Objektorientierte Programmierung mit C++

(Entwurf)

SS 05

Sven Eric Panitz

FH Wiesbaden

Version 17. Juni 2005

Die vorliegende Fassung des Skriptes ist ein Entwurf für die Vorlesung des Sommersemesters und wird im Laufe der Vorlesung erst seine endgültige Form finden. Kapitel können dabei umgestellt, vollkommen revidiert werden oder gar ganz wegfallen.

Dieses Skript entsteht begleitend zur Vorlesung des SS 05. Es basiert auf den Skripten zur Einführung in die Programmierung mit der Programmiersprache Java an der TFH-Berlin [Pan03b][Pan03c][Pan04] und der Programmiervorlesung zur Einführung in C++ aus dem WS03/04 an der TFH-Berlin [Pan03d].

Naturgemäß ist es in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler werden sich im Eifer des Gefechtes nicht vermeiden lassen und wahrscheinlich in nicht geringem Maße auftreten. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Teil I

Die Sprache C++

Kapitel 1

Einführung

1.1 Ziel der Vorlesung

Im ersten Semester wurden Grundkenntnisse der Programmierung in der Sprache C erlernt. Dabei wurden zuvorderst klassischen imperativen Konzepten, wie z.B. Kontrollstrukturen (Schleifen) erlernt. Ein Programmierkurs in der Sprache C ist sehr nah an der zugrundeliegenden Rechnerarchitektur gebunden. Die Konzepte der Sprache C spiegeln direkt die Grundprinzipien der zugrunde liegenden realen Maschine wieder. Dieser kann man besonders gut an dem Konzept der Reihungen (arrays)[1] sehen. Eine Reihung ist ein zusammenhängender Speicherbereich und eine Variable einer Reihung ist lediglich die Adresse, an der dieser zusammenhängende Speicherbereich beginnt. Eine Indexierung in eine Reihung ist dabei schon eine Zeigerarithmetik: die Adresse der indexierten Daten wird durch Addition auf den Beginn des Speicherbereichs erhalten. Ein zu höher Index greift aus diesen Speicherbereich hinaus, führt dazu, daß unkontrolliert Daten aus dem Hauptspeicher gelesen wurden.

In diesem Semester wollen wir uns von der realen Maschine entfernen und nicht ausgehend von den Möglichkeiten der Maschine das Programmieren betrachten, sondern ausgehend von dem Problemfeld, für das eine Lösung zu finden ist.

Die Programmiersprache C++ ist sehr umfangreich und vereint eine Vielzahl unterschiedlicher Konzepte. In einer zweistündigen einsemestrigen Vorlesung kann nicht die ganze Programmiersprache C++ gelehrt werden. Wir werden uns darauf konzentrieren, einige die wesentlichen Konzepte der deklarativen Programmierung zu zeigen. Dabei soll mehr der Fokus auf diesen Konzepten und weniger auf Details der Programmiersprache C++ liegen, so daß die erworbenen Kenntnisse leicht auf andere objektorientierte Programmiersprachen übertragen werden können. Daher werden sich immer mal wieder Hinweise im Skript befinden, die auf unterschiede zur Programmiersprache Java aufmerksam machen. Für Detailfragen zu C++ ist jeweils ein umfassendes Lehrbuch zu konsultieren. Eine ausführliche kommentierte Literaturliste findet sich im Skript von Herrn Grude [Gru01]. Ein relativ kompaktes und preisgünstiges Lehrbuch, das mir bei der Vorbereitung geholfen hat, ist [NH02]. Viele C++ Kurse finden sich im Netz. Beispielhaft sei [Wie99] hier erwähnt.

[1] Der im deutschen oft gebräuchliche Begriff *Feld* für *arrays* wird in der objektorientierten Nomenklatur für die im Englischen als *field* bezeichneten Attribute einer Klasse benutzt. Daher wird in diesem Skript für *arrays* der ebenso gebräuchliche deutsche Begriff *Reihung* benutzt.

1.2 Geschichtlicher Abriss

Die Programmiersprache C wurde anfang der 70er Jahre von Dennis M. Richie an den Bell Laboratories entworfen. Eine Vorläufersprache von C trug den Namen B, die im Gegensatz zu C allerdings vollkommen ungetypt war. C erfreute sich bald großer Beliebtheit als Alternative zur Programmiersprache Fortran der damals allmächtigen Firma IBM. Viele Programmierer empfanden Fortran als eine zu große Einschränkung ihrer Freiheit. C entwickelte sich als die Programmiersprache der Wahl für Unix-Programmierer. Die meisten Tools in Unix sind in C geschrieben[2] und Unix selbst ist zu über 90% in C implementiert. Erst 1989 erfolgte eine Standartisierung von C durch das ANSI X3J11 Komitee.

C++ erweitert die Programmiersprache C um objektorientierte Konstrukte, wie sie zuvor z.B. in der Programmiersprache *Smalltalk*[Ing78] eingeführt wurden. Die Erweiterung von C ist dabei so vorgenommen worden, daß weitgehendst jedes C-Programm auch ein C++-Programm ist.

Ein wichtiges Entwurfskriterium von C ist, den Programmierer in keinster Weise einzuschränken. Man kann lapidar sagen: alles ist erlaubt, der Programmierer weiß, was er tut. Dieses geht zu Lasten der Sicherheit von Programmen. Ein C-Compiler erkennt weniger potentielle Programmierfehler als z.B. ein Javacompiler oder gar ein Haskellcompiler[PJ03]. Das hat zur Folge, daß fehlerfrei kompilierte Programme häufig unerwartete Laufzeitfehler haben. Solche Fehler können schwer zu finden sein. Um das Risiko solcher Fehler zu verringern ist es notwendig in der Programmierung wesentlich strengere Programmierrichtlinien einzuhalten als z.B. in Java, oder in Haskell, wo der Programmierer noch nicht einmal die Typen von Funktionen in einer Signatur festlegen muß. Der Compiler ist in der Lage die Signatur selbst herzuleiten. In C hingegen empfiehlt es sich, sogar im Namen einer Variablen kenntlich zu machen, um was für einen Typ es sich bei ihr handelt.

Mit der vollen Kontrollen in C erhält ein Programmierer allerdings auch die Macht, sein Programm möglichst optimiert zu schreiben. Inwiefern ein C-Programmierer heutzutage allerdings die Fähigkeiten hat, diese Macht auszuspielen gegenüber einen klug optimierenden Compiler, sei dahingestellt.

Von diesen Entwurfskriterium wurde auch nicht bei der Erweiterung von C nach C++ abgegangen.

Die Programmiersprache C verfolgt in vielen Fällen einen gewissen Pragmatismus. Bestimmte Konvertierungen von Typen werden unter der Hand vorgenommen, insbesondere gilt dies für Zeiger und Referenztypen. So praktisch solch ein Umgang im Alltag sein kann; so stellt dieser Pragmatismus für den Anfänger oft eine starke Hürde da. Es gibt kein einfaches mathematisches Grundkonzept für die Übersetzung und Ausführung von C Programmen. Sehr viele Ausnahmen für bestimmte Typen und Konstrukte müssen erlernt werden. In der Praxis, so hässlich dieses Konglomerat aus Pragmatismus und Ausnahmen, kleinen Tricks und Fallstricken für den Theoretiker sein mag, hat sich C durchgestzt und ist derzeit und sicherlich noch auf lange Jahre hin, als das Zentrum des Programmieruniversums anzusehen. Wohl kaum eine zweite Programmiersprache kann mit so vielen Bibliotheken und Beispielprogrammen aufwarten wie C. Didaktisch hingegen ist C für eine Einführung in die Programmierung nicht unproblematisch; daher finden von jeher Programmiereinführungen oft auf anderen Sprachen statt: Pascal, Modula, Ada zeitweise oft Algol68 und Scheme oder seit wenigen Jahren Java. Die auch für die Zukunft nach wie vor zuerwartende Bedeutung von C++ in der Praxis rechtfertigt trotzdem den Einstieg in die Programmierung mit C/C++ an einer Fachhochschule.

[2] Eine interessante Ausnahme ist der Editor *emacs*, der in Lisp geschrieben ist.

Das Erlernen von C/C++ hat einen hohen praktischen Nutzen. Nicht nur, weil viele Projekte heute mit C++ realisiert werden, sondern auch, weil viele moderne Sprachen sich an C/C++ anlehnen. So hat Java eine große syntaktische Ähnlichkeit mit C/C++, auch wenn hier z.T. grundsätzlich andere Konzepte verwirklicht sind. Ein anderes Beispiel ist C#, eine Programmiersprache, die Microsoft als Erweiterung von C++ in ihrem .NET Framework propagiert. Schließlich gibt es noch viele Programmiersprachen für sehr spezielle Anwendungen, die sich an C/C++ anlehnen, z.B. NVIDIA's Cg (C for Graphics) mit dem Grafikprozessoren programmiert werden können. Hintergrund ist, dass heute bei den meisten Programmierern C/C++ - Kenntnisse erwartet werden und die Entwickler neuer Sprachen sich von der Anlehnung an C/C++ versprechen, dass den Programmierern der Umstieg auf ihre neue Sprache leichter fällt.

Zusätzlich kann man die große Flexibilität von C++ auch didaktisch als Chance begreifen. Es läßt sich in sehr unterschiedlichen Stile und nach sehr unterschiedlichen Entwurfskriterien Software in C++ schreiben. Unter Umständen sind Programme in unterschiedlichen Stilen kaum noch als zu einer gemeinsamen Sprache gehörig zu erkennen und es kommt wahrscheinlich auch nicht von ungefähr, daß es einen *obfuscated C contest* (`http://www.de.ioccc.org/main.html`) gibt, in dem es gerade um ungewähnlich und verwirrend codierte Programme geht.

1.3 Compiler und Werkzeuge

1.3.1 Editoren

Es für C++ eine Vielzahl mächtiger integrierter Entwicklungsumgebungen. Hier seien z.B. `kdevelop` (`http://www.kdevelop.org/`) und `anjuta` (`http://anjuta.sourceforge.net/`) genannt. Während `anjuta` sich darauf beschränkt C/C++-Projekte zu unterstützen, dafür aber viel Hilfe zur Einbindung verschiedener GUI-Bibliotheken gibt, können in `kdevelop` Projekte in unterschiedlichsten Sprachen entwickelt werden, wie Ada, Fortran, Haskell, Java, Pascal, Perl Php, Python und Ruby. Für den Anfänger empfiehlt es sich zunächst einmal auf eine integrierte Entwicklungsumgebung zu verzichten und lediglich mit einem Texteditor und einem Kommandozeilenübersetzer zu beginnen. Eine Entwicklungsumgebung setzt oft viel implizites Wissen über die Sprache und ihr Ausführungs-/Übersetzungsmodell voraus. Zudem werden wir zunächst nur kleinere Testprogramme schreiben, für die die Benutzung einer Entwicklungsumgebung Overkill wäre. Erst wenn wir Programme sozusagen von Hand entwickelt haben, lernen wir auch die Vorteile einer Entwicklungsumgebung zu schätzen und sinnvoll zu nutzen.

So werden wir unsere Programme mit einem Texteditor unserer Wahl entwickeln. Die meisten allgemeinen Texteditoren haben einen Modus für C++ Programme, so daß sie syntaktische Konstrukte erkennen und farblich markieren. Hier steht natürlich an erster Stelle der allgegenwärtige Editor emacs (`www.gnu.org/software/emacs/`) , der sowohl für Windows- als auch für unixartige Betriebssysteme zur Verfügung steht; aber jedem sei freigestellt nach eigenen Vorlieben einen Editor seiner Wahl zu benutzen, z.B `kate` (`http://kate.kde.org/`) , KEdit (`http://docs.kde.org/en/3.3/kdeutils/kedit/`) oder eines der vielen `vi`-Derivate wie vim (`http://www.vim.org/`) .

1.3.2 Compiler

Für C++ stehen viele verschiedene Übersetzer zur Verfügung. Wir stellen die drei gebräuchlichsten hier vor.

gcc

gcc ist der C und C++ Compiler von Gnu. gcc steht auf Linux standardmäßig zur Verfügung und ist in der Regel auf Unixsystemen installiert. g++ ist ein Befehl, der den C Compiler mit einer Option aufruft, so daß C++ Quelldateien verstanden werden. gcc steht kostenlos zur Verfügung.

gcc ist schon längst nicht mehr allein ein C-Compiler sondern eine Sammlung von unterschiedlichen Übersetzern. So kann mit gcc neben C++ und C mitlerweile auch Java kompiliert werden. Der entsprechende Aufruf hierzu ist gcj. gcj ist nicht nur in der Lage Klassendateien für Java-Quelltext zu erzeugen, sondern auch platformabhängige Objektdateien.

cygwin gcc steht auch dem Windowsprogrammierer zur Verfügung. Unter der Bezeichnung cygwin kann man auf einem Windowssystem alle gängigen Entwicklungstools von Linux auch auf Windows installieren. Damit läßt sich auf Windows exakt wie auf Unix arbeiten. cygwin ist kostenfrei und kann vom Netz (www.cygwin.com) heruntergeladen werden.

Borland

Die Firma Borland hat eine C++-Entwicklungsumgebung. Der Compiler wird für nichtkomerzielle Anwender von der Firma Borland frei zur Verfügung gestellt. Auch er kann vom Netz (www.borland.com/bcppbuilder/freecompiler) heruntergeladen werden.

Microsoft

Schließlich bietet Microsoft unter den Namen Visual C++ für sein Windowsbetriebssystem einen C Entwicklungsumgebung an. Zur Benutzung dessen Compilers ist allerdings ein Lizenzvertrag von Nöten.

Die Beispiele dieses Skriptes sind alle mit gcc Version 3.3.1 auf Linux übersetzt worden. Einige Programmen wurden mit gcc zusätzlich auch unter Windows getestet.

1.4 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.

HelloWorld.cpp

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

Wir sehen, daß zunächst einmal die Bibliothek für Ein- und Ausgabeströme zu inkludieren ist: `#include <iostream>`.

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, daß 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.

`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 audführen:

```
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
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.

1.5 Kompilierung: Mach mal!

Es gibt ein wunderbares kleines Programm mit Namen `make`. Starten wir dieses Programm einmal mit einem Argument:

```
panitz@fozzie(~)$ make love
make: *** No rule to make target ‘love’.  Stop.
panitz@fozzie(~)$
```

Nun, etwas anderes hätten wir von einem Computer auch kaum erwartet. Wenn er etwas machen soll, so müssen wir ihm in der Regel sagen, wie er das machen kann. Hierzu liest das Programm `make` standardmäßig eine Datei mit Namen `Makefile`. In dieser sind Ziele beschrieben und die Operationen, die zum Erzeugen dieser Ziele notwendig sind.

Ein Ziel ist in der Regel eine Datei, die zu erzeugen ist. Syntaktisch wird ein Ziel so definiert, daß der Zielname in der ersten Spalte einer Zeile steht. Ihm folgt ein Doppelpunkt, dem auf derselben Zeile Ziele aufgelistet folgen, die zuerst erzeugt werden müssen, bevor dieses Ziel gebaut werden

kann. Dann folgt in den folgenden Zeilen durch einen Tabulatorschritt eingerückt, welche Befehle zum Erreichen des Ziels ausgeführt werden müssen. Eine einfache Makedatei zum Bauen des Programms `helloworld` sieht entsprechend wie folgt aus.

makeTheHelloWorld.

```
1 helloworld: HelloWorld.o
2         g++ -o helloworld HelloWorld.o
3
4 helloworld.o: HelloWorld.cpp
5         g++ -c helloworld.cpp
```

Es gibt in dieser Makedatei zwei Ziele: die ausführbare Datei `helloworld` und die Objektdatei `HelloWorld.o`. Zum Erzeugen der Objektdatei, muß die Datei `HelloWorld.cpp` vorhanden sein, zum Erzeugen der ausführbaren Datei, muß erst die Objektdatei generiert worden sein. Wir können jetzt `make` bitten, unsere `helloworld`-Applikation zu bauen. Die Makedatei nicht `Makefile` heißt, welches die standardmäßig von `make` als Steuerdatei gelesene Datei ist, müssen wir `make` mit der Option `-f` als Argument mitgeben, in welcher Datei die Ziele zum Machen definiert sind.

Für die Erzeugung des Programms `helloworld` reicht also folgender Aufruf von `make`:

```
sep@pc216-5:~/fh/cpp/student/src> make -f makeTheHelloWorld
g++    -c -o HelloWorld.o HelloWorld.cpp
g++ -o helloworld HelloWorld.o
sep@pc216-5:~/fh/cpp/student/src> ./helloworld
hello world
sep@pc216-5:~/fh/cpp/student/src> make -f makeTheHelloWorld
make: ≫helloworld≪ ist bereits aktualisiert.
sep@pc216-5:~/fh/cpp/student/src>
```

Gibt man `make` nicht als zusätzliches Argument an, welches Ziel zu bauen ist, so baut `make` das erste in der Makedatei definierte Ziel.

Die wahre Stärke von `make` ist aber, daß nicht stur alle Schritte zum Erzeugen eines Ziels durchgeführt werden, sondern anhand der Zeitstempel der Dateien geschaut wird, ob es denn überhaupt notwendig ist, bestimmte Ziele neu zu bauen, oder ob sie schon aktuell im Dateisystem vorliegen. So kommt es, daß beim zweiten Aufruf von `make` oben, nicht erneut kompiliert wurde.

Um gute und strukturierte Makedateien zu schreiben, kennt `make` eine ganze Anzahl weiterer Konstrukte. Zuvorderst Kommentare. Kommantarzeilen beginnen mit einem Gattersymbol `#`. Dann lassen sich Konstanten definieren. Es empfiehlt sich in einer Makedatei alle Pfade und Werkzeuge, wie z.B. der benutzte Compiler als Konstante zu definieren, damit diese leicht global geändert werden können. Schließlich lassen sich noch allgemeine Regeln, wie man von einem Dateitypen zu einem anderen Dateitypen gelangt, definieren; z.B. das der Aufruf `g++ -c` zu benutzen ist, für eine CPP-Datei eine Objektdatei zu generieren.

Eine typische Makedatei, die all diese Eigenschaften ausnutzt, sieht für ein C++-Projekt dann wie folgt aus.

Makefile.

```
1 #der zu benutzende C++ Compiler
2 CC = g++
3 EXE = TestReadRoman
```

```
4
5  #das Hauptprogramm
6  TestReadRoman: TestReadRoman.o ReadRoman.o
7          $(CC) -o TestReadRoman TestReadRoman.o ReadRoman.o
8
9  #allgemeine Regel zum Aufruf des Compilers zum generieren
10 #der Objektdatei aus der Quelltextdatei
11 .cpp.obj:
12         $(CC) -c $<
13
14 #alles wieder löschen
15 clean:
16         rm $(EXE)
17         rm *.o
18
19 #just for fun
20 love:
21         @echo "I do not know, how to make love.";
22
```

Für eine genaue Beschreibung der Möglichkeiten einer Makedatei konsultiere man die Dokumentation von `make`. Nahezu alle C++-Projekte benutzen `make` zum Bauen des Projektes. Für Javaprojekte hat sich hingegen das Programm `ant` (`http://ant.apache.org/`) durchgesetzt.

Auch die verschiedenen Versionen dieses Skripts werden mit Hilfe von `make` erzeugt.

Aufgabe 1 Kopieren Sie sich den Quelltextordner dieses Skripts und rufen Sie die darin enthaltene obige Makefile-Datei auf. Rufen Sie dann `make` direkt nocheinmal auf. Ändern Sie anschließend die Datei `ReadRoman.cpp` und starten Sie erneut `make`. Was beobachten Sie in Hinblick auf `TestReadRoman.o`. Starten Sie schließlich `make` mit dem Ziel `love`.

1.6 Dokumentation

Heutzutage schreibt niemand mehr ein einzelnen Programm ganz allein. In der Regel wird komplexere Software entwickelt. Die meiste Software wird damit nicht einmal ein Programm sein, sondern eine ganze Bibliothek, die in anderen Softwareprodukten verwendet wird.

Um die von einer Bibliothek bereitgestellte Funktionalität sinnvoll verwenden zu können, ist dieser auf einheitliche Weise zu dokumentieren. Zur Dokumentation von C++-Bibliotheken kann man das Werkzeug Doxygen (`http://www.stack.nl/~dimitri/doxygen/index.html`) einsetzen. Dieses Programm liest die C++ Kopfdateien und generiert eine umfassende Dokumentation in verschiedenen Formaten. Am häufigsten wird dabei wahrscheinlich die HTML-Dokumentation verwendet.

1.6.1 Benutzung von Doxygen

Doxygen kommt mit einem Kommandozeilenprogramm `doxygen`. Dieses erwartet als Eingabe eine Steuerdatei zur Konfiguration. In dieser Konfigurationsdatei können eine große Menge

von Parametern eingestellt werden, wie die Sprache, in der die Dokumentation erstellt werden soll oder auch den Projektnamen, des zu dokumentierenden Projekts. Zum Glück ist `doxygen` in der Lage, sich selbst eine solche Konfigurationsdatei zu generieren, die man dann manuell nacheditieren kann. Hierzu benutzt man das Kommandozeilenargument `-g` von `doxygen`.

Ein kurzer Start mit `doxygen` sieht also wie folgt aus:

- Generierung einer Konfigurationsdatei mit:

```
sep@pc216-5:~/fh/cpp/student/src> doxygen -g MeinProjekt

Configuration file ‘MeinProjekt’ created.

Now edit the configuration file and enter

  doxygen MeinProjekt

to generate the documentation for your project

sep@pc216-5:~/fh/cpp/student/src>
```

- Anschließend editiert man die erzeugte Datei `MeinProjekt`. Für den Anfang sei zu empfehlen `EXTRACT_ALL = YES` zu setzen. Selbst wenn keine eigentlich für Doxygen gedachte Dokumentation im Quelltext steht, werden alle Programmteile in der Dokumentation aufgelistet.
- Die Eigentliche Generierung der Dokumentation.

```
sep@pc216-5:~/fh/cpp/student/src> doxygen  MeinProjekt
sep@pc216-5:~/fh/cpp/student/src>
```

Jetzt befindet sich die generierte Dokumentation im in der Konfigurationsdatei angegebenen Ausgabeordner.

Aufgabe 2 Laden Sie sich die kompletten Quelltextdateien dieses Skriptes von der Webseite und generieren Sie mit `Doxygen` eine Dokumentation für diese.

1.6.2 Dokumentation des Quelltextes

Bisher konnte sich *Doxygen* lediglich auf den Quelltext beziehen, um eine Dokumentation zu erzeugen. Natürlich reicht das nicht aus. Daher kann man im Quelltext Kommentare anbringen, die *Doxygen* auswertet. Es gibt verschiedene Syntax hierzu, z.B. die klassische aus C bekannte Art zu kommentieren:

DoxygenMe.h

```
1 /**
2   Dies ist eine einfache Funktion.
3 Sie ist in der Lage ganz tolle Dinge zu tun.
4  */
5 void f();
```

In diesen Kommentarblöcken, die über dem zu dokumentierenden Code stehen, können mit dem Backslashsymbol bestimmte Attribute gesetzt werden. So gibt es z.B. das Attribut `\param` für einen Funktionsparameter oder das Attribut `\return` für das Funktionsergebnis.

DoxygenMe.h

```
6  /**
7    Die Additionsfunktion.
8    Sie kann dazu benutzt werden, um einen Funktionszeiger
9    auf die Addition
10   zur Verfügung zu haben. Sie ruft in ihrem Rumpf lediglich den
11   Additionsoperator auf.
12
13   \param x der erste Operand.
14   \param y der zweite (rechte) Operand.
15   \return die Summe der beiden Operanden.
16 */
17 int add(int x, int y);
```

Aufgabe 3 Betrachten Sie, die für die Funktionen `f` und `add` in der letzten Aufgabe generierte Dokumentation. *Doxygen* kennt viele solcher Attribute und viele verschiedene mächtige unterschiedliche Weisen der Dokumentation. Zum Glück ist *Doxygen* selbst recht gut erklärt, so daß mit dem hier gegebenen Einstieg es hoffentlich leicht möglich ist, sich mit dem Programm auseinanderzusetzen.

Aufgabe 4 Beschäftigen Sie sich mit der Dokumentation von *Doxygen* und testen einige der darin beschriebenen Möglichkeiten der Dokumentation.

1.7 Debuggen

Wie Ernie in der Sesamstraße bereits gesungen hat: *Jeder macht mal Fehler ich und du*; so müssen wir davon ausgehen, daß unsere Programme nicht auf Anhieb die gewünschte Funktionalität bereitstellen. Insbesondere da C uns relativ viel Möglichkeiten und Freiheiten in der Art der Programmierung ermöglicht, läßt es uns auch die Freiheiten, viele Fehler zu machen.

Um einen Fehler aufzuspüren können wir auf zwei Techniken zurückgreifen:

- Logging: Während der Ausführung wird in bestimmten Programmzeilen eine Statusmeldung in eine Log-Datei geschrieben. Diese läßt sich hinterher analysieren. Die einfachste Form des Loggings ist, ab und zu eine Meldung auf die Konsole auszugeben.
- Debuggen: mit Hilfe eines Debuggers läßt sich schrittweise das Programm ausführen und dabei in den Speichern schauen.

Der Begriff *debuggen*, den man wörtlich mit *Entwanzen* übersetzen könnte, bezieht sich auf die Benutzung des Wortes *bug* für Macken in einem technischen System. Der Legende nach stammt diese Benutzung des Wortes *bug* aus der Zeit der ersten Rechner, die noch nicht mit Halbleitern geschaltet haben, sondern mit Relais, ähnlich, wie man sie von Flipperautomaten kennt. Es soll einmal zu einem Fehler im Programmablauf gekommen sein, der sich schließlich

darauf zurückführen ließ, daß ein Käfer zwischen ein Relais geklettert war. Eine schöne Ankdote, die leider falsch ist, denn aus schon früheren Dokumenten geht hervor, daß der Begriff *bug* schon im 19. Jahrhundert für technische Fehler gebräuchlich war.

Um ein Programm mit einem Debugger zu durchlaufen, ist es notwenig, daß der Debugger Informationen im ausführbaren Code findet, die sich auf den Quelltext beziehen. Dieses sind umfangreiche Informationen. Der `gcc` generiert die notwendige Information in die Objektdateien, wenn man ihn mit der Option `-g` startet.

1.7.1 Debuggen auf der Kommandozeile

Die rudimentärste Art des Debuggens ist mit Hilfe des Kommandozeilenprogramms `gdb`. Wir wollen das einmal ausprobieren. Hierzu brauchen wir natürlich ein entsprechendes Programm, welches hier zunächst definiert sein. Es handelt sich dabei um ein Programm zur Konvertierung von als Strings vorliegenden römischen Zahlen in die entsprechende arabische Zahl als `int`.

ReadRoman.h

```
1  #ifndef READ_ROMAN__H
2  #define  READ_ROMAN__H ;
3  #include <string>
4  int romanToInt(std::string roman);
5  #endif
```

Eine rudimentäre Implementierung dieser Kopfdatei:

ReadRoman.cpp

```
1  #include <string>
2  #include "ReadRoman.h"
3
4  int romanCharToInt(char c){
5    switch (c){
6      case 'I': return 1;
7      case 'V': return 5;
8      case 'X': return 10;
9      case 'L': return 50;
10     case 'C': return 100;
11     case 'D': return 500;
12     case 'M': return 1000;
13   }
14   throw "illegal Roman digit";
15 }
16
17 int romanToInt(std::string roman){
18   int result=0;
19   int last=0;
20   for (int i=0;i<roman.size();i++){
21     int current=romanCharToInt(roman[i]);
22     if (current>last) result = result-2*last;
23     result=result+current;
24     last=current;
```

```
25    }
26    return result;
27  }
```

Wir benutzen hier schon ein paar wenige C++ Eigenschaften, die uns aber vorerst nicht stören sollen. Es folgt ein minimaler Test dieses Programms:

TestReadRoman.cpp

```
1  #include "ReadRoman.h"
2  #include <iostream>
3  int main(){
4    std::cout<<romanToInt("MMMCDXLIX") << std::endl;
5  }
```

Aufgabe 5 Übersetzen Sie das Programm mit der `-g`-Option des `gcc` und lassen Sie das Programm ausführen.

Jetzt können sie den `gnu debugger` mit dem Programm `TestReadRoman` aufrufen. Sie geraten in einen Kommandozeileninterpreter des Debuggers. Der Debugger kennt eine Reihe Kommandos, üer die er mit dem Kommando `help` gerne informiert:

```
sep@pc216-5:~/fh/cpp/student/bin> gdb TestReadRoman
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-suse-linux"...
  Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Das naheliegendste Kommando ist das Kommando `run`. Es führt dazu, daß das Programm ausgeführt wird.

```
(gdb) run
Starting program: /home/sep/fh/cpp/student/bin/TestReadRoman
3449

Program exited normally.
(gdb)
```

Wir wollen aber nicht, daß das Programm einmal komplett ausgeführt wird, sondern schrittweise die Ausführung nachverfolgen. Hierzu müssen wir einen sogenannten *breakpoint* setzen. Wir können diesen auf die erste Zeile des Programms setzen:

```
(gdb) break 4
Breakpoint 1 at 0x80488fd: file TestReadRoman.cpp, line 4.
(gdb)
```

Wenn wir jetzt `run` sagen, stoppet der Debugger in der Zeile vier:

```
(gdb) run
Starting program: /home/sep/fh/cpp/student/bin/TestReadRoman

Breakpoint 1, main () at TestReadRoman.cpp:4
4         std::cout<<romanToInt("MMMCDXLIX") << std::endl;
(gdb)
```

Jetzt können wir z.B. Zeile für Zeile das Programm vom Debugger ausführen lassen. Hierzu gibt es den Befehl `step`:

```
(gdb) step
romanToInt (roman=
        {static npos = 4294967295
        , _M_dataplus = {<std::allocator<char>> = {<No data fields>}
                        , _M_p = 0x804a014 "MMMCDXLIX"}
                        , static _S_empty_rep_storage = {0, 0, 0, 0}})
                                        at ReadRoman.cpp:18
18        int result=0;
(gdb) step
19        int last=0;
(gdb) step
20        for (int i=0;i<roman.size();i++){
(gdb)
```

Wie man sieht gibt der Debugger immer die aktuelle Zeile aus, und beim Funktionsaufruf auch den Wert des Parameters. Wollen wir zusätzlich in den Speicher schauen, so gibt es den Befehl `print`, der dazu da ist, sich den Wert einer Variablen anzuschauen:

```
(gdb) step
19        int last=0;
(gdb) step
20        for (int i=0;i<roman.size();i++){
(gdb) print i
$1 = 134514991
(gdb) print roman
$2 = {static npos = 4294967295,
  _M_dataplus = {<std::allocator<char>> = {<No data fields>},
   _M_p = 0x804a014 "MMMCDXLIX"}, static _S_empty_rep_storage = {0, 0, 0, 0}}
(gdb) step
```

```
21          int current=romanCharToInt(roman[i]);
(gdb) step
romanCharToInt (c=77 'M') at ReadRoman.cpp:5
5         switch (c){
(gdb) step
12          case 'M': return 1000;
(gdb) step
15      }
(gdb) step
romanToInt (roman=
        {static npos = 4294967295
        , _M_dataplus = {<std::allocator<char>> = {<No data fields>}
                        , _M_p = 0x804a014 "MMMCDXLIX"}
                        , static _S_empty_rep_storage = {0, 0, 0, 0}})
                                                    at ReadRoman.cpp:22
22          if (current>last) result = result-2*last;
(gdb) step
23          result=result+current;
(gdb) step
24          last=current;
(gdb) print result
$7 = 1000
(gdb)
```

Wenn auch mühselig, so können wir so doch ganz genau verfolgen, wie das Programm arbeitet.

Aufgabe 6 Spielen Sie die obige Session des Debuggers einmal nach und experimentieren sie mit den verschiedenen Befehlen des Debuggers.

1.7.2 Graphische Debugging Werkzeuge

Aufbauend auf die Möglichkeiten des Debuggens über die Kommandozeilenschnittstelle, lassen sich leicht konfortable graphische Debugger implementieren. Ein recht schöner solcher Debugger ist das Programm `ddd`. Hier können Breakpoints mit der Maus im Programmtext gesetzt werden und die verschiedenen Speicherbelegungen betrachtet werden. Abbildung 1.1 zeigt das Programm in Aktion.

Aufgabe 7 Üben Sie den Umgang mit dem Debugger `ddd` anhand des Programms `TestReadRoman`.

Wir haben unser Rüstzeug beisammen und können im nächsten Kapitel loslegen, das Programmieren mit C++ zu erkunden.

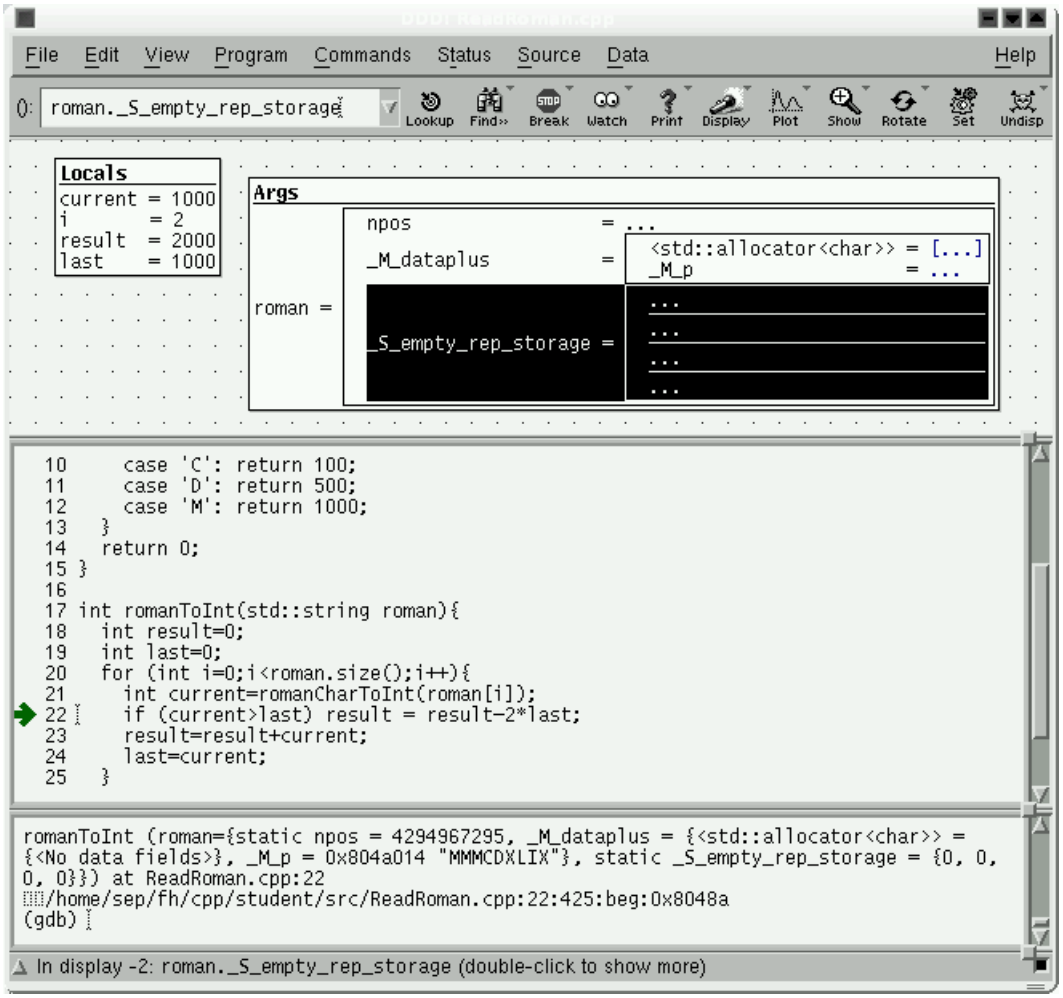

Abbildung 1.1: ddd in Aktion.

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 geinigt 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. 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.

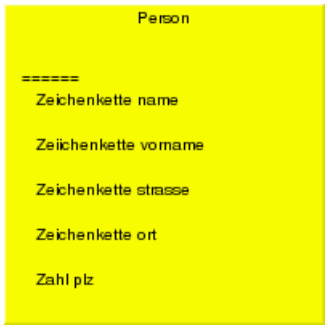

Abbildung 2.1: Modellierung einer Person.

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.

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

[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.

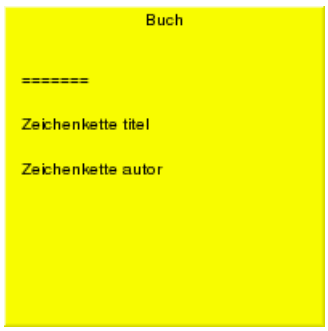

Abbildung 2.2: Modellierung eines Buches.

Abbildung 2.3: Modellierung eines Datums.

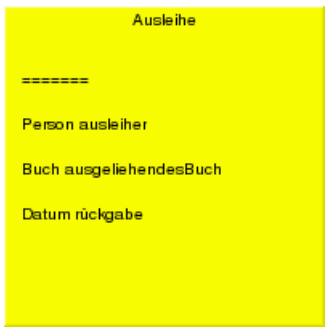

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

Klassen in C++

Wir haben in einem Modellierungsschritt im letzten Abschnitt verschiedene Objektarten identifiziert und ihre Eigenschaften spezifiziert. Dazu haben wir vier Karteikarten geschrieben. Jetzt können wir versuchen, diese Modellierung in C++ umzusetzen. In C++ beschreibt eine Klasse eine Menge von Objekten gleicher Art. Damit entspricht eine Klasse einer der Karteikarten in unserer Modellierung. Die Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören. Zusätzlich können wir in einer Klasse noch schreiben, welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt.

In C++ wird eine Klasse durch das Schlüsselwort `class`, gefolgt von dem Namen, den man für die Klasse gewählt hat, deklariert. Anschließend folgt in geschweiften Klammern der Inhalt der Klasse bestehend aus Felddefinitionen und Methodendefinitionen. Am Ende einer Klassendefinition steht ein Semicolon.

Die einfachste Klasse, die in C++ denkbar ist, ist eine Klasse ohne Felder oder Methoden:

Minimal.cpp

```
1 class Minimal {
2 };
```

Die Stringklasse Für C++ gibt es eine Vielzahl von Bibliotheken, die eine großen Anzahl von Klassen zur Verfügung stellen. Es müssen also nicht alle Klassen neu vom Programmierer definiert werden. Eine sehr häufig benutzte Klasse ist die Klasse `std::string`. Sie repräsentiert Objekte, die eine Zeichenkette darstellen, also einen Text, wie wir ihn in unserer ersten Modellierung bereits vorausgesetzt haben. Die Klasse `std::string` ist in der *standard template library* definiert.

Leider ist die Stringklasse der std in C++ nicht die einziege Stringklasse. Aus historischen Gründen kommen sehr viel C++-Bibliotheken mit einer eigenen Stringklasse daher. So kennen Qt, Gtkmm oder xerces eigene Implementierungen von Stringklassen. Allen diesen Klassen ist eigen, daß sie die als Reihung von Zeichen dargestellten Strings aus C in Objekte ihrer Klasse bei der Zuweisung konvertieren können.

Objekte

Eine Klasse deklariert, wie die Objekte, die die Klasse beschreibt, aussehen können. Um konkrete Objekte für eine Klasse zu bekommen, müssen diese irgendwann im Programm einmal erzeugt werden. Dieses wird in C++ syntaktisch gemacht, indem einem Schlüsselwort `new` der Klassenname mit einer in runden Klammern eingeschlossenen Argumentliste folgt. Dadurch erhält man einen Zeiger, auf das neu erzeugte Objekt. Ein Objekt der obigen minimalen C++-Klasse läßt sich entsprechend durch folgenden Ausdruck erzeugen:

Minimal.cpp

```
3 int main(){
4   Minimal* min = new Minimal();
5 }
```

Wenn ein Objekt eine Instanz einer bestimmten Klasse ist, spricht man auch davon, daß das Objekt den Typ dieser Klasse hat.

Objekte der Stringklasse Für die Klasse `std::string` gibt es eine besondere Art, Objekte zu erzeugen. Die aus C bekannten Strings können als Objekte der Klasse `std::string` benutzt werden. Ein in Anführungsstrichen eingeschlossener Text erzeugt somit ein Objekt der Klasse `std::string`. Hierzu muß die Definition der Klasse `string` inkludiert werden

Aus zwei Objekten der Stringklasse läßt sich ein neues Objekt erzeugen, indem diese beiden Objekte mit einem Pluszeichen verbunden werden:

StringAppend.cpp

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

int main(){
  std::string hallo= "hallo ";
  std::string welt = "welt";
  std::string hallowelt = hallo+welt;
  std::cout << hallowelt <<std::endl;
}
```

Hier werden die zwei Stringobjekte `"hallo "` und `"welt"` erzrugt und schließlich zum neuen Objekt `"hallo welt"` verknüpft.

Die Stringklasse ist so konzipiert, daß man mit ihr auch wie auf klassischen Reihungen arbieten kann. So lassen sich einzelne Zeichen eines Strings mit Hilfe der Indexierung über eckige Klammern ansprechen und verändern:

StringIndex.cpp

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

int main(){
  std::string s="hallo";
  for (int i=0;i<5;i++)
    std::cout<<s[i]<<std::endl;

  for (int i=0;i<5;i++)
    s[i]=s[i]-32;

  std::cout << s <<std::endl;
}

```

2.1.2 Felder und Methoden

Im obigen Abschnitt haben wir gesehen, wie eine Klasse definiert wird und Objekte einer Klasse erzeugt werden können. Allerdings war unsere erste Klasse noch vollkommen ohne Inhalt: es gab weder die Möglichkeit, Daten zu speichern, noch wurden Programmteile definiert, die hätten ausgeführt werden können. Hierzu können Felder und Methoden deklariert werden. Die Gesamtheit der Felder und Methoden nennt man mitunter auch *Features* oder, um einen griffigen deutschen Ausdruck zu verwenden, *Eigenschaften*.

Felder

Zum Speichern von Daten können Felder für eine Klasse definiert werden. In einem Feld können Objekte für eine bestimmte Klasse gespeichert werden. Bei der Felddeklaration wird angegeben, welche Art von Objekten in einem Feld abgespeichert werden sollen. Die Felder entsprechen dabei genau den Eigenschaften, die wir auf unsere Karteikarten geschrieben haben.

Syntaktisch wird in C++ der Klassenname des Typs, von dem Objekte gespeichert werden sollen, den frei zu wählenden Feldnamen vorangestellt. Eine Felddeklaration endet mit einem Semikolon.

Vorerst werden wir in einer Klasse, die Deklaration von Feldern und Methoden mit `public:` einleiten. Damit wollen wir kennzeichnen, daß auf die Eiegenschaften außerhalb der Klasse zugegriffen werden kann

Beispiel:
Im Folgenden schreiben wir eine Klasse mit zwei Feldern:

ErsteFelder.cpp

```
1  #include <string>
2
3  class Minimal {
4  };
5
6  class ErsteFelder {
7   public:
8    Minimal* meinFeld;
9    std::string  meinTextFeld;
10 };
```

Das erste Feld soll dabei ein Objekt unserer minimalen Klasse enthalten und das andere eine Zeichenkette.

Aufgabe 8 Schreiben Sie für die vier Karteikarten in der Modellierung eines Bibliotheksystems entsprechende Klassen mit den entsprechenden Feldern.

Methoden

Methoden[2] sind die Programmteile, die in einer Klasse definiert sind und für jedes Objekt dieser Klasse zur Verfügung stehen. Die Ausführung einer Methode liefert meist ein Ergebnisobjekt. Methoden haben eine Liste von Eingabeparametern. Ein Eingabeparameter ist durch den gewünschten Klassennamen und einen frei wählbaren Parameternamen spezifiziert.

Methodendeklaration In C++ wird eine Methode deklariert durch:

- den Rückgabetyp,
- den Namen der Methode,
- der in Klammern eingeschlossenen durch Kommas getrennten Parameterliste
- und den in geschweiften Klammern eingeschlossenen Programmrumpf.

[2]Der Ausdruck für *Methoden* kommt speziell aus der objektorientierten Prorammierung. In der imperativen Programmierung spricht man von *Prozeduren*, die funktionale Programmierung von *Funktionen*. Weitere Begriffe, die Ähnliches beschreiben, sind *Unterprogramme* und *Subroutinen*.

Im Programmrumpf wird mit dem Schlüsselwort `return` angegeben, welches Ergebnisobjekt die Methode liefert. Eine Quelle von Fehlern kann sein, daß ein C++-Compiler nicht unbedingt prüft, ob die Methode mit einer `return`-Anweisung beendet wird. Dann kann es sein, das zur Laufzeit undefinierte Werte durch die Methode zurückgegeben werden.

Beispiel:
Als Beispiel definieren wir eine Klasse, in der es eine Methode `addString` gibt, die den Ergebnistyp `std::string` und zwei Parameter vom Typ `String` hat:

StringUtilMethod.cpp

```
1  #include <string>
2
3  class StringUtilMethod {
4   public:
5    std::string addStrings
6              (std::string leftText, std::string rightText){
7      return leftText+rightText;
8    }
9  };
10
```

Zugriff auf Felder im Methodenrumpf In einer Methode stehen die Felder der Klasse zur Verfügung.

Beispiel:
Wir können mit den bisherigen Mitteln eine kleine Klasse definieren, die es erlaubt, Personen zu repräsentieren, so daß die Objekte dieser Klasse eine Methode haben, um den vollen Namen der Person anzugeben:

PersonExample1.cpp

```
1  #include <string>
2
3  class PersonExample1 {
4   public:
5    std::string vorname;
6    std::string nachname;
7
8    std::string getFullName(){
9      return (vorname+" "+nachname);
10   }
11 };
12
```

Methoden ohne Rückgabewert Es lassen sich auch Methoden schreiben, die keinen eigentlichen Wert berechnen, den sie als Ergebnis zurückgeben. Solche Methoden haben keinen Rückgabetyp. In C++ wird dieses gekennzeichnet, indem das Schlüsselwort `void` statt eines Typnamens in der Deklaration steht. Solche Methoden haben keine `return`-Anweisung.

Beispiel:
Folgende kleine Beispielklasse enthält zwei Methoden zum Setzen neuer Werte für ihre Felder:

PersonExample2.cpp

```
#include <string>

class PersonExample2 {
 public:
  std::string vorname;
  std::string nachname;

  void setVorname(std::string newName){
    vorname = newName;
  }
  void setNachname(std::string newName){
    nachname = newName;
  }
};

```

Obige Methoden weisen konkrete Objekte den Feldern des Objektes zu.

