Programmieren III
(Entwurf)

WS 03/04

Prof. Dr. Sven Eric Panitz

TFH Berlin

Version 17. März 2004

Die vorliegende Fassung des Skriptes ist ein roher Entwurf für die Vorlesung des kommenden Semesters 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 WS 03/04 vollkommen neu. Es stellt somit eine Mitschrift der Vorlesung dar. 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.

Die Vorlesung setzt die Kerninhalte der Vorgängervorlesungen [Pan02] und [Pan03b] voraus.

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. Der XML-Quelltext verweist direkt auf ein XSLT-Skript, das eine HTML-Darstellung erzeugt, so daß ein entsprechender Browser mit XSLT-Prozessor die XML-Datei direkt als HTML-Seite darstellen kann. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Einführung

Ziel der Vorlesung

Mit den Grundtechniken der Programmierung am Beispiel von Java wurde in der Vorgängervorlesungen vertraut gemacht. Jetzt ist es an der Zeit, die gelernten Konzepte in einer anderen objektorientierten Programmiersprache umzusetzen. Hierzu wird die Sprache C++ benutzt. C++ abstrahiert viel weniger als Java von der real zugrundeliegenden Maschine; auch ist der Sprachumfang von C++ wesentlich umfangreicher; eher ein Nachteil, insbesondere für Neulinge, die mit einer Vielzahl konkurrierender Konzepte in ein und derselben Sprache konfrontiert werden.

In einer zweistündigen einsemestrigen Vorlesung kann nicht die ganze Programmiersprache C++ gelehrt werden. Wir werden uns darauf konzentrieren, einige interessante Unterschiede zu Java herauszuarbeiten und die wesentlichen Konzepte zu zeigen. Mit Chance entsteht eine Einführung C++-kompakt, oder wie die Angelsachsen es gerne nennen: *in a nutshell*. Für Detailfragen 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]. Eine schöne Einführung in C ohne C++ findet sich auch im zweiten Teil des Skripts Programmieren II für den Studiengang Technische Informatik von Frau Ripphausen-Lipa [RL03]. Viele C++ Kurse finden sich im Netz. Beispielhaft sei [Wie99] hier erwähnt.

Aufbau der Vorlesung

Aus Java bekannte Konstrukte wie zusammengestzte Befehle, Schleifen etc. werden nicht erklärt werden. Die Vorlesung konzentriert sich auf die Besonderheiten von C. Hierzu gehören insbesondere imperative Konzepte und Zeiger. Daher werden im ersten Kapitel C++ Konstrukte, die nicht der Objektorientierung zugehörig sind, vorgestellt. Dabei werden insbesondere auch Konstrukte genauer unter die Lupe genommen, die gleich bekannten Konzepten aus Java scheinen, aber ein anderes Verhalten an den Tag legen. Im zweiten Kapitel wird dann vorgestellt, in wieweit sich die in Java eingeübten objektorientierten Techniken in C++ umsetzen lassen. Schließlich ist ein drittes Kapitel geplant, in dem ein Beispiel für graphische und GUI Programmierung vorgestellt wird. Hier könnten GTK und/oder OpenGL zum Einsatz kommen.

Im Laufe dieses Skriptes werden wir etwas lax mit der Bezeichnung C sein. Wenn wir von C sprechen werden wir in der Regel auch C++ meinen. Nur in speziellen Fällen wird auf eine genaue Unterscheidung zwischen C und C++ Wert gelegt.

Kapitel 1

Die Programmiersprache C++

1.1 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[1] 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[**?**]. 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.

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

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

Java im Gegensatz versucht mehr von der Maschine zu abstrahieren und ein logisch konsistentes Bild nach außen zu präsentieren. In Java ist nicht alles erlaubt, was gefällt. Allerdings steht Java syntaktisch sehr in der Tradition von C. Daher werden wir als Javaprogrammierer kaum Schwierigkeiten haben, viele der C Konstrukte zu lesen. Schleifenkonstrukte und ähnliche zusammengesetzte Befehle sehen in beiden Sprachen fast vollkommen identisch aus. Trotzdem ist, wie immer zwischen zwei äußerlich ähnlichen Sprachen, vor falschen Freunden zu warnen. Als falsche Freunde bezeichnet man Wörter, die man aus einer Sprache kennt, die aber in einer anderen Sprache eine andere Bedeutung haben. So heißt das englische *eventually* eben nicht *eventuell* sondern *schließlich* oder das englische *pregnant* hat eine Bedeutung, die selten in einem Fachtext der Informatik gebraucht wird.

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 vollkommen ungeeignet; daher fanden von jeher Programmiereinführungen auf andere Sprachen statt: Pascal, Modula zeitweise oft Algol68 und Scheme oder seit wenigen Jahren Java.

1.2 Compiler und Werkzeuge

1.2.1 Editoren

Ähnlich wie für Java gibt es natürlich auch für C eine Vielzahl mächtiger integrierter Entwicklungsumgebungen. Für den Anfänger empfiehlt es sich zunächst einmal auf diese 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.

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*, der sowohl für Windows- als auch für unixartige Betriebssysteme zur Verfügung steht.

