis

Abbildungsverzeichnis