2.1.3 Konstruktoren

Wir haben oben gesehen, wie prinzipiell Objekte einer Klasse mit dem `new`-Konstrukt erzeugt werden. In unserem obigen Beispiel würden wir gerne bei der Erzeugung eines Objektes gleich konkrete Werte für die Felder mit angeben, um direkt eine Person mit konkreten Namen erzeugen zu können. Hierzu können Konstruktoren für eine Klasse definiert werden. Ein Konstruktor kann als eine besondere Art der Methode betrachtet werden, deren Name der Name der Klasse ist und für die kein Rückgabetyp spezifiziert wird. So läßt sich ein Konstruktor spezifizieren, der in unserem Beispiel konkrete Werte für die Felder der Klasse übergeben bekommt:

Person1.cpp

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

class Person1 {
 public:
  std::string vorname;
  std::string nachname;

  Person1(std::string derVorname, std::string derNachname){
    vorname  = derVorname;
    nachname = derNachname;
  }

  std::string getFullName(){
    return (vorname+" "+nachname);
  }
};
```

Jetzt lassen sich bei der Erzeugung von Objekten des Typs `Person1` konkrete Werte für die Namen übergeben.

Beispiel:
Wir erzeugen ein Personenobjekt mit dem für die entsprechende Klasse geschriebenen Konstruktor:

Person1.cpp

```
18 int main(){
19   Person1* p1 =  new Person1("Nicolo","Paganini");
```

Wie man sieht, machen wir mit diesem Personenobjekt noch nichts. Das Programm hat keine Ausgabe oder Funktion. Dieses werden wir im nächsten Abschnitt hinzufügen.

Syntaktischer Zucker für Konstruktoren

Typischer Weise bekommen Konstruktoren Werte für die Felder einer Klasse. In der Regel werden im Konstruktor die übergebenen Werte den Feldern zugewiesen. Wir erhalten Zeilen der Form:

```
nachname=derNachname;
```

In C++ gibt es eine Notation, die speziell für solche Fälle vorgesehen ist. Vor dem Rumpf des Konstruktors können diese Zuweisungen in der Form: *feldName(wert)* erfolgen. Man spricht auch von der Initialisierung der Felder. Die Feldinitialisierungen stehen zwischen Parameterliste des Konstruktors und dem Konstruktorrumpf. Sie werden per Doppelpunkt von der Parameterliste abgetrennt und untereinander durch Komma getrennt.

Den Konstruktor der Klasse Person1 hätten wir demnach auch wie folgt schreiben können:

Person2.cpp

```
1  #include <string>
2  #include <iostream>
3
4  class Person2 {
5   public:
6    std::string vorname;
7    std::string nachname;
8
9    Person2(std::string derVorname, std::string derNachname)
10    :vorname(derVorname),nachname(derNachname){}
11 };
```

2.1.4 Zugriff auf Eigenschaften eines Objektes

Wir wissen jetzt, wie Klassen definiert und Objekte einer Klasse erzeugt werden. Es fehlt uns schließlich noch, die Eigenschaften eines Objektes anzusprechen. Wenn wir ein Zeiger auf ein Objekt haben, lassen sich die Eigenschaften dieses Objekts über den Pfeiloperator `->` und den Namen der Eigenschaften ansprechen.

Feldzugriffe

Wenn wir also ein Objekt in einem Feld `x` abgelegt haben und das Objekt ist vom Typ `Person1*`, der ein Feld `vorname` hat, so erreichen wir das Objekt, das im Feld `vorname` gespeichert ist, durch den Ausdruck: `x->vorname`.

Person1.cpp

```
1  Person1* p2 = new Person1("August","Strindberg");
2  std::string name = p1->vorname;
3  std::cout<< name<<std::endl;
4  name = p2->nachname;
5  std::cout<< name<<std::endl;
```

Methodenaufrufe

Ebenso können wir auf den Rückgabewert einer Methode zugreifen. Wir nennen dieses dann einen Methodenaufruf. Methodenaufrufe unterscheiden sich syntaktisch darin von Feldzugriffen, daß eine in runden Klammern eingeschlossene Argumentliste folgt:

Person1.cpp

```
6  name = p2->getFullName();
7  std::cout<< name<<std::endl;
8  }
9
```

Aufgabe 9 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

- Personen mit Namen, Vornamen, Straße, Ort und Postleitzahl.
- Bücher mit Titel und Autor.
- Datum mit Tag, Monat und Jahr.
- Buchausleihe mit Ausleiher, Buch und Datum.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine

Methode `std::string toString()` mit dem Ergebnistyp `std::string`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.

c) Schreiben Sie eine Hauptmethode, in der Sie Objekte für jede der obigen Klassen erzeugen und die Ergebnisse der `toString`-Methode auf den Bildschirm ausgeben.

2.1.5 der this-Bezeichner

Eigenschaften sind an Objekte gebunden. Es gibt in C++ eine Möglichkeit, in Methodenrümpfen über das Objekt, für das eine Methode aufgerufen wurde, zu sprechen. Dieses geschieht mit dem Schlüsselwort `this`. *this* ist zu lesen als: dieses Objekt, in dem du dich gerade befindest. Häufig wird der `this`-Bezeichner in Konstruktoren benutzt, um den Namen des Parameters des Konstruktors von einem Feld zu unterscheiden:

UseThis.cpp

```
1  #include <string>
2  #include <iostream>
3
4  class UseThis {
5   public:
6    std::string aField ;
7    UseThis(std::string aField){
8      this->aField = aField;
9    }
10 };
11 int main(){
12   std::cout << (new UseThis("hallo! ")) -> aField << std::endl;
13 }
14
```

this-Bezeichner in Konstruktoren vermeiden

Über die besondere Inititialisierungssyntax für Konstruktoren lassen sich Zeilen wie:

```
this->aField = aField;
```

im Konstruktorrumpf vermeiden. Im Konstrukt `feldName(feldWert)` kann sich `feldName` nur auf ein Feld der Klasse beziehen.

UseThis2.cpp

```
1  #include <string>
2  #include <iostream>
3
4  class UseThis2 {
5   public:
6    std::string aField ;
7    UseThis2(std::string aField):aField(aField){}
8  };
9  int main(){
10   std::cout << (new UseThis2("hallo! ")) -> aField << std::endl;
11 }
12
```

2.2 Kopfdateien für Klassen

In der Kopfdatei zu einer Bibliothek schreibt man in der Regel die Klasse mit allen ihren Eigenschaften, läßt jedoch für jede Methode den Methodenrumpf fort. Für die Klasse `Person`

erhalten wir so die folgende Kopfdatei.[3]

P2.h

```
#include <string>

class Person {

public:
  std::string nachname;
  std::string vorname;
  std::string getFullName();
  std::string toString();
  Person(std::string n,std::string v);
};
```

In der eigentlichen Bibliotheksdatei, braucht die Klassendefinition nun nicht mehr wiederholt zu werden. Sie kann ja inkludiert werden. Die fehlenden Funktionsimplementierungen können jetzt getätigt werden. Hierzu ist den Methoden jeweils anzuzeigen, zu welcher Klasse sie gehören. Dieses geschieht durch das Voranstellen des Klassennamens vor den Funktionsnamen. Als Seperator wird ein zweifacher Doppelpunkt benutzt. Für die Klasse Person erhalten wir die folgende Implementierungsdatei:

P2.cpp

```
#include "P2.h"

std::string Person::getFullName(){return vorname+" "+nachname;}
std::string Person::toString(){
  return "Person: "+getFullName();}

Person::Person(std::string n,std::string v){
  vorname=v;
  nachname=n;
}

```

Jetzt kann eine Klasse benutzt werden, indem man lediglich die Kopfdatei inkludiert:

TestP2.cpp

```
#include "P2.h"
#include <iostream>

int main(){
  Person* p1 = new Person("Walther","Zabel");
  std::cout << p1 -> getFullName() << std::endl;
  Person* p2 = p1;
  std::cout << p2 -> getFullName() << std::endl;
  p2 -> nachname = "Hotzenplotz";
  std::cout << p1->getFullName() << std::endl;
  std::cout << p2->getFullName() << std::endl;
}
```

[3] Üblich ist auch die Dateiendung `.hpp` für C++ Kopfdateien.

Das Programm verhält sich wieder wie folgt:

```
sep@pc216-5:~/fh/cpp/tutor> bin/TestP2
Zabel Walther
Zabel Walther
Zabel Hotzenplotz
Zabel Hotzenplotz
sep@pc216-5:~/fh/cpp/tutor>
```

2.2.1 Verschränkt rekursive Klassen

Ein C++-Compiler liest eine Quelltext immer nur genau von vorne bis hinten durch. Daher dürfen Methoden und Funktionen erst benutzt werden, wenn sie deklariert wurden. Gleiches gilt für Klassen. Somit muß es eine Möglichkeit geben, einen Klassennamen bekannt zu machen, ohne schon den Inhalt er Klasse zu spezifizieren, damit es möglich ist, verschränkt rekursive Klassen zu definieren.

Beispiel:
Als kleines Beispiel schreiben wir zwei klassen die einen Mann bzw. eine Frau repräsentieren. In den beiden Klassen gibt es jeweils ein Feld von der anderen Klasse. Wir haben die Möglichkeit, dem C++-Compiler zunächst einmal die Klasse `Mann` bekannt zu machen, ohne ihren Inhalt zu spezifizieren. Dann können wir bei der Definition der Klasse `Frau` von der Existenz der Klasse `Mann` ausgehen. Schließlich holen wir die eigentliche Spezifikation der Klasse `Mann` nach.

Heirat.h

```
1  #include <string>
2
3  class Mann;
4
5  class Frau{
6   public:
7     std::string name;
8     Mann* verheiratetMit;
9     Frau(std::string name);
10 };
11
12 class Mann{
13  public:
14    std::string name;
15    Frau* verheiratetMit;
16    Mann(std::string name);
17 };
18 void heirat(Mann* m,Frau* f);
```

Das folgende kleine Programm implementiert und testet die verschränkt sich aufeinander beziehenden Klassen `Mann` und `Frau`.

Heirat.cpp

```
1  #include "Heirat.h"
2  #include <iostream>
3
```

```
4  Mann::Mann(std::string name):name(name),verheiratetMit(NULL){}
5  Frau::Frau(std::string name):name(name),verheiratetMit(NULL){}
6
7  void heirat(Mann* m,Frau* f){
8    m->verheiratetMit=f;
9    f->verheiratetMit=m;
10 }
11
12 int main(){
13   Mann* adam = new Mann("Adam");
14   Frau* eva  = new Frau("Eva");
15   heirat(adam,eva);
16   std::cout<<adam->name<<" ist mit "
17            <<adam->verheiratetMit->name
18            <<" verheiratet."<<std::endl;
19 }
20
```

2.2.2 Statische Eigenschaften

2.3 Destruktoren

2.3.1 der Operator delete

Mit dem `new`-Konstrukt reservieren wir Speicherplatz für ein Objekt einer Klasse und lassen uns die Adresse dieses Speicherplatzes in Form eines Zeigers geben. C++ kennt ebensowenig wie C eine automatische Speicherverwaltung, die erkennt, wenn Speicherplatz, der mit `new` reserviert wurde, nicht mehr gebraucht wird. Wenn die Daten, die in diesem Speicherplatz abgelegt sind, nicht mehr im weiteren Verlauf des Programms benötigt werden. Damit wir nicht unkontrolliert immer neuen Speicherplatz anfordern, ist es notwendig, diesen wieder freizugeben. Hierzu gibt es einem zu `new` dualen Operator, den Operator `delete`. Dieser wird auf einen Zeiger angewendet und sorgt dafür, daß der Speicherplatz wieder für neue Benutzung freigegeben wird.

Wir können mit mit recht simplen Programmen testen, was passiert, wenn wir immer neuen Speicher anfordern, diesen aber nicht mehr freigeben. Hierzu schreiben wir einmal ein Programm, das hundert millionen Objekte erzeugt und deren Speicher nicht wieder frei gibt:

Undeleteted.cpp

```
1  class Minimal {};
2
3  int main(){
4    for (int i=0;i<100000000;i++)
5      new Minimal();
6  }
```

Und ein zweites Programm, das genauso viele Objekte erzeugt, den für sie reservierten Speicher aber sofort wieder frei gibt.

Deleteted.cpp

```
class Minimal {};
int main(){
  for (int i=0;i<100000000;i++){
    Minimal* m=new Minimal();
    delete m;
  }
}
```

Starten wir diese Programme, so stellen wir fest, daß `Undeleted` mehrere 1000 Megabyte Speicher belegt und unseren ganzen Rechner lahmlegt, `Deleted` hingegen in konstanten Speicher von wenigen hundert Bytes nach wenigen Sekunden beendet wird.

2.3.2 Destruktoren definieren

Ebenso, wie durch die Definition eines Konstruktors gesteuert werden kann, was an zusätzlicher Initialisierung bei einem Aufruf des Operators `new` für das erzeugte Objekt vorgenommen wird, kann man durch die Spezifikation eines Destruktors für eine Klasse beeinflussen, welcher Code ausgeführt wird, wenn der Operator `delete` für ein Objekt aufgerufen wird. Syntaktisch hat der Destruktor den Namen der Klasse mit einer vorangestellten Tilde `~`.

Beispiel:
In der folgenden Klasse gibt es einen Konstruktor und einen Destruktor, die jeweils davon, wenn sie aufgerufen werden, auf der Konsole eine Ausgabe tätigen.

Destruction.cpp

```
#include <iostream>
class Destruction {
 public:
  Destruction(){std::cout<<"Destruction erzeugt"<<std::endl;}
  ~Destruction(){std::cout<<"Destruction gelöscht"<<std::endl;}
};
int main(){
  for (int i=0;i<100;i++){
    Destruction* m=new Destruction();
    delete m;
  }
}

```

Starten wir dieses Programm, können wir sehen, daß `new` und `delete` den Code von Konstruktor bzw. Destruktor aufrufen.

Sehr häufig enthalten Objekte Zeiger, auf weitere Objekte. Wenn das äußere Objekt durch ein `delete` aus dem Speihcer gelöscht wird, so will man in der Regel auch Objekte auf die das zu löschende Objekt referenziert wieder löschen. Hierzu kann man innerhalb des Destruktors auch wieder Aufrufe des Operators `delete` tätigen. Dann stößt dieses einen weiteren Aufruf eines Destruktors an.

Beispiel:
Wir schreiben eine Klasse, in der es eine Referenz auf ein weiteres Objekt dieser Klasse gibt. Im Destruktor wird dieses referenzierte Objekt zunächst aus dem Speicher gelöscht.

DeleteChildren.cpp

```
1  #include <iostream>
2
3  class DeleteChildren{
4   public:
5    bool isEnd;
6    DeleteChildren* next;
7    DeleteChildren():isEnd(true),next(NULL){}
8    DeleteChildren(DeleteChildren* nect):isEnd(false),next(next){}
9    ~DeleteChildren(){
10     if (!isEnd) delete next;
11     std::cout<<"deleted"<<std::endl;
12   }
13 };
14
15 DeleteChildren* create(int last){
16   DeleteChildren* result=new DeleteChildren();
17   for (int i=0;i<last;i++){
18     result=new DeleteChildren();
19   }
20   return result;
21 }
22
23 int main(){
24   DeleteChildren* x = create(100);
25   delete x;
26 }
```

Starten wir dieses Programm, so können wir beobachten, daß alle mit `new` erzeugten Objekte auch wieder gelöscht werden.

2.4 Objekte direkt, Ohne Zeiger

Bisher haben wir, mit Ausnahme der Stringobjekte, Objekte nur über eine Zeigervariable angesprochen. Wir kennen bisher nur die Möglichkeit Objekte einer Klasse mit Hilfe des Operators `new` zu erzeugen. Dieser liefert uns einen Zeiger, auf den Speicherbereich, an dem das neue Objekt abgelegt wurde. Wir können aber auch in C++ direkt mit Objekten arbeiten. Dann ändert sich einiges. Objekte erhalten wir automatisch dadurch, daß wir Variablen ihres Typs deklarieren. Auf die Eigenschaften der entsprechenden Objekte wird nicht mit Pfeiloperator `->` sondern dem Punktoperator zugegriffen. Wie man sieht, können Objekt in der Weise benutzt werden, in der Strukturen in C verwendet wurden. Insbesondere bedeutet das, daß bei der Parameterübergabe eine Kopie des Objektes angelegt wird, ebenso bei der Zuweisung.

PointerLess.cpp

```
1  #include "P2.h"
2  #include <iostream>
```

```
3
4  void doesNotChangeName(Person x){x.vorname="Nathalie";}
5
6  int main(){
7    Person schmidt("Harald","Schmidt");
8    Person andrak("Manuel","Andrak");
9    std::cout << andrak.toString() << std::endl;
10   doesNotChangeName(andrak);
11   std::cout << andrak.toString() << std::endl;
12   schmidt=andrak;
13   schmidt.vorname="Nathalie";
14   std::cout << schmidt.toString() << std::endl;
15   std::cout << andrak.toString() << std::endl;
16 }
```

Starten wir dieses Programm, so erkennen wir gut die Kopiersemantik:

```
sep@pc216-5:~/fh/cpp/student> bin/PointerLess
Person: Andrak Manuel
Person: Andrak Manuel
Person: Nathalie Manuel
Person: Andrak Manuel
sep@pc216-5:~/fh/cpp/student>
```

Es gibt einen wichtigen Zusammenhang zwischen den Konstruktoren einer Klasse und der Deklaration von Variablen des Typs dieser Klasse. Sobald eine Variable des Objektstyps deklariert wird, ist auch das entsprechende Objekt zu erzeugen. Hierzu wird implizit der Konstruktor aufgerufen. Wenn nach dem Variablennamen keine Parameterliste folgt, so wird der Konstruktor ohne Parameter aufgerufen. Gibt es diesen nicht, so beschwert sich der Compiler darüber

PointerLessError.cpp

```
1  #include "P2.h"
2  #include <iostream>
3
4
5  int main(){
6    Person schmidt;
7    schmidt.nachname="Schmidt";
8    schmidt.vorname="Harald";
9  }
```

Für dieses Programm bekommen wir folgenden Compilerfehler:

```
PointerLessError.cpp: In function ‘int main()’:
PointerLessError.cpp:6: error: no matching function for call to ‘Person::Person
   ()’
P2.h:3: error: candidates are: Person::Person(const Person&)
P2.h:10: error:                Person::Person(std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >, std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >)
```

Die eigentlichen Konzepte der Objektorientierung, die wir im nächsten Abschnitt kennenlernen werden, können nur ausgenutzt werden, wenn wir die Objekte über einen Zeiger[4] ansprechen. Viele C++ Bibliotheken versuchen trotzdem ein API bereitzustellen, in dem es nicht notwendig ist mit Zeigern auf Objekten zu arbeiten. Dabei sind die Objekte oft in Containerklassen gekapselt, die wie ein Zeiger auf das eigentliche Objekt funktionieren. Die GUI-Bibliothek Gtkmm, die wir im zweiten Teil des Skripts kennenlernen werden, hat als Entwurfsgrundlage, möglichst zeigerfrei benutzt werden zu können.

2.5 Vererben und Erben

Eines der grundlegendsten Ziele der objektorientierten Programmierung ist die Möglichkeit, bestehende Programme um neue Funktionalität erweitern zu können. Hierzu bedient man sich der Vererbung. Bei der Definition einer neuen Klassen hat man die Möglichkeit, anzugeben, daß diese Klasse alle Eigenschaften von einer bestehenden Klasse erbt.

Wir haben in einer früheren Kapitel die Klasse `Person` geschrieben:

Wenn wir zusätzlich eine Klasse schreiben wollen, die nicht beliebige Personen speichern kann, sondern Studenten, die als zusätzliche Information noch eine Matrikelnummer haben, so stellen wir fest, daß wir wieder Felder für den Namen und die Adresse anlegen müssen; d.h. wir müssen die bereits in der Klasse `Person` zur Verfügung gestellte Funktionalität ein weiteres Mal schreiben:

StudentOhneVererbung.cpp

```
1  #include <string>
2
3  class StudentOhneVererbung {
4   public:
5    std::string vorname ;
6    std::string nachname;
7    int matrikelNummer;
8
9    StudentOhneVererbung
10          (std::string vorname, std::string nachname,int nr){
11     this->vorname = vorname;
12     this->nachname = nachname;
13     matrikelNummer = nr;
14   };
15   std::string getFullName(){
16     return (vorname+" "+nachname);
17   }
18   std::string toString(){
19     return    "Person:" +getFullName() +" ist ein Student";
20   }
21 };
22
```

Mit dem Prinzip der Vererbung wird es ermöglicht, diese Verdoppelung des Codes, der bereits für die Klasse `Person` geschrieben wurde, zu umgehen.

[4] oder den Referenztypen, den wir noch kennenlernen werden.

Wir werden in diesem Kapitel schrittweise eine Klasse `Student` entwickeln, die die Eigenschaften erbt, die wir in der Klasse `Person` bereits definiert haben.

Zunächst schreibt man in der Klassendeklaration der Klasse `Student`, daß deren Objekte alle Eigenschaften der Klasse `Person` erben. Hierzu wird das Symbol des Doppelpunkts `:` verwendet:

Student.h

```
1 #include "P2.h"
2
3 class Student:public Person {
```

Mit dieser `:`-Klausel wird angegeben, daß die Klasse von einer anderen Klasse abgeleitet wird und damit deren Eigenschaften erbt. Jetzt brauchen die Eigenschaften, die schon in der Klasse `Person` definiert wurden, nicht mehr neu definiert zu werden.

Mit der Vererbung steht ein Mechanismus zur Verfügung, der zwei primäre Anwendungen hat:

- **Erweitern**: zu den Eigenschaften der Oberklasse werden weitere Eigenschaften hinzugefügt. Im Beispiel der Studentenklasse soll das Feld `matrikelNummer` hinzugefügt werden.
- **Verändern**: eine Eigenschaft der Oberklasse wird umdefiniert. Im Beispiel der Studentenklasse soll die Methode `toString` der Oberklasse in ihrer Funktionalität verändert werden.

2.5.1 Hinzufügen neuer Eigenschaften

Unser erstes Ziel der Vererbung war, eine bestehende Klasse um neue Eigenschaften zu erweitern. Hierzu können wir jetzt einfach mit der `:`-Klausel angeben, daß wir die Eigenschaften einer Klasse erben. Die Eigenschaften, die wir zusätzlich haben wollen, lassen sich schließlich wie gewohnt deklarieren:

Student.h

```
4  public:
5   int matrikelNummer;
6   int getMatrikelNummer();
7   std::string toString();
8   Student(std::string vorname,std::string nachname,int nr);
9 };
```

Hiermit haben wir eine Klasse geschrieben, die drei Felder hat: `nachname` und `vorname`, die von der Klasse `Person` geerbt werden und zusätzlich das Feld `matrikelNummer`. Diese drei Felder können für Objekte der Klasse `Student` in gleicher Weise benutzt werden.

Student.cpp

```
1 #include "Student.h"
2 #include <iostream>
```

Ebenso so wie Felder lassen sich Methoden hinzufügen. Z.B. eine Methode, die die Matrikelnummer als Rückgabewert hat:

Student.cpp

```
3  int Student::getMatrikelNummer(){
4    return matrikelNummer;
5  }
```

2.5.2 Überschreiben bestehender Eigenschaften

Unser zweites Ziel ist, durch Vererbung eine Methode in ihrem Verhalten zu verändern. In unserem Beispiel soll die Methode `toString` der Klasse `Person` für Studentenobjekte so geändert werden, daß das Ergebnis auch die Matrikelnummer enthält. Hierzu können wir die entsprechende Methode in der Klasse `Student` einfach neu schreiben:

Student.cpp

```
6  std::string Student::toString(){
7    return    "Person:" +getFullName() +" ist ein Student";
8  }
```

Obwohl Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, benutzen sie nicht die Methode `toString` der Klasse `Person`, sondern die neu definierte Version aus der Klasse `Student`.

Um eine Methode zu überschreiben, muß sie dieselbe Signatur bekommen, die sie in der Oberklasse hat.

2.5.3 Konstruktion

Um für eine Klasse konkrete Objekte zu konstruieren, braucht die Klasse entsprechende Konstruktoren. In unserem Beispiel soll jedes Objekt der Klasse `Student` auch ein Objekt der Klasse `Person` sein. Daraus folgt, daß, um ein Objekt der Klasse `Student` zu erzeugen, es auch notwendig ist, ein Objekt der Klasse `Person` zu erzeugen. Wenn wir also einen Konstruktor für `Student` schreiben, sollten wir sicherstellen, daß mit diesem auch ein gültiges Objekt der Klasse `Person` erzeugt wird. Hierzu kann man den Konstruktor der Oberklasse aufrufen. Dieses geschieht, indem vor dem Rumpf des Konstruktors der der Konstruktor der Oberklasse aufgerufen wird. Für die Klasse `Student`, rufen wir so als erstes den Konstruktor der Klasse `Person` auf, bevor wir im Rumpf des Konstruktors noch das zusätzliche Feld initioalisieren.

Student.cpp

```
9  Student::Student(std::string vorname,std::string nachname,int nr)
10                                      :Person(vorname,nachname){
11   matrikelNummer = nr;
12 }
```

In unserem Beispiel bekommt der Konstruktor der Klasse `Student` alle Daten, die benötigt werden, um ein Personenobjekt und ein Studentenobjekt zu erzeugen.

Ein Objekt der Klasse `Student` kann wie gewohnt konstruiert werden:

TestStudent.cpp

```
13 #include "Student.h"
14 #include <iostream>
```

```
15
16 int main(){
17   Student* s = new Student("Martin"," Müller",755423);
18   std::cout<<s->toString()<<std::endl;
19 }
```

2.5.4 Zuweisungskompatibilität

Objekte einer Klasse sind auch ebenso Objekte ihrer Oberklasse. Daher können sie benutzt werden wie die Objekte ihrer Oberklasse, insbesondere bei einer Zuweisung. Da in unserem Beispiel die Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, dürfen diese auch Feldern des Typs `Person` zugewiesen werden:

TestStudent1.cpp

```
1 #include "Student.h"
2 #include <iostream>
3
4 int main(){
5   Person* p = new Student("Martin"," Müller",7463456);
6   std::cout<<p->toString()<<std::endl;
7 }
8
```

Alle Studenten sind auch Personen.

Hingegen die andere Richtung ist nicht möglich: nicht alle Personen sind Studenten. Folgendes Programm wird vom Compiler mit einem Fehler zurückgewiesen:

StudentError1.cpp

```
1 #include "Student.h"
2 #include <iostream>
3 int main(){
4   Student* s = new Person("Martin"," Müller");
5 }
6
```

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.cpp: In function ‘int main()’:
StudentError1.cpp:4: error: invalid conversion from ‘Person*’ to ‘Student*’
```

Der Compiler weist diese Klasse zurück, weil eine Person nicht unbedingt ein Student ist.

Gleiches gilt für den Typ von Methodenparametern. Wenn die Methode einen Parameter vom Typ `Person` verlangt, so kann man ihm auch Objekte eines spezielleren Typs geben, in unserem Fall der Klasse `Student`.

TestStudent2.cpp

```
#include "Student.h"
#include <iostream>

void printPerson(Person* p){
  std::cout<< p->toString() <<std::endl;
}

int main(){
  Student* s = new Student("Martin"," Müller",754545);
  printPerson(s);
}

```

Der umgekehrte Fall ist wiederum nicht möglich. Methoden, die als Parameter Objekte der Klasse `Student` verlangen, dürfen nicht mit Objekten einer allgemeineren Klasse aufgerufen werden:

StudentError2.cpp

```
#include "Student.h"
#include <iostream>

void printStudent(Student* p){
  std::cout<<p->toString());
}

int main(){
  Person* s
     = new Person("Martin"," Müller");
  printStudent(s);
}

```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

```
StudentError2.cpp: In function ‘int main()’:
StudentError2.cpp:11: error: invalid conversion from ‘Person*’ to ‘Student*’
```

2.5.5 Zugriff auf Methoden der Oberklasse

Vergleichen wir die Methoden `toString` der Klassen `Person` und `Student`, so sehen wir, daß in der Klasse `Student` Code der Oberklasse verdoppelt wurde. Aus der Klasse `Person`:

```
std::string Person::toString(){
  return "Person: "+getFullName();}
```

wird in der Klasse `Student` der komplette Anfang übernommen uns nur noch angehängt, daß es sich um einen Studenten handelt:

```
1 std::string Student::toString(){
2   return    "Person:" +getFullName() +" ist ein Student";}
```

Der Ausdruck `"Person:" +getFullName()` wiederholt die Berechnung der `toString`-Methode aus der Klasse `Person`. Es wäre schön, wenn an dieser Stelle die entsprechende Methode aus der Oberklasse benutzt werden könnte. Auch dieses ist in C++ möglich. Ähnlich, wie der Konstruktor der Oberklasse explizit aufgerufen werden kann, können auch Methoden der Oberklasse explizit aufgerufen werden. Auch in diesem Fall ist das durch dem mit einem dopppelten Doppelpunkt der Methode vorangestellten Klassennamens als Qualifikation ausgedrückt. Somit läßt sich die `toString`-Methode der Klasse `Student` auch wie folgt schreiben:

```
1   std::string Student::toString(){
2     return      //call toString of super class
3                 Person::toString()
4                 //add Student information
5             + " ist ein Student";
6   }
```

2.5.6 Späte Bindung (late binding) durch virtuelle Methoden

Wir haben gesehen, daß wir Methoden überschreiben können. Interessant ist, wann welche Methode ausgeführt wird. In unserem Beispiel gibt es je eine Methode `toString` in der Oberklasse `Person` als auch in der Unterklasse `Student`.

Welche dieser zwei Methoden wird wann ausgeführt? Wir können dieser Frage experimentell nachgehen:

TestNoLateBinding.cpp

```
1  #include "Student.h"
2  #include <string>
3  #include <iostream>
4
5  int main(){
6    Student* s = new Student("Martin"," Müller",756456);
7    Person* p1 = new Person("Harald","Schmidt");
8
9    std::cout<<s->toString()<<std::endl;
10   std::cout<<p1->toString()<<std::endl;
11
12   Person* p2 = new Student("Martin","Müller",756456);
13   std::cout<<p2->toString()<<std::endl;
14 }
15
```

Dieses Programm erzeugt folgende Ausgabe:

```
sep@pc216-5:~/fh/cpp/tutor> bin/TestNoLateBinding
```

```
Person: Müller Martin ist ein Student
Person: Schmidt Harald
Person: Müller Martin
sep@pc216-5:~/fh/cpp/tutor>
```

Die ersten beiden Ausgaben entsprechen sicherlich den Erwartungen: es wird eine Student und anschließend eine Person ausgegeben. Die dritte Ausgabe ist interessant. Wir haben ein Objekt vom Type Student erzeugt und dieses über eine Variablen vom Typ `Person*` aufgerufen. Wie zu sehen ist, wurde die Methode `toString` der Klasse `Person` ausgeführt und nicht der Klasse `Student`.

virtuelle Methoden

Jetzt werden wir die Klassen `Student` und `Person` minimal verändern: die Methode `toString` wird mit dem Attribut `virtual` markiert. Ansonsten haben wir mehr oder weniger die identische Implementierung zu oben:

P3.h

```
1  #include <string>
2  class Person {
3   public:
4    std::string nachname;
5    std::string vorname;
6    virtual std::string toString();
7    Person(std::string n,std::string v);
8  };
9  class Student:public Person {
10  public:
11   int matrikelNummer;
12   virtual std::string toString();
13   Student(std::string vorname,std::string nachname,int nr);
14 };
15
```

Entsprechend auhc die eigentliche Implemengtierung dieser Klassen:

P3.cpp

```
1  #include "P3.h"
2  Person::Person(std::string v,std::string n){
3    vorname=v;
4    nachname=n;
5  }
6
7  std::string Person::toString(){
8    return "Person: "+vorname+" "+nachname;}
9
10 Student::Student(std::string v,std::string n,int nr)
11                   :Person(v,n){
12   matrikelNummer=nr;}
13
```

```
14 std::string Student::toString(){
15   return Person::toString() + " ist ein Student";}
```

Jetzt betrachten wir wieder dasselbe Programm:

TestLateBinding.cpp

```
1  #include "P3.h"
2  #include <string>
3  #include <iostream>
4
5  int main(){
6    Person* p1 = new Person("Harald","Schmidt");
7    std::cout<<p1->toString()<<std::endl;
8
9    Student* s = new Student("Martin","Müller",756456);
10   std::cout<<s->toString()<<std::endl;
11
12   Person* p2 = new Student("Martin","Müller",756456);
13   std::cout<<p2->toString()<<std::endl;
14 }
15
```

Nun ist die Ausgabe eine andere:

```
sep@pc216-5:~/fh/cpp/tutor> ./bin/TestLateBinding
Person: Harald Schmidt
Person: Martin Müller ist ein Student
Person: Martin Müller ist ein Student
sep@pc216-5:~/fh/cpp/tutor>
```

Obwohl der Befehl:

```
1 std::cout<<p2->toString();
```

die Methode `toString` auf einem Feld vom Typ `Person*` ausführt, wird die Methode `toString` aus der Klasse `Student` ausgeführt. Dieser Effekt entsteht, weil das Objekt, das im Feld `p2` gespeichert wurde, als Student und nicht als Person erzeugt wurde. Die Idee der Objektorientierung ist, daß die Objekte die Methoden in sich enthalten. In unserem Fall enthält das Objekt im Feld `p2` seine eigene `toString`-Methode. Diese wird ausgeführt, weil mit dem Attribut `virtual` der Methode in Klasse `Person` markiert wurde, daß eventuell ein Objekt seine eigene Methode `toString` enthält und dann diese vorzuziehen ist. Der Ausdruck `p2->toString()` ist also zu lesen als:

> Objekt, das in Feld `p2` referenziert wird, führe bitte deine Methode `toString` aus.

Da dieses Objekt, auch wenn wir es dem Feld nicht ansehen, ein Objekt der Klasse `Student` ist, führt es die entsprechende Methode der Klasse `Student` und nicht der Klasse `Person` aus.

Dieses Prinzip der virtuellen Mehtoden, wird als *late binding* bezeichnet. Die Bindung erfolgt in dem Sinne spät, weil der Compiler eventuell gar nicht weiß, welche Methode schließlich

ausgeführt wird, sondern dieses erst zur Laufzeit aufgelöst wird.[5] Der Compiler muß noch nicht einmal etwas von der Existenz der Klasse `Student` wissen.

Mit dem Prinzip des *late-bindings* lassen sich hervorragend Bibliotheken modellieren. Die Bibliothek stellt die allgemeinen Fälle als Klassen bereit, in der konkreten Anwendung können spezialisierte Unterklassen dieser allgemeinen Klassen definiert werden. Die nächste Aufgabe illustriert dieses am Beispiel einer minimalen GUI-Implementierung.

Aufgabe 10 (4 Punkte) In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden C++ Klassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

ButtonLogic.h

```
#ifndef _BUTTON_LOGIC__H
#define _BUTTON_LOGIC__H
#include <string>
class ButtonLogic {
 public:
  virtual std::string getDescription();
  virtual std::string eval(std::string x);
};
#endif

```

ButtonLogic.cpp

```
#include "ButtonLogic.h"
std::string ButtonLogic::getDescription(){return "Drück mich";}
std::string ButtonLogic::eval(std::string x){return x;}
```

Dialogue.h

```
#include <gtkmm.h>
#include "ButtonLogic.h"

class Dialogue : public Gtk::Window{

public:
  Dialogue(ButtonLogic* l);

protected:
  ButtonLogic* logic;
  virtual void on_button_clicked();
  Gtk::Entry entry;
  Gtk::Button button;
  Gtk::Label  label;
  Gtk::VBox  box;
};
```

[5] Achtung: *late binding* ist in Java der Standard für alle Methoden. In Java sind alle Methoden virtuell, ohne daß es deklariert werden muß.

Dialogue.cpp

```
#include "Dialogue.h"
#include <iostream>
#include <string>

Dialogue::Dialogue(ButtonLogic* l)
  : logic(l)
  , button(l->getDescription())
  , label(""){
  set_border_width(10);

  button.signal_clicked()
        .connect(sigc::mem_fun
              (*this, &Dialogue::on_button_clicked));

  box.add(entry);
  box.add(button);
  box.add(label);
  add(box);
  show_all();
}

void Dialogue::on_button_clicked(){
  label.set_text(logic->eval(entry.get_text()));
}
```

TestDialogue.cpp

```
#include <gtkmm/main.h>
#include "Dialogue.h"

int main (int argc, char *args[]){
  Gtk::Main kit(argc,args);
  Dialogue dialogue(new ButtonLogic());
  Gtk::Main::run(dialogue);
}

```

a) Übersetzen Sie die drei obigen Klassen und starten Sie das Programm. Hierzu müssen Sie den Compiler beim Übersetzen der C-Quelltexte mit dem Kommandozeilenargument ‘`pkg-config gtkmm-2.4 --cflags`‘ und beim Linken mit ‘`pkg-config gtkmm-2.4 --libs`‘ aufrufen.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, daß der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie römische Zahlen in arabische Zahlen umwandeln können. Benutzen Sie hierzu die Funktionen

`romanToInt` aus dem Einführungskapitel und `itos` aus dem nächsten Kapitel. Testen Sie Ihr Programm.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie arabische Zahlen in römische Zahlen umwandeln können. Testen Sie Ihr Programm.

e) Schreiben Sie jetzt ein Guiprogramm, daß eine Zahl aus ihrer Darstellung zur Basis 10 in eine Darstellung zur Basis 2 umwandelt. Testen Sie.

2.5.7 abstrakte Methoden

In der letzten Aufgabe sollten Unterklassen der Klasse `ButtonLogic` schreiben. In dieser gab es zwei Methoden, die Sie dabei überschrieben haben. In der Klasse `ButtonLogic` gab es auch Implementierungen dieser Methoden, jedoch waren diese relativ allgemein gehalten: `getDescription` hat einen relativ allgemeinen Text zurückgegeben und `eval` war die Identität. Eigentlich wurde die Klasse nur dazu geschrieben, um zu spezifizieren, welche zwei Methoden die Unterklassen zu implementieren haben. Es ging also allein um Schnittstelle. Diese Situation kommt in Bibliotheken relativ häufig vor. C++ bietet hierzu abstrakte Methoden an. Eine abstrakte Methode in einer Klasse hat nur eine Signatur, es gibt keine Implementierung. Syntaktisch wird das durch `=0` nach der Signatur ausgedrückt. Eine abstrakte Methode hat keinen Methodenrumpf. Somit hätten wir die Klasse `ButtonLogic` wie folgt definieren können:

ButtonLogicAbstr.h

```
1 #include <string>
2 class ButtonLogicAbstr {
3  public:
4   virtual std::string getDescription()=0;
5   virtual std::string eval(std::string x)=0;
6 };
7
```

Von Klassen, die abstrakte Methoden enthalten, können keine Objekte erzeugt werden. Wie denn auch, es gibt keinen Code für die abstrakten Methoden dieses Objekts. Diese Methoden könnten für das Objekt nicht ausgeführt werden.

ButtonLogicAbstrError.cpp

```
1 #include "ButtonLogicAbstr.h"
2 int main(){
3  ButtonLogicAbstr* bL = new ButtonLogicAbstr();
4 }
```

Daher fürhrt der Versuch ein Objekt einer Klasse mit abstrakten Methoden zu erzeugen zu einen Übersetzungsfehler:

```
ButtonLogicAbstrError.cpp: In function ‘int main()’:
ButtonLogicAbstrError.cpp:3: error: cannot allocate an object of type ‘
   ButtonLogicAbstr’
ButtonLogicAbstrError.cpp:3: error:   because the following virtual functions
   are abstract:
ButtonLogicAbstr.h:5: error:    virtual std::string
```

```
   ButtonLogicAbstr::eval(std::basic_string<char, std::char_traits<char>,
   std::allocator<char> >)
ButtonLogicAbstr.h:4: error:    virtual std::string
   ButtonLogicAbstr::getDescription()
```

Wir können nur Objekte von Unterklassen der Klasse `ButtonLogicAbstr` erzeugen, in denen die abstrakten Methoden eine konkrete Implementierung haben:

ButtonLogicAbstrOK.cpp

```
1  #include <iostream>
2  #include "ButtonLogicAbstr.h"
3
4  class MyButtonLogic: public ButtonLogicAbstr{
5   public:
6    std::string getDescription(){return "nun nicht mehr abstrakt";}
7    std::string eval(std::string x){return x.replace (3,0," HUCH");}
8  };
9
10 int main(){
11  ButtonLogicAbstr* bL = new MyButtonLogic();
12  std::cout<<bL->getDescription()<<" "
13           <<bL->eval("Na, was ist das")<<std::endl;
14 }
```

Bereits mehrfach haben wir Klassen geschrieben, in denen wir eine Methode `toString`, die eine textuelle Darstellung des Objekts generiert hat, implementiert haben. Wir können mit einer abstrakten Methode jetzt ausdrücken, daß ein Objekt die `toString`-Methode implementieren soll.

ToString.h

```
1  #include <string>
2  #ifndef __TO_STRING_H
3  #define __TO_STRING_H ;
4  class ToString{
5   public:
6    virtual std::string toString()=0;
7  };
8
9  std::string itos(int i);
10 #endif
```

Die Klasse `ToString` braucht keine eigene Implementierung, wir nutzen allerdings die Gelegenheit, eine Funktion zu schreiben, die eine ganze Zahl als einen `string` darstellen kann:

ToString.cpp

```
1  #include <string>
2  #include <sstream>
3
4  using namespace std;
5
```

```
6  // convert int to string
7  std::string itos(int i){
8    stringstream s; s << i;
9    return s.str();}
```

2.5.8 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

```
1  #include <iostream>
2  #include "ButtonLogicAbstr.h"
3  #include "ToString.h"
4
5  class MyToStringButtonLogic:public ButtonLogicAbstr,public ToString{
6   public:
7    std::string getDescription(){return "nun nicht mehr abstrakt";}
8    std::string eval(std::string x){return x.replace (3,0," HUCH");}
9    std::string toString(){return "Blödsinnige GUI Anwendung";}
10 };
11
12 int main(){
13   MyToStringButtonLogic* bL = new MyToStringButtonLogic();
14   std::cout<<bL->getDescription()<<" "
15            <<bL->eval("Na, was ist das")<<" "
16            <<bL->toString()<<" "<<std::endl;
17 }
```

Besonders in Zusammenhang mit Klassen, die nur abstrakte Methoden bereitstellen, hat mehrfaches Erben[6] 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

```
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->tuDas()<<std::endl;
10 }
```

[6] 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.

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
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, sonern nur Schnittstellenbeschreibungen.

2.6 Zeiger und Referenzen

Um noch einmal die Unterschiede der verschiedenen Typen in C++ verstehen zu können, müssen wir uns ein wenig konkreter mit dem Speichermodell eines Rechners beschäftigen.

2.6.1 Der Speicher

Der Hauptspeicher eines Computers besteht aus einzelnen durchnummerierten Speicherzellen. Im Prinzip stehen in jeder Speicherzelle nur Zahlen. Als was diese Zahlen interpretiert werden, hängt von dem Programm ab, das die Zahl in der Speicherzelle benutzt. Dabei kann die Zahl interpretiert werden als:

- tatsächlich eine Zahl im herkömmlichen Sinne.
- als ein Wert eines anderen primitiven Typs, als die diese Zahl interpretiert wird, z.B. einen Zeichen in einem Zeichensatz.
- als die Adresse einer Speicherzelle.
- als ein Befehl der Maschinensprache des Prozessors.

Die letzten beiden Interpretationen der Zahl in einer Speicherzelle schlagen sich nicht auf jede Programmiersprache durch. Es gibt keine Datentypen, die die Adresse einer Speicherzelle darstellen und es gibt keinen Datentyp, der für den Code einer Methode steht. In C findet sich dieses wieder.

Folgende kleine Tabelle illustriert den Speicher. Wir sehen die Speicherzellen 1017 bis 1026 und die angedeutete Bedeutung des Inhalts dieser Speicherzellen.

...	...
1017	42
1018	1025
1019	pop
1020	load r1
1022	15
1023	x
1024	1019
1025	A
1026	14
...	...

2.6.2 Zeiger

Deklaration von Zeigertypen

In C können Variablen angelegt werden, deren Inhalt die Adresse einer Speicherzelle ist. Der Typ einer solchen Variablen endet mit einem Sternzeichen `*`. Diesem vorangestellt ist Typ der Daten, die in dieser Speicherzelle liegen. So bezeichent der Typ `int*` die Adresse einer Speicherzelle, in der der Wert einer ganzen Zahl steht; im Gegensatz zum Typ `int`, der direkt eine ganze Zahl darstellt.

Erzeugen von Zeigern

Eine Variable mit der Deklaration `int i;` entspricht einer Speicherzelle für eine ganze Zahl. Mit dem Zeichen `&` als einstelligen Operator kann in C für eine Variable die Adresse dieser Speicherzelle erhalten werden. Der Operator `&` dient also dazu Zeigerwerte zu erzeugen.

> **Beispiel:**
> Wir deklarieren drei Variablen: eine vom Typ `int`, einen Zeiger auf eine Variable vom Typ `int` und einen Zeiger auf eine Variable, die wieder ein Zeiger ist. Für alle drei Variablen werden die Werte ausgegeben.

Zeiger1.cpp

```
1  #include <iostream>
2
3  int main(){
4    int i = 5;
5    int* pI = &i;
6    int** ppI = &pI;
7
8    std::cout << i
9      << ", " << pI
10     <<", "  << ppI
11     << std::endl;
12 }
```

Das Programm kann z.B. folgende Ausgabe haben:

```
sep@swe10:~/fh/prog3/examples> ./Zeiger1
5, 0xbffff1b8, 0xbffff1b4
sep@swe10:~/fh/prog3/examples>
```

Es empfiehlt sich, Variablen, die Zeiger sind, auch im Variablennamen als solche zu kennzeichnen. Eine gebräuchliche Methode ist, sie mit einem `p` für *pointer* beginnen zu lassen.

Zeiger auflösen

Der Wert eines Zeigers selbst, wie das obige Beispiel gezeigt hat, ist recht uninteressant. Es handelt sich um die Nummer einer Speicheradresse. Diese hat wenig mit der eigentlichen Programmlogik zu tun. Ein Zeiger ist in der Regel dafür interessant, um ihm dazu zu benutzen, wieder auf den Inhalt der Speicherzelle, auf die der Zeiger zeigt, zuzugreifen. Eine Zeigervariable sozusagen zu benutzen, um mal in die Zelle, dessen Adresse der Zeiger darstellt, hineinzuschauen. Hierzu stellt C das Malzeichen `*` als einstelligen Operator zur Verfügung. Er ist sozusagen der inverse Operator zum Operator `&`. Er überführt einen Zeiger auf eine Variable, wieder in den Wert der Variablen.

Beispiel:
Mit dem Operator `*` läßt sich wieder der Wert der Speicherzelle, auf den ein Zeiger weist, erhalten.

Zeiger2.cpp

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

```
11      <<", "  << ppI
12      << ", " << *ppI
13      << ", " << **ppI
14      << std::endl;
15  }
```

Wir bekommen z.B. folgende Ausgabe:

```
sep@swe10:~/fh/prog3/examples> ./Zeiger2
5, 0xbffff1b8, 5, 0xbffff1b4, 0xbffff1b8, 5
sep@swe10:~/fh/prog3/examples>
```

Zuweisungen auf Speicherzellen

Der Operator * kann auch in Ausdrücken auf der linken Seite des Zuweisungsoperators benutzt werden.

Beispiel:
Wir verändern eine Variable einmal dirket und zweimal über Zeiger auf diese Variable.