1.2.2 Compiler

Anders als in Java, wo fast ausschließlich ein Compiler benutzt wird, nämlich der Compiler `javac` der Firma Sun, stehen für C 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.

1.3 Kompilieren und Ausführen

C unterscheidet sich sowohl in der Art der Übersetzung als auch in der Ausführung vollkommen von Java.

- **Übersetzung:** Ein C Compiler betrachtet immer genau ein C-Programm, das in genau einer Datei steht. Alle Informationen müssen in dieser Datei enthalten sein. Anders als beim Javacompiler, der unter der Hand auf Klassendateien, die mitunter in einer jar-Datei verpackt sind, zugreift, implizit die Standardklassen mit berücksichtigt und mehrere Javadateien auf einen Rutsch übersetzt.
- **Ausführung:** Der C Compiler erzeugt direkt maschinenabhängigen ausführbaren Programmcode. Es wird kein weiteres Programm, das eine virtuelle Maschine interpretiert, benötigt.

1.3.1 Das übliche Programm: hello world

Wie in den meisten Programmierkursen, wollen wir uns auch nicht um das übliche *hello world* Programm drücken und geben hier die C++-Version davon an.

Hello0.cpp
```
#include <iostream>


int main(){
  std::cout << "Hallo Welt\n";
}
```

Das Programm speichern wir in einer Datei mit der Endung `.cpp` ab. Alternative gebräuchliche Endungen sind: `.cc`, `.C`, `.cxx` und `.c++`.

Auffällig ist, daß der Name des Programms (`Hello0`) im Programmtext nicht auftaucht. Anders als in Java, indem jeder Quelltextdatei eine Klasse mit dem Namen der Datei enthält. In C folgen direkt nach dem Import von Bibliotheken die Funktionen. Es muß keine Klasse wie in Java definiert werden.

Wir sehen, daß zunächst einmal die Bibliothek für Ein- und Ausgabeströme zu importieren ist: `#include <iostream>`. Auf derartige Zeilen, die mit den Zeichen `#` beginnen, werden wir noch genauer eingehen.

Die Hauptmethode läßt sich ebenso schreiben wie in Java. Sie hat allerdings den Rückgabetyp `int`.[2] Auf das aus Java bekannte Argument (`String [] args`) kann in C++ verzichtet werden. Obwohl die Methode einen Rückgabetyp verlangt, braucht die Hauptmethode keinen `return`-Befehl.

Ein- und Ausgabe wird in C++ über Operatoren ausgedrückt. Zur Ausgabe gibt es den Operator `<<`. `std::out` 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.

Stringliteralen können wie in Java durch Anführungszeichen markiert ausgedrückt werden.

Wie in Java endet auch in C++ jeder Befehl mit dem hässlichen Semikolon.

Der Operator `<<` liefert als Ergebnis einen Ausgabestrom, so daß elegant eine Folge von Ausgaben durch eine Kette von `<<` Anwendungen ausgedrückt werden kann.

Hello1.cpp
```
#include <iostream>

int main(){
  std::cout << "Hallo " << "Welt" << std::endl;
}
```

[2] Manche Compiler lassen für die Hauptmethode auch den Rückgabetyp `void` zu. So z.B. auch gcc in Version 2.95.3.

1.3.2 Kompilieren

Zur Übersetzung dieses einfachen Programmes ist der C-Compiler `gcc` mit der Quelltextdatei als Argument aufzurufen. Als zusätzliches Argument kann mit der Option `-o` der Name des zu erzeugenen ausführbaren Programms mit angegeben werden. Das erzeugte Programm kann anschließend direkt ausgeführt werden:

```
sep@swe10:~/fh/prog3/examples> g++ -o Hello1 Hello1.cpp
sep@swe10:~/fh/prog3/examples> ./Hello1
Hallo Welt
sep@swe10:~/fh/prog3/examples>
```

1.4 Funktionen und Variablen

Die klassischen Strukturierungsmöglichkeiten einer Programmiersprache sind Funktionen und Variablen, oder Methoden und Felder, wie wir sie in Java genannt haben. Natürlich gibt es beides auch in C.

1.4.1 Schreiben von Funktionen

In den ersten beiden Programmen haben wir schon eine Funktion geschrieben, die Funktion `main`. Wie zu sehen war unterscheidet sich eine Funktionsdefinition in C kaum von einer Methode in Java. Das Attribut `static` wie wir es aus Java für statische Methoden kennen, wird in C nicht gesetzt. In C ist der Standardfall, daß eine Funktion statisch ist. Erst in C++ können Klassen mit nicht statischen Objektmethoden definiert werden.

Wir können versuchen ein Programm mit einer zweiten Funktion in C zu schreiben:

Hello2.cpp
```
#include <iostream>
#include <string>

std::string getHallo(){
  return "hallo welt\n";
};

int main(){
  std::cout <<getHallo();
}
```

Wir definieren, die parameterlose Funktion `getHallo` und rufen sie in der Funktion `main` auf. Man beachte, daß der Typ `string` in C im Gegensatz zu Java klein geschrieben wird.

Von Java sind wir gewohnt, daß die Reihenfolge der Methoden in einer Klasse beliebig ist. Der Kompiler hat jeweils bereits die ganze Quelldatei studiert, bevor er die Rümpfe der Methoden übersetzt. Wir können versuchen, ob dieses für C auch gilt. Hierzu vertauschen wir die Reihenfolge der beiden Funktionen im letzten Programm.

Hello3.cpp
```
#include <iostream>
#include <string>

int main(){
  std::cout << getHallo();
}

std::string getHallo(){
  return "hallo welt\n";
}
```

Der Versuch dieses Programm zu übersetzen führt zu einem Fehler:

```
sep@swe10:~/fh/prog3/examples> g++ Hello3.cpp
Hello3.cpp: In function ‘int main(...)’:
Hello3.cpp:6: implicit declaration of function ‘int getHallo(...)’
sep@swe10:~/fh/prog3/examples>
```

Offensichtlich können in C Funktionen nur benutzt werden, nachdem sie vorher im Programm definiert wurden. Diese Einschränkung hat sicherlich mit den Stand der Compilerbautechnik am Anfang der 70er Jahre zu tun. Ein Compiler, der nur einmal sequentiell das Programm von vorne nach hinten durchgeht, läßt sich leichter bauen, als ein Compiler, der mehrmals über eine Quelldatei gehen muß, um nacheinander verschiedene Informationen zu sammeln.

Wollen wir die Reihenfolge der beiden Funktionen, wie wir sie in der Datei `Hello3.cpp` geschrieben haben, beibehalten, so können wir das in C, indem wir die Signatur der Funktion `getHallo` vorwegnehmen. Dieses sieht dann ähnlich aus, wie eine Signatur in einer Javaschnittstelle. Man kann in C in ein und derselben Datei die Signatur einer Funktion deklarieren und später in der Datei diese Funktion erst implementieren.

Hello4.cpp
```
#include <iostream>
#include <string>

std::string getHallo();

int main(){
  std::cout << getHallo();
}

std::string getHallo(){
  return "hallo welt\n";
}
```

Jetzt übersetzt das Programm wieder fehlerfrei. Mit dieser Technik simuliert man gewissermaßen, was ein Javacompiler sieht, wenn er mehrfach über einen Quelltext geht. Zunächst sieht er die Signaturen aller Funktionen und erst dann betrachtet er ihre Rümpfe. Sobald die Signatur einer Funktion bekannt ist, kann die Funktion benutzt werden.

Header-Dateien

Nach dem geraden gelernten, scheint es sinnvoll in einem C Programm zunächst die Signaturen aller Funktionen zu sammeln und dann die einzelnen Funktionen zu implementieren. Die Sammlung der Signaturen ergibt eine schöne Zusammenfassung aller von einem Programm angebotener Funktionen. Es liegt nahe, diese in eine getrennte Datei auszulagern. Hierzu bedient man sich in C der sogenannten *header*-Dateien, die mit der Dateiendung `.h` abgespeichert werden.

Für unser letztes Beispiel können wir eine *header*-Datei schreiben, in der die Signatur der Funktion `getHallo` steht:

Hello5.h
```
#include <string>

std::string getHallo();
```

Statt jetzt in einem Programm diese Signatur am Anfang des Programms zu schreiben, sagen wir dem Compiler, er soll die *header*-Datei mit in das Programm einfügen. Hierzu benutzen wir das `#include` Konstrukt. Unser Programm sieht dann wie folgt aus:

Hello5.cpp
```
#include <iostream>
#include <string>

#include "Hello5.h"

int main(){
  std::cout << getHallo();
}

std::string getHallo(){
  return "hallo welt\n";
}
```

Wichtig ist hier zu verstehen, daß mit dem `include`, bevor der Compiler anfängt das Programm zu übersetzen, die Datei `Hello.h` komplett wörtlich an der entsprechenden Stelle eingefügt wird. Dieses Einfügen wird nicht vom eigentlichen Compiler gemacht, sondern vom einem Programm, das vor dem Compiler den Quelltext manipuliert, den sogenannten Präprozessor. Der Präprozessor liest die Zeilen des Quelltext, die mit dem Zeichen `#` beginnen. Diese nennt man Direktiven. Sie sind nicht Teil des eigentlichen Programms, sondern erklären dem Präprozessor, wie das eigentliche Programm zusammenzusetzen ist.

Objektdateien und Linker

Wer unsere bisherigen Programme übersetzt hat, wird festgestellt haben, daß Dateien mit der Endung `.o` ins Dateisystem geschrieben wurden. Diese Dateien heißen Objektdateien. Der eigentliche Compiler erzeugt nicht das ausführbare Programm, sondern Objektdateien. Die Objektdateien werden dann erst durch den sogenannten Linker zu ausführbaren Dateien zusammengefasst.

Mit der Option `-c` kann man den Compiler dazu bringen, nur die Objektdatei zu erzeugen und nicht anschließend noch das ausführbare Programm zu erzeugen.