Zeiger3.cpp

```
1   #include <iostream>
2
3   int main(){
4     int i = 5;
5     int *pI = &i;
6     int **ppI = &pI;
7     std::cout << i << ", " << *pI << ", " << **ppI << std::endl;
8
9     i = 10;
10    std::cout << i << ", " << *pI << ", " << **ppI << std::endl;
11
12    *pI = 20;
13    std::cout << i << ", " << *pI << ", " << **ppI << std::endl;
14
15    **ppI = 42;
16    std::cout << i << ", " << *pI << ", " << **ppI << std::endl;
17  }
```

Das Programm erzeugt folgende Ausgabe. Die Änderung der Variablen wird durch jeden Zeigerzugriff auf sie sichtbar.

```
sep@swe10:~/fh/prog3/examples> ./Zeiger3
5, 5, 5
10, 10, 10
20, 20, 20
42, 42, 42
sep@swe10:~/fh/prog3/examples>
```

Rechnen mit Zeigern

Zeiger sind technisch nur die Adressen bestimmter Speicherzellen, also nichts weiteres als Zahlen. Und mit Zahlen können wir rechnen. So können tatsächlich arithmetische Operationen auf Zeigern durchgeführt werden.

Beispiel:
Wir erzeugen mehrere Zeiger auf ganze Zahlen und betrachten die Nachbarschaft eines dieser Zeiger.

Zeiger4.cpp

```
1  #include <iostream>
2
3  int main(){
4    int i = 5;
5    int j = 42;
6    int *pI = &i;
7    int *pJ = &j;
8
9    int *pK;
10
11   std::cout << i
12     << ", " << pI
13     << ", " << pI+1
14     <<", "  << *(pI+1)
15     << ", " << pI-1
16     <<", "  << *(pI-1)
17     << ", " << pJ
18     << ", " << *pJ
19     << std::endl;
20
21   *pK = 12;
22     std::cout << pK <<", " << *pK << std::endl;
23 }
```

Dieses Programm erzeugt z.B. folgende interessante Ausgabe:

```
sep@swe10:~/fh/prog3/examples> ./Zeiger4
5, 0xbffff1b8, 0xbffff1bc, -1073745416, 0xbffff1b4, 42, 0xbffff1b4, 42
0xbffff1c8, 12
sep@swe10:~/fh/prog3/examples>
```

Das Ergebnis einer Zeigerarithmetik hängt sehr davon ab, wie die einzelnen Variablen im Speicher abgelegt werden. In der Regel wird man die Finger von ihr lassen und nur ganz bewußt auf sie zurückgreifen, wenn man genau weiß, was man tut, um eventuell Optimierungen durchführen zu können.

Zeiger als Parameter

Wenn man in C einer Funktion einen primitiven Typ als Parameter übergibt, sind innerhalb des Methodenrumpf gemachte Zuweisungen außerhalb der Funktion nicht sichtbar:

Beispiel:
Folgendes Programm gibt die `0` und nicht die `42` aus.

CIntAssign.cpp

```
1  #include <iostream>
2
3  void f(int x){x=42;}
4
5  int main(){
6    int y = 0;
7    f(y);
8    std::cout << y << std::endl;
9  }
```

Der Grund für dieses Verhalten ist, daß der Wert der ganzen Zahl, an die Funktion `f` übergeben wird, und nicht die Adresse, an der diese Zahl steht.

Mit Zeigern kann man in C die Adresse, in der eine Zahl steht, als Parameter übergeben:

CPIntAssign.cpp

```
1  #include <iostream>
2
3    void f(int* pX){*pX=42;}
4
5    int main(){
6      int y = 0;
7      int *pY = &y;
8      f(pY);
9      std::cout << y << std::endl;
10   }
```

Dadurch, daß wir einen Zeiger auf eine Speicherzelle übergeben, können wir diesen Zeiger benutzen, um die Speicherzelle zu verändern. Jetzt ist das Ergebnis die `42`. Die Variable `y` wird effektiv geändert.

Im Vergleich dazu in Java haben wir keine Zeiger direkt zur Verfügung. Wollen wir den Wert einer ganzen Zahl, der als Parameter einer Methode übergeben wird, effektiv in dieser Methode ändern, so geht das nur, wenn diese ganze Zahl das Feld eines Objektes ist, daß der Methode übergeben wird.

Beispiel:
Eine typische Klasse, die nur Objekte für eine ganze Zahl erzeugt:

IntBox.java

```
1  class IntBox{
2    int value;
3    public IntBox(int v){value=v;}
4  }
```

Objekte dieser Klasse können Methoden übergeben werden. Dann kann das Objekt als Referenz auf seine Felder benutzt werden, so daß die Felder in diesem Objekt effektiv verändert werden können.

JIntBoxAssign.java

```
class JIntBoxAssign {
  static void f(IntBox xBox){xBox.value=42;}
  public static void main(String [] _){
    IntBox yBox = new IntBox(0);
    f(yBox);
    System.out.println(yBox.value);
  }
}
```

Die Ausgabe dieses Programms ist `42`.

Dynamisch Adresse anfordern

new Operation Bisher haben wir Zeiger immer über den Operator `&` erzeugt, d.h. wir haben für eine bestehende Variable uns die Adresse geben lassen. Variablen werden ein für allemal fest im Programmtext definiert. Ihre Anzahl ist also bereits statisch im Programmtext festgelegt. C erlaubt eine dynamische Anforderung von Speicherzellen. Hierzu gibt es den Operator `new`. Ihm folgt der Name eines Typs. Er erzeugt einen neuen Zeiger auf eine Speicherzelle bestimmten Typs.

`new` bewirkt, daß vom Betriebssystem eine neue Speicherzelle bereitgestellt wird. In dieser Speicherzelle können dann Werte gespeichert werden.

Beispiel:
`new` ist gefährlich, denn jedesmal, wenn der Operator `new` ausgeführt wird, bekommen wir eine frische Speicherzelle für die Programmausführung zugewiesen. Wenn wir das wiederholt machen, so werden wir immer mehr Speicher vom Betriebssystem anfordern. Irgendwann wird das Betriebssystem keinen Speicher mehr zur Verfügung stellen können. Folgendes Programm durchläuft eine Schleife und fordert jedesmal eine neue Speicherzelle an.

News.cpp

```
#include <iostream>

int main(){
    while (true){
        int *pX = new int;
        std::cout << pX << std::endl;
    }
}
```

Startet man dieses Programm und betrachtet parallel (z.B. mit dem Befehl `top` auf Unix oder im *Task Manager* von Windows) die Systemresourcen, so wird man sehen, daß der vom Programm `News` benutzte Bereich des Hauptspeichers immer mehr anwächst, bis das Programm den ganzen Hauptspeicher belegt und das Betriebssystem lahmlegt.

Solche Situationen, in denen ein Programm immer mehr Speicher anfordert, ohne diesen wirklich zu benötigen, nennt man auf Englisch *space leak*.

delete Operation Mit `new` läßt sich in C eine frische Speicherzelle vom Betriebssystem anfordern. Damit das nicht unkontrolliert passiert, stellt C einen dualen Operator zur Verfügung, der Speicherzellen wieder dem Betriebssystem zurück gibt. Dieser Operator heißt `delete`. Der `delete`-Operator wird auf Zeigerausdrücke angewendet.

Beispiel:
Das Speicherloch des letzten Beispiels läßt sich schließen, indem der Zeiger nach seiner Ausgabe immer wieder gelöscht, d.h. dem Betriebssystem zurückgegeben wird.

Deleted.cpp

```
1  #include <iostream>
2
3  int main(){
4      while (true){
5          int *pX = new int;
6          std::cout << pX << std::endl;
7          delete pX;
8      }
9  }
```

Startet man dieses Programm, so stellt man fest, daß es einen konstanten Bedarf an Speicherkapazität benötigt. Zusätzlich kann man beobachten, daß die Speicheradresse stets die gleiche ist, also wiederverwendet wird. Das Betriebssystem gibt uns gerade wieder die Adresse, die wir eben wieder freigegeben haben.

Die `new`-Operation war gefährlich, weil wir Speicherlöcher bekommen konnten. Auch die `delete`-Operation ist gefährlich. Wenn wir Speicher zu früh freigeben, obwohl er noch benötigt wird, so können wir auch Laufzeitfehler bekommen.

Beispiel:
Wir erzeugen mit `new` einen Zeiger. Diesen Zeiger kopieren wir in einen zweiten Zeiger, über den schließlich die Speicherzelle wieder freigegeben wird. Jetzt haben wir einen Zeiger `pI`, dessen Zieladresse vom Programm neu verwendet werden kann.

WrongDelete.cpp

```
1  #include <iostream>
2
3  void f(int* pI){
4    int* pX = pI;
5    delete pX;
6  }
7
8  int main(){
9    int *pI = new int;
10   *pI = 42;
11   std::cout << *pI << std::endl;
12   f(pI);
13   int *pK = new int;
14   *pK = 16;
15   std::cout << *pI <<  std::endl;
16 }
```

Und tatsächlich: führen wir dieses Programm aus, so steht plötzlich in der Zieladresse des Zeigers `pI` die Zahl `16`, obwohl wir diese hier nie hineinschreiben wollten.

```
sep@linux:~/fh/prog3/examples/src> g++ -o WrongDelete WrongDelete.cpp
sep@linux:~/fh/prog3/examples/src> ./WrongDelete
42
16
sep@linux:~/fh/prog3/examples/src>
```

Wie man sieht, kann die dynamische Speicherverwaltung bei sorglosem Umgang zu vielfältigen, schwer zu verstehenden Fehlern führen.

Zeiger auf lokale Variablen

Wir können nicht darauf vertrauen, daß alle Zeiger, die auf eine Speicherzelle zeigen, immer noch in eine benutzte Speicherzelle zeigen. Es kann passieren, daß wir einen Zeiger haben, die Speicherzelle, auf die dieser zeigt, aber schon längst nicht mehr für eine Variable benutzt wird. Dieses ist insbesondere der Fall, wenn wir einen Zeiger auf eine lokale Variable einer Funktion haben. Sobald die Funktion verlassen wird, existieren die lokalen Variablen der Funktion nicht mehr. Haben wir danach noch Zeiger auf diese Variablen, do zeigen diese in Speicherbereiche, die wieder freigegeben wurden.

Beispiel:
Betrachten wir folgendes Programm. Die Funktion `refToSquare` hat als Rückgabewert einen Zeiger auf ihre lokale Variable `j`.

LocalVarPointer.cpp

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

Der Übersetzer warnt uns davor, diesen Zeiger nach ausserhalb der Funktion zurückzugeben:

```
sep@linux:~/fh/prog3/examples/src> c++ -o LocalVarPointer LocalVarPointer.cpp
LocalVarPointer.cpp: In function ‘int* refToSquare(int)’:
LocalVarPointer.cpp:4: Warnung: address of local variable ‘j’ returned
sep@linux:~/fh/prog3/examples/src>
```

Und tatsächlich, scheint er Recht mit seiner Warnung zu haben, denn das Programm liefert einen überraschenden undefinierten Wert während der Ausführung:

```
sep@linux:~/fh/prog3/examples/src> ./LocalVarPointer
42
-1073745324
sep@linux:~/fh/prog3/examples/src>
```

2.6.3 Funktionszeiger

Alles wird irgendwo im Hauptspeicher des Rechners abgelegt. Zeiger sind bestimmte Stellen im Speicher. Da auch die Funktionen, also der Programmcode im Speicher abgelegt sind, lassen sich entsprechend Zeiger auf Funktionen definieren. Die Syntax, um aus einem Funktionsnamen eine Zeiger auf diese Funktion zu erzeugen, ist der bereits bekannte Operator `&`.

Beispiel:
Wir definieren eine einfache Funktion, die 5 auf eine Zahl addiert. Wir greifen auf diese Funktion über ihren Adresszeiger in der Anwendung zu.

Function.cpp

```
1  #include <iostream>
2
3  int add5(int x){return x+5;}
4
5  int main(){
6    int (*f) (int) = &add5;
7    int i = (*f)(4);
8
9    std::cout << i << std::endl;
10 }
```

Im obigen Beispiel haben wir Gebrauch von den beiden Operatoren `*` und `&` zum Referenzieren und Dereferenzieren der Methode `add5` gemacht. Wir brauchen dieses nicht einmal. Das Programm läßt sich ebenso ohne diese Operatoren schreiben:

Function2.cpp

```
1  #include <iostream>
2
3  int add5(int x){return x+5;}
4
5  int main(){
6    int (*f) (int) = add5;
7    int i = f(4);
8
9    std::cout << i << std::endl;
10 }
```

Auch dieses Programm übersetzt fehlerfrei und liefert dasselbe Ergebnis wie sein Vorgänger. Der C-Übersetzer verhält sich wieder pragmatisch und wandelt den Funktionsnamen `add5` automatisch in einen Zeiger auf den Funktionscode um; und ebenso wird die Zeigervariable `f` implizit dereferenziert.

Zeigerarithmetik hat ihre Grenzen. Für Funktionszeiger kann keine Arithmetik angewendet werden. Foldendes Programm, in dem man annehmen könnte, daß der Aufruf der Methode `eval` inmitten der Methode `f` springt, wird vom Übersetzer zurückgewiesen:

WrongPointerArith.cpp

```
#include <iostream>

int f(){
    return 1+1+1+1+1+1+1+1+1+1+1+1;
}

int eval(int (*f)()){
    return f();
}

int main(){
  std::cout << eval(*f + 8) << std::endl;
}
```

Der durchaus verständliche Fehlertext lautet:

```
sep@linux:~/fh/prog3/examples/src> g++ WrongPointerArith.cpp
WrongPointerArith.cpp: In function ‘int main()’:
WrongPointerArith.cpp:12: error: pointer to a function used in arithmetic
sep@linux:~/fh/prog3/examples/src>
```

Funktionszeiger sind eine mächtige Abstraktion. Eine typische Anwendung von Funktionszeigern in C ist wieder die GUI-Programmierung. In einer GUI-Bibliothek kann in der Regel als Funktion das Verhalten der einzelnen Komponenten übergeben werden.

Aufgabe 11 In dieser Aufgabe sollen Sie wie in der letzten Aufgabe eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben. Dieses Mal übergeben Sie aber die Anwendungslogik dirket als Funktionszeiger und nicht als Objekt.

Gegeben sei die folgenden C++ Klasse, wobei Sie die Klasse `FDialogue` nicht zu analysieren oder zu verstehen brauchen:

FDialogue.h

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

typedef std::string (*ButtonFunction)(std::string);

class FDialogue : public Gtk::Window{
public:
  FDialogue(ButtonFunction f,std::string description);

protected:
  ButtonFunction eval;
  virtual void on_button_clicked();
  Gtk::Entry entry;
  Gtk::Button button;
  Gtk::Label  label;
  Gtk::VBox  box;
};
```

FDialogue.cpp

```
#include "FDialogue.h"
#include <iostream>
#include <string>

FDialogue::FDialogue(ButtonFunction f,std::string description)
  : eval(f)
  , button(description)
  , label(""){
  set_border_width(10);
  button.signal_clicked()
   .connect(sigc::mem_fun(*this,&FDialogue::on_button_clicked));
  box.add(entry);
  box.add(button);
  box.add(label);
  add(box);
  show_all();
}

void FDialogue::on_button_clicked(){
  label.set_text(eval(entry.get_text()));
}
```

TestFDialogue.cpp

```
#include <gtkmm/main.h>
#include "FDialogue.h"
#include "ReadRoman.h"
#include "ToString.h"

std::string fromRoman(std::string x){return itos(romanToInt(x));}
int main (int argc, char *args[]){
  Gtk::Main kit(argc,args);
  FDialogue dialogue(fromRoman,"nach arabisch");
  Gtk::Main::run(dialogue);
}

```

Schreiben Sie mit Hilfe von `FDialogue` eine GUI-Applikation, die es ermöglicht Dezimalzahlen in Oktalzahlen umzuwandeln.

2.6.4 Referenzen

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.

Daher hat man sich in C++ entschlossen einen weiteren Typ einzuführen, der die Vorteile von Zeigern hat, aber weniger gefährlich in seiner Anwendung ist, 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[7]. 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:

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>
```

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

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>
```

2.7 Einfach verkettete Listen von String

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Da Sammlungstypen oft gebraucht werden, gibt es Bibliotheken, die Sammlungstypen zur Verfügung stellen. Bevor wir uns aber diesen bereits vorhandenen Klassen zuwenden, wollen wir in diesem Kapitel Listen selbst spezifizieren und programmieren.

2.7.1 Formale Spezifikation

Wir werden Listen als abstrakten Datentyp formal spezifizieren. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Methoden. Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktormethoden spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testmethoden spezifiziert, die angeben, mit welchem Konstruktor ein Datum erzeugt wurde.

Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testmethoden wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie bei jeder rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[8]

Konstruktoren

Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[9] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

[8] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach $n + 1$ dem Hinzufügen eines neuen Elements zu einer Liste.

[9] Entgegen der Notation in C++, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

- Empty: () →**List**
- Cons: (**string**,**List**) →**List**

Selektoren

Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Cons** hat zwei Parameter. Für **Cons**-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: **(List)→string**
- tail: **(List)→List**

Der funktionale Zusammenhang von Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$$\begin{aligned} head(Cons(x, xs)) &= x \\ tail(Cons(x, xs)) &= xs \end{aligned}$$

Testmethoden

Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine **Cons**-Liste. Hierzu bedarf es noch einer Testmethode, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testmethode *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: **List→boolean**

Das funktionale Verhalten der Testmethode läßt sich durch folgende zwei Gleichungen spezifizieren:

$$\begin{aligned} isEmpty(Empty()) &= true \\ isEmpty(Cons(x, xs)) &= false \end{aligned}$$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert und Listen können nach der Art ihrer Konstruktion unterschieden werden.

Listenalgorithmen

Allein diese fünf Funktionen beschreiben den ADT der Listen. Wir können aufgrund dieser Spezifikation Algorithmen für Listen schreiben.

Länge Und ebenso läßt sich durch zwei Gleichungen spezifizieren, was die Länge einer Liste ist:

$$\begin{aligned} \mathit{length}(\mathit{Empty}()) &= 0 \\ \mathit{length}(\mathit{Cons}(x, xs)) &= 1 + \mathit{length}(xs) \end{aligned}$$

Mit Hilfe dieser Gleichungen läßt sich jetzt schrittweise die Berechnung einer Listenlänge auf Listen durchführen. Hierzu benutzen wir die Gleichungen als Ersetzungsregeln. Wenn ein Unterausdruck in der Form der linken Seite einer Gleichung gefunden wird, so kann diese durch die entsprechende rechte Seite ersetzt werden. Man spricht bei so einem Ersetzungsschritt von einem Reduktionsschritt.

Beispiel:
Wir errechnen in diesem Beispiel die Länge einer Liste, indem wir die obigen Gleichungen zum Reduzieren auf die Liste anwenden:

$$\begin{aligned} & \mathit{length}(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & 1 + \mathit{length}(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + \mathit{length}(\mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + \mathit{length}(\mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + 0)) \\ \rightarrow\ & 1 + (1 + 1) \\ \rightarrow\ & 1 + 2 \\ \rightarrow\ & 3 \end{aligned}$$

Letztes Listenelement Wir können mit einfachen Gleichungen spezifizieren, was wir unter dem letzten Element einer Liste verstehen.

$$\begin{aligned} \mathit{last}(\mathit{Cons}(x, \mathit{Empty}())) &= x \\ \mathit{last}(\mathit{Cons}(x, xs))) &= \mathit{last}(xs) \end{aligned}$$

Auch die Funktion `last` können wir von Hand auf einer Beispielliste einmal per Reduktion ausprobieren:

$$\begin{aligned} & \mathit{last}(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & \mathit{last}(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & \mathit{last}(\mathbf{Cons}(c, \mathbf{Empty}())) \\ \rightarrow\ & c \end{aligned}$$

Listenkonkatenation Die folgenden Gleichungen spezifizieren, wie zwei Listen aneinandergehängt werden:

$$\begin{aligned} \mathit{concat}(\mathit{Empty}(), ys) &= ys \\ \mathit{concat}(\mathit{Cons}(x, xs), ys) &= \mathit{Cons}(x, \mathit{concat}(xs, ys)) \end{aligned}$$

Auch diese Funktion läßt sich beispielhaft mit der Reduktion einmal durchrechnen:

$$
\begin{aligned}
& \mathit{concat}(\mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Empty}())), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathit{concat}(\mathbf{Cons}(j, \mathbf{Empty}()), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathit{concat}(\mathbf{Empty}(), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))
\end{aligned}
$$

Schachtel- und Zeiger-Darstellung

Listen lassen sich auch sehr schön graphisch visualisieren. Hierzu wird jede Liste durch eine Schachtel mit zwei Feldern dargestellt. Von diesen beiden Feldern gehen Pfeile aus. Der erste Pfeil zeigt auf das erste Element der Liste, dem `head`, der zweite Pfeil zeigt auf die Schachtel, die für den Restliste steht dem *tail*. Wenn eine Liste leer ist, so gehen keine Pfeile von der Schachtel aus, die sie repräsentiert.

Die Liste *Cons(a,Cons(b,Cons(c,Empty())))* hat somit die Schachtel- und Zeiger- Darstellung aus Abbildung 2.5.

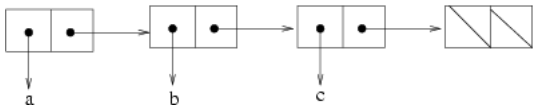

Abbildung 2.5: Schachtel Zeiger Darstellung einer dreielementigen Liste.

In der Schachtel- und Zeiger-Darstellung läßt sich sehr gut verfolgen, wie bestimmte Algorithmen auf Listen dynamisch arbeiten. Wir können die schrittweise Reduktion der Methode `concat` in der Schachtel- und Zeiger-Darstellung gut nachvollziehen:

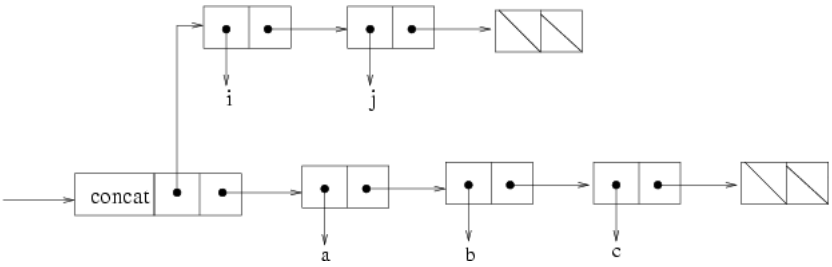

Abbildung 2.6: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

Abbildung 2.6 zeigt die Ausgangssituation. Zwei Listen sind dargestellt. Von einer Schachtel, die wir als die Schachtel der Funktionsanwendung von `concat` markiert haben, gehen zwei Zeiger aus. Der erste auf das erste Argument, der zweite auf das zweite Argument der Funktionsanwendung.

Abbildung 2.7 zeigt die Situation, nachdem die Funktion `concat` einmal reduziert wurde. Ein neuer Listenknoten wurde erzeugt. Dieser zeigt auf das erste Element der ursprünglich ersten Argumentliste. Der zweite zeigt auf den rekursiven Aufruf der Funktion `concat`, diesmal mit der Schwanzliste des ursprünglich ersten Arguments.

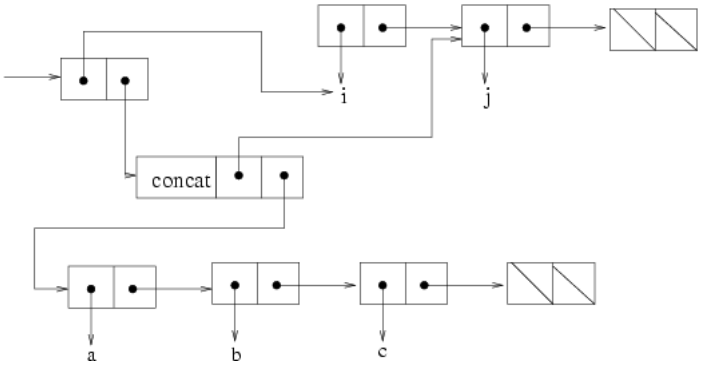

Abbildung 2.7: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

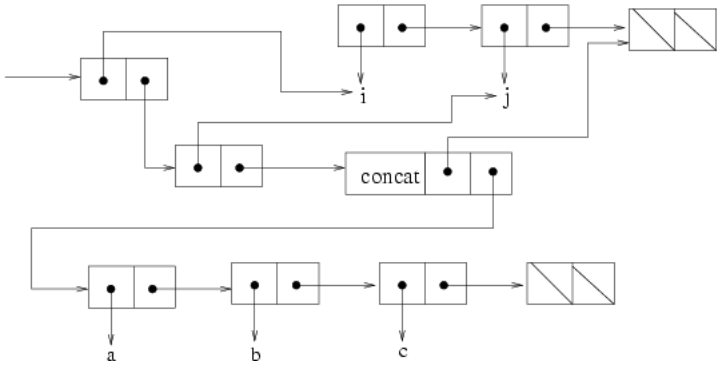

Abbildung 2.8: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

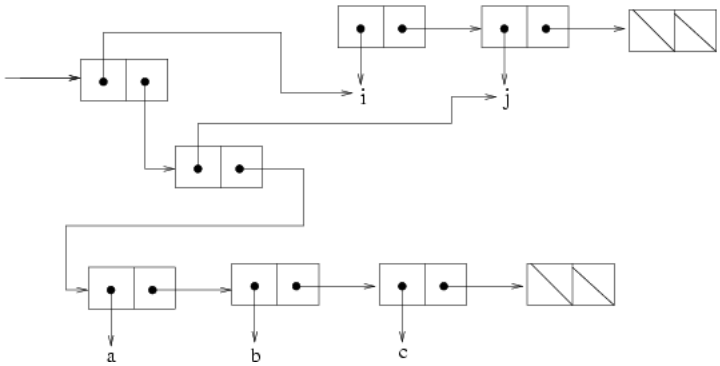

Abbildung 2.9: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

Abbildung 2.8 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 2.9 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, daß die beiden ursprünglichen Listen noch vollständig erhalten sind. Sie wurden nicht gelöscht. Die erste Ar-

gumentliste wurde quasi kopiert. Die zweite Argumentliste teilen sich gewisser Maßen die neue Ergebnisliste der Funktionsanwendung und die zweite ursprüngliche Argumentliste.

2.7.2 Modellierung

C++ kennt keine direkte Unterstützung für ADTs, die nach obigen Prinzip spezifiziert werden.[10]

Nachdem wir im letzten Abschnitt formal spezifiziert haben, wie Listen konstruiert werden, wollen wir in diesem Abschnitt betrachten, wie diese Spezifikation geeignet mit unseren programmiersprachlichen Mitteln modelliert werden kann, d.h. wie viele und was für Klassen werden benötigt. Wir werden in den folgenden zwei Abschnitten zwei alternative Modellierungen der Listenstruktur angeben.

Modellierung als Klassenhierarchie

Laut Spezifikation gibt es zwei Arten von Listen: leere Listen und Listen mit einem Kopfelement und einer Schwanzliste. Es ist naheliegend, für diese zwei Arten von Listen je eine eigene Klasse bereitzustellen, jeweils eine Klasse für leere Listen und eine für **Cons**-Listen. Da beide Klassen zusammen einen Datentyp Liste bilden sollen, sehen wir eine gemeinsame Oberklasse dieser zwei Klassen vor. Wir erhalten die Klassenhierarchie in Abbildung: 2.10.

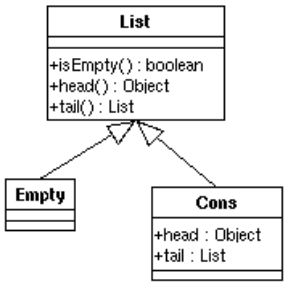

Abbildung 2.10: Modellierung von Listen mit drei Klassen.

Wir haben uns in diesem Klassendiagramm dazu entschieden, daß die Selektormethoden für alle Listen auch für leere Listen zur Verfügung stehen. Es ist in der formalen Spezifikation nicht angegeben worden, was in dem Fall der Methoden *head* und *tail* auf leere Listen als Ergebnis erwartet wird. Wir können hier Fehler geben oder aber bestimmte ausgezeichnete Werte als Ergebnis zurückgeben.

In einer Klasse

Eine alternative Modellierung der Listenstruktur besteht aus nur genau einer Klasse. Diese Klasse braucht hierzu aber zwei Konstruktoren. Wie in vielen objektorientierten Sprachen ist es in C++ auch möglich, für eine Klasse mehrere Konstruktoren mit unterschiedlichen Parametertypen zu schreiben. Wir können also eine Klasse `List` modellieren, die zwei Konstruktoren hat: einen mit keinem Parameter und einen mit zwei Parametern.

[10] Gleiches gilt für Java. Es gibt jedoch eine Javaerweiterung namens *Pizza*[OW97], oder die relativ neue Sprache *Scala*[Oa04] die eine solche Unterstützung eingebaut hat.

2.7.3 Codierung

Die obigen beiden Modellierungen der Listen lassen sich jetzt direkt in C++-Code umsetzen.

Implementierung als Klassenhierarchie

Die Oberklasse List Die zusammenfassende Oberklasse der Listenstruktur `List` stellt für die Listenmethoden `head`, `tail` und `isEmpty` jeweils eine prototypische Implementierung zur Verfügung. Die Methode `isEmpty` setzen wir in unserer Umsetzung standardmäßig auf `false`.

List.h

```
1 #include <string>
2 #ifndef __List_H
3 #define __List_H ;
4
5 class List {
6  public:
7   virtual bool isEmpty()=0;
8   virtual std::string head();
9   virtual List* tail();
```

List.cpp

```
1 #include "List.h"
2 std::string List::head(){return NULL;}
3 List* List::tail(){return NULL;}
```

Die Methoden `tail` und `head` können nur für nichtleere Listen Objekte zurückgeben. Daher haben wir uns für die prototypischen Implementierung in der Klasse `List` dazu entschieden, den Wert `NULL` zurückzugeben. `NULL` steht in C++ für das Fehlen eines Objektes. Der Versuch, auf Felder und Methoden von `NULL` zuzugreifen, führt in C++ zu einem Fehler.

Für die Klasse `List` schreiben wir keinen Konstruktor. Wir wollen diese Klasse nie direkt instanziieren, d.h. nie einen Konstruktor für die Klasse `List` aufrufen.[11]

Die Klasse Empty Die Klasse, die die leere Liste darstellt, ist relativ leicht abzuleiten. Wir stellen einen Konstruktor zur Verfügung und überschreiben die Methode `isEmpty`.

Empty.h

```
1 #include "List.h"
2
3 class Empty :public List {
4  public:
5   Empty();
6   virtual bool isEmpty();
```

Empty.cpp

```
1 #include "Empty.h"
2 Empty::Empty(){}
3 bool Empty::isEmpty(){return true;}
```

[11] In diesem Fall generiert C++ automatisch einen Konstruktor ohne Argumente.

Die Klasse Cons Schließlich ist noch die Klasse `Cons` zu codieren. Diese Klasse benötigt nach unserer Modellierung zwei Felder, in denen Kopf und Schwanz der Liste abgespeichert werden können. Die Methoden `head` und `tail` werden so überschrieben, daß sie entsprechend den Wert eines dieser Felder zurückgeben. Der Konstruktor initialisiert diese beiden Felder.

Cons.h

```
#include "List.h"
class Cons:public List{
 private:
  std::string hd;
  List* tl ;
 public:
  Cons(std::string x,List* xs);
  virtual std::string head();
  virtual List* tail();
  virtual bool isEmpty();
```

Cons.cpp

```
#include "Cons.h"
Cons::Cons(std::string x,List* xs):hd(x),tl(xs){}
std::string Cons::head(){return hd;}
List* Cons::tail(){return tl;}
bool Cons::isEmpty(){return false;}
```

Damit ist die Listenstruktur gemäß unserer formalen Spezifikation vollständig implementiert.

Einfache Tests Für Algorithmen auf Listen werden wir nur die zwei Konstruktoren, die zwei Selektoren und die Testmethode `isEmpty` benutzen. Folgendes kleine Testprogramm konstruiert eine Liste mit drei Elementen des Typs `String`. Anschließend folgen einige Tests für die Selektormethoden:

TestFirstList.cpp

```
#include "Cons.h"
#include "Empty.h"

#include <iostream>

int main(){
  List* xs = new Cons("friends",
             new Cons("romans",
             new Cons("countrymen",
             new Empty())));

  std::cout<< "1. Element: "+xs->head()<<std::endl;
  std::cout<< "2. Element: "+xs->tail()->head()<<std::endl;
  std::cout<< "3. Element: "+xs->tail()->tail()->head()<<std::endl;

  if (xs->tail()->tail()->tail()->isEmpty()){
    std::cout<< "leere Liste nach drittem Element."<<std::endl;
```

```
18    }
19
20    std::cout<< xs->tail()->tail()->tail()->tail()->head()<<std::endl;
21  }
```

Tatsächlich bekommen wir für unsere Tests die erwartete Ausgabe:

```
sep@linux:~/fh/prog1/examples/classes> ./TestFirstList
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

Implementierung als eine Klasse

In der zweiten Modellierung haben wir auf eine Klassenhierarchie verzichtet. Was wir in der ersten Modellierung durch die verschiedenen Klassen mit verschieden überschriebenen Methoden ausgedrückt haben, muß in der Modellierung in einer Klasse über ein bool'sches Feld vom Typ `bool` ausgedrückt werden. Die beiden unterschiedlichen Konstruktoren setzen ein bool'sches Feld `_isEmpty`, um für das neu konstruierte Objekt zu markieren, mit welchem Konstruktor es konstruiert wurde. Die Konstruktoren setzen entsprechend die Felder, so daß die Selektoren diese nur auszugeben brauchen.

L1.h

```
1  #include <string>
2  #include "ToString.h"
3  #ifndef __L1_H
4  #define __L1_H ;
5
6  class L1 :public ToString {
7   private:
8    bool _isEmpty;
9    std::string hd;
10   L1* tl;
11
12  public:
13   L1();
14   L1(std::string hd,L1* tl);
15   virtual ~L1();
16   std::string head();
17   L1* tail();
18   bool isEmpty();
19   std::string toString();
20
21  private:
22   std::string toStringAux();
```

L1.cpp

```
#include <string>
#include "L1.h"

L1::L1():_isEmpty(true){}
L1::L1(std::string hd,L1* tl):hd(hd),tl(tl),_isEmpty(false){}
L1::~L1(){if (!isEmpty()) delete tl;}
std::string L1::head(){return hd;}
L1* L1::tail(){return tl;}
bool L1::isEmpty(){return _isEmpty;}

std::string L1::toString(){
  if (isEmpty()) return "";
  return "["+head()+tail()->toStringAux();
};

std::string L1::toStringAux(){
  if (isEmpty()) return "]";
  return ","+head()+tail()->toStringAux();
}

```

Zum Testen dieser Listenimplementierung sind nur die Konstruktoraufrufe bei der Listenkonstruktion zu ändern. Da wir nur noch eine Klasse haben, gibt es keine zwei Konstruktoren mit unterschiedlichen Namen, sondern beide haben denselben Namen:

TestFirstLi.cpp

```
#include "L1.h"
#include <iostream>

int main(){
  L1* xs = new L1("friends",
           new L1("romans",
           new L1("countrymen",
           new L1())));


  std::cout<< "1. Element: "+xs->head()<<std::endl;
  std::cout<< "2. Element: "+xs->tail()->head()<<std::endl;
  std::cout<< "3. Element: "+xs->tail()->tail()->head()<<std::endl;
  std::cout<< "toString: "+xs->toString() <<std::endl;

  if (xs->tail()->tail()->tail()->isEmpty()){
    std::cout<< "leere Liste nach drittem Element."<<std::endl;
  }

  std::cout<< xs->tail()->tail()->tail()->tail()->toString();
}
```

Wie man aber sieht, ändert sich an der Benutzung von Listen nichts im Vergleich zu der Modellierung mittels einer Klassenhierarchie. Die Ausgabe ist ein und dieselbe für beide Implementierungen:

```
sep@linux:~/fh/prog1/examples/classes> java TestFirstLi
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

2.7.4 Methoden für Listen

Mit der formalen Spezifikation und schließlich Implementierung von Listen haben wir eine Abstraktionsebene eingeführt. Listen sind für uns Objekte, für die wir genau die zwei Konstruktormethoden, zwei Selektormethoden und eine Testmethode zur Verfügung haben. Unter Benutzung dieser fünf Eigenschaften der Listenklasse können wir jetzt beliebige Algorithmen auf Listen definieren und umsetzen.

Länge

Eine interessante Frage bezüglich Listen ist die nach ihrer Länge. Wir können die Länge einer Liste berechnen, indem wir durchzählen, aus wievielen **Cons**-Listen eine Liste besteht. Wir haben bereits die Funktion *length* durch folgende zwei Gleichungen spezifiziert.

$$\begin{aligned} length(Empty()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

Diese beiden Gleichungen lassen sich direkt in C++-Code umsetzen. Die zwei Gleichungen ergeben genau zwei durch eine `if`-Bedingung zu unterscheidende Fälle in der Implementierung. Wir können die Klassen `List` und `L1` um folgende Methode `length` ergänzen:

L1.h

```
22 public: int length();
```

L1.cpp

```
23 int L1::length(){
24   if (isEmpty()) return 0;
25   return 1+ tail()->length();
26 }
```

In den beiden Testklassen läßt sich ausprobieren, ob die Methode `length` entsprechend der Spezifikation funktioniert. Wir können in der `main`-Methode einen Befehl einfügen, der die Methode `length` aufruft:

TestLength.cpp

```
1 #include "L1.h"
2 #include <iostream>
3
```

```
4  int main(){
5    L1* xs =new L1("friends",new L1("romans"
6                  ,new L1("countrymen",new L1())));
7
8    std::cout << xs->length()<<std::endl;
9  }
10
```

Und wie wir bereits schon durch Reduktion dieser Methode von Hand ausgerechnet haben, errechnet die Methode `length` tatsächlich die Elementanzahl der Liste:

```
sep@linux:~/fh/prog1/examples/classes> ./TestLength
3
sep@linux:~/fh/prog1/examples/classes>
```

Alternative Implementierung im Fall der Klassenhierarchie Im Fall der Klassenhierarchie können wir auch die `if`-Bedingung der Methode `length` dadurch ausdrücken, daß die beiden Fälle sich in den unterschiedlichen Klassen für die beiden unterschiedlichen Listenarten befinden. In der Klasse `List` kann folgende prototypische Implementierung eingefügt werden:

List.h

```
1    virtual int length()=0;
2  };
3  #endif
```

Empty.h

```
1    virtual int length();
2  };
```

Cons.h

```
1    virtual int length();
2  };
```

Für die Klasse `Empty` ist die Länge der Liste 0:

Empty.cpp

```
1  int Empty::length(){return 0;}
```

In der Klasse `Cons`, die ja Listen mit einer Länge größer 0 darstellt, ist dann die Methode entsprechend der Definition zu überschreiben:

Cons.cpp

```
1  int Cons::length(){return 1+tail()->length();}
```

Auch diese Längenimplementierung können wir testen:

TestListLength.java

```
1  class TestListLength {
2    static  List XS = new Cons("friends",new Cons("romans",
3                      new Cons("countrymen",new Empty())));
4
5    public static void main(String [] args){
6      System.out.println(XS.length());
7    }
8  }
```

An diesem Beispiel ist gut zu sehen, wie durch die Aufsplittung in verschiedene Unterklassen beim Schreiben von Methoden `if`-Abfragen verhindert werden können. Die verschiedenen Fälle einer `if`-Abfrage finden sich dann in den unterschiedlichen Klassen realisiert. Der Algorithmus ist in diesem Fall auf verschiedene Klassen aufgeteilt.

Iterative Lösung Die beiden obigen Implementierungen der Methode `length` sind rekursiv. Im Rumpf der Methode wurde sie selbst gerade wieder aufgerufen. Natürlich kann die Methode mit den entsprechenden zusammengesetzten Schleifenbefehlen in C++ auch iterativ gelöst werden.

Wir wollen zunächst versuchen, die Methode mit Hilfe einer `while`-Schleife zu realisieren. Der Gedanke ist naheliegend. Solange es sich noch nicht um die leere Liste handelt, wird ein Zähler, der die Elemente zählt, hochgezählt:

L1.h

```
9    int lengthWhile();
```

L1.cpp

```
10 int L1::lengthWhile(){
11   int erg=0;
12   L1* xs = this;
13   while (!xs->isEmpty()){
14     erg = erg +1;
15     xs = xs->tail();
16   }
17   return erg;
18 }
```

Schaut man sich diese Lösung genauer an, so sieht man, daß sie die klassischen drei Bestandteile einer `for`-Schleife enthält:

- die Initialisierung einer Laufvariablen: `L1* xs = this;`
- eine bool'sche Bedingung zur Schleifensteuerung: `!xs->isEmpty()`
- eine Weiterschaltung der Schleifenvariablen: `xs = xs->tail();`

Damit ist die Schleife, die über die Elemente einer Liste iteriert, ein guter Kandidat für eine `for`-Schleife. Wir können das Programm entsprechend umschreiben:

L1.h

```
19   int lengthFor();
```

L1.cpp

```
20 int L1::lengthFor(){
21   int erg=0;
22   for (L1* xs=this;!xs->isEmpty();xs=xs->tail()){
23     erg = erg +1;
24   }
25   return erg;
26 }
```

Eine Schleifenvariable ist also nicht unbedingt eine Zahl, die hoch oder herunter gezählt wird, sondern kann auch ein Objekt sein, von dessen Eigenschaften abhängt, ob die Schleife ein weiteres Mal zu durchlaufen ist.

In solchen Fällen ist die Variante mit der `for`-Schleife der Variante mit der `while`-Schleife vorzuziehen, weil somit die Befehle, die die Schleife steuern, gebündelt zu Beginn der Schleife stehen.

Akkumulative Rekursion Die rekursive Lösung des Längenfunktion hat einen kleinen Nachteil: das Programm berechnet erst die Additionen, wenn das Ende der Liste erreicht wurde. In der Reduktion der Längenfunktion wird erst die Liste bis zum Ende aufgerollt. Das programm muß sich so lange die einzelnen `1+` auf dem Stack zwischenspeichern. Der Stack wächst um die noch zu berechenenden Additionen:

$$
\begin{aligned}
& 1 + (1 + (1 + \mathit{length}(\mathbf{Empty}()))) \\
\rightarrow\ & 1 + (1 + (1 + 0)) \\
\rightarrow\ & 1 + (1 + 1) \\
\rightarrow\ & 1 + 2 \\
\rightarrow\ & 3
\end{aligned}
$$

Mit einem Trick kann man dieses Problem umgehen. Hierzu schreiben wir die Längenfunktion mit einem Akkumulator. Ein Akkumulator ist ein Parameter, an dem schrittweise das Gesamtergebnis akkumuliert wird. Die Funktion wird mit einem Akkumulator wird inital mit dem Ergebniswert für die leere Liste aufgerufen.

L1.h

```
27   int lengthAkk(int result);
28   int lengthAkk();
29 };
30
31 #endif
```

Die Implementierung ist ähnlich einfach, wie bei den übrigen Versionen der Längenfunktion. Wenn die Liste leer ist, so steht im Akkumulator das Ergebnis, andernfalls wird die Funktion mit um eins erhöten Akkumulator auf den Schwanz der Liste ausgeführt.

L1.cpp

```
32 int L1::lengthAkk(int result){
33   if (isEmpty()) return result;
34   return tail()->lengthAkk(result+1);
35 }
36 int L1::lengthAkk(){return lengthAkk(0);}
```

Betrachten wir die Reduktion der akkumulativen Längenfunktion in der funktionalen Schreibweise:

$$
\begin{aligned}
& \mathit{lengthAkk}(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))), 0) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())), 0 + 1) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())), 1) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Cons}(c, \mathbf{Empty}()), 1 + 1) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Cons}(c, \mathbf{Empty}()), 2) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Empty}(), 2 + 1) \\
\rightarrow\ & \mathit{lengthAkk}(\mathbf{Empty}(), 3) \\
\rightarrow\ & 3
\end{aligned}
$$

Wie man sieht wird die Addition jeweils sofort duchgeführt. Es gibt außerhalb des Funktionsaufrufs für `lengthAkk` keine Informationen mehr, die zur Berechnung des Ergebnisses nötig sind. Kluge Compiler können diese Tatsache nutzen, um keinen neuen Speicherbereich auf dem Stack für den rekursiven Aufruf zu reservieren.

Aufgabe 12 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

a) `std::string last()`: gibt das letzte Element der Liste aus.

b) `L1 concat(L1 other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `std::string elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$
\begin{aligned}
\mathit{elementAt}(\mathit{Cons}(x, xs), 1) &= x \\
\mathit{elementAt}(\mathit{Cons}(x, xs), n + 1) &= \mathit{elementAt}(xs, n)
\end{aligned}
$$

2.8 Objektorientierung in C

Für einen kurzen Moment wollen wir einmal unsere neu erworbenen Kenntnisse der Klassen in C++ vergessen und wieder zurück zum einfachen C gehen. Können wir Ojekte in C simulieren?

Die ursprünglichen C Strukturen sind bereits sehr nah an der objektorientierten Programmierung. Drei entscheidene Eigenschaften der Objektorientierung fehlen:

- Objektmethoden
- Vererbung
- und late-binding

In diesem Abschnitt wollen wir einmal versuchen, ob die Sprache C nicht schon mächtig genug ist, um diese drei Konzepte zu implementieren; und wir werden sehen, daß mit einigen Klimmzügen dieses tatsächlich möglich ist.

2.8.1 Objektmethoden für Strukturen

Zunächst einmal gibt es einen ganz einfachen Trick, wie in Strukturen Methoden integriert werden können. Hierzu kann eine Methode außerhalb der Struktur definiert werden, um dann in der Struktur selbst einen Funktionszeiger auf diese Funktion gespeichert werden. Damit wir in den Objektmethoden auf die Felder des Objektes zugreifen können, muß die Funktion einen Zeiger auf dieses Objekt zur Verfügung haben, den sogenannten `this`-Zeiger.

Versuchen wir mit diesen Mitteln eine Pseudoklasse für Personen in C zu definieren. Hierzu definieren wir eine Struktur mit den entsprechenden Feldern, und einen Feld für den Funktionszeiger auf die Objektmethode `getFullName`.

CPerson.h

```
1  // pseudo class for persons
2  struct CPerson
3    {char* name
4    ;char* vorname
5    ;char* strasse
6    ;char* stadt
7    ;int plz
8    ;char* (*getFullNameFunction)(struct CPerson*)
9    ;};
10
11 //pseudo instance method for some this object
12 char* getFullName(struct CPerson * this);
13
14 //constructor for CPerson
15 struct CPerson* newPerson
16   (char* n, char* v,char* str,char* st,int p);
```

2.8.2 Unterklassen für Strukturen

Als nächstes versuchen wir eine Unterklasse der Pseudoklasse `CPerson` zu definieren. Diese soll alle Eigenschaften von `CPerson` haben, und zusätzlich noch eine Matrikelnummer. Jetzt können wir uns die Tatsache zu Nutze machen, daß die einzelnen Attribute einer Struktur direkt hintereinander im Speicher abgelegt werden. Es reicht somit aus, ein Attribut für eine Instanz der Pseudosuperklasse vorzusehen und danach die zusätzlichen Attribute zu definieren.

CPerson.h

```
17 // pseudo subclass of CPerson for students
18 struct CStudent
```

```
19   {struct CPerson super
20   ;int matrNr;
21   };
22
23 //constructor for CStudent
24 struct CStudent* newStudent
25   (char* n, char* v,char* str,char* st,int p,int ma);
```

Soweit die Schnittstellenbeschreibung der beiden gewünschten Pseudoklassen. Bevor wir diese als objektorientierte Bibliothek implementieren, können wir zunächst ein einfaches Testbeispiel für diese Kopfdatei schreiben. Wir wollen in der Lage sein, Personen und Studenten zu erzeugen. Studenten sollen in Variablen für Felder abgelegt werden können. Der Aufruf der Funktion `getFullName` soll über *late-binding* jeweils die entsprechende Objektmethode für `CPerson` bzw. `CStudent` ausführen.

TestCPerson.c

```
1  #include "CPerson.h"
2
3  int main(){
4    //create a person object
5    struct CPerson* p1
6     = newPerson("Schmidt","Harald","studio449","koeln",54566);
7
8    //create a student object, but store it in a Person variable
9    struct CPerson* p2
10    = newStudent("Andrak","Manuel","studio449","koeln",54566
11                 ,565456);
12
13   //print the full names of the two person objects
14   printf("die erste Person ist: %s\n",getFullName(p1));
15   printf("der zweite Person ist: %s\n",getFullName(p2));
16
17   //make a cast of the second person to student
18   printf("mit Matrikel-Nr.: %i\n",((struct CStudent*)p2)->matrNr);
19 }
```

Soweit schon einmal der Testfall für unsere kleine Bibliothek. Es gilt jetzt die beschriebenen Schnittstellen so zu implementieren, daß wir verschiedene Methoden `getFullName` für Personen und Studenten erhalten und über späte Bindung auf die für das ursprünglich erzeugte Objekt geschriebene Methode zugegriffen wird. Zunächst einmal schreiben wir die zwei Methoden, die für Personen bzw. Studenten den vollen Namen errechnen. Da wir uns auf einfaches C beschränken wollen, haben wir keine Klasse zur Darstellung von Zeichenketten sondern nur Reihungen von Zeichen und müssen Funktionen für diese aus den Standardbibliotheken benutzen:

CPerson.c

```
1  #include "CPerson.h"
2  #include <string.h>
3  #include <stdlib.h>
4
5  char* getFullPersonName(struct CPerson* this){
```

```
6   size_t nameLength = strlen (this->name);
7   size_t vornameLength = strlen (this->vorname);
8   char* result
9    = malloc((nameLength + vornameLength+3) * sizeof(char));
10  char* end = result;
11  end=(char*)stpcpy(end,this->name);
12  end=(char*)stpcpy(end,", ");
13  end=(char*)stpcpy(end,this->vorname);
14  return result;
15 }
16
17 char* getFullStudentName(struct CPerson* this){
18  size_t nameLength = strlen (this->name);
19  size_t vornameLength = strlen (this->vorname);
20  char* result
21   = malloc((nameLength + vornameLength+3) * sizeof(char));
22  char* end = result;
23  end=(char*)stpcpy(end,this->vorname);
24  end=(char*)stpcpy(end," ");
25  end=(char*)stpcpy(end,this->name);
26  return result;
27 }
```

Es fehlen uns schließlich nur noch die Implementierungen der Konstruktormethoden. Diese splitten wir in zwei Teile, einen erzeugenen Teil und einen Initialisierungsteil. Hier zunächst der Initialisierungsteil für Personen. Einem Personenobjekt werden die einzelnen Werte für seine Felder zugewiesen:

CPerson.c

```
28 void initPerson(struct CPerson* result
29                ,char* n, char* v,char* str,char* st,int p){
30   result->name = n;
31   result->vorname = v;
32   result->strasse = str;
33   result->stadt = st;
34   result->plz = p;
35   result->getFullNameFunction = &getFullPersonName;
36 }
```

Insbesondere sei bemerkt, daß die Funktion zur Berrechnung eines vollen Namens mit der entsprechenden Funktion für Personen initialisiert wird.

Der eigentliche Konstruktor für Personen, fordert über die C Funktion `malloc` genügend Speicherplatz für ein Personenobjekt an und führt die Initialisierung in diesem neuen Speicherbereich durch:

CPerson.c

```
37 struct CPerson* newPerson
38    (char* n, char* v,char* str,char* st,int p){
39   struct CPerson* result
```

```
40      = (struct CPerson*) malloc(sizeof(struct CPerson));
41    initPerson(result, n,  v, str, st, p);
42    return result;
43  }
```

Der Konstruktor für Studenten fordert zunächst den notwendigen Speicher für einen Studenten an, initialisiert dann mit den Initialisator der Superklasse den Personenteil für Studenten. Dieses entspricht durchaus den Aufruf des Superkonstruktors, wie wir ihn aus C++ kennen. Schließlich wird der Funktionszeiger überschrieben mit der für Studenten spezifischen Funktion, und zu guter letzt wird das neue Feld der Matrikelnummer initialisiert.

CPerson.c

```
44  struct CStudent* newStudent
45     (char* n, char* v,char* str,char* st,int p,int ma){
46    struct CStudent* result
47      = (struct CStudent*) malloc(sizeof(struct CStudent));
48    initPerson(&(result->super), n,  v, str, st, p);
49
50    (result->super).getFullNameFunction = &getFullStudentName;
51
52    result->matrNr = ma;
53
54    return result;
55  }
```

Wollte man noch eine Unterklasse von `CStudent` schreiben, so wäre es vorteilhaft gewesen, auch für Studenten schon zwischen Konstruktorfunktion und Initialisator zu unterscheiden.

Schließlich können wir noch eine Funktion vorsehen, die für eine Person oder eine beliebige Unterklasse davon, die Objektfunktion `getFullName` aufruft.

CPerson.c

```
56  char* getFullName(struct CPerson * this){
57    return (*(this->getFullNameFunction))(this);
58  };
```

Und tatsächlich, unser Testprogramm zeigt, daß wir es geschafft haben, späte Bindung in C zu implementieren:

```
sep@linux:~/fh/prog3/examples/src> gcc -o TestCPerson TestCPerson.c CPerson.c
TestCPerson.c: In function ‘main’:
TestCPerson.c:11: Warnung: initialization from incompatible pointer type
sep@linux:~/fh/prog3/examples/src> ./TestCPerson
die erste Person ist: Schmidt, Harald
der zweite Person ist: Manuel Andrak
mit Matrikel-Nr.: 565456
sep@linux:~/fh/prog3/examples/src>
```

Wir bekommen lediglich eine Warnung bei der Zuweisung eines Studenten auf eine Variable für Personenreferenzen. Von dieser Warnung lassen wir uns aber in diesem Fall nicht stören.

Kapitel 3

Generische Programmierung

3.1 Generische Funktionen

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.

Java kannte bis zur Version 1.5 keine Möglichkeit der generischen Programmierung. Mitlerweile ist der Spezifikationsprozess der Javagemeinde abgeschlossen, so daß ab Version 1.5 auch in Java generisch programmiert werden kann[BCK+01]. In C++ kann generisch über sogenannte Formulare englisch *templates* programmiert werden.[1] In Vergleich zu Javas generischen Funktionen sind die C++-Formulare etwas primitiver technisch umgesetzt.

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 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.h

```
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.

[1] Auf deutsch findet sich auch die Bezeichnung: Schablone.

Trace.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  }
```

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. In Javas generischen Typen wäre dieses auch der Fall, in C++ kann man dieses nicht sagen. 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,
   _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. In Javas generischen Typen ist dieses möglich.

Aufgabe 13 Schreiben Sie eine Funktion zur Funktionakomposition 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.

3.1.1 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.h

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

Und die entsprechende Implementierungsdatei:

T.cpp

```
1 #include <iostream>
2 #include "T.h"
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.h"
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 1.5 realisiert ist. Hier findet sich in der Klassendatei nur ein Code für eine generische Funktion. Als Konsequenz werden intern in Java die generischen Typen als vom Typ `Object` betrachtet, und nach jeder Anwendung einer generischen Methode wird eine Typzusicherung nötig.

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.

3.1.2 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:
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>
```

3.1.3 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 14 (2 Punkte) 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){return x>y.length()?x:y.length();}
9
10 int main(){
11   int xs [] = {1,2,3,4,5,4,3,2,1};
12   std::cout << fold(xs,9,add,0) << std::endl;
13   std::cout << fold(xs,9,mult,1)<< std::endl;
14   std::cout << fold(xs,9,gr,0)  << std::endl;
15   std::string ys [] = {"john","paul","george","ringo","stu"};
16   std::cout << fold(ys,5,strLaenger,0)  << std::endl;
17 }
```

Schreiben Sie ein paar zusätzliche Beispielaufrufe von `fold`.

Aufgabe 15 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester vorgestellt wahrscheinlich 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.

Lösung

Zunächst die Sortierung von ints nach ihrer Größe:

Bubble.cpp

```
1  #include <string>
2  #include <iostream>
3
4  void bubbleSort(int xs [],int l ){
5    for (int i=l-1;i>=0;i--){
6      for (int j=0;j<i;j++){
7        if (xs[j]>xs[j+1]){
8          int k=xs[j];
9          xs[j]=xs[j+1];
10         xs[j+1]=k;
11       }
12     }
13   }
14 }
```

Nun so verallgemeinert, daß die Sortierrelation als Funktion übergeben wird:

Bubble.cpp

```
15 void bubbleSortBy(int xs [],int l, bool (*smaller) (int,int) ){
16   for (int i=l-1;i>=0;i--){
17     for (int j=0;j<i;j++){
18       if (smaller(xs[j+1],xs[j])){
19         int k=xs[j];
20         xs[j]=xs[j+1];
21         xs[j+1]=k;
22       }
23     }
24   }
25 }
```

Und schließlich weiter verallgemeinert, so daß der Typ der Elemente allgemein gehalten wurde:

Bubble.cpp

```
26 template <typename a>
27 void bubbleSortByGen(a xs [],int l, bool (*smaller) (a,a) ){
28   for (int i=l-1;i>=0;i--){
29     for (int j=0;j<i;j++){
30       if (smaller(xs[j+1],xs[j])){
31         a k=xs[j];
32         xs[j]=xs[j+1];
33         xs[j+1]=k;
34       }
35     }
36   }
37 }
```

Zum Abschluß ein paar Beispielaufrufe:

Bubble.cpp

```
38 bool smallerInt(int x, int y){return x<y;}
39
40 bool smallerString(std::string x, std::string y){
41   return x.length()<y.length();}
42
43 bool smallerString2(std::string x, std::string y){
44   return x<y;}
45
46 template <typename a>
47 void printArray(a xs [],int l){
48   std::cout<<"{";
49   for (int i=0;i<l-1;i++){
50     std::cout << xs[i]<<", ";
51   }
52   std::cout << xs[l-1] <<"}"<<std::endl;
53 }
```

```
54
55 int main(){
56   int xs1 [] = {32,4532,32,54,12,43,54,2};
57   printArray(xs1,8);
58   bubbleSort(xs1,8);
59   printArray(xs1,8);
60
61   int xs2 [] = {32,4532,32,54,12,43,54,2};
62   printArray(xs2,8);
63   bubbleSortBy(xs2,8,smallerInt);
64   printArray(xs2,8);
65
66   int xs3 [] = {32,4532,32,54,12,43,54,2};
67   printArray(xs3,8);
68   bubbleSortByGen(xs3,8,smallerInt);
69   printArray(xs3,8);
70
71   std::string xs4 []
72     = {"lear","macbeth","hamlet","othello","prospero"};
73   printArray(xs4,5);
74   bubbleSortByGen(xs4,5,smallerString);
75   printArray(xs4,5);
76
77   std::string xs5 []
78     = {"lear","macbeth","hamlet","othello","prospero"};
79   printArray(xs5,5);
80   bubbleSortByGen(xs5,5,smallerString2);
81   printArray(xs5,5);
82 }
```

Im besten Fall erkennt man aus den drei Sortierfunktionen, wie man sich aus einer speziellen Lösung nach und nach generellere Lösungen ableiten kann

3.2 Überladen

3.2.1 Überladen von Funktionen

Im letzen Abschnitt haben wir eine Funktionsdefinition geschrieben, die als Schablone für beliebig viele Typen stehen konnte. Jetzt gehen wir gerade den umgekehrten Weg. Wir werden Funktionen überladen, so daß wir mehrere Funktionsdefinitionen für ein und dieselbe Funktion erhalten.

Beispiel:
Jetzt schreiben wir die `trace` Funktion nicht mit Hilfe einer Funktionsschablone, sondern 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>
```

3.2.2 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:
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.h

```
1  #include <string>
2  #include <iostream>
3
4  class Complex {
5    public:
6      float im;
7      float re;
8
9      Complex(float re,float im);
10
11     Complex operator+(const Complex& other)const;
12     Complex operator*(const Complex& other)const;
13     float operator()()const;
14
15     std::string toString()const;
16 };
```

Die Operatoren lassen sich wie reguläre Funktionen implementieren:

Complex.cpp

```
1  #include "Complex.h"
2  #include <sstream>
3
4  Complex::Complex(float re,float im){
5    this->re=re;this->im=im;
6  }
7
8  Complex Complex::operator+(const Complex& other)const{
9    return Complex(re+other.re,im+other.im);
10 }
11
12 Complex Complex::operator*(const Complex& other)const{
13   return
14    Complex(re*other.re-im*other.im,re*other.im+im*other.re);
15 }
16
17 float Complex::operator()()const{
18   return re*re+im*im;
19 }
20
21 std::string Complex::toString()const{
22   std::string result="";
23   std::stringstream ss;
24   ss<<re;
25   ss>>result;
26   result=result+"+";
```

```
27
28    std::string resulti="";
29    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.h

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

```
22      std::string toString()const;
23  };
```

Im folgenden die naheliegende Implementierung dieser Listen.

StringLi.cpp

```
1   #include <string>
2   #include "StringLi.h"
3
4   StringLi::StringLi(std::string h,const StringLi* t):empty(false){
5     this->hd=h;
6     this->tl=t;
7   }
8
9   StringLi::StringLi():empty(true){}
10
11  StringLi::~StringLi(){
12    if (!isEmpty()) delete tl;
13  }
14
15  bool StringLi::isEmpty()const{return empty;}
16  std::string StringLi::head()const{return hd;}
17  const StringLi* StringLi::tail()const{return tl;}
18
19  StringLi* StringLi::clone()const{
20    if (isEmpty()) return new StringLi();
21    return new StringLi(head(),tail()->clone());
22  }
23
24  const StringLi* StringLi::operator+(const StringLi* other) const{
25    if (isEmpty()) return other->clone();
26    return new StringLi(head(),(*tail())+other);
27  }
28
29  std::string StringLi::toString()const{
30    if (isEmpty()) return "()";
31    std::string result="["+head();
32    for (const StringLi* it=this->tail()
33        ; !it->isEmpty();it=it->tail()){
34      result=result+","+it->head();
35    }
36    return result+"]";
37  }
```

Listen lassen sich jetzt durch die Addition konkatenieren:

StringLiTest.cpp

```
1   #include <iostream>
2   #include "StringLi.h"
3
```

```
4  int main(){
5    const StringLi* xs
6      = new StringLi("friends",
7        new StringLi("romans",
8        new StringLi("contrymen",
9        new StringLi())));
10
11
12   const StringLi* ys
13    = new StringLi("lend",
14      new StringLi("me",
15      new StringLi("your",
16      new StringLi("ears",
17      new StringLi()))));
18
19   const 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.

Aufgabe 16 Ergänzen Sie die obige Listenklasse um weitere Listenoperatoren.

- `const std::string operator[](const int i) const;` soll das i-te Element zurückgeben.
- `const StringLi* operator-(const int i) const;` soll eine neue Liste erzeugen, indem von der Liste die ersten i Elemente weggelassen werden.

- `const bool operator<(const StringLi* other) const;` soll die beiden Listen bezüglich ihrer Länge vergleichen.
- `const bool operator==(const StringLi* other) const;` soll die Gleichheit auf Listen implementieren.

Testen Sie ihre Implementierung an einigen Beispielen.

3.3 Standardparameter

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:
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:
Jetzt simulieren wir, Standardparameter durch eine überladene Funktionsdefinition.

OverloadedInsteadOfDefaults.cpp

```
#include <iostream>

int f(int x,int y){return x+y;}
int f(int x){return f(x,40);}

int main(){
    std::cout << "f(1,2) = "<< f(1,2) << std::endl;
    std::cout << "f(2)   = "<< f(2) << std::endl;
}
```

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:~>
```

3.4 Generische Klassen

Wir werden in der Programmierung häufig in die Situation kommen, daß sich zwei Klassen lediglich im Typ eines ihrer Felder unterscheiden. Um verdoppelten Code zu verhindern, bietet C++ an, auch Klassen generisch zu schreiben.

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

BoxString.cpp

```
#include <string>

class BoxString{
  public:
    std::string contents;
    BoxString(std::string c) {contents = c;}
};
```

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

BoxInt.cpp

```
class BoxInt{
  public:
    int contents;
    BoxInt(int c) {contents = c;}
};
```

Über den Formularmechanismus von C++ können wir statt dieser beiden Klassen eine einzige Formularklasse, 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 Formularklasse mit genau dieser Typvariablen für die Klasse handeln soll:

Box.h

```
#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 die von den generischen Methoden bereits bekannte Notation mit den 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.h"

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:

Pair.h

```
#include <string>
#include "Box.h"

template <class fstType,class sndType>
class Pair: public Box<fstType>{
```

```
6   public:
7     sndType snd;
8     Pair(fstType c,sndType snd):Box<fstType>(c),snd(snd) {}
9
10    std::string toString(){
11        return "("+contents+","+snd+")";
12    }
13  };
```

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

TestPair.cpp

```
1  #include "Pair.h"
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.

UniPair.h

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

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

TestUniPair.cpp

```
1  #include "UniPair.h"
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 }
```

3.4.1 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.

Zunächst sehen wir zwei Hilfsfunktionen vor, die uns helfen werden, die Methoden `contains` und `toString` möglichst generisch zu formulieren:

Li.h

```
1  #include <string>
2  #include <sstream>
3
4  template <typename X>
5  bool equals(X x,X y){return x==y;}
6
7  std::string callToString(std::string x ){return x;}
8  std::string callToString(int x ){
9
10   std::stringstream ss;
11   std::string result;
12
13   ss << x;
14   ss >> result;
15
16   return result;
17 }
18
19 template <typename X>
20 std::string callToString(X* x ){return x->toString();}
```

Jetzt können wir mit der eigentlichen Klasse beginnen. Wir haben zunächst die gängigen Definitionen für Konstruktoren, Selektoren und Testmethode:

Li.h

```
21 template <typename A>
22 class Li {
23   private:
24     A hd;
25     const Li<A>* tl;
26     const bool empty;
27
28   public:
29     Li(A hd,const Li<A>* tl):empty(false){
30       this->hd=hd;
31       this->tl=tl;
32     }
33
```

```
34      Li():empty(true){}
35
36
37      bool isEmpty()const{return empty;}
38      A head()const{return hd;}
39      const Li<A>* tail()const{return tl;}
```

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

Li.h

```
40      virtual ~Li(){
41        if (!isEmpty()) delete tl;
42      }
```

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

Li.h

```
43      Li<A>* clone()const{
44        if (isEmpty()) return new Li<A>();
45        return new Li<A>(head(),tail()->clone());
46      }
47
48      const Li<A>* operator+(const Li<A>* other) const{
49        if (isEmpty()) return other->clone();
50        return new Li<A>(head(),(*tail())+other);
51      }
52
53      int size() const {
54        if (isEmpty()) return 0;
55        return 1+tail()->size();
56      }
```

Die Methode `contains`, die testen soll, ob ein Element in der Liste enthalten ist, sehen wir in zwei Versionen vor. Der einen Versionen kann eine Gleichheitsfunktion für die Elemente mitgegeben werden, die andere benutzt hierfür den Operator `==`.

Li.h

```
57      bool contains(A el, bool (*eq)(A,A))const{
58        if (isEmpty())return false;
59        if (eq(head(),el)) return true;
60        return tail()->contains(el,eq);
61      }
62
63      bool contains(A el)const{
64        return contains(el,equals);
65      }
```

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.h

```
66      template <typename B>
67      Li<B>* map(B(*f)(A))const{
68        if (isEmpty()) return new Li<B>();
69        return new Li<B>(f(head()),tail()->map(f));
70      }
```

Und schließlich die Methode, um eine Stringdarstellung der Liste zu erhalten. Diese Methode erwartet vom Elementtyp `A`, daß auf disem eine Funktion `callToString` definiert ist. Diese haben wir oben für Zeichenketten und für Zeiger auf Klassen, die eine Methode `toString` haben, definiert.

Li.h

```
71      std::string toString(){
72        if (isEmpty()) return "[]";
73        std::string result="["+(callToString(this->head()));
74        for (const Li<A>* it=this->tail()
75            ; !it->isEmpty();it=it->tail()){
76          result=result+","+(callToString(it->head()));
77        }
78        return result+"]";
79      }
80  };
```

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 `map`. Auch die Methode `toString` läßt sich für die verschiedenen Elementtypen ausführen.

TestLi.cpp

```
1   #include "Li.h"
2   #include "UniPair.h"
3   #include <iostream>
4   #include <string>
5
6   int callSize(std::string s){return s.size();}
7
8   std::string getContents(UniPair<std::string>* p){
9    return p->contents;
10  }
11
12  int main(){
13    Li<std::string>* xs
14      = new Li<std::string>("Schimmelpfennig",
15        new Li<std::string>("Moliere",
16        new Li<std::string>("Shakespeare",
```

```
17        new Li<std::string>("Bernhard",
18        new Li<std::string>("Brecht",
19        new Li<std::string>("Salhi",
20        new Li<std::string>("Achternbusch",
21        new Li<std::string>("Horvath",
22        new Li<std::string>("Sophokles",
23        new Li<std::string>("Calderon",
24        new Li<std::string>())))))))))));
25
26    std::cout << xs->toString() << std::endl;
27    std::cout << xs->contains("Brecht") << std::endl;
28    std::cout << xs->contains("Zabel") << std::endl;
29
30    Li<int>* is = xs->map(callSize);
31    delete xs;
32
33    std::cout << is->toString() << std::endl;
34    delete is;
35
36    Li<UniPair<std::string>* >* ys =
37    new Li<UniPair<std::string>* >(new UniPair<std::string>("A","1")
38   ,new Li<UniPair<std::string>* >(new UniPair<std::string>("B","2")
39   ,new Li<UniPair<std::string>* >(new UniPair<std::string>("C","3")
40   ,new Li<UniPair<std::string>* >())));
41
42    std::cout << ys->toString() << std::endl;
43
44    xs=ys->map(getContents);
45    std::cout << xs->toString() << std::endl;
46  }
```

Der Test führt zu folgenden Ausgaben:

```
sep@linux:~/fh/prog3/examples/src> ./TestLi
[Schimmelpfennig,Moliere,Shakespeare,Bernhard,Brecht,Salhi,Achternbusch,Horvath,Sophokles,Calderon]
1
0
[15,7,11,8,6,5,12,7,9,8]
[(A,1),(B,2),(C,3)]
[A,B,C]
sep@linux:~/fh/prog3/examples/src>
```

3.4.2 Closures, Funktionsobjekte und Schönfinkeleien

In diesem Kapitel sollen ein Paar Konzepte aus der funktionalen Programmierung in C++ vorgestellt werden. Wir werden dabei nicht mehr allein auf C Funktionszeigern zurückgreifen, sondern Klassen schreiben, die Funktionsobjekte repräsentieren.

Closures

Zunächst wollen wir eine Klasse vorstellen, die lediglich einen Wert speichern kann; wir erhalten eine Klasse entsprechend der generischen Klasse `Box` aus dem vorangegangenen Abschnitt. Wie sehen also ein Feld vor, in dem ein beliebiger Wert abgespeichert werden kann. Zum Initialsieren sei ein Konstruktor vorhanden. Ein leerer Konstruktor ermöglicht auch uninitialisierte Werte zu haben:

Closure.h

```
1  template <class A>
2  class Closure {
3
4    private:
5      A value;
6
7    public:
8      Closure <A> (A a) {
9        this->value=a;
10     };
11
12     Closure <A>(){}
```

Auf den abgespeicherten Wert soll über eine `get`-Methode zugegriffen werden. Statt aber eine Methode `getValue` vorzusehen, überladen wir den Operator der Funktionsanwendung. Dieses ist der Operator bestehend aus den rundem Klammerpaar.

Closure.h

```
13     virtual  A operator()() {
14       return value;
15     };
16 };
```

Damit können wir ein `Closure`-Objekt wie eine Funktion ohne Parameter benutzen, die den einmal abgespeicherten Wert als Rückgabe hat:

TestSimpleClosure.cpp

```
1  #include "Closure.h"
2  #include <iostream>
3
4  int main(){
5     Closure<int> f1 = Closure<int>(42);
6     std::cout << f1() << std::endl;
7  }
```

Mit der Klasse `Closure` lassen sich also einfache konstante Funktionen beschreiben. Nun wollen wir dieses Konzept erweitern:

Ein *closure* soll eine noch nicht ausgerechnete Funktionsanwendung bezeichnen. Hierzu betrachten wir zunächst erst einmal Funktionen mit genau einem Argument. Solche Funktionen haben einen Parametertyp und einen Rückgabetyp. Diese beiden Typen lassen wir generisch und beschreiben sie durch eine Typvariable `aT` für *Argumenttyp* und `rT` für *Rückgabetyp*.

ApplyFun.h

```
1  #include "Closure.h"
2  #include <iostream>
3
4  template <class aT,class rT>
5  class ApplyFun : public Closure<rT> {
```

Ein *closure* wird durch eine Funktion und den Wert, auf den diese Funktion angewendet werden soll ausgedrückt:

ApplyFun.h

```
6    private:
7      rT (*f)(aT);
8      aT arg;
```

Zusätzlich sehen wir Felder vor, in denen, sobald es berechnet wurde, das Ergebnis der Funktionsanwendung gespeichert werden kann und die Information, ob das Ergebnis bereits berechnet wurde:

ApplyFun.h

```
9      rT result;
10     bool resultIsEvaluated;
```

Der Konstruktor initialisiert die beiden Felder der Funktion und des Arguments. Für ein neu erzeugtes Objekt wird das Ergebnis der Funktionsanjwendung noch nicht ausgerechnet:

ApplyFun.h

```
11   public:
12     ApplyFun <aT,rT> (rT(*f)(aT),aT a) {
13       this->f=f;
14       this->arg=a;
15       resultIsEvaluated=false;
16     };
17
18     ApplyFun <aT,rT> () {resultIsEvaluated=false;}
```

Entscheidend ist nun, wie der Operator der Funktionsanwedung überschrieben wird: hier soll nun endlich die Funktions auf ihr Argument angewendet werden. Wurde das Ergebnis bereits ausgewertet, so findet es sich im Feld `result`, ansonsten ist es erst zu berechnen und für eventuelle spätere Aufrufe im Feld `result` zu speichern.

ApplyFun.h

```
19     virtual  rT operator()() {
20       if (!resultIsEvaluated){
21         resultIsEvaluated = true;
22         result=f(arg);
23       }
24       return result;
25     };
26 };
```

Mit der Klasse `ApplyFun` haben wir zwei Dinge erreicht:

- wir können ausdrücken, daß wir eine Funktion auf ein Argument zwar gerne aufrufen wollen, aber nicht sofort, sondern erst bei späteren Bedarf.
- der *closure* arbeitet mit einem Ergebnisspeicher. Wurde das Ergenis einmal ausgerechnet, so wird bei einem späteren Aufruf des Funktionsoperators das bereits errechnete Ergebnis benutzt und nicht nochmal ausgerechnet.

Zeit nun endlich einmal mit Objekten der Klasse `ApplyFun` zu arbeiten:

TestClosure.cpp

```
#include "ApplyFun.h"
#include "Li.h"
#include <iostream>
#include <string>
```

Hierzu definieren wir uns zunächst ein paar sehr einfache Funktionen, die wir in *closures* verpacken wollen:

TestClosure.cpp

```
int add1(int x){std::cout << "add1" << std::endl;return x+1;}
int add2(int x){std::cout << "add2" << std::endl;return x+2;}
int add3(int x){std::cout << "add3" << std::endl;return x+3;}
int add4(int x){std::cout << "add4" << std::endl;return x+4;}
int getSize(std::string xs){return xs.length();}
```

Wir können ein paar *closures* anlegen:

TestClosure.cpp

```
int main(){
  Closure<int>* c1 = new ApplyFun<int,int>(add1,41);
  Closure<int>* c2 = new ApplyFun<int,int>(add2,40);
  Closure<int>* c3 = new ApplyFun<int,int>(add3,39);
  Closure<int>* c4 = new ApplyFun<int,int>(add4,38);
```

Erst durch expliziten Aufruf des Funktionsklammeroperators führt dazu, daß tatsächlich eine Funktion auf ihr Argument angewendet wird:

TestClosure.cpp

```
  std::cout << (*c2)() << std::endl;
```

Wir können jetzt z.B. eine Liste von *closures* anlegen.

TestClosure.cpp

```
  Li<Closure<int>* >* xs =
      new Li<Closure<int>* >(c1,
      new Li<Closure<int>* >(c2,
      new Li<Closure<int>* >(c3,
      new Li<Closure<int>* >(c4,
      new Li<Closure<int>* >())));
```

Berechnen wir die Länge einer solchen Liste von *closures*, so werden die einzelnen Werte der Listenelemente nicht ausgerechnet, sie werden dazu ja nicht benötigt:

TestClosure.cpp

```
22   std::cout << xs->size() << std::endl;
```

Erst wenn wir explizit ein Element der Liste berechnet haben wollen, wird dessen Berechnungen angestoßen.

TestClosure.cpp

```
23   std::cout << (*(xs->tail()->tail()->head()))() << std::endl;
24   delete xs;
```

Abschließend ein Beispiel für ein *closure*, der auf Strings arbeitet:

TestClosure.cpp

```
25   Closure<int>*  c5
26     = new ApplyFun<std::string,int>(getSize,"witzelbritz");
27   std::cout << (*c5)() << std::endl;
28 }
```

Das Programm hat folgende leicht nachzuvollziehende Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> ./TestClosure
add2
42
4
add3
42
11
sep@linux:~/fh/prog3/examples/src>
```

In den Programmiersprachen Miranda[Tur85], Clean (`www.cs.kun.nl/~clean`) und Haskell[PJ03] sind *closures* zum Standardprinzip erhoben worden. Für jede Funktionsanwendung wird zunächst ein nicht ausgewerteter *closure* im Speicher angelegt. Erst wenn durch eine `if-` oder `case`-Bedingung das Ergebnis der Berechnung benötigt wird, stößt die Laufzeitumgebung die Berechnung des Ergebnisses der Funktionsanwendung an. Solche Programmiersprachen werden als *lazy* bezeichnet.

Unendliche Listen Über noch nicht ausgewertete Funktionsanwendungen lassen sich jetzt potentiell unendliche Listen beschreiben. Hierzu sehen wir eine Listenklasse vor, die nicht die Schwanzliste in einem Feld gespeichert hat, sondern einen *closure*, in dem lediglich gespeichert ist, wie der Rest Liste zu berechnen ist, falls denn einmal die Methode `tail` aufgerufen wird.

Wir können also unsere Listenklasse so umändern, daß im Feld `tl` nun ein *closure* gespeichert ist und die Methode `tail` zur Auswertung dieser führt. Ansonsten läßt sich unsere bisherige Listenimplementierung übernehmen:

LazyList.h

```
1  #include "ApplyFun.h"
2  #include <iostream>
```

```
3
4  template <class A>
5  class LazyList {
6    private:
7      A hd;
```

Statt also direkt des Schwanz der Liste zu speichern, speichern wir intern einen *closure* für diesen Schwanz:

LazyList.h

```
8      Closure<LazyList<A>*>* tl;
9      const bool empty;
```

Wir sehen sowohl einen `Cons`-Konstruktor für eine direkt übergebene Schwanzliste, als auch für eine *closure*, der die Schwanzliste berechnet vor:

LazyList.h

```
10   public:
11     LazyList(A hd, Closure<LazyList<A>*>* tl):empty(false){
12       this->hd=hd;
13       this->tl=tl;
14     }
15
16     LazyList(A hd,LazyList<A>* t):empty(false){
17       this->hd=hd;
18       this->tl= new Closure<LazyList<A> >(t);
19     }
```

Der Konstruktor für leer Listen fehlt natürlich nicht:

LazyList.h

```
20     LazyList():empty(true){}
```

Die Methoden `head` und `isEmpty` funktionieren, wie bereits bekannt:

LazyList.h

```
21     bool isEmpty()const{return empty;}
22     A head()const{return hd;}
```

Die Methode `tail` sorgt jetzt dafür, daß die Schwanzliste endlich ausgerechnet wird:

LazyList.h

```
23     LazyList<A>* tail(){
24       return (*tl)();
25     }
26 };
```

Jetzt können wir z.B. die Liste aller natürlichen Zahlen definieren. Trotzdem läuft uns der Speicher nicht voll, weil wir immer nur die nächste Restliste erzeugen, wenn die Methode `tail` aufgerufen wird. Hierzu schreiben wir eine Methode `from`, die einen Listenknoten von `Lazylist` erzeugt, und für den `tail` einen *closure* erzeugt:

TestLazyList.cpp

```
#include "LazyList.h"
#include <iostream>

LazyList<int> *  from(int fr){
  Closure<LazyList<int>* >* tl
   = new ApplyFun<int,LazyList<int>*>(from,fr+1);
  return new LazyList<int>(fr,tl);
}
```

Folgender kleiner Test illustriert, die Benutzung der verzögerten Listen:

TestLazyList.cpp

```
int main(){
  LazyList<int>* xs = from(2);
  for (int i=0;i<200000;i=i+1){
    LazyList<int>* ys = xs->tail();
    std::cout << xs->head() << std::endl;
    delete xs;
    xs = ys;
  }
}
```

Wenn wir dieses Programm starten bekommen wir nacheinander die natürlichen Zahlen ausgegeben. Das Programm hat einen konstanten Speicherbedarf.

Aufgabe 17 **(optional)** Schreiben Sie weitere Funktionen, die Objekte der Klasse `LazyList` erzeugen und testen Sie diese:

a) Schreiben Sie eine generische Funktion:

```
template <typename A>
LazyList<A>* repeat(A x)
```

die eine Liste erzeugt, in der unendlich oft Wert `x` wiederholt wird.

b) Schreiben Sie eine Funktion, die eine unendliche Liste von Pseudozufallszahlen erzeugt. Benutzen sie hierzu folgende Formel:
$z_{n+1} = (91 * z_n + 1) \bmod 347$

Funktionsobjekte

Die Objekte der Klasse `Closure` des letzten Abschnitts haben konstante Funktionen ohne Parameter repräsentiert. In diesen Abschnitt wollen wir echte Funktionsobjekte modellieren, die Funktionen mit einem Parameter darstellen. Hierzu kapseln wir einfach einen Funktionszeiger in einem Objekt und überladen den Funktionsanwendungsoperator für diese Klasse, so daß er direkt zur Funktionsanwendung führt. Wir erhalten die folgende einfache Klasse:

Lambda.h

```
#ifndef LAMBDA__H
#define LAMBDA__H  ;

template <class aT, class rT>
class Lambda {
  private:
    rT (*f)(aT);

  public:
    Lambda <aT,rT> (rT(*_f)(aT)){
      this->f=_f;
    };

    Lambda <aT,rT> (){}

    virtual  rT operator()(aT arg) {
      f(arg);
    };
};
#endif
```

Ein Objekt des Typs `Lambda<argType,resultType>` kann nun ebenso benutzt werden wie ein Funktionszeiger des Typs: `resultType (* f)(argType)`.

TestLambda.cpp

```
#include "Lambda.h"
#include <iostream>

int add5(int x) {return x+5;}

int main(){
  Lambda<int,int> f1(add5);
  std::cout << add5(37) << std::endl;
  std::cout << f1(37)   << std::endl;
}
```

Currying

Im letzten Abschnitt haben wir eine Klasse geschrieben, die Funktionen darstellen kann, indem ein Funktionszeiger einfach gekapselt wird. Wir können jetzt Unterklassen der Klasse `Lambda` definieren, die uns erlauben auf verschiedene Weise Funktionsobjekte zu erzeugen.

Wir wollen eine Klasse schreiben, mit der sich Funktionsobjekte nach dem Prinzip des sogenannten *Currying* erzeugen lassen. Diesen Prinzip ist nicht nach einem indischen Reisgericht sondern nach dem Mathematiker Haskell B. Curry(1900-1982) benannt[2].

[2]Obwohl die Idee hierfür wohl ursprünglich von Moses Schönfinkel stammt, aber *Schönfinkeling* wäre wohl eine wenig attraktive Bezeichnung gewesen.

Die Grundidee des Currying ist, eine Funktion nur partiell auf nicht alle Parameter anzuwenden. Eine Funktion f mit zwei Argumenten, also mit dem Typ: $(a, b) \rightarrow c$ kann betrachtet werden als eine Funktion mit nur einem Parameter, deren Ergebnis eine Funktion mit einem zweiten Parameter ist, also mit dem Typ: $a \rightarrow (b \rightarrow c)$. Unter dieser Betrachtungsweise läßt sich eine Funktion f zunächst auf das erste Argument anwenden und dann das Ergebnis dieer Anwendung auf das zweite Argument. Also statt $f(x, y)$ kann die zweistellige Funktion betrachtet werden als: $(f(x))(y)$. Somit lassen sich unter der Voraussetzung, daß Funktionen Daten sind, die das Ergebnis einer anderen Funktion sein können, alle Funktionen als einstellige Funktionen modellieren.

Jetzt wollen wir in C++ eine Funktionsklasse schreiben, die für eine zweistellige Funktion und einen Wert für den ersten Paramter eine Funktion, die noch den zweiten Parameter erwartet, schreiben. Hierzu leiten wir eine generische Klasse `Curry` von unserer Funktionsklasse `Lamda` ab.

Curry.h

```
1  #include <iostream>
2  #include "Lambda.h"
3
4  template <typename a,typename b,typename c>
5  class Curry : public Lambda<b,c>{
```

Ein `Curry`-Objekt soll für eine zweistellige Funktion und einen Wert für den ersten Parameter eine Funktion erzeugen, die den zweiten Parameter erwartet. Hierzu sehen wir Felder für Funktion und ersten Parameter vor und initialisieren diese im Konstruktor:

Curry.h

```
6    public:
7      a i;
8      c (*f)(a x,b y);
9
10     Curry(c(*f)(a x, b y),a i):i(i),f(f){}
```

Im Funktionsanwenungsoperator wird schließlich die ursprünglich zweistellige Funktion auf die endlich vorliegenden zwei Parameter angewendet:

Curry.h

```
11     virtual c operator()(b y){ return f(i,y);}
12 };
```

Zeit für einen kleinen Test. Wir benutzen eine zweistellige Additionsfunktion. Aus dieser können wir neue einstellige Funktionen über Currying erzeugen, die auf einen Wert einen festen Zahlenwert draufaddieren:

TestCurry.cpp

```
1  #include "Curry.h"
2
3  int add(int x,int y){return x+y;}
4
5  int main(){
6    Lambda<int,int>* add5 = new Curry<int,int,int>(add,5);
```

```
7   int i = (*add5)(37);
8   std::cout << i << std::endl;
9 }
10
```

Das Prinzip des *Currying* kan ein nützliches Entwurfsmuster sein, wenn die Daten der Argumente für eine Funktionsanwednung nicht zu einem Zeitpunkt gleichzeitig vorliegen. Dann läßt sich die Funktion schon einmal auf die zunächst bereits vorliegenden Daten anwenden und das entstehende Funktionsobjekt für die weiteren Anwendungen speichern. Insbesondere in der Programmierung von Graphiken kann es eine solche Situation geben. Angenommen es gibt eine Funktion zum zeichnen von Vierecken mit 5 Argumenten:
`drawRect(Color c,Width w,Height h,Pos xPos,Pos yPos)`

Mittels *Currying* ließen sich jetzt neue Funktionen definieren, eine in der die Farbe bereits gesetzt ist, eine in der die Form auf Quadrate einer bestimmten Größe festgelegt ist:

```
1 drawRedRect = curry(drawRect,red);
2 drawRedSquare = curry(drawRedRect,3,3);
```

Die so neu gebauten Funktionsobjekte lassen sich nun benutzen, um an unterschiedlichen Positionen rote Quadrate zu zeichnen:

```
1 drawRedSquare(42,80);
2 drawRedSquare(17,4);
3 drawRedSquare(08,15);
4 drawRedSquare(47,11);
```

Kapitel 4

Ausnahmen, Namensräume, Konstantendeklaration

4.1 Namensräume

Die Namensräume in C++ entsprechen vom Konzept her den Paketen in Java. Unterschiede sind, daß nicht wie in Java sich die Paketstruktur in der Ordnerstruktur auf dem Dateisystem wiederspiegelt und daß es kein Sichtbarkeitsattribut, das sich auf Pakete bezieht, gibt.

Namensräume können hierarchisch sein, wie es in Java Unterpakete gibt, so gibt es also auch Unternamensräume.

Eigene Namensräume entsprechend der `package`-Klausel in Java werden in C++ über das Schlüsselwort `namespace` vereinbart. Es folgt in geschweiften Klammern ein Block, dessen Definitionen in dem gerade definierten Namensraum liegen.

Die Qualifizierung eines Bezeichners über seinen Namensraum wird in C++ mit einem zweifachen Doppelpunkt vorgenommen (in Java war dies einfach durch einen Punkt.).

Zur unqualifizierten Benutzung von Bezeichnern aus einem Namensraum kann die Klausel `using namespace` benutzt werden. Dieses entspricht dem `import` in Java. Die `using namespace` Deklaration gilt ähnlich wie Variablendeklarationen nur in dem Block, in dem sie getätigt wird, also nicht außerhalb des Blockes.

Beispiel:
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  }
```

4.2 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.

4.2.1 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.

4.2.2 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  }
```

4.2.3 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.

4.2.4 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

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

int fac(int n){
  assert(n>=0);
  if (n==0) return 1;
  return n*fac(n-1);
}

int main(){
  std::cout<<fac(5)<<std::endl;
  std::cout<<fac(-5)<<std::endl;
}
```

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

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

int fac(int n){
#ifdef DEBUG
  assert(n>=0);
#endif
  if (n==0) return 1;
  return n*fac(n-1);
}

int main(){
  std::cout<<fac(5)<<std::endl;
  std::cout<<fac(-5)<<std::endl;
}
```

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.

4.3 Konstantendeklarationen

Es lassen sich in C++ Werte als unveränderbar deklarieren.[1] Hierzu dient in C das Schlüsselwort `const`.

4.3.1 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 }
```

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 {
```

[1] 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.

```
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>
```

4.3.2 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 Funiktion `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.

Teil II

Interessante Bibliotheken

Kapitel 5

Die Standard Template Library

Die wichtigste Bibliothek in C++ stellt die *standard template library (STL)* dar. In ihr finden sich Klassen für Sammlungen, aber auch die Klasse für Strings, die wir bereits benutzt haben; und Klassen, die sich mit Funktionsobjekten beschäftigen. Die STL benutzt exessiv Klassenschablonen, auch für Klassen, bei denen man das nicht gleich erwarten würde, wie z.B. `string`. In der STL werden fast alle in den vorangegangenen Abschnitten vorgestellten Konzepte benutzt, wie z.B. generische Klassen und Funktionen, überladene Operatoren, Funktionen höherer Ordnung, verallgemeinerte Zeigertypen und verallgemeinerte Funktionstypen. Das Bestreben, die in der Bibliothek umgesetzte Funktionaliltät möglichst allgemein sowohl für generische Sammlungsklassen als auch für Reihungen anzubieten, führt dazu, daß die Methoden in der STL nicht Objektmethoden sind, sondern globale Funktionen, die in gleicher Weise für Reihungen als auch auf Sammlungen funktionieren. Daher war es auch sinnvoll, den Funktionen nicht allgemein Sammlungen bzw. Reihungen als Parameter zu übergeben, sondern ein verallgemeinertes Konzept von Zeigern auf das erste und das letzte Element der Sammlung, für die eine Funktion auszuführen ist.

5.1 ein kleines Beispiel

Als kleines Beispiel, an dem fast alle wichtigen Konzepte der STL zu erkunden ist, wollen wir eine Sammlung von Zahlen nehmen, der wir eine einstellige Funktion übergeben. Die Funktion soll auf jedes Element der Sammlung angewendet werden. Hierzu stellt die STL die Funktion `for_each` zur Verfügung. Sie hat drei Parameter, den Anfang und das Ende des Bereichs, auf dem die im dritten Parameter übergebene Funktion angewendet werden soll.

5.1.1 Anwendung auf Reihungen

Zunächst betrachten wir, wie die Funktion `for_each` für eine Reihung angewendet werden kann. Wir schreiben eine Funktion, die ganze Zahlen verdoppelt. In der Hauptmethode legen wir eine Reihung von drei Elementen an und wenden die Funktion durch einen Aufruf von `for_each` auf jedes Element der Reihung an.

ForEachArray.cpp

```
1 #include <algorithm>
2 #include <iostream>
```

```
3
4  void doppel(int& x){x=2*x;}
5
6  int main(){
7    int  ar[3] = {17,28,21};
8    std::for_each(ar,ar+3,doppel);
9
10   for (int i= 0;i<3;i=i+1){
11     std::cout << ar[i] << std::endl;
12   }
13 }
```

5.1.2 Anwendung auf Vektoren

Jetzt machen wir dasselbe Spiel nicht für eine Reihung sondern für eine Sammlung der STL-Klasse `vector`.

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 }
```

Wie man sieht, ist die Klasse `vector` generisch über ihren Elementtyp[1]. Der Operator `[]` ist überladen, so daß er ebenso wie bei Reihungen benutzt werden kann. Lediglich das Argument, das den Anfang des Elementbereichs bezeichnen, für den `for_each` anzuwenden ist, wird über die Objektmethode `begin()` der Klasse `vector` bestimmt. Ansonsten sieht das Programm fast identisch zu der Version Auf Reihungen aus.

Nach diesem kleinen motivierenden Beispiel betrachten wir in den folgenden Abschnitten die in der STL realisierten Konzepte im Einzelnen.

5.2 Konzepte

Eine Schwäche der C++ Schablonen haben wir schon angesprochen. Es gibt keine Möglichkeit in C++ auszudrücken, welche Typen für die Typvariablen eingesetzt werden können. Dieses

[1] So wie ja auch Reihungen über den Elementtyp generisch sind.

kann nicht in der Signatur oder in irgendeiner Weise in C++ ausgedrückt werden. Hier ist man auf Versuch und Irrtum angewiesen. Erst wenn man versucht, eine Schablone mit einem bestimmten Typ zu benutzen, kann es zu einem Fehler kommen, wenn sich herausstellt, daß dieser Typ nicht für die Typvariable eingesetzt werden konnte. Ein solches Beispiel haben wir in dem Programm `TraceWrongUse.cpp` bereits kennengelernt.

Den Entwicklern der STL war dieses Problem der C++ Formularklasse natürlich bewußt. Sie haben daher großen Wert darauf gelegt, möglichst genau in der Dokumentation zu beschreiben, mit was für Typen eine Typvariable instanziiert werden darf. Hierfür haben die STL Entwickler sogenannte Konzepte *(concepts)* eingeführt. Konzepte existieren nur in der STL-Dokumentation und sind kein Bestandteil von C++.