Der Compiler kann Objektdateien erzeugen, in denen noch nicht jede Funktion implementiert ist. Wir können das letzte Programm schreiben, ohne die Funktion `getHallo` implementiert zu haben:

Hello6.cpp
```
#include <iostream>
#include "Hello5.h"

int main(){
  std::cout << getHallo();
}
```

Mit der Option `-c` kann der Compiler fehlerfrei die Objektdatei erzeugen.

```
sep@swe10:~/fh/prog3/examples> g++ -c Hello6.cpp
sep@swe10:~/fh/prog3/examples> ls -l Hello6.*
-rw-r--r--    1 sep      users          84 Sep  3 17:36 Hello6.cpp
-rw-r--r--    1 sep      users        8452 Sep  3 17:40 Hello6.o
sep@swe10:~/fh/prog3/examples>
```

Wollen wir hingegen ein ausführbares Programm erzeugen, so bekommen wir einen Fehler, und zwar nicht vom Compiler sondern vom Linker. Der Linker versucht jede deklarierte Funktion mit konkreten Programmcode aufzulösen. Wenn er einen solchen Code nicht findet, so meldet er einen Fehler:

```
sep@swe10:~/fh/prog3/examples> g++ Hello6.cpp
/tmp/ccRKWUqK.o: In function ‘main’:
/tmp/ccRKWUqK.o(.text+0xe): undefined reference to ‘getHallo(void)’
collect2: ld returned 1 exit status
sep@swe10:~/fh/prog3/examples>
```

Um einen solchen Fehler zu produzieren, bedarf es keiner *header*-Datei. Wir können in einem Programm eine Funktion deklarieren ohne sie zu implementieren.

Hello7.cpp
```
#include <iostream>
#include <string>

std::string getHallo();

int main(){
  std::cout << getHallo();
}
```

Wir bekommen wieder einen Fehler des Linkers:

```
sep@swe10:~/fh/prog3/examples> g++ Hello7.cpp
/tmp/ccdm3ng0.o: In function ‘main’:
/tmp/ccdm3ng0.o(.text+0xe): undefined reference to ‘getHallo(void)’
collect2: ld returned 1 exit status
sep@swe10:~/fh/prog3/examples>
```


Der Linker hat im Gegensatz zum Compiler meherere Dateien als Eingabe, die er gemeinsam verarbeitet.

Wie man sieht, sind ein paar Dinge in C zu beachten, über die man sich in Java keine Gedanken machen mußte. Das liegt primär darin begründet, daß ein C-Compiler immer nur genau eine Datei übersetzt, in der alle Information für das Programm enthalten sein müssen; und daß der Compiler nur genau einmal den Quelltext sequentiell liest.

Aufgabe 1 Übersetzen Sie die Programme aus diesem Abschnitt und lassen Sie sie laufen.

Aufgabe 2 Schreiben Sie die rekursive Funktion der Fakultät. Hierzu brauchen Sie den `if`-Befehl, wie Sie ihn aus Java kennen. Testen Sie die Funktion mit ein paar Werten.

1.4.2 Variablen

Variablen lassen sich in C genauso deklarieren und benutzen wie in Java. Dem Variablennamen wird bei der Deklaration der Typ vorangestellt. Das Gleichheitszeichen fungiert als der Zuweisungsoperator. Es können Variablen lokal für eine Funktion oder einen anderen Block deklariert werden, oder aber global innerhalb eines Programmes.

Vars.cpp
```
#include <iostream>

int i = 2;

void f(int j){
  int k = 2*j;
  i=i+k;
}

int j = 2;

int main(){
  std::cout << i <<  std::endl;
  f(15);
  std::cout << i <<  std::endl;
  f(5);
  std::cout << i <<  std::endl;
  std::cout << j <<  std::endl;
}
```

Mehrfach inkludieren

Genauso wie Funktionen können Variablen können auch in Headerdateien deklariert sein.

C1.h
```
int x =0;
```

Ebenso können Headerdateien weitere Headerdateien inkludieren:

C2.h

```
#include "C1.h"
```

Hier kann es zu einem Fehler kommen, wenn beide diese Headerdateien auf die eine oder andere Art in eine Datei inkludiert wird.

C3.cpp

```
#include "C1.h"
#include "C2.h"

int main(){};
```

Der Versuch dieses Programm zu übersetzen führt zu einem Fehler.

```
sep@linux:~/fh/prog3/examples/src> g++ -c C3.cpp
In file included from C2.h:1,
                 from C3.cpp:2:
C1.h:1: error: redefinition of ‘int x’
C1.h:1: error: ‘int x’ previously defined here
sep@linux:~/fh/prog3/examples/src>
```

Um solche Fehler, die entstehen, wenn ein und dieselbe Datei mehrfach inkludiert wird, von vorneherein auszuschließen, bedient man sich standardmäßig einer weiteren Direktive. Mit der Direktive `#define` können Werte definiert werden. Mit der Direktive `#ifndef` kann abgefragt werden, ob ein Wert bereits definiert wurde. Nun kann man in einer Headerdatei eine für diese Datei spezifische Variable definieren. Üblicher Weise bildet man diese mit zwei Unterstrichen, gefolgt vom komplett in Großbuchstaben geschriebenen Dateinamen und der Endung `_H`. Den kompletten Inhalt der Headerdatei klammert man nun in einer `#ifndef`-Direktive. So wird der Inhalt einer Headerdatei nur dann eingefügt, wenn die spezifische Variable noch nicht definiert wurde. Also höchstens einmal.

Beherzigen wir diese Vorgehensweise, so erhalten wir statt der Datei `C1.h` von oben, eine entsprechende Datei, die wir zur Unterscheidung als `D.h` speichern.

D1.h

```
#ifndef __D1_H

#define __D1_H ;

int x =0;

#endif /* __D1_H */
```

Die Variable, die markiert, ob die Datei schon inkludiert wurde, heißt nach unserer entsprechenden Konvention `__D1_H`.

Diese Datei läßt sich jetzt woanders inkludieren.

D2.h

```
#include "D1.h"
```


Eine mehrfache Inklusion führt jetzt nicht mehr zu einen Fehler.

D3.cpp

```
#include "D1.h"
#include "D2.h"

int main(){};
```

Das Programm läßt sich jetzt fehlerfrei übersetzen:

```
sep@linux:~/fh/prog3/examples/src> g++ -o D3 D3.cpp
sep@linux:~/fh/prog3/examples/src> ./D3
```

In größeren C-Projekten kann man diese Vorgehensweise standardmäßig angewendet sehen.

Kapitel 2

Typen

2.1 Zeiger und Referenzen

In Java gab es im Prinzip nur zwei unterschiedliche Arten von Typen: Objekte und primitive Typen[1]. Damit schottet uns die Sprache Java von dem eigentlichen technischen Speichermodell eines von Neumann Rechners ab. Um die Unterschiede der verschiedenen Typen in C++ verstehen zu können, müssen wir uns ein wenig konkreter mit dem Speichermodell eines Rechners beschäftigen.

2.1.1 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 die Programmiersprache Java 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.

[1]Es gab auch noch Reihungen, die wir aber weitgehendst wie Objekte mit einer zusätzlichen Syntax betrachtet haben.

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

2.1.2 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
```
#include <iostream>

int main(){
  int i = 5;
  int* pI = &i;
  int** ppI = &pI;

  std::cout << i
    << ", " << pI
    <<", "  << ppI
    << std::endl;
}
```

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

```
#include <iostream>

int main(){
  int i = 5;
  int *pI = &i;
  int **ppI = &pI;

  std::cout << i
    << ", " << pI
    << ", " << *pI
    <<", "  << ppI
    << ", " << *ppI
    << ", " << **ppI
    << std::endl;
}
```

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

```
#include <iostream>

int main(){
  int i = 5;
  int *pI = &i;
  int **ppI = &pI;
  std::cout << i << ", " << *pI << ", " << **ppI << std::endl;

  i = 10;
  std::cout << i << ", " << *pI << ", " << **ppI << std::endl;

  *pI = 20;
  std::cout << i << ", " << *pI << ", " << **ppI << std::endl;

  **ppI = 42;
  std::cout << i << ", " << *pI << ", " << **ppI << std::endl;
}
```

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

```
#include <iostream>

int main(){
  int i = 5;
  int j = 42;
  int *pI = &i;
  int *pJ = &j;

```

```
  int *pK;

  std::cout << i
    << ", " << pI
    << ", " << pI+1
    <<", "  << *(pI+1)
    << ", " << pI-1
    <<", "  << *(pI-1)
    << ", " << pJ
    << ", " << *pJ
    << std::endl;

  *pK = 12;
    std::cout << pK <<", " << *pK << std::endl;
}
```

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

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

JIntAssign.java

```java
class JIntAssign {
  static void f(int x){x=42;}

  public static void main(String [] _){
    int y = 0;
    f(y);
    System.out.println(y);
  }
}
```

Entsprechend funktioniert auch das analoge `C` Programm:

CIntAssign.cpp

```cpp
#include <iostream>

void f(int x){x=42;}

int main(){
  int y = 0;
  f(y);
  std::cout << y << std::endl;
}
```

Auch hier ist das Ergebnis `0`.

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

```cpp
#include <iostream>

  void f(int* pX){*pX=42;}

  int main(){
    int y = 0;
    int *pY = &y;
    f(pY);
    std::cout << y << std::endl;
  }
```

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.