Als Beispiel betrachten wir jetzt einmal die Signatur der oben benutzten Funktion `for_each`:

```
1 template <typename InputIterator,typename UnaryFunction>
2 UnaryFunction for_each
3  (InputIterator first, InputIterator last, UnaryFunction f);
```

Wie man sieht, ist die Funktion über zwei Typen generisch geschrieben. Für die zwei generisch gelassene Typen werden die Typvariablen `UnaryFuntion` und `InputIterator` benutzt. Betrachtet man die STL Dokumentation, so stellt man fest, daß für die Typvariablennamen `UnaryFunction` und `InputIterator` je eine eigene Dokumentationsseite existiert. In dieser Seite ist beschrieben, was ein Typ, der für die Variable eingesetzt werden kann, alles für Funktionalität vorweisen können muß. Dieses wird in der STL Dokumentation als *Konzept* bezeichnet. So sind wir nicht überrascht, daß für das Konzept `UnaryFunction` verlangt wird, daß Typen dieses Konzepts den einstelligen Funktionsanwendungsoperator besitzen.

5.3 Iteratoren

Eines der wichtigsten Konzepte der STL sind Iteratoren. In der STL stellen Operatoren ein verallgemeinertes Prinzip von Zeigern auf Elementen einer Sammlung dar.

Auch Iteratoren sind in der STL als Konzept spezifiziert. Ein Iterator ist ein Typ, für den bestimmte Funktionen und Operationen zur Verfügung stehen.

5.3.1 Triviale Iteratorn

Das einfachste Konzept eines Iterators, dem alle anderen Iteratoren zu Grunde liegen, ist der `trivial iterator`.

Für alle Typen `X` des Konzepts `trivial iterator` wird verlangt:

- ein parameterloser Standardkonstruktor, es muß also möglich sein eine Variable ohne explizite Initialisierung zu erzeugen: `X x;`
- die Dereferenzierung muß überladen sein, also `*x` muß für Elemente diesen Typs möglich sein.
- Zuweisung an die dereferenzierte Stelle, also der Ausdruck: `*x = t`.
- Dereferenzierter Zugriff auf Attribute, also die Pfeiloperation muß zur Verfügung stehen: `x->m`.

5.3.2 InputIterator

Das erste Konzept eines Iterators ist uns in der Funktion `for_each` begegnet. Dabei handelte es sich um einen einfachen Operator, der zu den im Konzept des trivialen Iterators noch eine weitere Operation zuläßt, nämlich das Erhöhen des Iterators, so das er auf ein Nachfolgeelement zeigt. Dieses drückt sich dadurch aus, daß erwartet wird, daß der Operator `++` für Typen dieses Konzepts existiert. Es soll also möglich sein, den Iterator um eins zu erhöhen, so daß seine Dereferenzierung auf ein nächstes Element zeigt.

Wie man sieht, erfüllen Zeiger allgemein die Anforderungen eines `InpuIterators`. Über die Zeigerarithmetik können wir jeweils auf das nächste im Speicher anbgelegte Element gleichen Typs zugreifen.

Wenn wir zusätzlich noch eine Gleichheit auf InputIteratoren voraussetzen, so haben wir alles, um mit zwei Iteratoren, die Anfang und Ende eines zu iterierenden Elementbereichs bezeichnen, über alle Elemente dieses Bereichs einmal von vorne bis hinten durchzuiterieren:

Beispiel:
Wir schreiben eine generische Funktion, die ermöglicht das größte Element eines Iterationsbereichs zu bestimmen.

InputIterator.cpp

```
1  #include <iostream>
2  #include <string>
3
4  template <typename InputIterator,typename Element>
5  Element* maximum(InputIterator begin,InputIterator end){
6    Element* result = begin;
7    for (InputIterator it = begin+1;it<end;it++){
8      if ((*result) < (*it)){ result=it;}
9    }
10   return result;
11 }
12
13 int main(){
14   int  ar[3] = {17,28,21};
15   std::cout << * (maximum<int*,int>(ar,ar+3)) << std::endl;
16
17   std::string  sr[3] = {"shakespeare","calderon","moliere"};
18   std::cout << *(maximum<std::string*,std::string>(sr,sr+3))
19             << std::endl;
20 }
```

Eine Schwäche der Modellierung in der STL läßt sich erkennen: der Elementtyp spielt für einen Iterator keine Rolle. Es wird nicht ausgedrückt, daß die Elemente auf die ein Iterator zeigt von einen bestimmten Typ sein sollen. Dieses wird erst bei der Anwendung der generischen Funktion auf bestimmte Werte Typen geprüft.

5.4 Behälterklassen

Die STL stellt primär Behälterklasssen zur Verfügung. Hierzu gehören sowohl listenartige Sequenzen, als auch Mengen und Abbildungen und schließlich auch Strings.

5.4.1 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.[2]
- `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.

Beispiel:
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.

VectorTest.cpp

```
1  #include <vector>
2  #include <list>
3  #include <string>
4  #include <iostream>
5
6  template <typename Iterator>
7  void doPrint(Iterator begin,Iterator end){
8    std::cout << "[";
9    for (Iterator it = begin;it!=end;){
10     std::cout << *it;
11     it++;
12     if (it != end) std::cout << ",";
```

[2] Diese Klasse entspricht am ehesten der Klasse `java.util.ArrayList` aus Java.

```
13    }
14    std::cout << "]"<<std::endl;
15  }
16
17  int main (){
18    std::string words [] ={ "the","world","is","my","oyster"};
19    std::vector<std::string> sv;
20    std::list<std::string> sl;
21    std::cout << sv.capacity() << std::endl;
22
23    for (int i=0;i<5;i++){
24      sv.insert(sv.end(),words[i]);
25      sl.insert(sl.end(),words[i]);
26    }
27    std::cout << sv.capacity() << std::endl;
28
29    doPrint(sv.begin(),sv.end());
30    sl.reverse();
31    doPrint(sl.begin(),sl.end());
32  }
```

5.4.2 Mengen und Abbildungen

Die zweite große Gruppe von Behäterklassen 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`:
- `multiset`:
- `multimap`:

Entsprechend gibt es die gleichen Klassen mit dem Haschprinzip unter den Namen: `hash_set`, `hash_map`, `hash_multiset` und `hash_multimap`.

Beispiel:
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
7    public:
8      std::string strasse;
9      int hausnummer;
10     Address(){}
11     Address(std::string strasse,int hasunummer)
12         :strasse(strasse),hausnummer(hausnummer){}
13 };
14
15
16 int main(){
17   std::map<std::string, Address,std::less<std::string> >
18     addressBook;
19   addressBook.insert(
20    std::pair<std::string,Address>
21     ("Hoffmann",Address("Gendarmenmarkt",42)));
22   addressBook.insert(
23    std::pair<std::string,Address>
24     ("Schmidt",Address("Bertholt-Brecht-Platz",1)));
25
26   std::cout << addressBook["Hoffmann"].strasse << std::endl;
27
28   addressBook["Tierse"] = Address("Pariser Platz",1);
29
30   std::cout << addressBook["Tierse"].strasse << std::endl;
31   std::cout << addressBook["Andrak"].strasse << std::endl;
32 }
```

Man beachte, daß die Klasse `map` über drei Typen parameterisiert ist:

- der Typ des Suchschlüssels
- der Typ des assoziierten Wertes
- der Typ der Kleinerrelation auf dem Schlüssel

5.4.3 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:
Folgendes kleine Programm demonstriert einen sehr rudimentären Umgang mit Strings.

Pallindrom.cpp

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

using namespace std;

template <typename t>
bool isPallindrome(t w){
  t w2 = w;
  reverse(w2.begin(),w2.end());
  return w==w2;
}

int main(){
  cout << isPallindrome<string>("rentner")  << endl;
  cout << isPallindrome<string>("pensioner")<< endl;
  cout << isPallindrome<string>("koblbok")  << endl;
  cout << isPallindrome<string>("alexander")<< endl;
  cout << isPallindrome<string>("stets")    << endl;
  cout << isPallindrome<string>
                   ("ein neger mit gazellezag tim regen nie")
            << endl;
}
```

5.5 Algorithmen

Wir haben bereits gesehen, daß die STL nicht durchgängig objektorientiert entworfen wurde. Nur Basisfunktionalität ist in den einzelnen Behälterklassen implementiert. Darüberhinaus stehen Funktionen zur Verfügung, die verschiedenste Algorithmen auf Behälterklassen realisieren. Die Algorithmen bekommen jeweils den Start- und Enditerator des Abschnitts eines Behäterobjekts als Parameter übergeben.

Ein paar Beispiele dieser Algorithmen haben wir bereits kennengelernt, wie z.B. `reverse`, der die Reihenfolge in einer Sequenz umdreht. Eine weitere Funktion auf Behäterklassen war die Funktion `for_each`. Diese Funktion ist eine Funktion höherer Ordnung, indem sie nämlich eine weitere Funktion als Argument hat, die auf jedes Element angewendet werden soll.

Die STL stellt in ihrem Entwurf eine interessante Symbiose von Konzepten der objektorientierten Programmierung und Konzepten der funktionalen Programmierung dar. Viele Algorithmen für Behäterobjekte sind wie `for_each` höherer Ordnung, indem Sie eine Funktion als Parameter übergeben bekommen.

5.5.1 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. Auch kann über eine Funktion `bind1st` *currying* praktiziert werden.

Beispiel:
Folgendes Programm demonstriert Currying in der STL mit der Funktion `bind1st`.

STLCurry.cpp

```
1  #include <functional>
2  #include <iostream>
3
4  int main(){
5    std::cout <<bind1st(std::plus<int>(),17)(25) <<std::endl;
6  }
```

Beispiel:
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

```
1  #include <algorithm>
2  #include <functional>
3  #include <iostream>
4
5  using namespace std;
6
7  void print (int a){cout << a <<",";}
8
9  int main(){
10   int xs [] = {1,2,3,4,5};
11   int ys [] = {6,7,8,9,10};
12   int zs [5];
13   transform(xs,xs+5,ys,zs,plus<int>());
14   for_each(zs,zs+5,print);
15   cout << endl;
16 }
```

Beispiel:
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

```
1  #include <vector>
2  #include <list>
3  #include <string>
4  #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>
```

Aufgabe 18 Sie haben bereits einmal eine Funktion `fold` implementiert. Jetzt sollen Sie die Funktion noch einmal allgemeiner schreiben.

a) Implementieren Sie jetzt die Funktion `fold` zur Benutzung für allgemeine Behälterklassen im STL Stil entsprechend der Signatur:

STLFold.h

```
1  template <typename Iterator,typename BinFun,typename EType>
2  EType fold(Iterator begin,Iterator end,BinFun f,EType start);
```

b) Testen Sie Ihre Implementierung mit folgendem Programm:

STLFoldTest.cpp

```
1  #include <vector>
2  #include <string>
3  #include <functional>
4  #include <iostream>
5  #include "STLFold.h"
6
7  int addLength(int x,std::string s){return x+s.length();}
8
9  std::string addWithSpace(std::string x,std::string y){
10     return x+" "+y;
11 }
12
13 int main(){
14   int xs []={1,2,3,4,5};
15   int result;
16   result = fold(xs,xs+5,std::plus<int>(),0);
17   std::cout << result << std::endl;
18   result = fold(xs,xs+5,std::multiplies<int>(),1);
19   std::cout << result << std::endl;
20
21   std::vector<std::string> ys;
22   ys.insert(ys.end(),std::string("friends"));
23   ys.insert(ys.end(),std::string("romans"));
```

```
24    ys.insert(ys.end(),"contrymen");
25    result = fold(ys.begin(),ys.end(),addLength,0);
26    std::cout << result << std::endl;
27
28    std::string erg
29      = fold(ys.begin(),ys.end()
30            ,std::plus<std::string>(),std::string(""));
31    std::cout << erg << std::endl;
32
33    erg=fold(ys.begin(),ys.end(),addWithSpace,std::string(""));
34    std::cout << erg << std::endl;
35  }
```

c) Schreiben Sie weitere eigene Tests.

Kapitel 6

DOM: eine Bibliothek zur XML-Verarbeitung

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Beispiel:
Der eigentliche Text des Dokuments sei:

```
1 The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

```
1 <cd>
2   <artist>The Beatles</artist>
3   <title>White Album</title>
4 </cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem W3C (`http://www.w3c.org`) . Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)*, weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt. Die aktuelle Empfehlung für XML liegt seit anfang des 2004 als Empfehlung in der Version 1.1 vor [T. 04].

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran, daß es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so daß die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[1]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

6.1 XML-Format

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, daß es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

```
1 <?xml version="1.0"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.1. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, daß es jemals eine Version 2.0 von XML geben wird. Zu viele weitere Techniken und Empfehlungen basieren auf XML, so daß die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

6.1.1 Elemente

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen*, der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit `<name>` und endet mit `</name>`. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, daß es genau ein einziges oberstes Element hat.

Beispiel:
Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

```
1 <?xml version="1.0"?>
2 <myText>Dieses ist der Text des Dokuments. Er ist
3 mit genau einem Element markiert.
4 </myText>
```

[1] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Beispiel:
Folgendes Dokument enthält ein Element ohne Inhalt:

```
1 <?xml version="1.0"?>
2 <skript>
3   <page>erste Seite</page>
4   <page></page>
5   <page>dritte Seite</page>
6 </skript>
```

Leere Elemente

Für ein Element mit Tagnamen *name*, das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Beispiel:
Das vorherige Dokument läßt sich somit auch wie folgt schreiben:

```
1 <?xml version="1.0"?>
2 <skript>
3   <page>erste Seite</page>
4   <page/>
5   <page>dritte Seite</page>
6 </skript>
```

Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)*.

Beispiel:
Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

```
1 <?xml version="1.0"?>
2 <myText>Der <landsmann>Italiener</landsmann>
3 <eigename>Ferdinand Carulli</eigename> war  als Gitarrist
4 ebenso wie der <landsmann>Spanier</landsmann>
5 <eigename>Fernando Sor</eigename> in <ort>Paris</ort>
6 ansäßig.</myText>
```

XML-Dokumente als Bäume

Die wohl wichtigste Beschränkung für XML-Dokumente ist, daß sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder geschlossen wurden.

Beispiel:
Das folgende ist kein gültiges XML-Dokument. Das Element `<bf>` wird geschlossen bevor das später geöffnete Element `<em>` geschlossen worde.

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em>kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

Das Dokument wäre wie folgt als gültiges XML zu schreiben:

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift</bf><bf> <em>kursiv und fett</em></bf>
  <em>nur noch kursiv</em>.
</validDocument>
```

Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur von XML-Dokumenten läßt sich sehr schön veranschaulichen, wenn man die Darstellung von XML-Dokumenten in Webbrowsern wie Firefox oder Microsofts Internet Explorer betrachtet.

6.1.2 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet.

Beispiel:
Dokument mit einem Attribut für ein Element.

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```

6.1.3 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit `<!--` begonnen und mit `-->` beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes.

Beispiel:
Im folgenden Dokument ist ein Kommentar eingefügt:

```
1 <?xml version="1.0"?>
2 <drehbuch filmtitel="Ben Hur">
3 <akt>
4   <szene>Ben Hur am Vorabend des Wagenrennens.
5     <!--Diese Szene muß noch ausgearbeitet werden.-->
6   </szene>
7 </akt>
8 </drehbuch>
```

6.1.4 Character Entities

Sobald in einem XML-Dokument eine der spitze Klammern `<` oder `>` auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muß man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities*. Eine Character Entity beginnt in XML mit dem Zeichen `&` und endet mit einem Semikolon`;`. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

Entity	Zeichen	Beschreibung
`&lt;`	`<`	(less than)
`&gt;`	`>`	(greater than)
`&amp;`	`&`	(ampersant)
`&quot;`	`"`	(quotation mark)
`&apos;`	`'`	(apostroph)

Somit lassen sich in XML auch Dokumente schreiben, die diese Zeichen als Text beinhalten.

Beispiel:
Folgendes Dokument benutzt Character Entities um mathematische Formeln zu schreiben:

```
1 <?xml version="1.0"?>
2 <gleichungen>
3   <gleichung>x+1&gt;x</gleichung>
4   <gleichung>x*x&lt;x*x*x für x&gt;1</gleichung>
5 </gleichungen>
```

6.1.5 CDATA-Sections

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeutung haben. XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge: `]]>`. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden.

Beispiel:
Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>
```

6.1.6 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction* mit `<?` und endet mit `?>`. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle.

Beispiel:
Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, daß das Skript in eine HTML-Darstellung umwandelt:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

6.1.7 Namensräume

Die *Tagnamen* sind zunächst einmal Schall und Rauch. Erst eine externes Programm wird diesen Namen eine gewisse Bedeutung zukommen lassen, indem es auf die *Tagnamen* in einer bestimmten Weise reagiert.

Da jeder Autor eines XML-Dokuments zunächst vollkommen frei in der Wahl seiner *Tagnamen ist*, wird es vorkommen, daß zwei Autoren denselben *Tagnamen* für die Markierung gewählt haben, aber semantisch mit diesem Element etwas anderes ausdrücken wollen. Spätestens dann, wenn verschiedene Dokumente verknüpft werden, wäre es wichtig, daß *Tagnamen* einmalig mit einer Eindeutigen Bedeutung benutzt wurden. Hierzu gibt es in XML das Konzept der Namensräume.

Tagnamen können aus zwei Teilen bestehen, die durch einen Doppelpunkt getrennt werden:

- dem Präfix, der vor dem Doppelpunkt steht.
- dem lokalen Namen, der nach dem Doppelpunkt folgt.

Hiermit allein ist das eigentliche Problem gleicher *Tagnamen* noch nicht gelöst, weil ja zwei Autoren den gleichen Präfix und gleichen lokalen Namen für ihre Elemente gewählt haben können. Der Präfix wird aber an einem weiteren Text gebunden, der eindeutig ist. Dieses ist der eigentliche Namensraum. Damit garantiert ist, daß dieser Namensraum tatsächlich eindeutig ist, wählt man als Autor seine Webadresse, denn diese ist weltweit eindeutig.

Um mit Namensräume zu arbeiten ist also zunächst ein Präfix an eine Webadresse zu binden; dies geschieht durch ein Attribut der Art:
`xmlns:myPrefix="http://www.myAdress.org/myNamespace"`.

Beispiel:
Ein Beispiel für ein XML-Dokument, daß den Präfix `sep` an einem bestimmten Namensraum gebunden hat:

```
1 <?xml version="1.0"?>
2 <sep:skript
3    xmlns:sep="http://www.tfh-berlin.de/~panitz/dv2">
4   <sep:titel>Grundlagen der DV 2</sep:titel>
5   <sep:autor>Sven Eric Panitz</sep:autor>
6 </sep:skript>
7
```

Die Webadresse eines Namensraumes hat keine eigentliche Bedeutung im Sinne des Internets. Das Dokument geht nicht zu dieser Adresse und holt sich etwa Informationen von dort. Es ist lediglich dazu da, einen eindeutigen Namen zu haben. Streng genommen brauch es diese Adresse noch nicht einmal wirklich zu geben.

6.2 Codierungen

XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehrerern zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen die 8 Bit im Speicher belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem `ß` zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt.

Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Beispiel:
Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes.

Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

6.3 DOM: XML als Bäume in C++

XML-Dokumente sind logisch gesehen nichts weiteres als Bäume. Die allgemeine Schnittstellenbeschreibung für XML als Baumstruktur ist das *distributed object modell* kurz *dom*[Arn04], für das das W3C eine Empfehlung herausgibt. Die Ursprünge von *dom* liegen nicht in der XML-Programmierung sondern in der Verarbeitung von HTML-Strukturen im Webbrowser über Javascript. Mit dem Auftreten von XML entwickelte sich der Wunsch, eine allgemeine, platform- und sprachunabhänge Schnittstellenbeschreibung für XML- und HTML-Baumstrukturen zu bekommen, die in verschiedenen Sprachen umgesetzt werden kann.

Wir werden im folgenden die Implementierung von DOM im Projekt Xerces-c der Apache Gruppe (`http://xml.apache.org/xerces-c/`) benutzen. Zum Linken der Programme ist hierzu die Option `-lxerces-c` dem Compiler mitzugeben.

Das DOM APi definiert eine gemeinsame Oberklasse für alle Knotentypen des XML-Baums, die Klasse `DOMNode`. Unterklassen dieser Klasse repräsentieren die verschiedenen Knotenarten, die es im XML-Baum geben kann, wie `DOMElement` für Elemente, `DOMCharacterData` für die verschiedenen Textknoten oder auch `DOMAttr` für Attribute.

Abbildung 6.1 zeigt graphisch die Unterklassen von `DOMNode`.

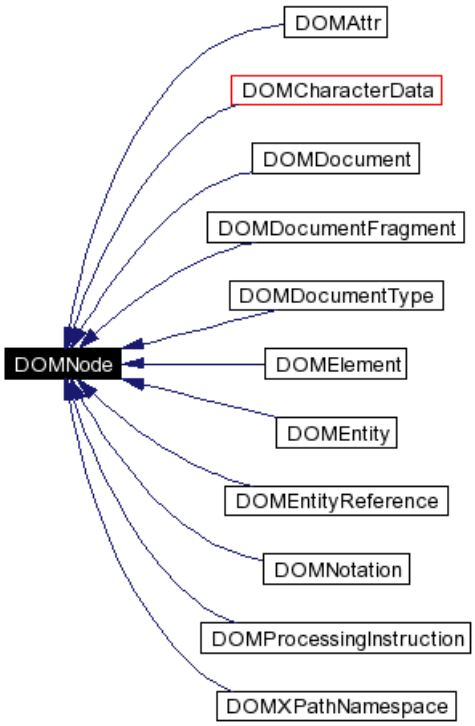

Abbildung 6.1: Vererbungsdiagramm für DOMNode.

6.3.1 Methoden in DOMNode

Knotentyp

Ein DOMknoten kann zunächst nach seinen Typ gefragt werden. Hierzu ist in einer Aufzählung definiert, welche Knotentypen unterschieden werden.

```
1 virtual short getNodeType ()
```

Das Ergebnis ist eines der Werte der folgenden Aufzählung:

```
1 enum NodeType
2  { ELEMENT_NODE                = 1
```

```
3   , ATTRIBUTE_NODE              = 2
4   , TEXT_NODE                   = 3
5   , CDATA_SECTION_NODE          = 4
6   , ENTITY_REFERENCE_NODE       = 5
7   , ENTITY_NODE                 = 6
8   , PROCESSING_INSTRUCTION_NODE = 7
9   , COMMENT_NODE                = 8
10  , DOCUMENT_NODE               = 9
11  , DOCUMENT_TYPE_NODE          = 10
12  , DOCUMENT_FRAGMENT_NODE      = 11
13  , NOTATION_NODE               = 12
14  }
```

Möchte man zum Beispiel testen, ob es sich bei einem vorliegenden Knoten `n` um einen Elementknoten handelt, so ist das mit folgendem Code möglich:

```
1   n->getNodeType() == DOMNode::ELEMENT_NODE
```

Knotenname

Ein Knoten kann nach seinem Namen mit der folgenden Methode gefragt werden:

```
1   virtual const XMLCh* getNodeName ()
```

Dabei geben Elementknoten den Tagnamen zurück, Attributknoten den Namen des Attributs zurück. Die meisten anderen Knoten geben keinen interessanten Wert zurück. So geben Textknoten z.B. nur den String `"text"` zurück.

Knotenwert

Ebenso wie einen Namen haben Knoten einen Wert, der erfragt werden kann.

```
1   virtual const XMLCh * getNodeValue ()
```

Für die meisten Knotentypen ist dieser Wert `null`. Nur für Textknoten ist es der eigentliche Inhalt des Textes, und bei Attributknoten der Wert des Attributs.

Elternknoten

Mit der Methode

```
1   virtual DOMNode* getParentNode ()
```

läßt sich jeder Knoten nach seinen Elternknoten fragen.

Kinderknoten

Zum Navigieren zu den Kindern des Knotens gibt es die Methode:

```
1 virtual DOMNodeList* getChildNodes()
```

Man bekommt ein spezielles Listenobjekt mit den Kinderreferenzen zurück. Die Klasse `DOMNodeList` kent genau zwei Methoden, eine zum erfragen der Länge und eine zum selektieren eines bestimmten Elements:

```
1 virtual DOMNode* item (XMLSize_t index);
2 virtual XMLSize_t getLength ();
```

Attribute

Die Attribute stehen außerhalb der eigentlichen Baumstruktur und stehen nicht in der Liste der Kinderknoten. Die Attribute eines Knotens lassen sich gesondert erfragen mit der Methode:

```
1 virtual DOMNamedNodeMap* getAttributes()
```

Sie wichtigste methode der Klasse `DOMNamedNodeMap` gibt den Wert zu einem bestimmten Attributnamen zurück.

```
1 virtual DOMNode* DOMNamedNodeMap::getNamedItem (const XMLCh* name)
```

Wenn kein Attribut mit diesem Namen existiert, wird `null` zurückgegeben.

6.3.2 Strings in Xerces

Auch beim Arbeiten mit Xerces werden wir mit der unangenehmen Tatsache konfrontiert, daß sich in C++ noch keine Stringklasse durchgesetzt hat, die unicodefähig ist. Daher kommt auch diese Bibkliothek mit einer eigenen Lösung für Unicode-Strings nämlich als `XMLCh*`. Wir wollen uns in dieser Vorlesung mit diesem Problem nicht weiter belasten und bieten eine einfache KOnversion von den Xerces-Strings zu standard Stringobjekten an und verzichten damit auf die Unicodezeichen:

ChToString.h

```
1  #ifndef CHTOSTRING__H
2  #define CHTOSTRING__H
3  #include <string>
4  #include <xercesc/dom/DOM.hpp>
5
6  XERCES_CPP_NAMESPACE_USE
7  std::string xmlCh2String(const XMLCh* in);
8  XMLCh* toxmlstr(const char* str );
9  XMLCh* toxmlstr(std::string str );
10 #endif
```

Hier die Implementierung, die nicht weiter hinterfragt werden soll.

ChToString.cpp

```
#include "ChToString.h"

std::string xmlCh2String(const XMLCh* inCH) {
  std::string result;
  while (*inCH!='\0'){result=result+ ((char)(*inCH));inCH++;}
  return result;
}

XMLCh* toxmlstr(const char* str ) {
  int length=0;
  while (str[length]!='\0') length++;
  XMLCh* xstr = new XMLCh[length-1];
  for (int i=0;i<length;i++) xstr[i]=str[i];
  xstr[length]='\0';
  return xstr;
}
XMLCh* toxmlstr(std::string str ){
  const char* cs = str.c_str();
  return toxmlstr(cs);
}
```

6.3.3 Aufbau des DOM-Objekts mit Xerces

Im vorangegangenen Abschnitt hjaben wir die Basisfunktionalität für das Navigieren durch XML-Bäume vorgestellt. Jetzt brauchen wir nur noch Baumobjekte, die wir bearbeiten werden. Xerces bietet einen Parser an, der Baumobjekte aus einer XML-Datei einlesen kann. Wir bieten hier eine einfach zu benutzende Funktion an, die diesen Parser startet.

ParseXML.h

```
#ifndef PARS_XML__H
#define PARS_XML__H

#include <xercesc/dom/DOM.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <string>
#include <iostream>

XERCES_CPP_NAMESPACE_USE
DOMNode* parseXML(std::string fileName);
#endif
```

Ein Dateiname wird übergeben und ein Zeiger auf den DOM-Knoten des in der Datei enthaltenen XML Dokuments. Auch diese Implementierung wollen wir nicht im Detail hinterfragen.

ParseXML.cpp

```
#include "ParseXML.h"

```

```
3  DOMNode* parseXML(std::string fileName){
4    try{
5      XMLPlatformUtils::Initialize();
6      static const XMLCh gLS[] = { chLatin_L, chLatin_S, chNull };
7      DOMImplementation *impl
8        = DOMImplementationRegistry::getDOMImplementation(gLS);
9      DOMBuilder *parser
10       = impl->createDOMBuilder(DOMImplementationLS::MODE_SYNCHRONOUS,0);
11     const char* fn=fileName.c_str();
12     DOMDocument* doc = parser->parseURI(fn);
13     XMLPlatformUtils::Terminate();
14     return doc->getDocumentElement();
15   }catch(...){std::cerr<<"error occured"<<std::endl;}
16 }
```

6.3.4 Beispiele

Jetzt kennen wir die wichtigsten Funktionen, um XML-Dateien zu verarbeiten. Fangen wir zunächst mit etwas einfachen an. Wir schreiben eine Funktion, die die Anzahl der Knoten in einem XML-Baum zählt:

Zählen der Knoten im Baum

CountElems.h

```
1  #ifndef COUNT_ELEMS__H
2  #define COUNT_ELEMS__H
3  #include "ParseXML.h"
4  int countChildElements(DOMNode* n);
5  #endif
```

Die Funktion läßt sich rekursiv realisieren. Für jeden Knoten setze wir das Ergebnis zunächst auf 1, nämlich für diesen Knoten. Dann Iterieren wir durch die Kinderknoten, sofern es sich um einen Elementknoten handelt, der Kinder hat, und rufen rekursiv die Funktion für jedes Kind auf. Die Ergebnisse der rekursiven Aufrufe werden zum Gesamtergebnis hinzuaddiert:

CountElems.cpp

```
1  #include <xercesc/dom/DOM.hpp>
2  #include <iostream>
3  #include "CountElems.h"
4  #include "ChToString.h"
5  int countChildElements(DOMNode* n){
6    int result = 1;
7    if (n->getNodeType() == DOMNode::ELEMENT_NODE){
8      const  XMLCh* name = n->getNodeName();
9      std::cout<< xmlCh2String(name)<<std::endl;
10
11    DOMNodeList* nl = n->getChildNodes();
12     for (int i=0;i<nl->getLength();i++)
13       result = result+countChildElements(nl->item(i));
```

```
14    }
15    return result;
16  }
```

Wir schreiben ein Hauptprogramm, das als Kommandozeilenparameter den Namen der zu bearbeitenden XML-Datei bekommt.

cxmlelems.cpp

```
1  #include "CountElems.h"
2
3  int main(int argc, char* argv[]){
4    std::cout<<countChildElements(parseXML(argv[1]))<<std::endl;
5  }
```

Sammeln aller Tagnamen

Im folgenden Beispiel soll die menge aller unterschiedlichen Tagnamen eines Dokuments errechnet werden.

GetTags.h

```
1  #ifndef GET_TAGS__H
2  #define GET_TAGS__H
3  #include <xercesc/dom/DOM.hpp>
4  #include <iostream>
5  #include "ChToString.h"
6  #include <set>
7
8  std::set<std::string> getTags(DOMNode* n);
9  #endif
10
```

Die Vorgehensweise ist wieder sehr ähnlich zum obigen Beispiel. Eine Ergebnisvariable wird angelegt, diesmal in Form einer Menge. Der Tagname eines Elementknotens wird dieser hinzugefügt. Anschließend folgt die Iteration über die Kinder und der rekursive Aufruf auf die Kinder. Schließlich werden die Teilergebnisse dem Gesamtergebnis zugefügt.

GetTags.cpp

```
1  #include "GetTags.h"
2
3  std::set<std::string> getTags(DOMNode* n){
4    std::set<std::string> result;
5    if (n->getNodeType() == DOMNode::ELEMENT_NODE){
6      result.insert(xmlCh2String(n->getNodeName()));
7      DOMNodeList* nl = n->getChildNodes();
8      for (int i=0;i<nl->getLength();i++){
9        std::set<std::string> childResult = getTags(nl->item(i));
10       result.insert(childResult.begin(),childResult.end());
11     }
12   }
```

```
13    return result;
14  }
```

Auch zu dieser Funktion schreiben wir eine kleine Anwendung, die uns alle Tagnamen eines XML-Dokuments auf der Kommandozeile ausgibt.

getxmltags.cpp

```
1  #include "GetTags.h"
2  #include "ParseXML.h"
3  #include <vector>
4  #include <iostream>
5
6  void print(std::string p){std::cout << p << ",";}
7
8  template <typename Iterator>
9  void printSeq(Iterator begin,Iterator end){
10     if (begin==end){std::cout << "[]" << std::endl;return;}
11   std::cout << "[";
12   for_each(begin,end-1,print);
13   std::cout << *(end-1) << "]"<<std::endl;
14 }
15
16 int main(int argc, char* argv[]){
17   std::set<std::string> result= getTags(parseXML(argv[1]));
18   std::vector<std::string> toPrint(result.begin(),result.end());
19
20   printSeq(toPrint.begin(),toPrint.end());
21 }
```

Maximale Tiefe im Baum

In der folgenden Funktion wollen wir die maximale Tiefe eines XML-Baumes berechnen.

MaxDepth.h

```
1  #ifndef MAX_DEPTH__H
2  #define MAX_DEPTH__H
3  #include "ParseXML.h"
4  int maxDepth(DOMNode* n);
5  #endif
```

Auch hier ist die vorgehensweise analog wie oben. Jedes Kind wird nach seiner maximalen Tiefe gefragt. Ist diese höher als die bisher gefundene, wird sie als aktuelle maximale Tiefe genommen. Schließlich wird zum Ergebnis noch eins hinzuaddiert, nämlich für die Kante vom aktuellen Knoten zu seinem Kind mit der maximalen Tiefe.

MaxDepth.cpp

```
1  #include <xercesc/dom/DOM.hpp>
2  #include <iostream>
3  #include "MaxDepth.h"
```

```
4
5  int maxDepth(DOMNode* n){
6    int result = 0;
7    if (n->getNodeType() == DOMNode::ELEMENT_NODE){
8     DOMNodeList* nl = n->getChildNodes();
9      for (int i=0;i<nl->getLength();i++){
10       int childDepth =maxDepth(nl->item(i));
11       result = childDepth>result?childDepth:result;
12     }
13   }
14   return result+1;
15 }
```

Auch dieses Programm können wir über die Kommandozeile testen.

xmldepth.cpp

```
1  #include "MaxDepth.h"
2
3  int main(int argc, char* argv[]){
4    std::cout<<maxDepth(parseXML(argv[1]))<<std::endl;
5  }
```

Extraktion aller Attributwerte für einen Namen

Bisher haben wir in keinem Beispiel die Attribute der Elemente betrachtet. In diesem Beispiel sollen zu einem Attributnamen alle Werte dieses Attributs, die in einem Dokument zu finden sind, etrahiert werden. Hierzu wird der Wurzelknoten des Dokuments übergeben, die Ergebnismenge, und der Attributname, nach dem gesucht wird.

Für jeden Elementknoten werden die Attribute geholt. Es wird in diesem nach dem gesucten Attribut gefragt. Wenn dieses nicht `NULL` ist, so wird dessen Wert der Ergebnismenge hinzugefügt. Anschließend wird in den Kinderknoten weitergesucht.

ExtractAttribute.cpp

```
1  #include <fstream>
2  #include <xercesc/dom/DOM.hpp>
3  #include "ParseXML.h"
4  #include "ChToString.h"
5  #include <set>
6  #include <vector>
7
8  using namespace std;
9
10 void getAttr(DOMNode* n,set<string>& result,XMLCh* attrName){
11   if (n->getNodeType() == DOMNode::ELEMENT_NODE){
12     const DOMNamedNodeMap* attrs=n->getAttributes();
13     const DOMNode* attrNNode = attrs->getNamedItem(attrName);
14
15     if (attrNNode!=NULL)
16       result.insert(xmlCh2String(attrNNode->getNodeValue()));
```

```
17
18      const DOMNodeList* nl = n->getChildNodes();
19      for (unsigned int i=0;i<nl->getLength();i++){
20        getAttr(nl->item(i),result,attrName);
21      }
22    }
23  }
```

Die Funktion machen wir in einer Hauptmethode von der Kommandozeile aus benutzbar:

ExtractAttribute.cpp

```
24  void print(std::string p){std::cout << p << ",";}
25
26  template <typename Iterator>
27  void printSeq(Iterator begin,Iterator end){
28      if (begin==end){std::cout << "[]" << std::endl;return;}
29    std::cout << "[";
30    for_each(begin,end-1,print);
31    std::cout << *(end-1) << "]"<<std::endl;
32  }
33
34  int main(int argc, char* args[]){
35    set<string> result;
36    getAttr(parseXML(args[1]),result,toxmlstr(args[2]));
37    std::vector<std::string> toPrint(result.begin(),result.end());
38    printSeq(toPrint.begin(),toPrint.end());
39  }
```

Jetzt lassen sich z.B. alle Programmnamen aus dem Quelltext dieses Skriptes extrahieren:

```
sep@pc305-3:~/fh/cpp/student> bin/ExtractAttribute ../student.xml class
[AllTogether,Apfel,ApplyFun,Ball,Box,BoxInt,BoxString,ButtonFunction,ButtonLogic,ButtonLogicAbstr,
ButtonLogicAbstrError,ButtonLogicAbstrOK,ButtonsAndLabels,CIntAssign,CPIntAssign,CPerson,Calculato
r,CalculatorGUI,CalculatorModell,ChToString,Closure,Complex,Cons,Count2,CountElems,CubeWorld,Curry
,DateiTest,DefaultParameter,DeleteChildren,Deleted,Deleteted,Destruction,Dialogue,DigitalClock,Dis
playApfel,DisplayCalculator,DisplayKeyApfel,DisplayMouseApfel,DoxygenMe,Empty,ErsteFelder,Exceptio
nTest,ExplicitTypeParameter,ExtractAttribute,FDialogue,FirstDraw,Fold,FoldTest,ForEachArray,ForEac
hVector,Function,Function2,GameCanvas,GenMap,GetTags,Graph,Heirat,HelloGTKmm,HelloLineLoop,HelloLi
neStrip,HelloLines,HelloPoints,HelloPolygon,HelloQuadStrip,HelloQuads,HelloSphere,HelloTriangleFan
,HelloTriangleStrip,HelloTriangles,HelloWindow,HelloWorld,InitRef,InputIterator,IntBox,JIntBoxAssi
gn,KeyApfel,L1,Lambda,Layout,LazyList,Li,List,LocalVarPointer,Makefile,MapTest,MaxDepth,Minimal,Mo
difyOrg,ModifyRef,MouseApfel,Movable,MultipleParents,NamespaceTest,News,NoInitRef,OverloadedInstea
dOfDefaults,OverloadedTrace,P2,P3,Paddle,Pair,Pallindrom,ParseXML,Person1,Person2,PersonExample1,P
ersonExample2,Ping,PlayPing,PointerLess,PointerLessError,ReadRoman,RefArg,RefToRef,RenderShape,STL
Curry,STLFold,STLFoldTest,ShowTree,SimpleTimer,SolidCone,SolidCube,SolidDodecahedron,SolidIcosahed
ron,SolidOctahedron,SolidSphere,SolidTeapot,SolidTetrahedron,SolidTorus,SomePoints,SortTest,StartD
igitalClock,StartSimpleTimer,Strichkreis,StringAppend,StringIndex,StringLi,StringLiTest,StringUtil
Method,Student,StudentError1,StudentError2,StudentOhneVererbung,T,TestButtonsAndLabels,TestCPerson
,TestClosure,TestComplex,TestCurry,TestDialogue,TestFDialogue,TestFirstLi,TestFirstList,TestGenBox
,TestLambda,TestLateBinding,TestLayout,TestLazyList,TestLength,TestLi,TestListLength,TestNoLateBin
ding,TestP2,TestPair,TestReadRoman,TestSimpleClosure,TestStudent,TestStudent1,TestStudent2,TestUni
Pair,ToString,Trace,TraceWrongUse,TransformTest,Tux,Undeleteted,UniPair,UseT,UseThis,UseThis2,Vate
rMuterKind,VaterMuterKindError,VectorTest,WireCone,WireCube,WireDodecahedron,WireIcosahedron,WireO
ctahedron,WireSphere,WireTeapot,WireTetrahedron,WireTorus,WrongCall,WrongDelete,WrongInitRef,Wrong
```

```
PointerArith,XMLTree,Zeiger1,Zeiger2,Zeiger3,Zeiger4,cxmlelems,getxmltags,makeTheHelloWorld,testFr
omRoman,testToDual,xmldepth]
sep@pc305-3:~/fh/cpp/tutor>
```

Aufgabe 19 (6 Punkte) Schreiben Sie ein Program `getcode` mit drei Kommandozeilenparametern:

```
1 getcode xmlFileName className langName
```

Es soll die XML-Datei `xmlFileName` lesen und dafür ein DOM-Objekt erzeugen. Dann soll es von jedem Element mit Tagnamen `code`, das für das Attribut `class` den Wert `className` und für das Attribut `lang` den Wert `langName` hat, den in diesem Element enthaltenen Text extrahieren und in eine Datei `className.langName` schreiben.

Als Beispieldokument befindet sich der Quelltext dieses Skripts (`http://www.panitz.name/cpp/student.xml`) auf der Webseite dieser Vorlesung.

Der Aufruf `getcode student.xml Li h` soll also die Datei `Li.h` schreiben mit den Code, der hierzu in `student.xml` zu finden ist.

Als Beispiel, wie eine Textdatei als Datenstrom geschrieben werden kann, betrachten Sie folgendes kleine Programm:

DateiTest.cpp

```
1 #include <fstream>
2
3 int main(){
4     std::fstream datei;
5     datei.open("testdatei",std::ios::out);
6     datei << "etwas text"<<std::endl;
7     datei << "in die datei schreiben"<<std::endl;
8     datei.close();
9 }
```

Kapitel 7

Gtkmm: graphische Oberflächen in C++

In diesem Kapitel sollen unsere frischen Erkenntnisse in C++ an einer GUI-Bibliothek vertieft werden. 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.

Gtkmm ist nur eine von viele Umsetzungen der Bibliothek Gtk+ für eine andere Programmiersprache. Es gibt fast für alle wichtigen Sprachen Portierungen dieser Bibliothek, darunter: Java, Perl, Python, Haskell, Ruby, Ada, C#, Common Lisp, Eiffel, Erlang, Pascal, PHP, Scheme, Smalltalk.

7.1 Widgets

Eine Graphikanwendung besteht aus graphischen Komponenten. Diese werde als *widget* bezeichnt. *widget* ist dabei ein Kunstwort, das aus der Bezeichnung *window gadget* abgeleitet wurde. Graphische Komponenten können dreierlei Art sein:

- mehr oder weniger atomare sichtbare Objekte, wie Knöpfe, Textfelder, Menüs.
- Sichbare top-level Komponenten, die weitere Komponenten enthalten, wie ein Fenster, das mehrere Knöpfe, Bilder, Statusleisten usw. als Unterkomponenten enthält.

- Zwischenkomponenten zum Gruppieren einzelner Komponenten. Diese Zwischenkomponenten haben keine eigene sichtbare Ausprägung. Typische solche Komponenten sind Boxen, die Unterkomponenten vertikal oder horizontal ausrichten.

Entsprechend gibt es in Gtkmm eine Oberklasse `Gtk::Widget` von der mehrere Unterklassen wie `Gtk::Button`, `Gtk::Label`, `Gtk::HBox`, `Gtk::VBox`, `Gtk::Menu` und auch `Gtk::Window` ableiten.

7.1.1 Hello Gtkmm

Für eine Gtkmm-Anwendung ist die entsprechende Bibliothek zu inkludieren. Am einfachsten kann man sie komplett inkludieren mit `#include <gtkmm.h>`. Allerdings sollte man in der Praxis dedizierter die einzelnen benötigten Teilbibliotheken inkludieren, z.B. `<gtkmm/button.h>`, wenn man Knöpfe benötigt.

Um mit der Bibliothek arbeiten zu können, muß ein Initialisierungsaufruf gemacht werden, sie quasi gestartet werden. Hierzu kann man der Klasse `Gtkmm::Main` de Kommandozeilenparameter übergeben. Eine einfache Gtkmm-Applikation beginnt demnach mit:

HelloGTKmm.cpp

```
1  #include <gtkmm.h>
2
3  using namespace Gtk;
4
5  int main(int argc,char** args){
6    Main kit(argc,args);
```

Jetzt lassen sich die einzelnen graphischen Komponenten, die für die Anwendung benötigt werden, definieren:

HelloGTKmm.cpp

```
7    Window window;
8    VBox vbox;
9    HBox hbox;
10   Button knopf1("Knopf 1");
11   Button knopf2("Knopf 2");
12   Label label1("hier steht Text drin");
13   Label label2("hier steht anderer Text drin");
```

Schließlich lassen sich die einzelnen Komponenten ineinanderstecken. Die Methode `add` fügt zu einem Widget ein anderes Widget als Kind hinzu:

HelloGTKmm.cpp

```
1    vbox.add(label1);
2    hbox.add(knopf1);
3    hbox.add(knopf2);
4    vbox.add(hbox);
5    vbox.add(label2);
6    window.add(vbox);
```

Wenn dergestalt das GUI definiert ist, kann es sichtbar gemacht werden und die Anwendung gestartet werden.

HelloGTKmm.cpp

```
1   window.show_all();
2
3   Main::run(window);
4 }
```

Damit ist die erste graphische Anwendung, wenn auch eine funktionslose, implementiert. Das Programm erzeugt das in Abbildung 7.1 angegebene Fenster.

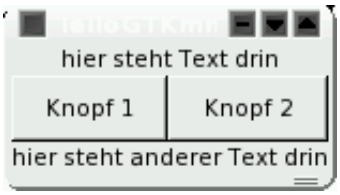

Abbildung 7.1: Erstes mit Gtkmm erzeugtes Fenster.

Beim Übersetzen von Gtkmm-Anwendungen sind eine ganze Menge von Pfaden und zu ladenen Bibliotheken anzugeben. Die Arbeit, dieses von Hand zu machen, nimmt einen das *tool* `pkg-config` ab. Geben Sie hierzu einmal die Befehle `pkg-config gtkmm-2.4 --cflags` und `pkg-config gtkmm-2.4 --libs` auf der Kommandozeile ein:

```
sep@pc216-5:~/fh/cpp> pkg-config gtkmm-2.4 --cflags
-I/usr/local/include/gtkmm-2.4 -I/usr/local/lib/gtkmm-2.4/include -I/usr/
local/include/glibmm-2.4 -I/usr/local/lib/glibmm-2.4/include -I/usr/local
/include/gdkmm-2.4 -I/usr/local/lib/gdkmm-2.4/include -I/usr/local/includ
e/pangomm-1.4 -I/usr/local/include/atkmm-1.6 -I/usr/local/include/sigc++-
2.0 -I/usr/local/lib/sigc++-2.0/include -I/usr/local/include/glib-2.0 -I/
usr/local/lib/glib-2.0/include -I/usr/local/include/pango-1.0 -I/usr/X11R
6/include -I/usr/include/freetype2 -I/usr/include/freetype2/config -I/usr
/local/include/atk-1.0 -I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include
sep@pc216-5:~/fh/cpp> pkg-config gtkmm-2.4 --libs
-L/usr/local/lib -lgtkmm-2.4 -lgdkmm-2.4 -latkmm-1.6 -lgtk-x11-2.0 -lpango
mm-1.4 -lglibmm-2.4 -lsigc-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -l
m -lpangoxft-1.0 -lpangox-1.0 -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -ldl
 -lglib-2.0
sep@pc216-5:~/fh/cpp>
```

Sie erhalten die Parameter, die zum Übersetzen bzw. Linken eines Gtkmm-Programms nötig sind. Den `pkg-config` Befehl können sie im Aufruf des Compilers integrieren. Damit läßt sich unser obiges Gtkmm-Programm übersetzen mit:

```
sep@pc216-5:~> g++ -o HelloGtkmm ‘pkg-config gtkmm-2.4 --cflags --libs‘ HelloGTKmm.cpp
sep@pc216-5:~>
```

7.1.2 Eigene Widgets definieren

Eine schöne Eigenschaft der Gtkmm Bibliothek ist, daß von den Klassen der Bibliothek abgelitten werden kann, um eigene spezialisierte *Widgets* zu erzeugen, die dann ganz genauso wie die vordefinierten *Widgets* der Bibliothek benutzt werden können.

So läßt sich eine eigene Widgetklasse definieren, die zwei Knöpfe und zwei Label in einem Widget vereint. Hierzu leiten wir von der Klasse `VBox` ab.

ButtonsAndLabels.h

```
1  #include <gtkmm.h>
2  #include <string>
3
4  using namespace Gtk;
5  using namespace std;
6
7  class ButtonsAndLabels : public VBox{
8   protected:
9    Label label1,label2;
10   Button knopf1,knopf2;
11   HBox hbox;
12
13  public:
14   ButtonsAndLabels(string l1,string l2,string b1,string b2);
15 };
```

In der Implementierung des Konstruktors lassen sich alle Unterkomponenten initialisieren:

ButtonsAndLabels.cpp

```
1  #include "ButtonsAndLabels.h"
2
3  ButtonsAndLabels::ButtonsAndLabels
4                     (string l1,string l2,string b1,string b2):
5     label1(l1),label2(l2),knopf1(b1),knopf2(b2){
6    add(label1);
7    hbox.add(knopf1);
8    hbox.add(knopf2);
9    add(hbox);
10   add(label2);
11 }
```

Ebenso können wir eine Unterklasse der Klasse `Window` definieren. Dieses Fenster soll ein widget der obigen Klasse `ButtonsAndLabels` enthalten.

TestButtonsAndLabels.cpp

```
1  #include "ButtonsAndLabels.h"
2  class MyWindow:public Window{
3   private:
4    ButtonsAndLabels content;
5
6   public:
```

```
7   MyWindow(string l1,string l2,string b1,string b2)
8    :content(l1,l2,b1,b2){
9     add(content);
10    show_all();
11  }
12 };
```

Unsere spezialisierte Fensterkomponente mit den zwei Knöpfen und zwei Labels läßt sich wie ein allgemeines Fenster erzeugen und sichtbar machen.

TestButtonsAndLabels.cpp

```
1 int main(int argc,char** args){
2   Main kit(argc,args);
3   MyWindow window("hallo","welt","knopf","button");
4   Main::run(window);
5 }
6
```

Das Programm erzeugt das in Abbildung 7.2 angegebene Fenster.

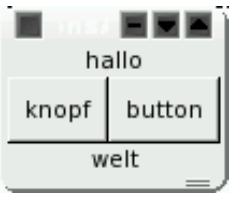

Abbildung 7.2: Gtkmm Fenster mit Labeln und Knöpfen .

7.2 Funktionalität der Komponenten

7.2.1 Ereignisbehandlung

Als nächstes wollen wir den graphischen Komponenten Funktionalität geben. Dieses wird in Gtkmm ereignisbasiert definiert. Dazu benutzt Gtkmm die Bibliothek *libsigc++*. Bestimmte graphische Komponenten haben bestimmte Ereignisse, die für sie auftreten können. Das einfachste und plastischste ist sicher das Drücken eines Knopfobjektes. Hierzu gibt es in der Klasse `Gtk::Button` die Methode `signal_clicked( )`, die ein Objekt des Typs `Glib::SignalProxy0<void>` zurückgibt. Diesem Objekt lassen sich nun Funktionen anheften, die ausgeführt werden sollen, wenn das Ereignis, das der Knopf gedrückt wird, eingetreten ist. Dazu enthält das Object die Funktion `connect`. Es gibt zwei Arten von Funktionen, die als Funktionalität zugefügt werden können:

- statische Funktionen: hierzu kapselt man den Funktionspointer mit `sigc::ptr_fun` in ein Objekt, das der Methode `connect` übergeben wird.
- Objektmethoden: hierzu erzeugt man mit der Funktion `sigc::mem_fun` aus dem Objekt und der Objektmethode die ausgeführt ist, ein Funktionsobjekt, das der Methode `connect` übergeben wird.

Betrachten wir ein simples Beispiel für einen Knopf, der mit zwei Funktionen darauf reagieren soll, wenn er gedrückt wird:

ButtonFunction.cpp

```
1  #include <gtkmm.h>
2  #include <iostream>
3  #include "ToString.h"
4
5  using namespace std;
6  using namespace Gtk;
7  using namespace sigc;
```

Zunächst schreiben wir die Funktion, die ausgeführt werden soll, wenn der Knopf gedrückt wird. Unsere Funktion soll davon einfach auf der Konsole Kenntnis geben.

ButtonFunction.cpp

```
8  void doWhenClicked(){
9   cout<<"Knopf wurde gedrückt!"<<endl;
10 }
```

Eine Klasse sei definiert, in der es einen Knopf und ein Label gibt, sowie eine interne Zahl:

ButtonFunction.cpp

```
11 class CountButton:public VBox{
12  private:
13   int i;
14   Label l;
15   Button b;
```

In dieser Klasse gibt es eine Objektmethode, in der der Zähler um eins erhöht und das Label mit dem aktuellen Wert des Zählers gesetzt wird:

ButtonFunction.cpp

```
16   void doWhenClickedToo(){
17     i=i+1;
18     l.set_text(itos(i));
19   }
```

Im Konstruktor werden zunächst die einzelnen Komponenten initialisiert und der Hauptkomponente hinzugefügt:

ButtonFunction.cpp

```
20  public:
21   CountButton():i(0),b("drücken"),l(itos(i)){
22     add(b);
23     add(l);
```

Schließlich fügen wir den Knopf Funktionalität hinzu, wenn er gedrückt wird. Hierzu benutzen wir die `connect` Methode auf dem durch `signal_clicked` erzeugten Objekt.

ButtonFunction.cpp

```
24      b.signal_clicked()
25       .connect(<redv>ptr_fun(&doWhenClicked)</redv>);
26      b.signal_clicked()
27       .connect(<redv>mem_fun(*this,&CountButton::doWhenClickedToo)</redv>);
28    };
29  };
```

Wie man sieht lassen sich durchaus mehrere Funktionen als Reaktion auf ein Ereignis an ein graphisches Objekt binden.

Die Hauptmethode öffne ein Fenster mit einem Objekt des Knopfes zum Zählen:

ButtonFunction.cpp

```
30  int main(int argc,char** args){
31    Main kit(argc,args);
32    Window w;
33    CountButton cb;
34    w.add(cb);
35    w.show_all();
36    Main::run(w);
37  }
38
```

Typen zu Ereignisbehandlung

Im letzten Beispiel haben wir dem Ereignis `signal_clicked` zwei nullstellige Funktionen mit der Rückgabetyp `void` angeheftet. Woher wußten wir, daß hier solche Funktionen erwartet waren. Hätten wir auch andere Funktionen anheften können? Die Antwort liegt im Typ der Ereignisbehandlung, die die Funktion `signal_clicked` zurückgibt, dieses war der Typ: `Glib::SignalProxy0<void>`. Er sagt aus, daß zur Ereignisbehandlung nullstellige Funktionen mit `void` als Rückgabetyp erwartet werden.

7.2.2 Überschreiben von Ereignisfunktionen

Im letzten Abschnitt haben wir Gui-Objekten mit Hilfe der Methode `connect` Ereignissbehandlungsmethoden hinzugefügt. Es gibt eine im Sinne der Objektorientierung etwas naheliegendere Art, wie man das Verhalten einer GUI-Komponente programmieren kann. Hierzu befinden sich in der gemeinsamen Oberklasse `Widget` eine Vielzahl von virtuellen Methoden, die sich auf ein bestimmtes Ereignis beziehen und ausgeführt werde, wenn immer dieses Ereignis auftritt. Wenn man die entsprechende Methode in seiner eigenen Widgetklasse überschreibt, so läßt sich das Verhalten für ein bestimmtes Ereigniss bestimmen.

Betrachten wir auch hierzu ein Beispiel. Wir definieren eine Unterklasse der Klasse `Button`, in der die Methode `on_clicked()` überschrieben ist.

Count2.cpp

```
1  #include <gtkmm.h>
2  #include "ToString.h"
3  using namespace Gtk;
```

```
4
5  class Count2: public VBox{
6   public:
7    Count2();
8   protected:
9    class MyButton:public Button{
10    public:
11     MyButton(char* s,Label& l);
12     virtual void on_clicked();
13     int i;
14     Label& l;
15   };
16   MyButton b;
17   Label l;
18 };
```

Der Abwechslung halber ist die Klasse `MyButton` innerhalb der Klasse `Count2` definiert. Ebenso wie in C Strukturen lokal definiert werden können, können auch Klassen in C++ lokal innerhalb anderer Klassen definiert werden.

Für beide Klassen wird der entsprechende Konstruktor implementiert:

Count2.cpp

```
19 Count2::Count2():l("0"),b("click",l){add(l);add(b);}
20
21 Count2::MyButton::MyButton(char* s,Label& l):Button(s),i(0),l(l){}
```

Schließlich überschreiben wir die Methode `on_clicked` für unseren spezialisierten Knoten.

Count2.cpp

```
22 void Count2::MyButton::on_clicked(){
23  i=i+1;l.set_text(itos(i));}
```

In der Hauptmethode läßt sich dieses Widget innerhalb eines Fensters öffnen:

Count2.cpp

```
24 int main(int argc,char** args){
25   Main kit(argc,args);
26   Window w;
27   Count2 c;
28   w.add(c);
29   w.show_all();
30   Main::run(w);
31 }
```

Somit lassen sich hier die beiden Möglichkeiten, für eine Oberfläche Funktionalität zu definieren, einmal durch Übergabe eines Funktionszeigers und einmal über das Überschreiben einer Funktion in der Unterklasse, auch in der Bibliothek Gtkmm anwenden.

7.3 Eigene Widgets zeichnen

Widgets haben eine graphische Repräsentation. Diese wird durch eine Methode bestimmt, die die Gtkmm-Umgebung aufruft, sobald das Widget wieder zu zeichnen ist, z.B. weil es kurzzeitig von einem anderen Fenster verdeckt war, nun aber wieder sichtbar geworden ist. In diesem Fall ruft die Gtkmm-Umgebung die Methode `on_expose_event` des Widgets auf. Die Widgets der Gtkmm-Bibliothek implementieren diese Methode, so daß ihre graphische Darstellung auf dem Bildschirm gezeichnet wird. Wir werden aber natürlich nicht nur auf die fertigen Widgets der Gtkmm-Bibliothek zurückgreifen wollen, sondern eigene Widgets zeichnen wollen. Hierzu kann man eigene Widgetsklassen definieren, in denen die Methode `on_expose_event` nach unseren Wünschen überschrieben ist. Hierzu bietet Gtkmm das Widget `DrawingArea` an, das selber noch keine Implementierung der Methode `on_expose_event` hat.

Neben der Methode `on_expose_event` ist auch noch die Methode `on_realize` zu überschreiben. Diese wird ausgeführt, wenn das Widget zum ersten Mal sichtbar erzeugt wird. Hier lassen sich allgemeine Initialisierungen durchführen, wie das Setzen der Hintergrundfarbe.

Ein einfaches Widget, das seine eigene Zeichenfunktion definiert, hat entsprechend folgende Kopfdatei:

FirstDraw.h

```
1 #include <gtkmm.h>
2 class FirstDraw:public  Gtk::DrawingArea{
3  protected:
4   virtual void on_realize();
5   virtual bool on_expose_event(GdkEventExpose* event);
6 };
```

7.3.1 Der graphische Kontext

Zum Zeichen bietet Gtkmm ein Objekt für den graphischen Kontext an. Dieses Objekt wird damit über ein Referenzerungsobjekt gekapselt angesprochen. Man erhält ein Objekt des Typs `Glib::RefPtr<Gdk::GC>`. Für die Erzeugung eines solchen Objekts steht eine statische Funktion zur Verfügung: `create (const Glib::RefPtr<Drawable>& drawable)`. Diese bekommt als Parameter ein Objekt auf dem gezeichnet werden soll mit. Dieses wiederum läßt sich für ein Widget mit der Methode:

```
1 Glib::RefPtr<const Gdk::Window>         get_window () const
```

efragen. Auf diesem Objekt finden dann auch die eigentlichen Zeichenoperationen statt.

Der graphische Kontext enthält Information darüber, wie Dinge zu zeichnen sind, z.B. in welcher Farbe, oder in welcher Strichbreite Linien zu zeichnen sind.

Wem das zu verwirrend klingt, mag im folgenden Beispiel sehen, wie entsprechend vorzugehen ist. Wir wollen einfach nur ein Paar Linien zeichnen, und erweitern hierzu die Klasse `Gtk::DrawingArea`:

Zeichnen.cpp

```
1 #include <gtkmm.h>
2
3 class Zeichne : public Gtk::DrawingArea{
```

Im Konstruktor dieser Klasse setzen wir zunächst die Größe der Leinwand, auf die wir zeichnen wollen fest:

Zeichnen.cpp

```
4  public:
5   Zeichne(){
6     set_size_request(400, 300);
7   }
```

Für den graphischen Kontext und das eigentliche Fensterobjekt zum Zeichnen sehen wir je ein Feld in der Klasse vor:

Zeichnen.cpp

```
8  protected:
9   Glib::RefPtr<Gdk::GC> gc;
10  Glib::RefPtr<Gdk::Window> window;
```

In der Methode `on_realize` werden diese beiden Felder initialisiert. Zuvor wird allerdings noch die entsprechende Methode der Oberklasse aufgerufen.

Zeichnen.cpp

```
11  virtual void on_realize(){
12    Gtk::DrawingArea::on_realize();
13    window = get_window();
14    gc = Gdk::GC::create(window);
15  }
```

Schließlich gelangen wir zu der eigentlichen Zeichenfunktion `on_expose_event`. In dieser löschen wir zunächst den gesamten Inhalt.

Zeichnen.cpp

```
16  virtual bool on_expose_event(GdkEventExpose* event){
17    window->clear();
```

Anschließend werden die Attribute zum Zeichnen von Linien gesetzt.

Zeichnen.cpp

```
18    gc->set_line_attributes
19         (1, Gdk::LINE_SOLID,Gdk::CAP_ROUND,Gdk::JOIN_MITER);
```

Und schließlich eine Linie im Fenster gezeichnet:

Zeichnen.cpp

```
20    window->draw_line(gc,10,10,390,10);
```

Zum Experimentieren Zeichnen wir noch ein paar weitere Linie mit unterschiedlichen Attributen:

Zeichnen.cpp

```
21      gc->set_line_attributes
22           (1, Gdk::LINE_ON_OFF_DASH,Gdk::CAP_ROUND,Gdk::JOIN_MITER);
23      window->draw_line(gc,10,30,390,30);
24
25      gc->set_line_attributes
26           (10, Gdk::LINE_SOLID,Gdk::CAP_ROUND,Gdk::JOIN_MITER);
27      window->draw_line(gc,10,70,390,70);
28
29      gc->set_line_attributes
30           (20, Gdk::LINE_SOLID,Gdk::CAP_BUTT,Gdk::JOIN_MITER);
31      window->draw_line(gc,10,100,390,100);
32
33      gc->set_line_attributes
34           (40, Gdk::LINE_SOLID,Gdk::CAP_NOT_LAST,Gdk::JOIN_MITER);
35      window->draw_line(gc,10,200,390,200);
36      return true;
37    }
38  };
```

In der Hauptmethode öffnen wir einmal das so gestaltete Fenster:

Zeichnen.cpp

```
39  int main(int argc, char** argv){
40    Gtk::Main main_instance (argc, argv);
41    Gtk::Window window;
42    Gtk::HBox box;
43
44    Zeichne z;
45    window.add(z);
46    window.show_all();
47
48    Gtk::Main::run(window);
49    return 0;
50  }
```

7.3.2 Einfaches Zeichnen

Nachdem im letztem Abschnitt gezeigt wurde, wie eigene graphische Komponenten definiert werden können, die eine eigene graphische Ausprägung haben, soll jetzt in Hinblick als Vorübung auf die nächste Aufgabe, eine Komponente erstellt werden, die einen Kreis zeichnet. Dieser Kreis soll durch Radiusliniengezeichnet werden. Hierzu sind verschiedene Radiuslinien mit unterschiedlichen Steigungen zu rechnen. Man kommt also nicht um ein wenig rudimentäre Kenntnisse der Geometrie aus. Zunächst definieren wir eine Konstante zum Umrechnen von Winkeln in Gradangabe zu Winkeln im Bogenmaß.

Strichkreis.cpp

```
1   #include <gtkmm.h>
2   #include <cmath>
```

```
3
4  const double DEG2RAD = 3.1415928 / 180.0;
```

Wir definieren eine Unterklasse von `DrawingArea`. Diese bekomme den Radius des zu zeichnenden Kreises im Konstruktor übergeben.

Strichkreis.cpp

```
5  class StrichKreis : public Gtk::DrawingArea{
6   public:
7    StrichKreis(int r);
8    ~StrichKreis();
```

Die beiden Methoden, die beim Zeichnen und Erzeugen der Komponente aktiv sind, werden überschrieben.

Strichkreis.cpp

```
9   protected:
10   virtual void on_realize();
11   virtual bool on_expose_event(GdkEventExpose* event);
```

Zusätzlich wird ein Feld für den graphischen Kontext und je ein Feld für die zwei benutzten Farben vorgesehen.

Strichkreis.cpp

```
12   Glib::RefPtr<Gdk::GC> gc;
13   Gdk::Color black, white;
14 };
```

Im Konstruktor werden die Farben initialisiert und in der Farbtafel eingetragen. Der Radius wird benutzt, um die Größe der Komponente festzulegen.

Strichkreis.cpp

```
15 StrichKreis::StrichKreis(int r){
16   Glib::RefPtr<Gdk::Colormap> colormap = get_default_colormap();
17   black = Gdk::Color("black");
18   white = Gdk::Color("white");
19
20   colormap->alloc_color(black);
21   colormap->alloc_color(white);
22   set_size_request(r, r);
23 }
24
25 StrichKreis::~StrichKreis(){}
```

In der Methode `on_realize` wird der graphische Kontext initialisiert und die HIntergrundfarbe auf Schwarz gesetzt.

Strichkreis.cpp

```
26 void StrichKreis::on_realize(){
27   Gtk::DrawingArea::on_realize();
28
29   Glib::RefPtr<Gdk::Window> window = get_window();
30   gc = Gdk::GC::create(window);
31   window->set_background(black);
32 }
```

Schließlich geht is in der Methode `on_expose_event`, um das eigentliche Zeichnen der Radiuslinien. Hierzu lasen wir uns zunächst von der Komponente die Ausmaße geben:

Strichkreis.cpp

```
33 bool StrichKreis::on_expose_event(GdkEventExpose*){
34   Glib::RefPtr<Gdk::Window> window = get_window();
35
36   // window geometry: x, y, width, height, depth
37   int winx, winy, winw, winh, wind;
38   window->get_geometry(winx, winy, winw, winh, wind);
```

Das Minimum von Weite und Höhe wird als der Durchmesser des Kreises genommen:

Strichkreis.cpp

```
39   int radius = winw/2;
40   if (winh<winw) radius=winh/2;
```

Und schließlich werden im Abstand von einem Grad 360 Radiuslinien gezeichne. Diese beginnen stets am Mittelpunkt des Kreises, der in der Mitte des Fensters liegt. Mit den trigeometrischen Funktionen `sin` und `cos` läßt sich für die Radiuslinie aus dem Steigungswinkel der Endpunkt bestimmen, um schließlich die Linie zu zeichnen:

Strichkreis.cpp

```
41   gc->set_foreground(white);
42   window->clear();
43   for (int i = 0; i<=360; i=i+1){
44     int x1=winw/2;
45     int y1=winh/2;
46     int x2 = x1 + (int) (radius * cos(DEG2RAD * i));
47     int y2 = y1 + (int) (radius * sin(DEG2RAD * i));
48     window->draw_line(gc, x1,y1,x2,y2 );
49   }
50   return true;
51 }
```

Die so erstellte Komponente `Strichkreis` kann mehrfach instanziiert und einer `Box` zugefügt werden.

Strichkreis.cpp

```
52 int main(int argc, char** argv){
53   Gtk::Main main_instance (argc, argv);
```

```
54    Gtk::Window window;
55    Gtk::HBox box;
56
57    StrichKreis kreis1(400);
58    StrichKreis kreis2(180);
59    StrichKreis kreis3(50);
60    StrichKreis kreis4(100);
61    StrichKreis kreis5(20);
62    box.add(kreis1);
63    box.add(kreis2);
64    box.add(kreis3);
65    box.add(kreis4);
66    box.add(kreis5);
67
68    window.add(box);
69    window.show_all();
70
71    Gtk::Main::run(window);
72    return 0;
73  }
```

Wir erhalten das in Abbildung 7.3 gezeigte Fenster.

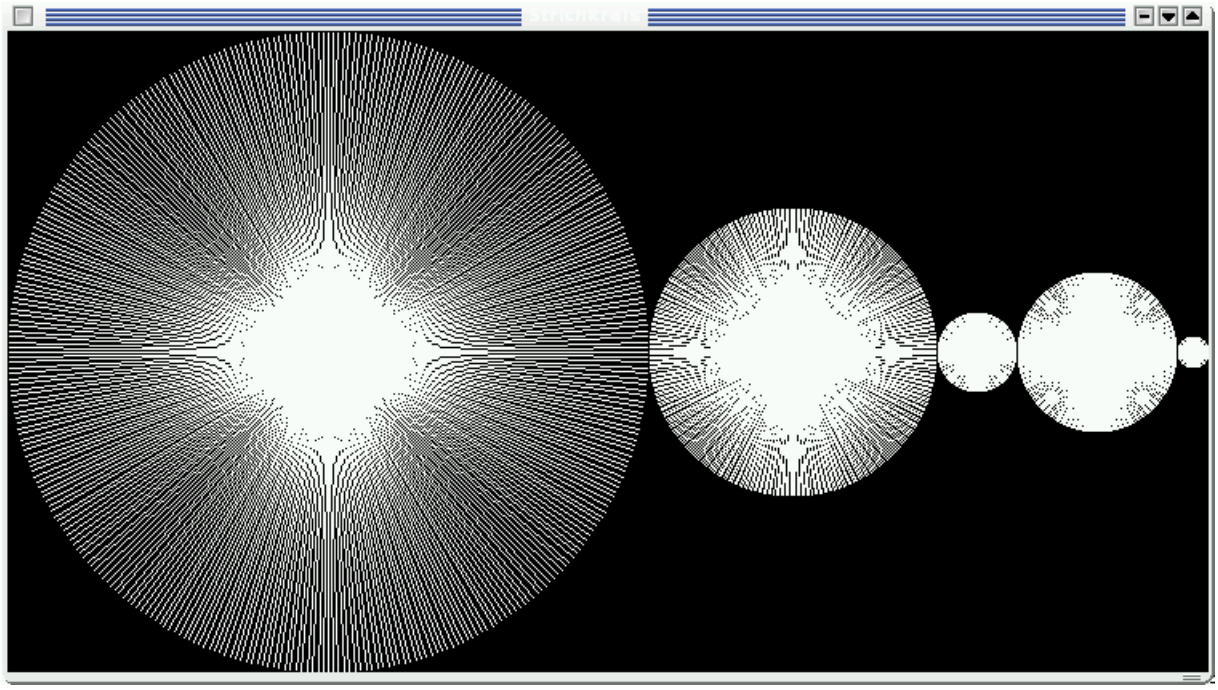

Abbildung 7.3: Strichkreis in Aktion.

7.3.3 Fraktale

Wir haben im letzten Beispiel zum ersten Mal mit Farben gearbeitet, indem die Standardeinstellung der Farben so verändert haben, daß der Hintergrund scharz und die Liienfarbe weiß war. Um noch ein wenig mit Farben zu spielen, zeichnen wir in diesem Abschnitt die berühmten Apfelmännchen. Apfelmännchen werden definiert über eine Funktion auf komplexen Zahlen. Zur Darstellung komplexer Zahlen haben wir bereits eine Klasse als Beispiel zur Überladung von Operatoren geschrieben.

Zu Apfelmännchen findet sich in Wikipedia (`http://de.wikipedia.org/wiki/Apfelmännchen`) folgender Eintrag:

> Die Mandelbrot-Menge, im allgemeinen Sprachgebrauch oft auch*Apfelmännchen* genannt, ist ein von Benoît Mandelbrot 1980 entdecktes Fraktal, das in der Chaostheorie eine bedeutende Rolle spielt.
>
> Außerhalb der Fachwelt wurde die Mandelbrot-Menge vor allem durch ihre computergenerierten graphischen Darstellungen bekannt, die über einen hohen ästhetischen Reiz verfügen, der durch geschickte Farbgestaltung des Außenbereichs, der nicht zur Menge gehört, noch erhöht wird. Die Mandelbrot-Menge wird oft als das formenreichste geometrische Gebilde bezeichnet, das überhaupt bekannt ist. Dieser außerordentliche Formenreichtum zeigt sich an stark vergrößerten Ausschnitten des Randes, die überdies schöne Beispiele für das Konzept der Selbstähnlichkeit bei Fraktalen liefern. Trotz dieser offensichtlichen hohen inneren Ordnung wurde die Mandelbrot-Menge zum Symbol für das mathematische Chaos, einem vom Laien meist missverstandenen Begriff.

Grundlage zum Zeichnen von Apfelmännchen ist folgende Iterationsgleichung auf komplexen Zahlen: $z_{n+1} = z_n^2 + c$. Wobei z_0 die komplexe Zahl $0 + 0i$ mit dem Real- und Imaginärteil 0 ist.

Zum Zeichnen der Apfelmännchen wird ein Koordinatensystem so interpretiert, daß die Achsen jeweils Real- und Imaginärteil von komplexen Zahlen darstellen. Jeder Punkt in diesem Koordinatensystem steht jetzt für die Konstante c in obiger Gleichung. Nun wir geprüft ob und für welches n die Norm von z_n größer eines bestimmten Schwellwertes ist. Je nach der Größe von $n$wird der Punkt im Koordinatensystem mit einer anderen Farbe eingefärbt.

Mit diesem Wissen können wir nun versuchen die Apfelmännchen zu zeichnen. Wir müssen nur geeignete Werte für die einzelnen Parameter finden. Wir schreiben eine eigene Klasse für das graphische Objekt, in dem ein Apfelmännchen gezeichnet wird.

Apfel.h

```
1  #ifndef APFEL__H
2  #define APFEL__H
3
4  #include <gtkmm.h>
5  #include "Complex.h"
6  using namespace Gtk;
7
8  class Apfel : public DrawingArea{
9   public:
10   Apfel();
11
```

```
 protected:
  virtual void on_realize();
  virtual bool on_expose_event(GdkEventExpose* event);
  int checkC(Complex c);
  void paintColorPoint(int x,int y,int it);

  Glib::RefPtr<Gdk::GC> gc;
  Gdk::Color black;
  Gdk::Color colAppleman;
  Glib::RefPtr<Gdk::Colormap> colormap;

  const int width, height, recDepth, schwellwert;
  float zelle,startX,startY;
};

#define WIDTH 480
#define HEIGHT 430
#define ZELLE 0.00625
#define START_X -2
#define START_Y  -1.35
#define REC_DEPTH   50
#define SCHWELLWERT  4

#endif
```

Im Konstruktor werden die einzelnen Felder der Klasse initialisiert.

Apfel.cpp

```
#include "Apfel.h"

Apfel::Apfel():width(WIDTH),height(HEIGHT),zelle(ZELLE)
              ,startX(START_X),startY(START_Y)
              ,recDepth(REC_DEPTH),schwellwert(SCHWELLWERT)
              ,colormap(get_default_colormap())
              ,black(Gdk::Color("black"))
{
  colormap->alloc_color(black);
  colAppleman.set_red(0);
  colAppleman.set_green(129*255);
  colAppleman.set_blue(190*255);
  colormap->alloc_color(colAppleman);
  set_size_request(width, height);
}
```

Die wichtigste Methode berechnet die Werte für die Gleichung $z_{n+1} = z_n^2 + c$. Der Eingabeparameter ist die komplexe Zahl c. Das Ergebnis dieser Methode ist das n, für das z_n größer als der Schwellwert ist:

Apfel.cpp

```
int Apfel::checkC(Complex c){;
  Complex zn(0,0);
```

```
18
19    for (int n=0;n<recDepth;n=n+1) {
20      Complex znp1 = zn*zn+c;
21      if (znp1() > schwellwert) return n;
22      zn=znp1;
23    }
24    return recDepth;
25  }
```

Beim realisieren des Objekts lassen wir uns den graphischen Kontext geben und setzen die Hintergrundfarbe:

Apfel.cpp

```
26  void Apfel::on_realize(){
27    Gtk::DrawingArea::on_realize();
28    Glib::RefPtr<Gdk::Window> window = get_window();
29    gc = Gdk::GC::create(window);
30    window->set_background(black);
31  }
```

Zum Zeichnen des Apfelmännchens iterieren wir über jeden sichtbaren Punkt. Lassen uns für diesen Punkt für die Gleichung den Schwellwert berechnen, und Färben für diesen Punkt entsprechend eines Schemas die Leinwand ein.

Apfel.cpp

```
32  bool Apfel::on_expose_event(GdkEventExpose*){
33    Glib::RefPtr<Gdk::Window> window = get_window();
34    window->clear();
35
36    for (int y=0;y<height;y=y+1) {
37      for (int x=0;x<width;x=x+1) {
38        const Complex current(startX+x*zelle,startY+y*zelle);
39        const int iterationenC = checkC(current);
40        paintColorPoint(x,y,iterationenC);
41      }
42    }
43    return true;
44  }
```

Zum einfärben schauen wir zunächst, ob es der Maximale Wert ist, wenn nicht, errechnen wir aus dem Schwellwert jeweils einen Rot-, Grün- und Blauanteil, um die Farbe zu bestimmen, in der der Punkt eingefärbt wird.

DisplayApfel.cpp

```
45  void Apfel::paintColorPoint(int x,int y,int it){
46    Gdk::Color color;
47    if (it==recDepth){color=colAppleman;}
48    else{
49      color.set_red(65535-5*it%1*255);
```

```
50      color.set_green(65535-it%5*30*255);
51      color.set_blue(65535-it%5* 50*255);
52      colormap->alloc_color(color);
53    }
54    gc->set_foreground(color);
55    get_window()->draw_point(gc, x, y);
56  }
```

Wir haben nun eine hübsche Komponente, die wir in einem Fenster öffnen können.

DisplayApfel.cpp

```
1  #include "Apfel.h"
2
3  int main(int argc, char** argv){
4    Gtk::Main main_instance (argc, argv);
5    Gtk::Window window;
6    Apfel apfel;
7    window.add(apfel);
8    window.show_all();
9    Gtk::Main::run(window);
10   return 0;
11 }
```

Und, voila, das Programm ergibt das Bild aus Abbildung 7.4.

7.4 Weitere Ereignisse

Wir haben bisher nur die einfachste Form von Ereignissen kennengelernt, die in einem GUI auftreten können, dem Drücken eines bestimmten Knopfes. In diesem Abschnitt sollen beispielhaft weitere häufige Ereignisse in GUIs programmiert werden.

7.4.1 Mausereignisse

Eine graphische Benutzerschnittstelle ist kaum ohne die Benutzung der Maus vorstellbar. Die Klasse `Apfel` des letzten Abschnitts soll jetzt so überschrieben werden, daß sie auf Mausereignisse reagiert. Durch Drücken, Bewegen und wieder Loslassen der Maus, soll ein Gebiet im Apfelfenster markiert werden, in das die Anwendung hineinzoomt. Hierzu sind die Methoden: `on_button_press_event` und `on_button_release_event` zu überschreiben. Wir erweitern einfach die Klasse `Apfel`, und überschreiben die Methoden, die sich auf Mausereignisse beziehen.

MouseApfel.h

```
1  #ifndef MOUSE_APFEL__H
2  #define MOUSE_APFEL__H
3
4  #include <gtkmm.h>
5  #include "Apfel.h"
6
```

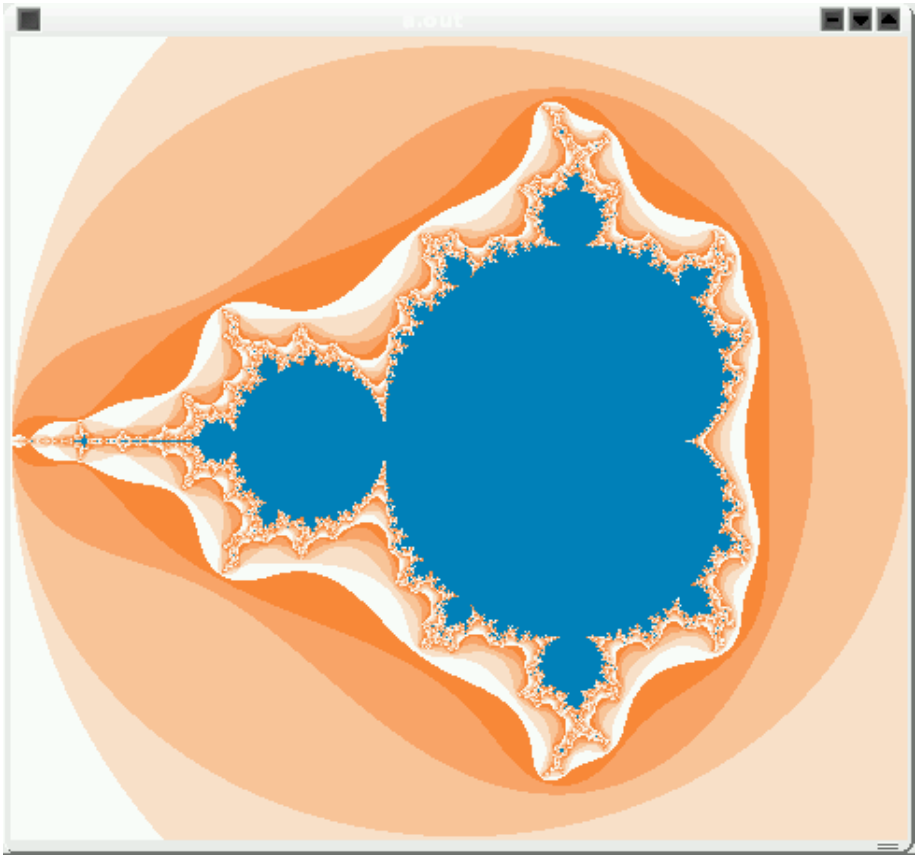

Abbildung 7.4: Apfelmännchen.

```
7  class MouseApfel : public Apfel{
8   public:
9    MouseApfel();
10  protected:
11   virtual void MouseApfel::on_realize();
12   virtual bool on_button_press_event(GdkEventButton* event);
13   virtual bool on_button_release_event(GdkEventButton* event);
```

Die Koordiniaten Maus beim Drücken des Knopfes sollen in zwei privaten Feldern zwischengespeichert werden:

MouseApfel.h

```
14  private:
15   gdouble mouseStartX;
16   gdouble mouseStartY;
17 };
18 #endif
```

Im Konstruktor werden diese beiden Werte mit `0` initialisiert.

MouseApfel.cpp

```
1  #include "MouseApfel.h"
2  #include <iostream>
3
4  MouseApfel::MouseApfel():Apfel(),mouseStartX(0),mouseStartY(0){}
```

Beim realisieren müssen wir Gtkmm deutlich machen, daß auch die Ereignisse des Drückens und Loslassen den Mausknopfs reagiert werden soll. Nur dann werden die entsprechenden überschriebenen Methoden überhaupt ausgeführt.

MouseApfel.cpp

```
5  void MouseApfel::on_realize(){
6    Apfel::on_realize();
7    add_events( Gdk::EXPOSURE_MASK
8              | Gdk::BUTTON_PRESS_MASK
9              | Gdk::BUTTON_RELEASE_MASK
10         );
11 }
```

Wenn der Mausknopf gedrückt wird, merken wir uns die entsprechende Mausposition:

MouseApfel.cpp

```
12 bool MouseApfel::on_button_press_event(GdkEventButton* event){
13   mouseStartX=event->x;
14   mouseStartY=event->y;
15   return false;
16 }
```

Wird der Knopf wieder losgelassen, wird der somit markierte Bereich als neuer Wertebereich auf den komplexen Zahlen genommen, für den die Gleichung des Apfelmännchens berechnet werden soll. Mit dem Aufruf `queue_draw()` fordern wir das Gtkmm-System auf, die Komponente so bald wie möglich wieder neu zu zeichnen.

MouseApfel.cpp

```
17 bool MouseApfel::on_button_release_event(GdkEventButton* event){
18   gdouble endX = event->x;
19   gdouble endY = event->y;
20   startX = startX+(mouseStartX*zelle);
21   startY = startY+(mouseStartY*zelle);
22   zelle  = zelle*(endX-mouseStartX)/width;
23   queue_draw();
24   return false;
25 }
```

Auch diese Komponente können wir in einer Hauptmethode in einem einfachen Fenster starten.

DisplayMouseApfel.cpp

```
1  #include "MouseApfel.h"
2
```

```
3  int main(int argc, char** argv){
4    Gtk::Main main_instance (argc, argv);
5    Gtk::Window window;
6    MouseApfel apfel;
7    window.add(apfel);
8    window.show_all();
9    Gtk::Main::run(window);
10   return 0;
11 }
```

7.4.2 Tastaturereignisse

Einer der wichtigsten Kommunikation mit dem Anwender läuft auch in graphischen benutzeroberflächen noch über die Tastatur. Auf ähnliche Weise wie auf Tastaturereignisse, läßt sich programmieren, wie das GUI auf Tastaturereignisse reagieren soll. Hierzu ist die Methode `on_key_press_event` zu überschreiben.

Bleiben wir bei unserem Beispiel der Apfelmännchen, können wir diese um die Behandlung von Tastaturereignisse erweitern, indem wir abermals eine Unterklasse schreiben. Dieses Mal soll die Methode `on_key_press_event` überschrieben und beim Realisieren des Fensters aktiviert werden. Die entsprechende Klasse wird demnach:

KeyApfel.h

```
1  #ifndef KEY_APFEL__H
2  #define KEY_APFEL__H
3
4  #include <gtkmm.h>
5  #include "MouseApfel.h"
6
7  class KeyApfel : public MouseApfel{
8   protected:
9    virtual void on_realize();
10   virtual bool on_key_press_event(GdkEventKey* event);
11 };
12 #endif
```

Zunächst ist sicher zu stellen, daß die Behandlung von Tastaturereignissen aktiviert ist. dazu muß auch der Fokus auf die Komponente gelegt werden können:

KeyApfel.cpp

```
1  #include "KeyApfel.h"
2  #include <iostream>
3
4  void KeyApfel::on_realize(){
5    MouseApfel::on_realize();
6    add_events(Gdk::KEY_PRESS_MASK);
7    set_flags(Gtk::CAN_FOCUS);
8  }
```

Wir reagieren auf zwei Arten von Ereignissen, dem Drücken der Taste 'q' und dem Drücken einer beliebigen anderen Taste. Im ersten Fall fahren wir die gesammte Applikation herunter, im zweitem Fall soomen wir wieder aus dem zahlenbereich, in dem das Apfelmännchen gezeichnet wird heraus.

KeyApfel.cpp

```
9  bool KeyApfel::on_key_press_event(GdkEventKey* event){
10   if(event->keyval == 'q') exit(0);
11   zelle  = zelle*2;
12   startX = startX-(width*zelle/4);
13   startY = startY-(height*zelle/4);
14   queue_draw();
15   return false;
16 }
```

Öffnen wir diese Komponente in einem Fenster, so haben wir eine Applikation die mit Maus und tastatur bedient werden kann.

DisplayKeyApfel.cpp

```
1  #include "KeyApfel.h"
2
3  int main(int argc, char** argv){
4    Gtk::Main main_instance (argc, argv);
5    Gtk::Window window;
6    KeyApfel apfel;
7    window.add(apfel);
8    window.show_all();
9    Gtk::Main::run(window);
10   return 0;
11 }
```

7.4.3 Zeitgesteuerte Ereignisse

es gibt auch Ereignisse, die nicht durch dem Anwender erzeugt werden, sondern zeitgesteuert auftreten. Häufig ist es notwendig zeitgesteuert in einem bestimmten Takt die GUI-Komponente neu zu zeichnen. Dieses kann z.B. für einfache bewegte Bilder benutzt werden. In bestimmten Zeitintervallen wird die Szenerie verändert und die Komponente neu gezeichnet. Hierzu bedient sich Gtkmm der Bibliothek Glibmm, in der es verschiedene Klassen zu Threads und zeitlichen Ereignissen gibt. Wir werden die Klasse `Glib::SignalTimeout` benutzen, an die wir eine bestimmte Funktion zur Ereignisbehandlung binden werden. Als erstes Beispiel schreiben wir eine GUI-Applikation, die jede Sekunde einen Zähler um eins erhöht. Hierzu definieren folgende einfache Klasse:

SimpleTimer.h

```
1  #ifndef SIMPLE_TIMER__H
2  #define SIMPLE_TIMER__H
3  #include <gtkmm.h>
4  #include "ToString.h"
5
6  using namespace Gtk;
```

```
7
8  class SimpleTimer: public Label{
9    public:
10     virtual bool SimpleTimer::timer_callback();
11     int i;
12     SimpleTimer();
13 };
14 #endif
```

Im Konstruktor lassen wir uns ein `SignalTimeout` Objekt über die Funktion `Glib::signal_timeout()` erzeugen. Wir wir es schon für einfache knopfereignisse gesehen haben, läßt sich jetzt eine Funktion zur Ereignisbehandlung an diesen Zeitsignal mit der Methode `connect` binden. Das zweite Argument der Methode `connect` zeigt in tausendstel Sekunden an, in welchem Intervall das Ereignis ausgeführt werden soll.

SimpleTimer.cpp

```
1  #include "SimpleTimer.h"
2  SimpleTimer::SimpleTimer():Label("0"),i(0){
3    Glib::signal_timeout()
4     .connect(sigc::mem_fun(*this,&SimpleTimer::timer_callback), 1000);
5  }
```

Jetzt ist nur noch auszuprgrammieren, was jede Sekunde für Code auszuführen ist. Wir erhöhen einfach den Zähler um eins und setzen den neuen Zahlenwert als neuen Text auf das Label.

SimpleTimer.cpp

```
6  bool SimpleTimer::timer_callback(){
7    i=i+1;
8    set_text(itos(i));
9    return 1;
10 }
```

Und mit wenigen Zeilen haben wir eine Anwendung, die zeitlich gesteuert wird.

StartSimpleTimer.cpp

```
1  #include "SimpleTimer.h"
2
3  int main(int argc,char** args){
4    Main kit(argc,args);
5    Window w;
6    SimpleTimer c;
7    w.add(c);
8    w.show_all();
9    Main::run(w);
10 }
```

Eine Digitaluhr

Nun läßt sich mit wenigen Zeilen eine graphische Uhrenkomponente realisieren. Hierzu kann einfach eine Unterklasse des einfachen Zählers geschrieben werden, die jede Sekunde die aktuelle Zeit auf dem Label anzeigt:

DigitalClock.h

```
#ifndef DIGITAL_CLOCK__H
#define DIGITAL_CLOCK__H
#include "SimpleTimer.h"

using namespace Gtk;

class DigitalClock: public SimpleTimer{
  public: virtual bool DigitalClock::timer_callback();
};
#endif
```

Nur die Methode `timer_callback` ist zu überschreiben. Die C-Bibliothek zur Uhrzeit wird inkludiert, um die aktuelle Zeit zu erfragen und als Text darzustellen.

DigitalClock.cpp

```
#include "DigitalClock.h"
#include <ctime>
using namespace std;
bool DigitalClock::timer_callback(){
  time_t curr=time(0);
  set_text(ctime(&curr));
  return 1;
}
```

Und schon haben wir eine Uhr!

StartDigitalClock.cpp

```
#include "DigitalClock.h"

int main(int argc,char** args){
  Main kit(argc,args);
  Window w;
  DigitalClock c;
  w.add(c);
  w.show_all();
  Main::run(w);
}
```

Aufgabe 20 Schreiben Sie ein Programm, das eine Analoguhr auf dem Bildschirm realisiert. Die Uhr soll dabei Sekunden, Minuten und Stundenzeiger haben.

Das Telespiel: Ping

jetzt haben wir alle Grundtechniken beieinander, um

Movable.h

```
#include <gtkmm/window.h>
#ifndef __MOVABLE_H
```

```
3  #define __MOVABLE_H ;
4
5  class Movable {
6    int _xPos;
7    int _yPos;
8    int _xDir;
9    int _yDir;
10
11   int _width;
12   int _height;
13
14 public:
15   Movable();
16   Movable(const int xPosVal,const int yPosVal);
17   virtual ~Movable();
18
19   void setXPos(const int val);
20   void setYPos(const int val);
21   void setXDir(const int val);
22   void setYDir(const int val);
23   void setWidth(const int val);
24   void setHeight(const int val);
25
26   int getXPos(){return _xPos;}
27   int getYPos(){return _yPos;}
28   int getXDir(){return _xDir;}
29   int getYDir(){return _yDir;}
30   int getWidth(){return _width;}
31   int getHeight(){return _height;}
32
33   bool touches(Movable &other);
34
35   virtual void move(const int maxX,const int maxY);
36   virtual void paint(Glib::RefPtr<Gdk::Window> window
37                      ,Glib::RefPtr<Gdk::GC> gc_);
38
39 private:
40   bool touchesHorizontal(Movable &other);
41   bool touchesVertical(Movable &other);
42 };
43
44 #endif /* __MOVABLE_H */
```

Movable.cpp

```
1  #include "Movable.h"
2  Movable::Movable(){
3    _xPos=0;
4    _yPos=0;
5    _xDir=1;
```

```
6     _yDir=1;
7
8     _width=10;
9     _height=10;
10  }
11
12  Movable::Movable(const int xPosVal,const int yPosVal){
13    _xPos=xPosVal;
14    _yPos=yPosVal;
15    _xDir=10;
16    _yDir=10;
17
18    _width=10;
19    _height=10;
20  }
21
22  Movable::~Movable(){}
23
24  void Movable::setXPos(const int val){_xPos=val;}
25  void Movable::setYPos(const int val){_yPos=val;}
26  void Movable::setXDir(const int val){_xDir=val;}
27  void Movable::setYDir(const int val){_yDir=val;}
28  void Movable::setWidth(const int val){_width=val;}
29  void Movable::setHeight(const int val){_height=val;}
30
31  void Movable::move(const int maxX,const int maxY){
32    _xPos = _xPos + _xDir;
33    _yPos = _yPos + _yDir;
34  }
35
36  void Movable::paint
37    (Glib::RefPtr<Gdk::Window> window,Glib::RefPtr<Gdk::GC> gc_){
38  }
39
40  bool Movable::touches(Movable &other){
41    return (   touchesHorizontal(other) && touchesVertical(other));}
42
43  bool Movable::touchesHorizontal(Movable &other){
44    const int otherRight = other._xPos+other._width;
45    const int thisRight  = _xPos+_width;
46
47    return (_xPos<=other._xPos  && other._xPos<=thisRight)
48        || (_xPos<=otherRight   && otherRight<=thisRight)
49        || (_xPos>=other._xPos  && otherRight>=_xPos);
50  }
51
52  bool Movable::touchesVertical(Movable &other){
53    const int otherUpper = other._yPos+other._height;
54    const int thisUpper  = _yPos+_height;
55
```