In Java haben wir keine Zeiger 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

```java
class IntBox{
  int value;
  public IntBox(int v){value=v;}
}
```

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
```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`.

Aufgabe 3 Schreiben Sie jetzt eine Version der Fakultätsfunktion, in der ein Zeiger auf die Variable, in der das Ergebnis akkumuliert wird, als zweites Argument übergeben wird:

FakAcc.h
```c
void fak(int x,int* result);
```

Schreiben Sie Testfälle für diese Funktion.

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`.[2] 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
```cpp
#include <iostream>

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

[2] `new` ist also ein falscher Freund, denn hier hat er kaum etwas mit dem entsprechenden Operator in Java zu tun.


```cpp
    }
}
```

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
```cpp
#include <iostream>

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

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.

In Java kennen wir keine direkte dynamische Speicherverwaltung über Zeiger. Dort wird jede Speicherverwaltung automatisch von der Javalaufzeitumgebung durchgeführt. Wenn neue Objekte erzeugt werden, wird neuer Speicher angefordert; Speicher, der nicht mehr benötigt wird, wird von Zeit zu Zeit wieder freigegeben.

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

```
#include <iostream>

void f(int* pI){
  int* pX = pI;
  delete pX;
}

int main(){
  int *pI = new int;
  *pI = 42;
  std::cout << *pI << std::endl;
  f(pI);
  int *pK = new int;
  *pK = 16;
  std::cout << *pI <<  std::endl;
}
```

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

```
#include <iostream>

int*  refToSquare(int i){
  int j = i*i;
  return &j;
```



```
}

int main(){
  int *pJ = refToSquare(5);
  int k = 42;
  std::cout << k <<std::endl;
  std::cout << *pJ <<std::endl;
}
```

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>
```

2.1.3 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

```
#include <iostream>

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

int main(){
  int (*f) (int) = &add5;
  int i = (*f)(4);

  std::cout << i << std::endl;
}
```

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
```
#include <iostream>

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

int main(){
  int (*f) (int) = add5;
  int i = f(4);

  std::cout << i << std::endl;
}
```

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>
```

Aufgabe 4 (2 Punkte) Schreiben Sie eine Funktion, die mit einer Intervallschachtellung für eine stetige Funktion eine Nullstelle approximiert.

Nullstellen.h
```
float nullstelle(float (*f) (float),float mi,float ma);
```

Testen Sie Ihre Implementierung mit folgendem Programm:

TesteNullstellen.cpp
```
#include "Nullstellen.h"
#include <iostream>

float f1(float x){return 2*x*x*x+5;}
float f2(float x){return 2*x +1 + x*x*x/4 ;}
float f3(float x){return 10*x+5;}
float f4(float x){return 0;}
float f5(float x){return x*x;}

void teste(float (*f)(float)){
  const float fn = nullstelle(f,-10,11);
  std::cout << "n = "<<fn<< ", f(n) = " << f(fn) << std::endl;
}

int main(){
  teste(f1);
  teste(f2);
  teste(f3);
  teste(f4);
  teste(f5);
}
```

2.1.4 Synonyme Typnamen

Wie wir im letztem Abschnitt gesehen haben, können Typnamen recht kompliziert werden. Eine Funktion, die einen `float` in einen `float` Wert überführt, hat bereits den Typ: `float (*f)(float)`. Wie sieht dann erst der Typ aus, der eine solche Funktion als Eingabe hat und eine andere Ausgabe erzeugt? Typen könnten theoretisch immer länger werden. Um noch griffig zu erkennen, um was es sich bei dem Typ handeln soll, kann man Typen einen Namen geben. Hierzu gibt es in C das `typedef` Konstrukt. Dem Schlüsselwort `typedef` folgt der eigentliche Typ und dem dann ein neuer frei zu wählender Bezeichner für das Typsynonym.

Beispiel:
Ein einfaches Typsynonym für Zahlen:

Zahl.cpp
```
#include <iostream>

typedef int zahl;

zahl f(zahl x){return 2*x;}

```

```
int main(){
  std::cout << f(21) << std::endl;
}
```

Für Typen, die in ihrer Syntax eigentlich einen Variablenbezeichner klammern, steht an der Stelle des Variablenbezeichners im `typedef`-Konstrukt der Name des Typsynonyms.

Beispiel:
Für führen ein Synonym für Funktionen auf `float`-Werten ein:

FloatFunction.cpp
```
#include <iostream>

typedef float zahl;
typedef zahl (*zahlenFunktion)(zahl);

zahl verdoppel(zahl x){return 2*x;}

zahl applyTwice(zahlenFunktion f, zahl x){
  return f(f(x));
}

int main(){
  std::cout << applyTwice(verdoppel,10.5) << std::endl;
}
```

Drei Vorteile ergeben sich durch das `typedef`Konstrukt:

- Es lassen sich sprechende Typnamen erfinden, die das Programm lesbarer machen.
- Man kann einen konkreten Typ leicht ändern, wenn man im gesamten Programm nur mit dem Typsynonym arbeitet; so kann man z.B. ein Typsynonym `zahl` einführen und legt nur in dessen Definition fest, ob es sich um eine `int` oder `long` Zahl handelt.
- Unhandliche C Typen, die aus mehreren Teilen bestehen, lassen sich durch ein Wort ausdrücken.

Die Definition eines Typsynonyms wird sinnvoller Weise in einer Header-Datei vorgenommen.

Ein schönes Beispiel für einen sprechenden Typnamen findet sich in der Blibliothek `cstddef`. Dort wird ein Typsynonym `size_t` definiert, um ihn für Reihungen als Typ für den Index zu benutzen.

2.1.5 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[3]. 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
```
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
```
int i;
int &j=i;
```

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

WrongInitRef.cpp
```
int i;
int &j1=i+2;
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>
```

[3]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

```cpp
#include <iostream>
int  main(){
    int i=42;
    int &j=i;
    std::cout << j << std::endl;
    i=i+1;
    std::cout << j << std::endl;
    i=i-1;
    std::cout << j << std::endl;
    int k=j;
    std::cout << k << std::endl;
    i=i+1;
    std::cout << k << std::endl;
}
```

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

```cpp
#include <iostream>

int main(){
  int i = 21;
  int &j=i;
```



```cpp

  j=2*j;
  std::cout << i << ", "<< j << std::endl;
}
```

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

```cpp
#include <iostream>

int main(){
  int i1 = 21;
  int &j1=i1;

  int i2 = 42;
  int &j2=i2;
  std::cout << i2 <<", "<< j2 << std::endl;

  j2 = j1;
  std::cout << i2 <<", "<< j2 << std::endl;

  j2 = 15;
  std::cout << i1 <<", "<< j1 << std::endl;
  std::cout << i2 <<", "<< j2 << std::endl;
}
```

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

```
#include <iostream>

void f(int &x){
  x=x+1;
}

int main(){
  int i = 41;
  f(i);
  std::cout << i << std::endl;
}
```

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

```
#include <iostream>

void f(int &x){}

int main(){
  int i = 41;
  f(i+1);
  f(1);
}
```

Wir bekommen wieder die schon bekannten Fehlermeldungen:

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


2.2 Konstruierte Typen

2.2.1 Aufzählungen

C ermöglicht es mit Aufzählungen, eine Menge möglicher Werte zu definieren. Diese Menge ist geordnet. Technisch handelt es sich lediglich um eine Untermenge der ganzen Zahlen. Auf Aufzählungstypen können Vergleichsoperationen angewendet werden. Arithmetische Operationen liefern ganze Zahlen als Ergebnis. Zahlen können durch eine Typzusicherung in Aufzählungen umgewandelt werden.

Beispiel:
Wir definieren ein Aufzählung von 6 Farben und benutzen diese.

Farben.cpp

```
#include <iostream>

enum farben
  {rot, gruen, gelb, blau, magenta, umbra};

int main() {
  enum farben f = gelb;

  std::cout << f <<std::endl;
  if (f== gelb) std::cout << "gelbe Farbe" <<std::endl;
  if (f== umbra) std::cout << "Farbe umbra" <<std::endl;
  if (f> rot) std::cout << "Farbe größer Rot" <<std::endl;
  f=(farben)(f+1);
  std::cout << f << std::endl;
  if (f== blau) std::cout << "blaue Farbe" <<std::endl;
  if (f> gelb) std::cout  << "Farbe größer Gelb" <<std::endl;
  f=gruen;
  std::cout << f << std::endl;

  f=(farben)(10);
  std::cout << "die Farbe 10: " << f << std::endl;
}
```

Das Programm erzeugt folgende Ausgabe:

```
sep@linux:~/fh/prog3/examples/bin> ./Farben
2
gelbe Farbe
Farbe größer Rot
3
blaue Farbe
Farbe größer Gelb
1
die Farbe 10: 10
sep@linux:~/fh/prog3/examples/bin>
```

Wie man als erstes sieht, werden die Werte der Aufzählung wie Zahlen behandelt.
Die Werte der Aufzählung beginnen bei `0`. Die Farbe `gelb` entspricht im Beispiel

also der Zahl 2. Sobald auf `farben` gerechnet wurde, wird eine Typzusicherung nötig.

Auch der bool'sche Datentyp der Wahrheitswerte, wird intern in C++ wie eine Aufzählung behandelt. Dieses wird besonders deutlich, wenn wir bool'sche Werte ausgeben. Es zeigt sich auch, daß zusammengesetzte Befehle mit Bedingungen Zahlen als Werte für die bool'sche Bedingung zulassen

ShowBool.cpp
```
#include <iostream>

int main(){
  std::cout << true << std::endl;
  std::cout << false << std::endl;

  if (10) std::cout << "10 ist wahr" << std::endl;
  if (0) ; else std::cout << "0 ist falsch" << std::endl;
}
```

Wir können uns entsprechend also einen eigenen Typ für Wahrheitswerte definieren:

Wahrheit.cpp
```
#include <iostream>

enum wahrheit {falsch, wahr};

int main(){
    if (wahr) std::cout << "wahr" << std::endl;

    wahrheit w = (wahrheit)(falsch||wahr);
    if (w) std::cout << "wahr" << std::endl;

    w = (wahrheit)(falsch&&wahr);
    if (w) std::cout << "wahr" << std::endl;

    w=(wahrheit)(1==1);
    if (w) std::cout << "wahr" << std::endl;
}
```

Wie man sieht, zwingt uns C immerhin dazu, eine Typzusicherung von bool'schen Werten auf unseren Aufzählungstyp zu machen.

2.2.2 Reihungen (Arrays)

Wir kennen Reihungen bereits aus Java. In Java waren Reihungen kaum etwas anderes als eine weitere Klasse, für die es eine besondere Syntax gab. Daher haben Reihungen in der Javavorlesung nur ein sehr wenig Raum eingenommen. Es ist dort für Reihungen nicht viel Neues zu den Javakonzepten zu lernen.