```
56    return   (_yPos<=other._yPos    && other._yPos<=thisUpper)
57          || (_yPos<=otherUpper     && otherUpper<=thisUpper)
58          || (_yPos>=other._yPos    && otherUpper>=thisUpper);
59  }
```

Ball.h

```
1   #include "Movable.h"
2
3   class Ball  : public Movable{
4   public:
5     Ball();
6     ~Ball();
7
8     virtual void move(const int maxX,const int maxY);
9     virtual void paint(Glib::RefPtr<Gdk::Window> window
10                      ,Glib::RefPtr<Gdk::GC> gc_);
11  };
```

Ball.cpp

```
1   #include <gdkmm/window.h>
2   #include <gtkmm/window.h>
3   #include "Ball.h"
4
5   Ball::Ball(){}
6   Ball::~Ball(){}
7
8   void Ball::move(const int maxX,const int maxY){
9     Movable::move(maxX,maxY);
10
11    if (getXPos()>=maxX+80) setXPos(30);
12    if (getXPos()<=-80) setXPos(maxX-40);
13
14    if (getYPos()>=maxY-getHeight() || getYPos()<=0)
15      setYDir(-getYDir());
16  }
17
18  void Ball::paint(Glib::RefPtr<Gdk::Window> window
19                  ,Glib::RefPtr<Gdk::GC> gc_){
20    window->draw_arc(gc_, 1,getXPos(),getYPos()
21                     ,getWidth(),getHeight(),0,360*64);
22  }
```

Paddle.h

```
1   #include "Movable.h"
2
3   class Paddle  : public Movable{
4   public:
5     Paddle(const int xPosVal,const int boardHeight);
```

```
6   ~Paddle();
7
8   virtual void move(const int maxX,const int maxY);
9   virtual void paint(Glib::RefPtr<Gdk::Window> window
10                    ,Glib::RefPtr<Gdk::GC> gc_);
11
12  static const int yMoveVal =11;
13 };
```

Paddle.cpp

```
1  #include <gdkmm/window.h>
2  #include <gtkmm/window.h>
3  #include "Paddle.h"
4
5  Paddle::Paddle(const int xPosVal,const int boardHeight)
6    :Movable(xPosVal,boardHeight-50){
7    setWidth(10);
8    setHeight(50);
9    setXDir(0);
10   setYDir(0);
11 }
12
13 Paddle::~Paddle(){}
14
15 void Paddle::move(const int maxX,const int maxY)
16 { Movable::move(maxX,maxY);
17   if (getYPos()>=maxY-getHeight()) setYPos(maxY-getHeight());
18
19   if (getYPos()<0) setYPos(0);
20 }
21
22 void Paddle::paint(Glib::RefPtr<Gdk::Window> window
23                   ,Glib::RefPtr<Gdk::GC> gc_){
24   window->draw_rectangle(gc_,1,getXPos()
25                          ,getYPos(),getWidth(),getHeight());
26 }
```

GameCanvas.h

```
1  #include <gtkmm/drawingarea.h>
2  #include <gdkmm/window.h>
3  #include <gtkmm/window.h>
4
5  class GameCanvas : public Gtk::DrawingArea{
6  public:
7    GameCanvas();
8    virtual ~GameCanvas();
9    bool timer_callback();
10
11   virtual void moveObjects(const int maxX,const int maxY);
```

```
12    virtual void paintObjects(Glib::RefPtr<Gdk::Window> window
13                              ,Glib::RefPtr<Gdk::GC> gc_);
14    virtual void doChecks();
15
16
17    //Overridden default signal handlers:
18    virtual void on_realize();
19    virtual bool on_expose_event(GdkEventExpose* event);
20
21    Glib::RefPtr<Gdk::GC> gc_;
22    Gdk::Color black_, white_;
23  };
```

GameCanvas.cpp

```
1   #include <gtkmm/drawingarea.h>
2   #include <gdkmm/colormap.h>
3   #include <gdkmm/window.h>
4   #include <gtkmm/window.h>
5   #include "GameCanvas.h"
6
7   GameCanvas::GameCanvas(){
8     Glib::RefPtr<Gdk::Colormap> colormap = get_default_colormap();
9     black_ = Gdk::Color("black");
10    white_ = Gdk::Color("white");
11
12    colormap->alloc_color(black_);
13    colormap->alloc_color(white_);
14    set_size_request(400, 200);
15
16    Glib::signal_timeout()
17     .connect(sigc::mem_fun(*this,&GameCanvas::timer_callback), 3);
18
19    add_events(Gdk::EXPOSURE_MASK);
20  }
21
22  GameCanvas::~GameCanvas(){}
23
24  void GameCanvas::on_realize(){
25    // We need to call the base on_realize()
26    Gtk::DrawingArea::on_realize();
27
28    // Now we can allocate any additional resources we need
29    Glib::RefPtr<Gdk::Window> window = get_window();
30    gc_ = Gdk::GC::create(window);
31    window->set_background(black_);
32
33    set_flags(Gtk::CAN_FOCUS);
34  }
35
```

```
36 bool GameCanvas::on_expose_event(GdkEventExpose*){
37   // we need a ref to the gdkmm window
38   Glib::RefPtr<Gdk::Window> window = get_window();
39   window->clear();
40   gc_->set_foreground(white_);
41   paintObjects(window,gc_);
42   return true;
43 }
44
45 bool GameCanvas::timer_callback(){
46   int winx, winy, winw, winh, wind;
47   Glib::RefPtr<Gdk::Window> window = get_window();
48   window->get_geometry(winx, winy, winw, winh, wind);
49
50   moveObjects(winw,winh);
51   doChecks();
52   queue_draw();
53   // never disconnect timer handler
54   return 1;
55 }
56
57 void GameCanvas::moveObjects(const int maxX,const int maxY){}
58
59 void GameCanvas::paintObjects(Glib::RefPtr<Gdk::Window> window
60                               ,Glib::RefPtr<Gdk::GC> gc_){}
61
62 void GameCanvas::doChecks(){}
```

Ping.h

```
1  #ifndef PING__H
2  #define PING__H
3  #include <gtkmm.h>
4  #include "Ball.h"
5  #include "Paddle.h"
6  #include "GameCanvas.h"
7
8  using namespace std;
9
10 class Ping : public GameCanvas{
11 public:
12   Ping();
13   ~Ping();
14   bool timer_callback();
15
16   int pointsLeftPlayer;
17   int pointsRightPlayer;
18   bool newBall;
19
20   Ball ball;
```

```
21    Paddle leftPaddel;
22    Paddle rightPaddel;
23
24    virtual void moveObjects(const int maxX,const int maxY);
25    virtual void paintObjects(Glib::RefPtr<Gdk::Window> window
26                              ,Glib::RefPtr<Gdk::GC> gc_);
27
28    virtual void doChecks();
29
30    virtual bool on_key_press_event(GdkEventKey* event);
31    virtual bool on_key_release_event(GdkEventKey* event);
32  };
33  #endif
```

Ping.cpp

```
1   #include "Ping.h"
2
3   Ping::Ping():GameCanvas(),leftPaddel(10,200),rightPaddel(380,200){
4     set_flags(Gtk::CAN_FOCUS);
5     pointsLeftPlayer = 0;
6     pointsRightPlayer = 0;
7   }
8
9   Ping::~Ping(){}
10
11  void Ping::paintObjects(Glib::RefPtr<Gdk::Window> window
12                          ,Glib::RefPtr<Gdk::GC> gc_){
13    ball.paint(window,gc_);
14    leftPaddel.paint(window,gc_);
15    rightPaddel.paint(window,gc_);
16  }
17
18  void Ping::moveObjects(const int maxX,const int maxY){
19    this->ball.move(maxX,maxY);
20    this->leftPaddel.move(maxX,maxY);
21    this->rightPaddel.move(maxX,maxY);
22  }
23
24  void Ping::doChecks(){
25    if (ball.touches(leftPaddel) || ball.touches(rightPaddel))
26      ball.setXDir(-ball.getXDir());
27
28    if (ball.getXPos()<0&& newBall){
29      pointsRightPlayer++;newBall=false;
30    }else if (ball.getXPos()>400&&newBall){
31      pointsLeftPlayer++;newBall=false;
32    }else if (!newBall && ball.getXPos()>40
33                       && ball.getXPos()<80) {
34      newBall=true;
```

```
35    }
36  }
37
38  bool Ping::on_key_press_event(GdkEventKey* event){
39    if(event->keyval == 'l')
40      rightPaddel.setYDir(- Paddle::yMoveVal);
41    else if(event->keyval == 'a')
42      leftPaddel.setYDir(- Paddle::yMoveVal);
43  }
44
45  bool Ping::on_key_release_event(GdkEventKey* event){
46    if(event->keyval == 'l')
47      rightPaddel.setYDir(Paddle::yMoveVal);
48    else if(event->keyval == 'a')
49      leftPaddel.setYDir(Paddle::yMoveVal);
50  }
```

PlayPing.cpp

```
1  #include "Ping.h"
2  int main(int argc, char** argv){
3    Gtk::Main main_instance (argc, argv);
4    Gtk::Window* window = new Gtk::Window();
5    Gtk::HBox m_box0(false,0);
6
7    Ping* p = new Ping();
8    m_box0.pack_start(*p, Gtk::PACK_EXPAND_WIDGET, 0);
9    window->add(m_box0);
10   window->show_all();
11
12   Gtk::Main::run(*window);
13   delete p;
14   delete window;
15   return 0;
16 }
```

7.5 Fragen des Layouts

Wir haben bisher gesehen, wie wir mit Hilfe der Klassen `VBox` und `HBox` die einzelnen Guikomponenten zueinander ausgerichtet werden können. Für größere graphische Oberflächen ist dies ein mühsames Unterfangen, bei dem man schnell die Übersicht vrlieren kann. In diesem Abschnitt werden zwei Möglichkeiten vorgeschlagen, diese Aufgabe besser zu bewältigen:

- mit einer kleinen Hilfsbibliothek, die ein kleines graphisches Layout im Programmtext ermöglicht.
- mit Hilfe eines GUI-Builder Programms.

7.5.1 Eine kleine Layoutbibliothek

In diesem Abschnitt wollen wir anhand der Frage des Layouts der GUI-Komponente in Gtkmm zeigen, wie flexibel C++ in der Definition von Bibliotheken ist. Die dabei entstehende Minibibliothek, das sei nicht verschwiegen, ist für den ernsthaften Gebrauch nicht gedacht, und soll lediglich die Möglichkeiten der programmiertechnischen Mittel in C++ demonstrieren und die Phantasie der Leser, diese auf vielfältige Weise zu nutzen, anspornen.

Plazieren relativ im 2-dimensionalen Raum

Der Bildschirm ist ein zweidimensionaler Raum. Eine der häufigsten Aufgaben in der graphischen und GUI-Programmierung in der Informatik ist das plazieren von grephischen Objekten im zweidimensionalen Raum. Dieses macht z.B. auch ein Textverareitungsprogramm, ein Notensatzprogramm, ein Programm zur Darstellung chemischer Strukturen ebenso wie Graph- und Baumlayoutprogramme. Und auch die einzelnen Steuerelemente einer GUI-Anwendung sind auf einem zweidimensionalen Raum zu plazieren.

Im Prinzip geibt es zwei fundamentale Wege wie zwei Objekte im graphischen Layout relativ zueinander ausgerichtet werden können:

- horizontal, also hintereinander. Hierzu stellt die Gtkmm-Bibliothek entsprechend auch das Widget `HBox` zur Verfügung.
- vertikal, also übereinander. Hierzu stellt die Gtkmm-Bibliothek entsprechend auch das Widget `VBox` zur Verfügung.

Statt die Widgets `HBox` und `VBox` explizit zu benutzen, werden wir jetzt eine kleine Klasse bereitstellen, in der Widgets gekapselt sind. Auf dieser Klasse werden wir dann Operatoren zur vertikalen und horizontalen Ausrichtung zweier solcher gekapselter Widgets definieren:

Layout.h

```
1  #ifndef LAYOUT__H
2  #define LAYOUT__H ;
3  #include <gtkmm.h>
4  using namespace Gtk;
5
6  class B{
7   public:
8    Widget* w;
9    B(Widget* w);
```

Als Operatoren zum graphischen Layout benutzen wir `|` für die vertikale Ausrichtung und `,` für die horizontale Ausrichtung:

Layout.h

```
10   B operator,(B b2);
11   B operator|(B b2);
```

Zusätzlich überladen wir die beiden Operatoren, daß sie auch ein direktes Widget mit einem als `B`-Objekt gekapselten Widget kombinieren können:

Layout.h

```
12   B operator,(Widget* w2);
13   B operator|(Widget* w2);
```

Für interen Zwecke merkt sich ein `B`-Objekt, ob es schon im vertikalen oder horizontalen Modus zwei Objekte kombiniert hat:

Layout.h

```
14
15  private:
16   bool isHorizontal;
17   bool isVertical;
18 };
19
20 #endif
```

Schreiten wir zur Implementierung. zunächst der Konstruktor, der einen Zeiger auf ein Widget übergeben bekommt und die Hilfsflags auf `false` setzt.

Layout.cpp

```
1  #include "Layout.h"
2
3  B::B(Widget* w):w(w),isHorizontal(false),isVertical(false){};
```

Beginnen wir mit der horizontalen Kombination. Hierzu prüfen wir zunächst, ob es sich bei dem gekapselten Widget bereits um eine horizontale Kombination handelt. Wenn ja, dann gehen wir davon aus, daß es sich um ein `HBox`-Objekt handelt und fügen diesem das linke Widget hinzu:

Layout.cpp

```
4  B B::operator,(Widget* w2){
5    if (isHorizontal){
6      ((HBox*)w)->add(*w2);
7      return *this;
```

Andernfalls legen wir eine neue `HBox` an und fügen das bisher bereits gespeicherte Widget sowie das neue (die rechte Seite des Operators) hinzu.

Außerdem merken wir uns, daß wir jetzt etwas horizontal ausgerichtet haben:

Layout.cpp

```
8    }else{
9      HBox* hbox = new HBox();
10     hbox->add(*w);
11     hbox->add(*w2);
12     w=hbox;
13     isHorizontal=true;
14     return *this;
15   }
16 }
```

Entsprechend analog läuft der Code für die vertikale Kombination.

Layout.cpp

```
17 B B::operator|(Widget* w2){
18   if (isVertical){
19     ((VBox*)w)->add(*w2);
20     return *this;
21   }else{
22     VBox* vbox = new VBox();
23     vbox->add(*w);
24     vbox->add(*w2);
25     w=vbox;
26     isVertical=true;
27     return *this;
28   }
29 }
```

Die Überladungen der Operatoren für die bereits als B-Objekt gekapselten Widgets, entpacken diese und rufen die bereits implementierten Varianten der Operatoren auf.

Layout.cpp

```
30 B B::operator,(B b2){return *this , b2.w;}
31 B B::operator|(B b2){return *this | b2.w;}
32
```

Starten wir mit einem kleinen Test der Layoutbibliothek:

TestLayout.cpp

```
1  #include <gtkmm.h>
2  #include <iostream>
3  #include "Layout.h"
4
5  using namespace Gtk;
6  using namespace std;
7
8  int main(int argc,char** args){
9    Main kit(argc,args);
10   Window window;
11
12   Label display("ein wenig layout");  B l1(&display);
13   Button knopf7("7"); B k7(&knopf7);
14   Button knopf8("8");
15   Button knopf9("9");
16   Button knopf4("4"); B k4(&knopf4);
17   Button knopf5("5");
18   Button knopf6("6");
19   Button knopf1("1"); B k1(&knopf1);
20   Button knopf2("2");
21   Button knopf3("3");
22   Button knopf0("0"); B k0(&knopf0);
```

```
23
24    B   content =
25                    l1
26           |(k7,&knopf8,&knopf9)
27           |(k4,&knopf5,&knopf6)
28           |(k1,&knopf2,&knopf3)
29           |         k0
30    ;
31
32    Widget* c= (content.w);
33    window.add(*c);
34    window.show_all();
35    Main::run(window);
36  }
37
```

Das kleine Testprogramm erzeugt das in Abbildung 7.5 zu bewundern.

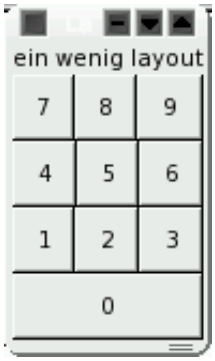

Abbildung 7.5: Mit der Layoutbibliothek erzeugtes Fenster.

Ein kleiner Taschenrechner

Im letzten Test haben wir es bereits angedeutet, wie sich mit den einfachen Knöpfen ein kleiner Taschenrechner programmieren läßt. In diesem Abschnitt wollen wir einen solchen Rechner mit Hilfe der Layoutbibliothek implementieren.

Das Modell Hierzu konzentrieren wir uns zunächst auf das Datenmodell, auf dem ein Taschenrechner agieren soll. In einem Taschenrechner sind wir es gewohnt die arithmetischen Operationen in Infixschreibweise einzugeben. Das bedeutet, daß wir uns zwei Zahlenwerte und die auf diese anzuwendene Operation merken müssen. Daher sehen wir entsprechende drei Felder in der Klasse `CalculatorModell` vor.

CalculatorModell.h

```
1  #ifndef CALCULATOR_MODELL__H
2  #define CALCULATOR_MODELL__H  ;
3
```

```
4  #include "Lambda.h"
5
6  typedef int(*BinOperator)(int,int);
7
8  class CalculatorModell {
9   public:
10   int currentValue;
11   int bufferedValue;
12   BinOperator op;
```

Wir merken uns in einem zusätzlichen Feld, welcher Wert gerade sichtbar sein soll. Außerdem benötigen wir einen Konstruktor für die Klasse:

CalculatorModell.h

```
13   int visibleValue;
14   CalculatorModell();
15 };
```

Für zwei Arten Operationen, die wir auf einem `CalculatorModell`-Objekt ausführen wollen, sehen wir entsprechende Funktionen vor. Eine die den Zustand verändern, wenn eine weitere Zifferntaste gedrückt wird, eine wenn eine Operatortaste gedrückt wird.

CalculatorModell.h

```
16 CalculatorModell& addDigit(int digit,CalculatorModell& m);
17 CalculatorModell& applyOperator(BinOperator,CalculatorModell& m);
```

Um Funktionszeiger für die vier Grundrechenarten zu haben, werden für diese Funktionen definiert:

CalculatorModell.h

```
18 int addOp(int x,int y);
19 int subOp(int x,int y);
20 int mulOp(int x,int y);
21 int divOp(int x,int y);
```

Wir werden in unserer Implementierung tatsächlich Currying verwenden.

CalculatorModell.h

```
22 typedef Lambda<CalculatorModell&,CalculatorModell&>* CalcFunction;
23
24 #endif
```

Die Implementierung dieser Klasse ist relativ geradeheraus. Wird eine Ziffer gedrückt, so wird diese als neue Einerstelle zum aktuellen Wert hinzugefügt. Dafür muß der alte aktuelle Wert um den Faktor 10 erhöht werden:

CalculatorModell.cpp

```
1  #include "CalculatorModell.h"
2  #include <iostream>
3
```

```
4  CalculatorModell& addDigit(int digit,CalculatorModell& m){
5    m.currentValue=10*m.currentValue+digit;
6    m.visibleValue=m.currentValue;
7    return m;
8  }
```

Die Funktion zur Ausführung eines Operators wird der gespeicherte Operator auf die beiden Werte angewendet. Das Ergebis wird als neuer linker Operand im Modellobjekt gespeichert. Der neue Operator wird im Operatorfeld abgelegt:

CalculatorModell.cpp

```
9  CalculatorModell& applyOperator(BinOperator op,CalculatorModell& m){
10   m.bufferedValue = m.op(m.bufferedValue,m.currentValue);
11   m.currentValue  = 0;
12   m.visibleValue=m.bufferedValue;
13   m.op=op;
14   return m;
15 }
```

Im Konstruktor werden alle Wert mit 0 initialisiert. Die Operatorfunktion wird mit der Addition initialisiert. Für diese ist die 0 ein neutrales Element.

CalculatorModell.cpp

```
16 CalculatorModell::CalculatorModell()
17   :currentValue(0),bufferedValue(0),visibleValue(0),op(addOp){}
```

Die Implementierung der Grundrechenarten sollte niemanden überraschen:

CalculatorModell.cpp

```
18 int addOp(int x,int y) {return x+y;}
19 int subOp(int x,int y) {return x-y;}
20 int mulOp(int x,int y) {return x*y;}
21 int divOp(int x,int y) {return x/y;}
22
```

Das Gui Damit steht unser Modell für den Taschenrechner. Um diesem herum wollen wir jetzt eine graphische Benutzeroberfläche entwickeln. Die graphische Grundkomponente hierzu wird ein Knopf des Rechners sein. Ein Knopf braucht eine Beschriftung, eine Funktione, die das Modell manipuliert und das Modell, auf das der Knopf wirkt:

CalculatorGUI.h

```
1  #ifndef CALCULATOR_GUI__H
2  #define CALCULATOR_GUI__H  ;
3
4  #include "CalculatorModell.h"
5  #include <gtkmm.h>
6
7  using namespace Gtk;
8  using namespace std;
```

```
9
10 class CalculatorButton :public Button{
11  public:
12   string l;
13   CalcFunction f;
14   CalculatorModell& m;
```

Zusätzlich muß ein Knopf wissen, auf welchem Label der aktuelle Zustand des Modells dargestellt wird. Außerdem benötigen wir eine Konstruktor und eine Methode, die auf das Ereignis des Knopfdrückens reagiert:

CalculatorGUI.h

```
15   Label& display;
16
17   CalculatorButton
18     (CalculatorModell& m,Label& display, string l,CalcFunction f);
19   void doWhenClicked();
20 };
```

Zwei spezialisierte Knopfarten gibt es an unserem Taschenrechner. Zunächst einmal Knöpfe mit Ziffern:

CalculatorGUI.h

```
21 class DigitButton :public CalculatorButton{
22  public:
23   DigitButton(CalculatorModell& m,Label& l,int d);
24 };
```

Und Knöpfe für operatoren:

CalculatorGUI.h

```
25 class OperatorButton :public CalculatorButton{
26  public:
27   OperatorButton(CalculatorModell& m
28                  ,Label& l,string s,BinOperator op);
29 };
30
31 #endif
```

Die Methoden dieser Klassen lassen sich sehr kurz implementieren. Zunächst der Konstruktor für die gemeinsame Oberklasse `CalculatorButton`.

CalculatorGUI.cpp

```
1 #include "CalculatorGUI.h"
2 #include "ToString.h"
3 #include "Curry.h"
4
5 #include <iostream>
6
```

```
7  CalculatorButton::CalculatorButton
8     (CalculatorModell& m,Gtk::Label& display, string l,CalcFunction f)
9           :Button(l),m(m),display(display),l(l),f(f){
10   signal_clicked()
11    .connect(mem_fun(*this,&CalculatorButton::doWhenClicked));
12 }
```

Wird ein Knopf gedrückt, so ist seine Funktion auf das Modell anzuwenden. Wir erhalten ein neuen Zustand. Dessen sichtbar zu machender Wert, wird im Label angezeigt.

CalculatorGUI.cpp

```
13 void CalculatorButton::doWhenClicked(){
14   m=(*f)(m);
15   display.set_text(itos(m.visibleValue));
16 }
```

Für die Konstruktoren der zwei Knopfarten wenden wir das Currying an. Mit `addDigit` und `applyOperator` haben wir zwei Funktionen, die noch nicht vom Typ `CalcFunction`. Eine `CalcFunction` überführt einen `CalculatorModell`-Zustand in einen neuen `CalculatorModell`-Zustand. Die beiden oben genannten Funktionen haben aber noch ein weiteres Argument, zum einen die Ziffer für den Knopf, zum anderen die Operatorfunktion für den Knopf. Wenn wir die Ziffer bzw. die Operatorfunktion kennen, können wir durch Currying die `CalcFunction` erhalten.

damit erhalten wir den folgenden Konstruktor für Knöpfen, die Ziffern darstellen:

CalculatorGUI.cpp

```
17 DigitButton::DigitButton(CalculatorModell& m,Label& l,int d)
18     :CalculatorButton
19   (m,l,itos(d),new Curry<int,CalculatorModell&,CalculatorModell&>
20                           (addDigit,d)){}
```

Ganz analog funktionieren die Knöpfe, die Operatoren darstellen:

CalculatorGUI.cpp

```
21 OperatorButton::OperatorButton
22       (CalculatorModell& m,Label& l,string s,BinOperator op)
23     :CalculatorButton
24   (m,l,s,new Curry<BinOperator,CalculatorModell&,CalculatorModell&>
25                           (applyOperator,op)){}
26
```

Und nun läßt sich alles zu einem kleinem Taschenerechner zusammenstecken.

Calculator.h

```
1  #ifndef CALCULATOR__H
2  #define CALCULATOR__H
3  #include <gtkmm.h>
4  #include "Layout.h"
5  #include "CalculatorGUI.h"
```

```
6  #include "CalculatorModell.h"
7
8  using namespace Gtk;
9  using namespace std;
10
11 class Calculator:public HBox{
12  public:
13   Calculator();
14 };
15 #endif
```

Calculator.cpp

```
1  #include "Calculator.h"
2  Calculator::Calculator():HBox(){
3    CalculatorModell m;
4
5    Label display("0");  B l1(&display);
6    DigitButton knopf7(m,display,7); B k7(&knopf7);
7    DigitButton knopf8(m,display,8);
8    DigitButton knopf9(m,display,9);
9    DigitButton knopf4(m,display,4); B k4(&knopf4);
10   DigitButton knopf5(m,display,5);
11   DigitButton knopf6(m,display,6);
12   DigitButton knopf1(m,display,1); B k1(&knopf1);
13   DigitButton knopf2(m,display,2);
14   DigitButton knopf3(m,display,3);
15   DigitButton knopf0(m,display,0); B k0(&knopf0);
16
17   OperatorButton addB(m,display,"+",addOp);
18   OperatorButton subB(m,display,"-",subOp);
19   OperatorButton mulB(m,display,"*",mulOp);
20   OperatorButton divB(m,display,"/",divOp);
21
22   B   content =
23                       l1
24          |(k7,&knopf8,&knopf9,&addB)
25          |(k4,&knopf5,&knopf6,&subB)
26          |(k1,&knopf2,&knopf3,&mulB)
27          |        (k0,&divB)
28   ;
29
30   add(*(content.w));
31 }
```

DisplayCalculator.cpp

```
1  #include <gtkmm.h>
2  #include "Calculator.h"
3
4  int main(int argc,char** args){
```

```
  Main kit(argc,args);
  Window window;

  Calculator m;
  window.add(m);
  window.show_all();
  Main::run(window);
}
```

7.5.2 Ein GUI-Builder

In der Praxis werden GUIs selten von Hand codiert, sondern es werden graphische Werkzeuge benutzt, um ein GUI zu definieren. Für Gtkmm

7.6 Gtk für C++ Beispielprogramme

7.6.1 Browser für XML-Dokumente

XMLTree.h

```
#ifndef XMLTREE_H
#define XMLTREE_H

#include <gtkmm.h>
#include <xercesc/dom/DOM.hpp>

XERCES_CPP_NAMESPACE_USE
class MyModelColumns : public Gtk::TreeModel::ColumnRecord{
 public:
   Gtk::TreeModelColumn<Glib::ustring>  tagname;
   MyModelColumns();
};

class ExampleWindow : public Gtk::Window{
 public:
  ExampleWindow(DOMNode* doc);
  virtual ~ExampleWindow();

 protected:
  Gtk::VBox m_VBox;
  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TreeView m_TreeView;
  Glib::RefPtr<Gtk::TreeStore> m_refTreeModel;
  DOMNode* doc;
};

#endif
```

XMLTree.cpp

```
#include <iostream>
#include "XMLTree.h"

bool justWhite(std::string str){
  char const* delims = " \t\r\n";
  int pos;

  std::string::size_type  notwhite = str.find_first_not_of(delims);
  str.erase(0,notwhite);
  return str=="";
}

static void buildTree
 (Glib::RefPtr<Gtk::TreeStore>treeModel
 ,Gtk::TreeModel::Row row
 ,DOMNode* n){
  MyModelColumns columnrecord;

  row = *(treeModel->append(row.children()));

  if (n->getNodeType() == DOMNode::ATTRIBUTE_NODE){
    std::string s =  XMLString::transcode(n ->getNodeName());
    s = s+"=\""+XMLString::transcode(n ->getNodeValue())+"\"";
    row[columnrecord.tagname] = s;
  }else if (n->getNodeType() == DOMNode::ELEMENT_NODE){
    std::string s =  XMLString::transcode(n ->getNodeName());
    row[columnrecord.tagname] ="<"+s+">";
  }else {
    std::string s =  XMLString::transcode(n ->getNodeValue());
    if (not(justWhite(s))) row[columnrecord.tagname] = s;
  }

  DOMNamedNodeMap* attributes = n->getAttributes();
  if (not (NULL==attributes)){
   for (int i=0;i<attributes->getLength();i++){
     DOMNode* attr = attributes->item(i);
     buildTree(treeModel,row,attr);
   }
  }

  if (n->hasChildNodes()){
    for (DOMNode *child = n->getFirstChild()
        ;child != 0
        ;child=child->getNextSibling()){
      buildTree(treeModel,row,child);
    }
  }
}

```

```
50 MyModelColumns::MyModelColumns(){add(tagname);}
51
52 ExampleWindow::ExampleWindow(DOMNode* doc): doc(doc){
53   MyModelColumns columnrecord;
54   m_refTreeModel = Gtk::TreeStore::create(columnrecord);
55   set_title("Gtk::TreeView (TreeStore) example");
56   set_border_width(5);
57   set_default_size(400, 200);
58
59   add(m_VBox);
60
61   //Add the TreeView, inside a ScrolledWindow, with the button underneath:
62   m_ScrolledWindow.add(m_TreeView);
63
64   //Only show the scrollbars when they are necessary:
65   m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);
66
67   m_VBox.pack_start(m_ScrolledWindow);
68   m_TreeView.set_model(m_refTreeModel);
69
70   Gtk::TreeModel::Row row = *(m_refTreeModel->append());
71
72   buildTree(m_refTreeModel,row,doc);
73
74   m_TreeView.append_column("", columnrecord.tagname);
75   show_all_children();
76 }
77
78 ExampleWindow::~ExampleWindow(){}
79
80
```

ShowTree.cpp

```
1  #include "XMLTree.h"
2
3  #include <gtkmm/main.h>
4  #include <xercesc/parsers/XercesDOMParser.hpp>
5  #include <iostream>
6
7  int main(int argc, char *argv[]){
8    Gtk::Main kit(argc, argv);
9    try{
10     XMLPlatformUtils::Initialize();
11     static const XMLCh gLS[] = { chLatin_L, chLatin_S, chNull };
12     DOMImplementation *impl
13       = DOMImplementationRegistry::getDOMImplementation(gLS);
14     DOMBuilder *parser
15       = impl->createDOMBuilder(DOMImplementationLS::MODE_SYNCHRONOUS,0);
16     DOMDocument *doc = parser->parseURI(argv[1]);
```

```
17
18      ExampleWindow window((DOMNode*)doc->getDocumentElement());
19
20      XMLPlatformUtils::Terminate();
21      Gtk::Main::run(window);
22    }catch (...) {std::cerr<<"error occured"<<std::endl;}
23    return 0;
24  }
25
```

AllTogether.cpp

```
1  #include <gtkmm.h>
2  #include "KeyApfel.h"
3  #include "DigitalClock.h"
4  #include "games/Ping.h"
5  #include "Layout.h"
6  #include "Calculator.h"
7
8  int main(int argc,char** args){
9    Main kit(argc,args);
10   Window window;
11   B apfel(new KeyApfel());
12   B clock(new DigitalClock());
13   B ping(new Ping());
14   B calc(new Calculator());
15   B content = apfel|clock|(ping,calc);
16   Widget* c= (content.w);
17   window.add(*c);
18   window.show_all();
19   Main::run(window);
20   return 0;
21 }
```

Kapitel 8

OpenGL: 3 dimensionale Graphiken

Die umfangreichste Standardwerk zu OpenGL ist das sogenannte *redbook*[WBN+97]. OpenGL kommt mit zwei zusätzlichen Bibliotheken: Eine Bibliothek nützlicher graphischer Objekte mit Namen GLU [CFH+98] und eine systemunabhängige GUI Bibliothek mit Namen GLUT [Kil96]. Ein kleines Tutorial für die Portierung von OpenGL für Haskell habe ich aufs Netz gestellt [Pan03a].

8.0.2 Eine Zustandsmaschine

8.0.3 Eine kleine GUI Bibliothek: GLUT

In der Bibliothek OpenGL existieren keine Funktionen zum Programmieren graphischer Benutzeroberflächen. Um aber eine minimale Benutzeroberfläche schreiben zu können, existiert die Bibliothek GLUT. Sie ist für kleine bis maximal mittelgroße Programme geeignet.

Ein GLUT-Programm hat im Wesentlichen immer die gleiche Struktur. Das GLUT System wird initialisiert, ein Basismodus für dieses System wird initialisiert, ein Fenster deklariert, diesem Fenster eine OpenGL Zechenfunktion übergeben und schließlich das Gesamtsystem gestartet.

In der Zeichenfunktion finden sich beliebig viele OpenGL-Befehle und sie wird in der Regel mit dem Befehl `flush()` bzw. bei doppel gepufferten Modus mit `glutSwapBuffers()` beendet.

Beispiel:
In diesem kleinen Beispiel wird ein Fenster geöffnet, dessen Zeichenfunktion lediglich den Fensterhintergrund mit einer definierten Löschfarbe löscht.

HelloWindow.cpp

```
1 #include <GL/glut.h>
2
3 void display(void){
4   glClearColor(0.0, 0.0, 0.0, 0.0);
5   glClear(GL_COLOR_BUFFER_BIT);
6   glFlush();
```

```
7  }
8
9  int main (int argc, char **argv){
10   glutInit(&argc, argv);
11   glutInitDisplayMode(GLUT_RGB);
12   glutCreateWindow("Hello Window");
13   glutDisplayFunc(display);
14   glutMainLoop();
15 }
```

Glut bietet über das Öffnen von Fenster eine ganze Reihe weiterer Bibliotheksfunktionen an. Insbesondere auch die Möglichkeit auf Tastatureingaben zu reagieren.

8.0.4 Vertexes und Figuren

Die Basiseinheit in OpenGL sind Punkte in einem 3-dimensionalen Raum, sogenannte *Vertexes*. Ein Vertex kann über den Funktionsaufruf `glVertex3f` definiert werden. In diesem Namen zeigt der Präfix `gl` an, daß es sich um eine OpenGL-Funktion handelt, die `3`, daß es drei Koordinaten gibt und daß `f`, daß diese Koordinaten als `float` Zahl übergeben werden. Die einzelnen Vertexes können dann zu unterschiedlichen vorgegebenen Figuren verbunden werden.

Wir definieren eine kleine Bibliothek, die es uns ermöglicht ein paar vorgegebene Vertexes als unterschiedliche Figuren in einen Fenster zu zeichenen. Hierzu definieren wir eine Funktion `moin`, die als zusätzlichen Parameter eine Figur enthält[1].

SomePoints.h

```
1  #include <GL/glut.h>
2  void moin(int argc, char **argv,int shape);
```

In der Implementierungsdatei sehen wir eine Reihe von Vertexes vor. In einer Variablen wird gespeichert, zu was für eine Figur diese Punkte verbunden werden sollen:

SomePoints.cpp

```
1  #include "SomePoints.h"
2  int shape=GL_POINTS;
3
4  void points(){
5     glVertex3f(0.2,-0.4,0);
6     glVertex3f(0.46,-0.26,0);
7     glVertex3f(0.6,0,0);
8     glVertex3f(0.6,0.2,0);
9     glVertex3f(0.46,0.46,0);
10    glVertex3f(0.2,0.6,0);
11    glVertex3f(0.0,0.6,0);
12    glVertex3f(-0.26,0.46,0);
13    glVertex3f(-0.4,0.2,0);
14    glVertex3f(-0.4,0,0);
```

[1] `moin` nehme ich als Norddeutscher gerne für Funktionen, die fast die Methode `main` sind, lediglich noch mehr Parameter haben.

```
15     glVertex3f(-0.26,-0.26,0);
16     glVertex3f(0,-0.4,0);
17  }
```

Die eigentliche Zeichenfunktion kappselt diese Punkte mit den Befehlen `glBegin` und `glEnd`. Der Funktion `glBegin` wird dabei als Parameter mitgegeben, zu was für einer Figur die Punkte verbunden werden sollen.

SomePoints.cpp

```
18  void pointsAsShape(){
19    glClear(GL_COLOR_BUFFER_BIT );
20    glColor3f (0.0, 0.0, 0.0);
21    glBegin(shape);
22      points();
23    glEnd();
24    glFlush();
25  }
```

Und schließlich noch die Pseudohauptmethode, die noch zusätzlich die zu erzeugende Figur als Parameter enthält

SomePoints.cpp

```
26  void moin(int argc, char **argv,int s){
27    shape=s;
28    glutInit(&argc, argv);
29    glutInitDisplayMode(GLUT_RGB);
30    glutCreateWindow("Hello");
31    glClearColor(1.0, 1.0, 1.0, 1.0);
32    glutDisplayFunc(pointsAsShape);
33    glutMainLoop();
34  }
```

Diese Bibliothek können wir benutzen, um die verschiedenen in OpenGL bekannten Figuren zu zeichnen. OpenGL kennt 10 Arten von Figuren, die aus Vertexes gebildet werden können:

- `GL_POINTS` nur als individuelle einzelne Punkte
- `GL_LINES` paarweise werden Linien verunden
- `GL_LINE_STRIP` ein Linienzug
- `GL_LINE_LOOP` ein geschlossener Linienzug. Der letzte Vertex wird wieder mit dem ersten verbunden.
- `GL_TRIANGLES` Tripel werden als Dreiecke verbunden.
- `GL_TRIANGLE_STRIP` zwei Dreiecke teilen sich einen Punkt.
- `GL_TRIANGLE_FAN` Ein Fächer von Dreiecke, die alle einen Punkt gemeinsam haben.
- `GL_QUADS` je vier Punkte als Vierecke verbunden.

- `GL_QUAD_STRIP` Ein Zug von miteinander verbundenen Vierecken.
- `GL_POLYGON` einfache konvexe Polygone.

Mit Hilfe unserer Beispielpunkte aus `SomePoints` lassen sich als einfache Einzeiler Beispiele für alle 10 Figuren schreiben:

HelloPoints.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_POINTS);}
```

HelloLines.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_LINES);}
```

HelloLineStrip.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_LINE_STRIP);}
```

HelloLineLoop.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_LINE_LOOP);}
```

HelloTriangles.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_TRIANGLES);}
```

HelloTriangleStrip.cpp
```
#include "SomePoints.h"
int main (int argc, char **as){moin(argc,as,GL_TRIANGLE_STRIP);}
```

HelloTriangleFan.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_TRIANGLE_FAN);}
```

HelloQuads.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_QUADS);}
```

HelloQuadStrip.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_QUAD_STRIP);}
```

HelloPolygon.cpp
```
#include "SomePoints.h"
int main (int argc, char **argv){moin(argc,argv,GL_POLYGON);}
```

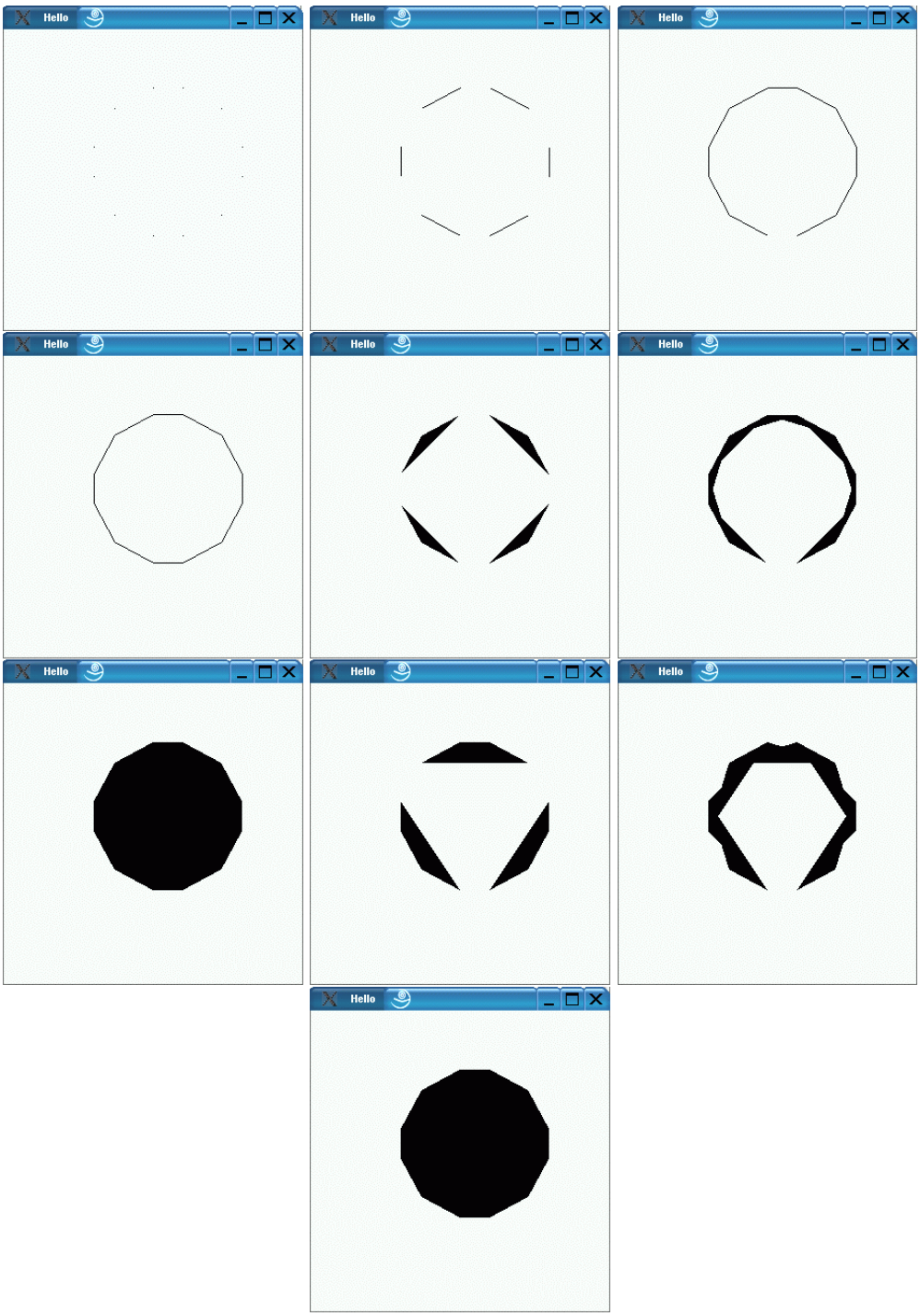

Abbildung 8.1: Alle 10 möglichen Figuren.

Die durch diese Programme erzeugten Fenster sind in Abbildung **??** zu bewundern.

In OpenGL gibt es keine Figuren mit runden Kanten oder Flächen. Diese müssen über Approximation von vielen kleinen geraden Flächen gebildet werden.

Beispiel:
Das folgende Programm zeichnet einen Funktionsgraphen:

Graph.cpp

```
1  #include <GL/glut.h>
2
3  float (*f) (float);
4  float square(float x) {return 3*x*x*x;};
5
6  void display(void){
7    glClearColor(0.0, 0.0, 0.0, 0.0);
8    glClear(GL_COLOR_BUFFER_BIT);
9
10   const int steps =100;
11   const float step = 2/(float)100;
12   glBegin(GL_LINE_STRIP);
13   for (int i = 0; i < steps; i=i+1){
14     const float x = -1+i*step;
15     const float y = f(x);
16     glVertex2f(x,y);
17   }
18   glEnd();
19   glFlush();
20 }
21
22 int main (int argc, char **argv){
23   glutInit(&argc, argv);
24   glutInitDisplayMode(GLUT_RGB);
25   glutCreateWindow("Function Graph Window");
26   f = square;
27   glutDisplayFunc(display);
28   glutMainLoop();
29 }
```

Aufgabe 21 Schreiben Sie eine OpenGL-Funktion
`void circle(int radius)`,
mit der Sie einen Kreis zeichnen können. Benutzen Sie hierzu die Figur `GL_POLYGON`.

8.0.5 Nah und Fern

8.0.6 Vordefinierte Figuren

Im letzten Abschnitt war zu sehen, daß OpenGL nur 10 primitive Arten kennt, wie Figuren gebaut werden können. Kompliziere Figuren, insbesondere runde Flächen sind approximativ aus diesen Grundfiguren zusammenzusetzen. In der Bibliothek GLUT sind eine Reihe von komplexeren Figuren bereits vordefiniert.

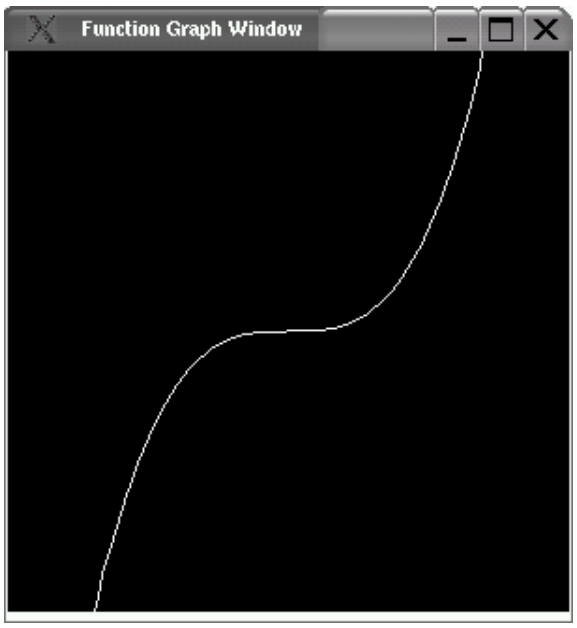

Abbildung 8.2: Der Graph einer Funktion.

- `Sphere`: eine Kugel.
- `Cube`: ein Würfel.
- `Torus`: ein runder Ring (Donut).
- `Icosahedron`: Körper mit 16 dreieckigen Flächen.
- `Octahedron`: Körper mit 8 dreieckigen Flächen.
- `Tetrahedron`: Körper mit 4 dreieckigen Flächen.
- `Dodecahedron`: Körper mit 12 dreieckigen Flächen. [2]
- `Cone`: ein Kegel.
- `Teapot`: eine Teekanne.

Alle diese dreidimesionalen Figuren gibt es einmal als solide geschlossene Form und einmal durch ein Netz dargestellt. Entsprechend gibt es jeweils zwei Funktionen, z.B. `glutWireCube(GLdouble size)` und `glutSolidCube(GLdouble size)` für Würfel.

Beispiel:
In diesem Beispiel wird eine Kugel definiert.

HelloSphere.cpp

```
1 #include <GL/glut.h>
2
3 void display(void){
```

[2]Entspricht in meiner installierten OpenGL Bibliothek nicht der Dokumentation.?

```
4     glClear(GL_COLOR_BUFFER_BIT );
5     glColor3f (1.0, 0.0, 0.0);
6     glutSolidSphere(0.8,10,10);
7     glFlush();
8   }
9
10  int main (int argc, char **argv){
11    glutInit(&argc, argv);
12    glutInitDisplayMode(GLUT_RGB);
13    glutCreateWindow("Hello Cube");
14    glClearColor(1.0, 1.0, 1.0, 1.0);
15    glutDisplayFunc(display);
16    glutMainLoop();
17  }
```

Das Ergebnis einer solchen dreidimensionalen Figur ist auf dem Bildschirm zunächst recht enttäuschend. Man sieht nur einen Kreis.

8.0.7 Bei Licht betrachtet

Um für dreidimensionale Figuren auch einen dreidimensionalen optischen Effekt zu erzeugen, braucht es Licht und Schatten auf einem Körper. OpenGL bietet die Möglichkeit, Lichtquellem zu definieren und für die Flächen einer Figur zu definieren, wie ihr Material auf unterschiedlichen Lichteinfluß reagieren.

Das Lichtmodell von OpenGL ist relativ komplex. Es gibt verschiedene Arten von Lichtquellen (*ambient*, *diffuse*, *specular*), die unterschiedlich stark gesetzt werden können. In diesem Kurzkapitel begnügen wir uns mit dem Beispiel der GLUT-Körper bei Licht betrachtet.

Um für alle 18 vorgefertigten Figuren aus GLUT ein Beispielprogramm zu bekommen, schreiben wir wieder einer einfache Bibliothek, in der eine Lichtquelle definiert wird:

RenderShape.h

```
1  #include <GL/glut.h>
2  int moin (int argc, char **argv,void(*display)(),char* title);
```

RenderShape.cpp

```
1  #include "RenderShape.h"
2
3  void (*shapeFun)();
4
5  void display(){
6    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
7    shapeFun();
8    glFlush();
9  }
10
11 int moin (int argc, char **argv,void(*shape)(),char* title){
12   shapeFun=shape;
13   glutInit(&argc, argv);
```

```
14    glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH);
15
16    glutCreateWindow(title);
17
18    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
19    GLfloat mat_shininess[] = { 50.0 };
20    GLfloat light_position[] = { 1.0, 1.0, -2.5, 0.0 };
21
22    glShadeModel (GL_SMOOTH);
23
24    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
25    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
26    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
27
28
29    glEnable(GL_LIGHTING);
30    glEnable(GL_LIGHT0);
31    glEnable(GL_DEPTH_TEST);
32
33    glClearColor(1.0, 0,0,0);
34    glutDisplayFunc(display);
35    glutMainLoop();
36  }
```

Und jetzt lassen sich auf einfache Weise Tests für alle Figuren erzeugen:

WireSphere.cpp

```
1  #include "RenderShape.h"
2  void f(){glutWireSphere(1,10,10); }
3  int main (int argc,char **argv){moin(argc,argv,f,"WireSphere");}
```

SolidSphere.cpp

```
1  #include "RenderShape.h"
2  void f(){glutSolidSphere(1,10,10); }
3  int main (int argc,char **argv){moin(argc,argv,f,"SolidSphere");}
```

WireCube.cpp

```
1  #include "RenderShape.h"
2  void f(){glutWireCube(1); }
3  int main (int argc,char **argv){moin(argc,argv,f,"WireCube");}
```

SolidCube.cpp

```
1  #include "RenderShape.h"
2  void f(){glutSolidCube(1); }
3  int main (int argc,char **argv){moin(argc,argv,f,"SolidCube");}
```

WireTorus.cpp

```
#include "RenderShape.h"
void f(){glutWireTorus(0.4,0.6,10,30); }
int main (int argc,char **argv){moin(argc,argv,f,"WireTorus");}
```

SolidTorus.cpp

```
#include "RenderShape.h"
void f(){glutSolidTorus(0.4,0.6,10,30); }
int main (int argc,char **argv){moin(argc,argv,f,"SolidTorus");}
```

WireIcosahedron.cpp

```
#include "RenderShape.h"
void f(){glutWireIcosahedron(); }
int main (int ac,char **av){moin(ac,av,f,"WireIcosahedron");}
```

SolidIcosahedron.cpp

```
#include "RenderShape.h"
void f(){glutSolidIcosahedron(); }
int main (int ac,char **av){moin(ac,av,f,"SolidIcosahedron");}
```

WireOctahedron.cpp

```
#include "RenderShape.h"
void f(){glutWireOctahedron(); }
int main (int ac,char **av){moin(ac,av,f,"WireOctahedron");}
```

SolidOctahedron.cpp

```
#include "RenderShape.h"
void f(){glutSolidOctahedron(); }
int main (int ac,char **av){moin(ac,av,f,"SolidOctahedron");}
```

WireTetrahedron.cpp

```
#include "RenderShape.h"
void f(){glutWireTetrahedron(); }
int main (int ac,char **av){moin(ac,av,f,"WireTetrahedron");}
```

SolidTetrahedron.cpp

```
#include "RenderShape.h"
void f(){glutSolidTetrahedron(); }
int main (int ac, char **av){moin(ac,av,f,"SolidTetrahedron");}
```

WireDodecahedron.cpp

```
#include "RenderShape.h"
void f(){glutWireDodecahedron(); }
int main (int ac,char **av){moin(ac,av,f,"WireDodecahedron");}
```

SolidDodecahedron.cpp

```
#include "RenderShape.h"
void f(){glutSolidDodecahedron(); }
int main (int ac,char **av){moin(ac,av,f,"SolidDodecahedron");}
```

WireCone.cpp

```
#include "RenderShape.h"
void f(){glutWireCone(0.3,0.8,20,25); }
int main (int ac,char **av){moin(ac,av,f,"WireCone");}
```

SolidCone.cpp

```
#include "RenderShape.h"
void f(){glutSolidCone(0.3,0.8,20,25); }
int main (int ac,char **av){moin(ac,av,f,"SolidCone");}
```

WireTeapot.cpp

```
#include "RenderShape.h"
void f(){glutWireTeapot(0.6); }
int main (int ac,char **av){moin(ac,av,f,"WireTeapot");}
```

SolidTeapot.cpp

```
#include "RenderShape.h"
void f(){glutSolidTeapot(0.6);}
int main (int ac,char **av){moin(ac,av,f,"SolidTeapot");}
```

Die Ergebnisse dieser Programme befinden sich in den Abbildungen 8.3 bzw. 8.4.

8.0.8 Transformationen

8.0.9 GLUT Tastatureingabe

Beispiel:

CubeWorld.cpp

```
#include <GL/glut.h>

void display(void);
void keyboard(unsigned char, int, int);
void resize(int, int);
void drawcube(int, int, void (*color) ());

int main (int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(600, 600);
  glutInitWindowPosition(40, 40);
```

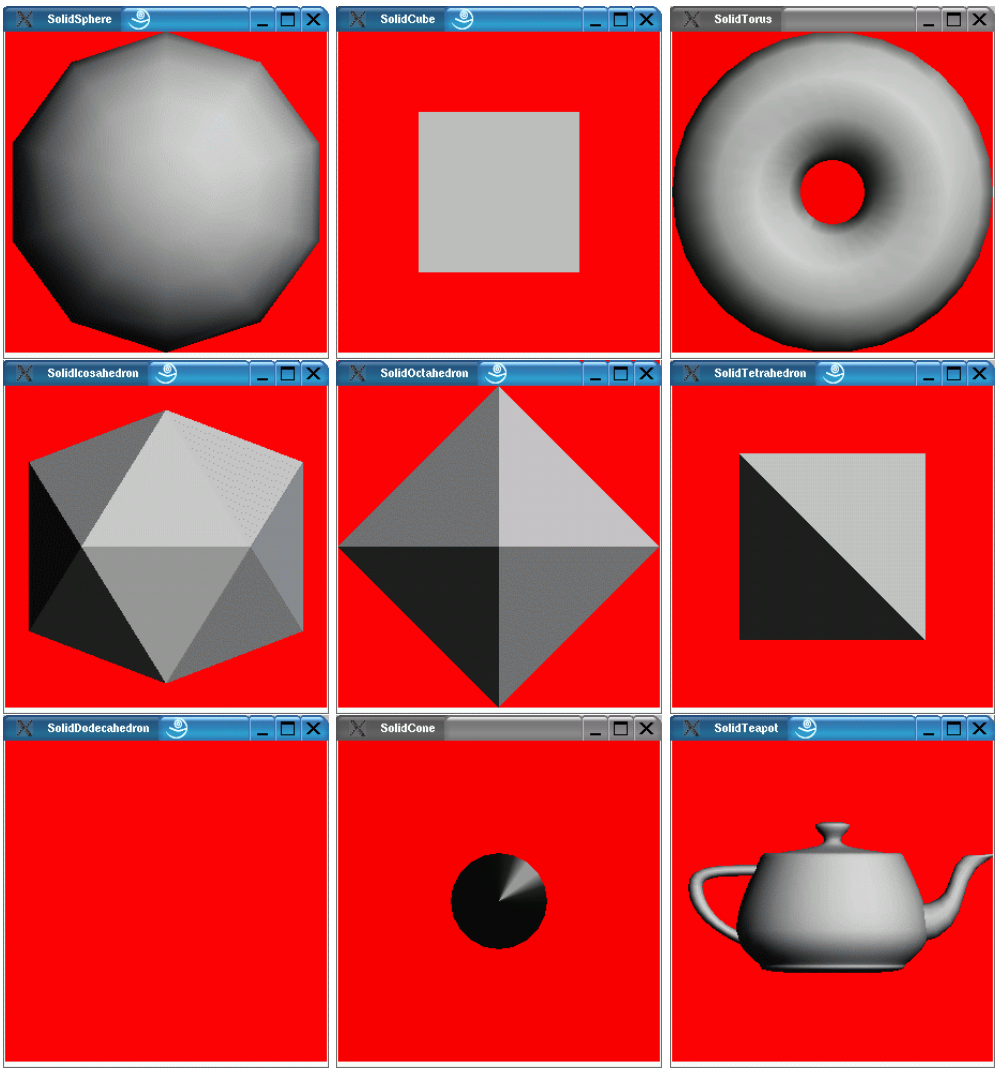

Abbildung 8.3: Alle 9 Glut Figuren mit Oberfläche.

```
13    glutCreateWindow("The Cube World");
14    glClearColor(0.0, 0.0, 0.0, 0.0);
15    glEnable(GL_DEPTH_TEST);
16    glutDisplayFunc(display);
17    glutKeyboardFunc(keyboard);
18    glutReshapeFunc(resize);
19    glutMainLoop();
20  }
21
22  void red()  {glColor3f(1.0,0.0,0.0);}
23  void blue() {glColor3f(0.0,1.0,0.0);}
24  void green(){glColor3f(0.0,0.0,1.0);}
```

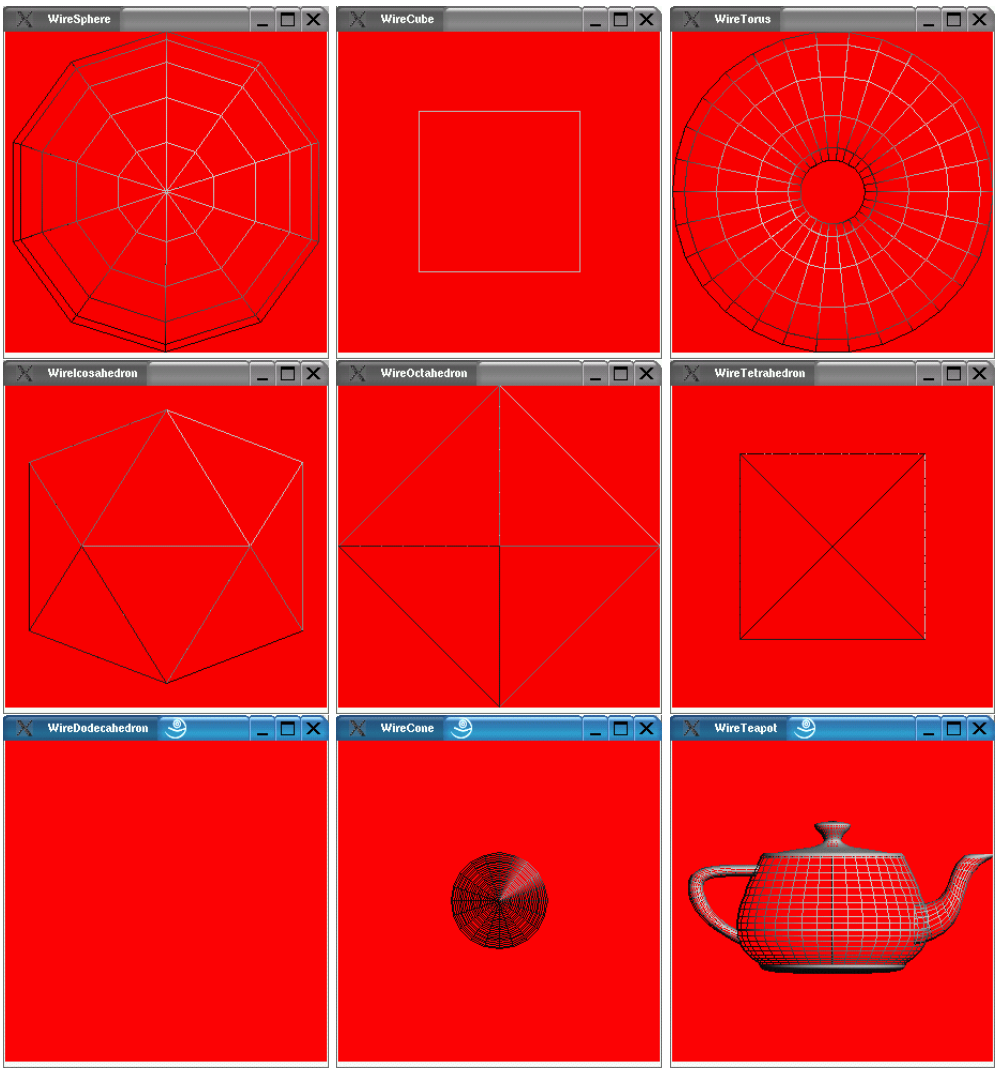

Abbildung 8.4: Alle 9 Glut Figuren als Netzwerk.

```
25 void grey() {glColor3f(0.8, 0.8, 0.8);}
26
27 void drawcube(int x_offset, int z_offset, void (*color)()){
28   color();
29   glPushMatrix();
30     glTranslatef(x_offset,0.0,z_offset);
31     glutSolidCube(10);
32   glPopMatrix();
33 }
34
35 void display(void){
36   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
37
38    /* draw the floor */
39    glPushMatrix();
40      glTranslatef(0,-5,20);
41      glScalef(30,0.1,30);
42      drawcube(0, 0, grey);
43    glPopMatrix();
44
45    /* draw 12 cubes with different colors */
46    drawcube(75, 57, red);
47    drawcube(-65, -12, green);
48    drawcube(50, -50, red);
49    drawcube(-56, 17, blue);
50    drawcube(67, 12, green);
51    drawcube(-87, 32, red);
52    drawcube(-26, 75, blue);
53    drawcube(57, 82, green);
54    drawcube(-3, 12, red);
55    drawcube(46, 35, blue);
56    drawcube(37, -2, green);
57
58    glutSwapBuffers();
59  }
60
61  void keyboard(unsigned char key, int _, int __){
62    switch (key){
63      case 'a': case 'A': glTranslatef(5.0, 0.0, 0.0); break;
64      case 'd': case 'D': glTranslatef(-5.0, 0.0, 0.0);break;
65      case 'w': case 'W': glTranslatef(0.0, 0.0, 5.0); break;
66      case 's': case 'S': glTranslatef(0.0, 0.0, -5.0);break;
67    }
68    display();
69  }
70
71  void resize(int width, int height){
72    glMatrixMode(GL_PROJECTION);
73    glLoadIdentity();
74    gluPerspective(45.0, width / height, 1.0, 400.0);
75    glTranslatef(0.0, -5.0, -150.0);
76    glMatrixMode(GL_MODELVIEW);
77  }
```

Die Klötzchenwelt ist in Abbildung 8.5 zu bewundern.

8.0.10 Beispiel: Tux der Pinguin

In einem abschließenden Beispiel benutzen wir alle oben vorgestellten OpenGL Konstrukte, um das Linuxmaskottchen *Tux* zu zeichnen.

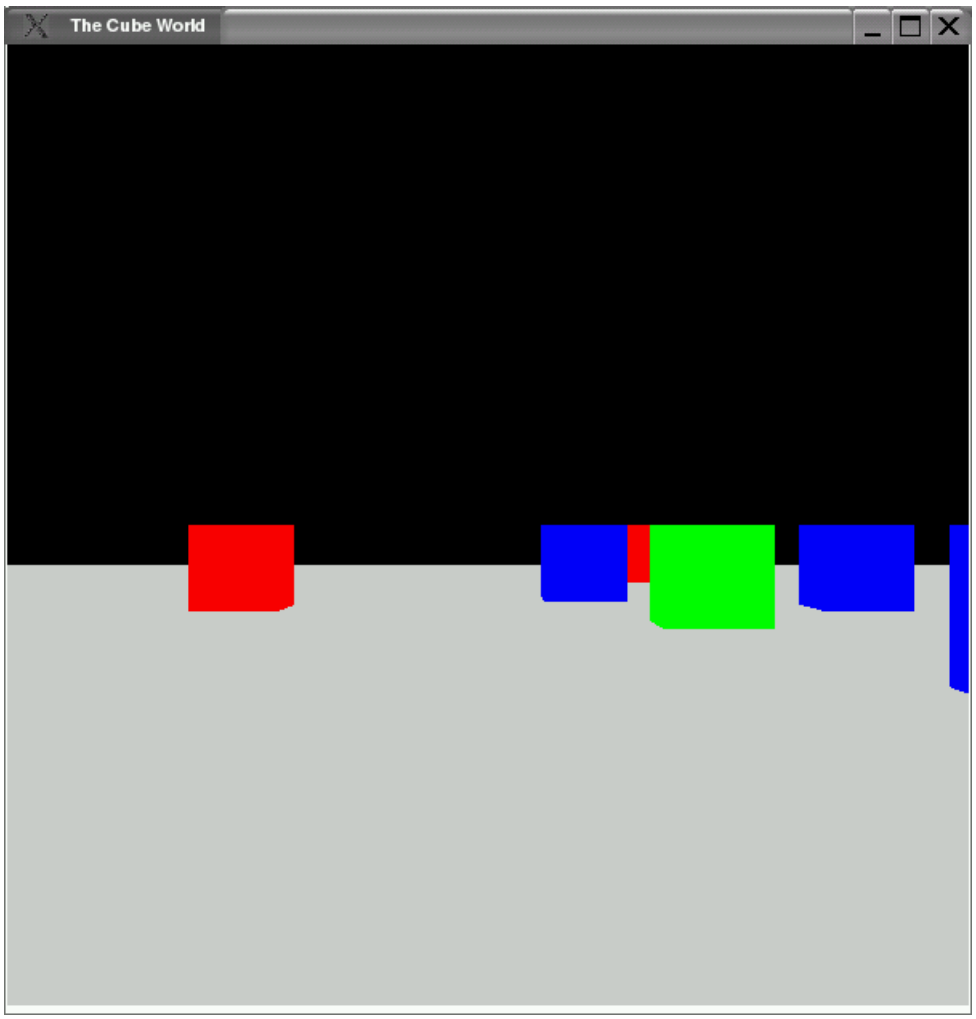

Abbildung 8.5: Dreidimensionale Klötzchenwelt.

Tux.cpp

```
1  #include <GL/glut.h>
2  #include <math.h>
3
4  float zPos = -1;
5
6  void display();
7  void keyboard(unsigned char, int, int);
8  void resize(int, int);
9
10 int main (int argc, char **argv){
11   glutInit(&argc, argv);
12   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
13   glutCreateWindow("Tux");
```

```
14    glClearColor(1.0, 0.0, 0.0, 0.0);
15
16    glEnable(GL_LIGHTING);
17    glEnable(GL_NORMALIZE);
18    glEnable(GL_DEPTH_TEST);
19
20    GLfloat lights[] = { 1, 1, 1, 1 };
21    GLfloat light_position[] = { 1, 0.4, 0.8, 1 };
22    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
23
24    glLightfv(GL_LIGHT0, GL_AMBIENT, lights);
25    glLightfv(GL_LIGHT0, GL_DIFFUSE, lights);
26    glLightfv(GL_LIGHT0, GL_SPECULAR, lights);
27    glEnable(GL_LIGHT0);
28
29    glutDisplayFunc(display);
30    glutKeyboardFunc(keyboard);
31    glutReshapeFunc(resize);
32    glutMainLoop();
33  }
34
35  void keyboard(unsigned char key, int _, int __){
36    switch (key){
37      case '+': zPos=zPos+0.1;break;
38      case '-': zPos=zPos-0.1;break;
39    }
40    display();
41  }
42
43  void resize(int w, int h){
44    glMatrixMode(GL_PROJECTION);
45    glLoadIdentity();
46    float near   = 0.001;
47    float    far    = 40;
48    float    fov    = 90;
49    float    ang    = (fov*3.14)/(360);
50    float    top    = near / ( cos(ang) / sin(ang) );
51    float    aspect = w/h;
52    float    right = top*aspect;
53    glFrustum(-right, right, -top, top, near, far);
54    glMatrixMode(GL_MODELVIEW);
55  }
56
57  void sphere(float r, float xs, float ys,float  zs){
58    glScalef(xs, ys, zs);
59    glutSolidSphere(r,100,100);
60  }
61
62  void rotateZ(float a){glRotatef(a,0,0,1);}
63  void rotateY(float a){glRotatef(a,0,1,0);}
```

```
64  void rotateX(float a){glRotatef(a,1,0,0);}
65
66  void crMat(float rd,float gd,float bd
67            ,float rs,float gs,float bs,float exp){
68    float diffuse [] ={rd, gd, bd,  1.0};
69    glMaterialfv(GL_FRONT,GL_DIFFUSE, diffuse);
70    glMaterialfv(GL_FRONT,GL_AMBIENT, diffuse);
71    float specular [] ={rs, gs, bs,  1.0};
72    glMaterialfv(GL_FRONT,GL_SPECULAR, specular);
73    float shine [] ={exp};
74    glMaterialfv(GL_FRONT,GL_SHININESS,shine);
75  }
76
77  void whitePenguin() {
78    crMat(0.58, 0.58, 0.58, 0.2, 0.2, 0.2, 50.0);
79  }
80  void blackPenguin() {crMat(0.1, 0.1, 0.1,0.5, 0.5, 0.5, 20.0);}
81  void beakColour()   {crMat(0.64, 0.54, 0.06,0.4, 0.4, 0.4, 5);}
82  void nostrilColour(){
83    crMat(0.48039, 0.318627, 0.033725,0.0,0.0,0.0, 1);
84  }
85  void irisColour()   {crMat(0.01, 0.01, 0.01,0.4, 0.4, 0.4, 90.0);}
86
87  void makeBody(){
88    glPushMatrix();
89      blackPenguin();
90      sphere(1, 0.95, 1.0, 0.8);
91    glPopMatrix();
92    glPushMatrix();
93      whitePenguin();
94      glTranslatef( 0, 0, 0.17);
95      sphere(1, 0.8, 0.9, 0.7);
96    glPopMatrix();
97  }
98
99  void createTorso(){
100   glPushMatrix();
101     glScalef( 0.9, 0.9, 0.9);
102     makeBody();
103   glPopMatrix();
104 }
105
106
107 void leftHand(){
108   glPushMatrix();
109     glTranslatef( -0.24, 0, 0);
110     rotateZ(20);
111     rotateX(90);
112     blackPenguin();
113     sphere( 0.5, 0.12, 0.05, 0.12);
```

```
114   glPopMatrix();
115 }
116
117
118 void leftForeArm(){
119   glPushMatrix();
120     glTranslatef(-0.23, 0, 0);
121     rotateZ( 20);
122     rotateX( 90);
123     leftHand();
124     blackPenguin();
125     sphere( 0.66, 0.3, 0.07, 0.15);
126   glPopMatrix();
127 }
128
129 void rightForeArm(){leftForeArm();}
130
131 void leftArm(){
132   glPushMatrix();
133     rotateY(180);
134     glTranslatef( -0.56, 0.3, 0);
135     rotateZ( 45);
136     rotateX( 90);
137     leftForeArm();
138     blackPenguin();
139     sphere( 0.66, 0.34, 0.1, 0.2);
140   glPopMatrix();
141 }
142
143 void rightArm(){
144   glPushMatrix();
145     glTranslatef(-0.56, 0.3, 0);
146     rotateZ( 45);
147     rotateX(-90);
148     rightForeArm();
149     blackPenguin();
150     sphere( 0.66, 0.34, 0.1, 0.2);
151   glPopMatrix();
152 }
153
154 void createShoulders(){
155   glPushMatrix();
156     glTranslatef( 0, 0.4, 0.05);
157     leftArm();
158     rightArm();
159     glScalef( 0.72, 0.72, 0.72);
160     makeBody();
161   glPopMatrix();
162 }
163
```

```
164
165 void createBeak(){
166   glPushMatrix();
167     glTranslatef( 0,-0.205, 0.3);
168     rotateX(10);
169     beakColour();
170     sphere( 0.8, 0.23, 0.12, 0.4);
171   glPopMatrix();
172   glPushMatrix();
173     beakColour();
174     glTranslatef( 0, -0.23, 0.3);
175     rotateX( 10);
176     sphere( 0.66, 0.21, 0.17, 0.38);
177   glPopMatrix();
178 }
179
180
181 void leftIris(){
182   glPushMatrix();
183     glTranslatef( 0.12,-0.045, 0.4);
184     rotateY( 18);
185     rotateZ( 5);
186     rotateX( 5);
187     irisColour();
188     sphere( 0.66, 0.055, 0.07, 0.03);
189   glPopMatrix();
190 }
191
192
193 void rightIris(){
194   glPushMatrix();
195     glTranslatef( -0.12,-0.045, 0.4);
196     rotateY( -18);
197     rotateZ( -5);
198     rotateX( 5);
199     irisColour();
200     sphere( 0.66, 0.055, 0.07, 0.03);
201   glPopMatrix();
202 }
203
204 void leftEye(){
205   glPushMatrix();
206     glTranslatef( 0.13,-0.03, 0.38);
207     rotateY( 18);
208     rotateZ( 5);
209     rotateX( 5);
210     whitePenguin();
211     sphere( 0.66, 0.1, 0.13, 0.03);
212   glPopMatrix();
213 }
```

```
214
215 void rightEye(){
216   glPushMatrix();
217     glTranslatef( -0.13,-0.03, 0.38);
218     rotateY( -18);
219     rotateZ( -5);
220     rotateX( 5);
221     whitePenguin();
222     sphere( 0.66, 0.1, 0.13, 0.03);
223   glPopMatrix();
224 }
225
226 void createEyes(){
227   leftEye();
228   leftIris();
229   rightEye();
230   rightIris();
231 }
232
233 void createHead(){
234   glPushMatrix();
235     glTranslatef( 0, 0.3, 0.07);
236     createBeak();
237     createEyes();
238     rotateY( 90);
239     blackPenguin();
240     sphere( 1, 0.42, 0.5, 0.42);
241   glPopMatrix();
242 }
243
244 void createNeck(){
245   glPushMatrix();
246     glTranslatef( 0, 0.9, 0.07);
247     createHead();
248     rotateY(90);
249     blackPenguin();
250     sphere( 0.8, 0.45, 0.5, 0.45);
251     glTranslatef( 0, -0.08, 0.35);
252     whitePenguin();
253     sphere( 0.66, 0.8, 0.9, 0.7);
254   glPopMatrix();
255 }
256
257 void footBase(){
258   glPushMatrix();
259     sphere( 0.66, 0.25, 0.08, 0.18);
260   glPopMatrix();
261 }
262
263
```

```
264 void toe1(){
265   glPushMatrix();
266     glTranslatef(-0.07, 0, 0.1);
267     rotateY( 30);
268     sphere(0.66, 0.27, 0.07, 0.11);
269   glPopMatrix();
270 }
271
272 void toe2(){
273   glPushMatrix();
274     glTranslatef(-0.07, 0, -0.1);
275     rotateY (-30);
276     sphere( 0.66,  0.27, 0.07, 0.11);
277   glPopMatrix();
278 }
279
280 void toe3(){
281   glPushMatrix();
282     glTranslatef(-0.08, 0, 0);
283     sphere( 0.66,  0.27, 0.07, 0.10);
284   glPopMatrix();
285 }
286
287 void foot(){
288   glPushMatrix();
289     glScalef(  1.1, 1.0, 1.3);
290     beakColour();
291     footBase();
292     toe1();
293     toe2();
294     toe3();
295   glPopMatrix();
296 }
297
298 void rightFoot(){
299   glPushMatrix();
300     glTranslatef( 0,-0.09, 0);
301     rotateY( 180);
302     foot();
303   glPopMatrix();
304 }
305
306 void leftFoot(){
307   glPushMatrix();
308     glTranslatef( 0,-0.09, 0);
309     rotateY( 100);
310     foot();
311   glPopMatrix();
312 }
313
```

```
314
315 void leftCalf(){
316   glPushMatrix();
317     glTranslatef( 0,-0.21, 0);
318     rotateY( 90);
319     leftFoot();
320     beakColour();
321     sphere( 0.5, 0.06, 0.18, 0.06);
322   glPopMatrix();
323 }
324
325 void rightCalf(){
326   glPushMatrix();
327     glTranslatef( 0,-0.21, 0);
328     rightFoot();
329     beakColour();
330     sphere( 0.5, 0.06, 0.18, 0.06);
331   glPopMatrix();
332 }
333
334 void hipBall(){
335   glPushMatrix();
336     blackPenguin();
337     sphere( 0.5, 0.09, 0.18, 0.09);
338   glPopMatrix();
339 }
340
341 void leftTigh(){
342   glPushMatrix();
343     rotateY( 180);
344     glTranslatef(-0.28,-0.8, 0);
345     rotateY( 110);
346     hipBall();
347     leftCalf();
348
349     rotateY (-110);
350     glTranslatef( 0,-0.1, 0);
351     beakColour();
352     sphere( 0.5, 0.07, 0.3, 0.07);
353   glPopMatrix();
354 }
355
356 void rightTigh(){
357   glPushMatrix();
358     glTranslatef(-0.28,-0.8, 0);
359     rotateY( -110);
360     hipBall();
361     rightCalf();
362
363     glTranslatef( 0,-0.1, 0);
```

```
364     beakColour();
365     sphere( 0.5, 0.07, 0.3, 0.07);
366   glPopMatrix();
367 }
368
369 void createTail(){
370   glPushMatrix();
371     glTranslatef( 0,-0.4,-0.5);
372     rotateX (-60);
373     glTranslatef( 0, 0.15, 0);
374     blackPenguin();
375     sphere( 0.5, 0.2, 0.3, 0.1);
376   glPopMatrix();
377 }
378
379 void tux(){
380   glPushMatrix();
381     glScalef( 0.35, 0.35, 0.35);
382     rotateY (-180);
383     createTorso();
384     createShoulders();
385     createNeck();
386     leftTigh();
387     rightTigh();
388     createTail();
389   glPopMatrix();
390 }
391
392 void display(){
393   glLoadIdentity();
394   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
395   gluLookAt(0,0,zPos,0,0,0,0,1,0);
396   tux();
397   glutSwapBuffers();
398 }
```

Der so gezeichnete kleine Kerl ist in Abbildung 8.6 zu sehen.

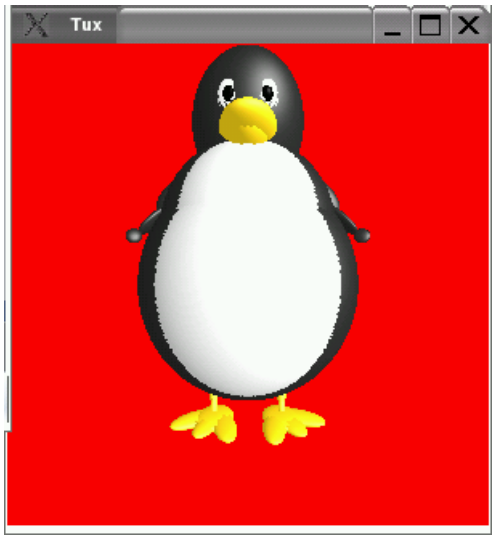

Abbildung 8.6: Tux der Pinguin.

Anhang A

Gesammelte Aufgaben

Aufgabe 1 Kopieren Sie sich den Quelltextordner dieses Skripts und rufen Sie die darin enthaltene obige Makefile-Datei auf. Rufen Sie dann `make` direkt nocheinmal auf. Ändern Sie anschließend die Datei `ReadRoman.cpp` und starten Sie erneut `make`. Was beobachten Sie in Hinblick auf `TestReadRoman.o`. Starten Sie schließlich `make` mit dem Ziel `love`.

Aufgabe 2 Laden Sie sich die kompletten Quelltextdateien dieses Skriptes von der Webseite und generieren Sie mit `Doxygen` eine Dokumentation für diese.

Aufgabe 3 Betrachten Sie, die für die Funktionen `f` und `add` in der letzten Aufgabe generierte Dokumentation.

Aufgabe 4 Beschäftigen Sie sich mit der Dokumentation von *Doxygen* und testen einige der darin beschriebenen Möglichkeiten der Dokumentation.

Aufgabe 5 Übersetzen Sie das Programm mit der `-g`-Option des `gcc` und lassen Sie das Programm ausführen.

Aufgabe 6 Spielen Sie die obige Session des Debuggers einmal nach und experimentieren sie mit den verschiedenen Befehlen des Debuggers.

Aufgabe 7 Üben Sie den Umgang mit dem Debugger `ddd` anhand des Programms `TestReadRoman`.

Aufgabe 8 Schreiben Sie für die vier Karteikarten in der Modellierung eines Bibliotheksystems entsprechende Klassen mit den entsprechenden Feldern.

Aufgabe 9 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

- Personen mit Namen, Vornamen, Straße, Ort und Postleitzahl.
- Bücher mit Titel und Autor.
- Datum mit Tag, Monat und Jahr.
- Buchausleihe mit Ausleiher, Buch und Datum.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine

Methode `std::string toString()` mit dem Ergebnistyp `std::string`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.

c) Schreiben Sie eine Hauptmethode, in der Sie Objekte für jede der obigen Klassen erzeugen und die Ergebnisse der `toString`-Methode auf den Bildschirm ausgeben.

Aufgabe 10 (4 Punkte) In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden C++ Klassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

ButtonLogic.h

```
#ifndef _BUTTON_LOGIC__H
#define _BUTTON_LOGIC__H
#include <string>
class ButtonLogic {
 public:
  virtual std::string getDescription();
  virtual std::string eval(std::string x);
};
#endif

```

ButtonLogic.cpp

```
#include "ButtonLogic.h"
std::string ButtonLogic::getDescription(){return "Drück mich";}
std::string ButtonLogic::eval(std::string x){return x;}
```

Dialogue.h

```
#include <gtkmm.h>
#include "ButtonLogic.h"

class Dialogue : public Gtk::Window{

public:
  Dialogue(ButtonLogic* l);

protected:
```

```
10    ButtonLogic* logic;
11    virtual void on_button_clicked();
12    Gtk::Entry entry;
13    Gtk::Button button;
14    Gtk::Label  label;
15    Gtk::VBox  box;
16  };
```

Dialogue.cpp

```
1   #include "Dialogue.h"
2   #include <iostream>
3   #include <string>
4
5   Dialogue::Dialogue(ButtonLogic* l)
6     : logic(l)
7     , button(l->getDescription())
8     , label(""){
9     set_border_width(10);
10
11    button.signal_clicked()
12          .connect(sigc::mem_fun
13                (*this, &Dialogue::on_button_clicked));
14
15    box.add(entry);
16    box.add(button);
17    box.add(label);
18    add(box);
19    show_all();
20  }
21
22  void Dialogue::on_button_clicked(){
23    label.set_text(logic->eval(entry.get_text()));
24  }
```

TestDialogue.cpp

```
1   #include <gtkmm/main.h>
2   #include "Dialogue.h"
3
4   int main (int argc, char *args[]){
5     Gtk::Main kit(argc,args);
6     Dialogue dialogue(new ButtonLogic());
7     Gtk::Main::run(dialogue);
8   }
9
```

a) Übersetzen Sie die drei obigen Klassen und starten Sie das Programm. Hierzu müssen Sie den Compiler beim Übersetzen der C-Quelltexte mit dem Kommandozeilenargument ‘`pkg-config gtkmm-2.4 --cflags`‘ und beim Linken mit ‘`pkg-config gtkmm-2.4 --libs`‘ aufrufen.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, daß der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie römische Zahlen in arabische Zahlen umwandeln können. Benutzen Sie hierzu die Funktionen `romanToInt` aus dem Einführungskapitel und `itos` aus dem nächsten Kapitel. Testen Sie Ihr Programm.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie arabische Zahlen in römische Zahlen umwandeln können. Testen Sie Ihr Programm.

e) Schreiben Sie jetzt ein Guiprogramm, daß eine Zahl aus ihrer Darstellung zur Basis 10 in eine Darstellung zur Basis 2 umwandelt. Testen Sie.

Aufgabe 11 In dieser Aufgabe sollen Sie wie in der letzten Aufgabe eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben. Dieses Mal übergeben Sie aber die Anwendungslogik dirket als Funktionszeiger und nicht als Objekt.

Gegeben sei die folgenden C++ Klasse, wobei Sie die Klasse `FDialogue` nicht zu analysieren oder zu verstehen brauchen:

FDialogue.h

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

typedef std::string (*ButtonFunction)(std::string);

class FDialogue : public Gtk::Window{
public:
  FDialogue(ButtonFunction f,std::string description);

protected:
  ButtonFunction eval;
  virtual void on_button_clicked();
  Gtk::Entry entry;
  Gtk::Button button;
  Gtk::Label  label;
  Gtk::VBox  box;
};
```

FDialogue.cpp

```
#include "FDialogue.h"
#include <iostream>
#include <string>

FDialogue::FDialogue(ButtonFunction f,std::string description)
  : eval(f)
  , button(description)
```

```
  , label(""){
  set_border_width(10);
  button.signal_clicked()
   .connect(sigc::mem_fun(*this,&FDialogue::on_button_clicked));
  box.add(entry);
  box.add(button);
  box.add(label);
  add(box);
  show_all();
}

void FDialogue::on_button_clicked(){
  label.set_text(eval(entry.get_text()));
}
```

TestFDialogue.cpp

```
#include <gtkmm/main.h>
#include "FDialogue.h"
#include "ReadRoman.h"
#include "ToString.h"

std::string fromRoman(std::string x){return itos(romanToInt(x));}
int main (int argc, char *args[]){
  Gtk::Main kit(argc,args);
  FDialogue dialogue(fromRoman,"nach arabisch");
  Gtk::Main::run(dialogue);
}

```

Schreiben Sie mit Hilfe von `FDialogue` eine GUI-Applikation, die es ermöglicht Dezimalzahlen in Oktalzahlen umzuwandeln.

Aufgabe 12 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

a) `std::string last()`: gibt das letzte Element der Liste aus.

b) `L1 concat(L1 other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `std::string elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} elementAt(Cons(x, xs), 1) &= x \\ elementAt(Cons(x, xs), n+1) &= elementAt(xs, n) \end{aligned}$$

Aufgabe 13 Schreiben Sie eine Funktion zur Funktionakomposition 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.

Aufgabe 14 (2 Punkte) 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:

fold(xs,l,op,st) = op(...op(op(op(st,xs[0]),xs[1]),xs[2])...,xs[l-1])

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

fold($\{x_0, x_1,, \ldots, x_{l-1}\}$,l,op,st) = st op x_0 op x_1 op x_2 op ... 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){return x>y.length()?x:y.length();}
9
10 int main(){
11   int xs [] = {1,2,3,4,5,4,3,2,1};
12   std::cout << fold(xs,9,add,0) << std::endl;
13   std::cout << fold(xs,9,mult,1)<< std::endl;
14   std::cout << fold(xs,9,gr,0)  << std::endl;
15   std::string ys [] = {"john","paul","george","ringo","stu"};
16   std::cout << fold(ys,5,strLaenger,0)  << std::endl;
17 }
```

Schreiben Sie ein paar zusätzliche Beispielaufrufe von `fold`.

Aufgabe 15 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester vorgestellt wahrscheinlich 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.

Aufgabe 16 Ergänzen Sie die obige Listenklasse um weitere Listenoperatoren.

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

Testen Sie ihre Implementierung an einigen Beispielen.

Aufgabe 17 **(optional)** Schreiben Sie weitere Funktionen, die Objekte der Klasse `LazyList` erzeugen und testen Sie diese:

a) Schreiben Sie eine generische Funktion:

```
1 template <typename A>
2 LazyList<A>* repeat(A x)
```

die eine Liste erzeugt, in der unendlich oft Wert `x` wiederholt wird.

b) Schreiben Sie eine Funktion, die eine unendliche Liste von Pseudozufallszahlen erzeugt. Benutzen sie hierzu folgende Formel:
$z_{n+1} = (91 * z_n + 1) \bmod 347$

Aufgabe 18 Sie haben bereits einmal eine Funktion `fold` implementiert. Jetzt sollen Sie die Funktion noch einmal allgemeiner schreiben.

a) Implementieren Sie jetzt die Funktion `fold` zur Benutzung für allgemeine Behälterklassen im STL Stil entsprechend der Signatur:

STLFold.h

```
template <typename Iterator,typename BinFun,typename EType>
EType fold(Iterator begin,Iterator end,BinFun f,EType start);
```

b) Testen Sie Ihre Implementierung mit folgendem Programm:

STLFoldTest.cpp

```
#include <vector>
#include <string>
#include <functional>
#include <iostream>
#include "STLFold.h"

int addLength(int x,std::string s){return x+s.length();}

std::string addWithSpace(std::string x,std::string y){
    return x+" "+y;
}

int main(){
  int xs []={1,2,3,4,5};
  int result;
  result = fold(xs,xs+5,std::plus<int>(),0);
  std::cout << result << std::endl;
  result = fold(xs,xs+5,std::multiplies<int>(),1);
  std::cout << result << std::endl;

  std::vector<std::string> ys;
  ys.insert(ys.end(),std::string("friends"));
  ys.insert(ys.end(),std::string("romans"));
  ys.insert(ys.end(),"contrymen");
  result = fold(ys.begin(),ys.end(),addLength,0);
  std::cout << result << std::endl;

  std::string erg
    = fold(ys.begin(),ys.end()
          ,std::plus<std::string>(),std::string(""));
  std::cout << erg << std::endl;

  erg=fold(ys.begin(),ys.end(),addWithSpace,std::string(""));
  std::cout << erg << std::endl;
}
```

c) Schreiben Sie weitere eigene Tests.

Aufgabe 19 (6 Punkte) Schreiben Sie ein Program `getcode` mit drei Kommandozeilenparametern:

```
1 getcode xmlFileName className langName
```

Es soll die XML-Datei `xmlFileName` lesen und dafür ein DOM-Objekt erzeugen. Dann soll es von jedem Element mit Tagnamen `code`, das für das Attribut `class` den Wert `className` und für das Attribut `lang` den Wert `langName` hat, den in diesem Element enthaltenen Text extrahieren und in eine Datei `className.langName` schreiben.

Als Beispieldokument befindet sich der Quelltext dieses Skripts (`http://www.panitz.name/cpp/student.xml`) auf der Webseite dieser Vorlesung.

Der Aufruf `getcode student.xml Li h` soll also die Datei `Li.h` schreiben mit den Code, der hierzu in `student.xml` zu finden ist.

Als Beispiel, wie eine Textdatei als Datenstrom geschrieben werden kann, betrachten Sie folgendes kleine Programm:

DateiTest.cpp

```
1 #include <fstream>
2
3 int main(){
4     std::fstream datei;
5     datei.open("testdatei",std::ios::out);
6     datei << "etwas text"<<std::endl;
7     datei << "in die datei schreiben"<<std::endl;
8     datei.close();
9 }
```

Aufgabe 20 Schreiben Sie ein Programm, das eine Analoguhr auf dem Bildschirm realisiert. Die Uhr soll dabei Sekunden, Minuten und Stundenzeiger haben.

Aufgabe 21 Schreiben Sie eine OpenGL-Funktion
`void circle(int radius)`,
mit der Sie einen Kreis zeichnen können. Benutzen Sie hierzu die Figur `GL_POLYGON`.

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Arn04] Arnaud Le Hors and Philippe Le Hégaret and Lauren Wood and Gavin Nicol and Jonathan Robie and Mike Champion and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification Version 1.0. W3C Recommendation, April 2004. `http://www.w3.org/TR/P3P/`.

[BCK+01] G. Bracha, N. Cohen, Chr. Kemper, St. Marx, S. E. Panitz, D. Stoutamire, K. Thorup, and Ph. Wadler. Adding generics to the java programming language: Participant draft specification. Java Community Process 14, 4 2001.

[CFH+98] Norman Chin, Chris Frazier, Paul Ho, Zicheng Lui, and Kevin P. Smith. The OpenGL Graphics System Utility Library, 1998. `ftp://ftp.sgi.com/opengl/doc/opengl1.2/glu1.3.ps`.

[Gru01] Ulrich Grude. C++ für Java-Programmierer. `www.tfh-berlin.de/~grude/SkriptCPPSS01.pdf`, 2001. Skript, TFH Berlin.

[Ing78] Daniel H. H. Ingalls. Smalltalk-76 programming system. In *Proc. 5th Annual ACM Symposium on Principles of Programming Languages*, 1978.

[Kil96] Mark J. Kilgard. The OpenGL Utility Toolkit (GLUT), 1996. `www.opengl.org/developers/documentation/glut/glut-3.spec.ps`.

[NH02] Alexander Niemann and Stefan Heitsiek. *Das Einsteigerseminar: C++ Objektorientierte Programmierung.* verlag moderne industrie Buch AG & Co KG, Bonn, 2002.

[Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan03a] Sven Eric Panitz. HOpenGL – 3D Graphics with Haskell, A small Tutorial. Online Script, TFH Berlin, 2003. `www.panitz.name/hopengl/index.html`.

[Pan03b] Sven Eric Panitz. Programmieren I. Skript zur Vorlesung, 2. revidierte Auflage, 2003. `www.panitz.name/prog1/index.html`.

[Pan03c] Sven Eric Panitz. Programmieren II. Skript zur Vorlesung, TFH Berlin, 2003. `www.panitz.name/prog2/index.html`.

[Pan03d] Sven Eric Panitz. Programmieren III. Skript zur Vorlesung, TFH Berlin, 2003. `www.panitz.name/prog3/index.html`.

[Pan04] Sven Eric Panitz. Programmieren IV. Skript zur Vorlesung, TFH Berlin, 2004. `www.panitz.name/prog4/index.html`.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[T. 04] T. Bray, and al. XML 1.1. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml11`.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer, 1985.

[WBN+97] Mason Woo, OpenGL Architecture Review Board, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide 3rd Edition*. Addison-Wesley, Reading, MA, 1997.

[Wie99] Greg Wiegand. Teach Yourself C++ in 21 Days, 1999. `www.mcp.com/sams`.