In C begegnen uns Reihungen als alte Freunde wieder; aber wir sollten einen genauen Blick auf sie werfen. Sie werden sich als falsche Freunde erweisen. An ihnen ist gut die Entwurfsphilosophie von C im Gegensatz zu Java zu erkennen.

Die Syntax für Reihungen entspricht bis auf einige Ausnahmen der aus Java bekannten. Der Reihungstyp besteht zuerst aus dem Elementtyp, dann dem Variablennamen, der schließlich von einem eckigen Klammernpaar gefolgt wird. Reihungen können mit in geschweiften Klammern eingeschlossenen Elementauflistungen initialisiert werden. Zugriff auf Reihungselemente erfolgt durch den in einem eckigen Klammerpaar eingeschlossenen Index. Der Reihungsindex startet von 0.

FirstArray.cpp
```
#include <iostream>

int main(){
  int xs [] = {1,2,3};
  std::cout << xs[1];
}
```

Die aus Java bekannte Syntax, in die Typbezeichnung einer Reihung keine Klammer um einen Variablennamen bildet, ist in C leider nicht erlaubt.

WrongArray.cpp
```
int main(){
  int [] xs = {1,2,3}; // Java Syntax
}
```

Das macht die Typsprache von C in vielen Fällen sehr unübersichtlich.

Größendeklaration

Ein auffälliger Unterschied zu Java ist, daß bei der Deklaration einer Reihungsvariablen die Anzahl der Elemente bekannt sein muß:

UnknownSize.cpp
```
int main(){
  int xs [];
}
```

Die Übersetzung dieses Programms führt zu einem Fehler:

```
sep@linux:~/fh/prog3/examples/src> g++ UnknownSize.cpp
UnknownSize.cpp: In function ‘int main()’:
UnknownSize.cpp:2: error: storage size of ‘xs’ isn’t known
sep@linux:~/fh/prog3/examples/src>
```

Der C-Übersetzer muß immer in der Lage sein, die Anzahl der Elemente einer Reihung abzuleiten. Der Grund dafür ist, daß sobald eine Variable deklariert wird, der C Compiler den Speicherplatz für die entsprechende Variable anfordern muß. In C bedeutet das bei einer Reihung, daß der Speicher für eine komplette Reihung angefordert wird und nicht ein Zeiger auf die Variable. Hierfür muß die Anzahl der Elemente bekannt sein.

KnownSize.cpp
```
#include <iostream>

int main(){
  int xs [15];
  std::cout << xs[1];
}
```

Das folgende Javaprogramm würde bei einer direkten Übertragung nach C bereits mehrere Fehler werfen:

ArrayTest.java
```
class ArrayTest {

    public static void main(String [] _){
        int [] ys = {1,2,3};
        int [] zs;
    }
}
```

Reihungen scheinen wirklich falsche Freunde zu sein.

Die Elementanzahl für eine Reihung muß nicht zur Übersetzungszeit bekannt sein, sondern kann dynamisch errechnet werden und z.B. als Parameter einer Funktion übergeben werden:

LocalArray.cpp
```
#include <iostream>

void f(int l){
  int xs [l];
  for (int i= 0; i<l;i=i+1){
    xs[i]=i;
    std::cout << xs[i] << ",";
  }
  std::cout << std::endl;
}

int main(){
  f(7);
  f(2);
}
```

Wie wir uns in der Ausgabe überzeugen können, werden in diesem Programm in der Funktion `f` nacheinander zwei Reihungen unterschiedlicher Länge erzeugt.[4]

```
sep@linux:~/fh/prog3/examples/src> ./LocalArray
0,1,2,3,4,5,6,
0,1,
sep@linux:~/fh/prog3/examples/src>
```

[4] Angeblich sollen bestimmte Versionen des Microsoft C++ Compilers derartige Programme nicht übersetzen.

Die Größe einer Reihung

In Java enthalten Reihungen ein Feld, in dem die Anzahl der Elemente der Reihung gespeichert ist. Eine Reihung kann wie ein Objekt behandelt werden. Das Reihungsobjekt kann einfach nach diesem Längenfeld gefragt werden:

ArrayLength.java
```
class ArrayLength {

  public static void main(String [] _){
    int xs [] = {1,2,3};
    System.out.println(xs.length);
  }
}
```

In C geht dieses nicht. Zwar ist C recht pienzig, wenn wir Reihungen deklarieren und verlangt, daß die Elementeanzahl ableitbar ist, hingegen kann diese Information von einer Reihung nicht direkt abgefragt werden. Reihungen sind hier lediglich ein zusammenhängender Block von Speicherzellen. Es ist keine zusätzliche Information zu diesem Block gespeichert. Daher muß der Programmierer selber Sorge dafür tragen, die Elementanzahl einer Reihung in einer zusätzlichen Variablen zu speichern.

In C++ gibt es einen Trick, die Anzahl der Elemente zu erfragen: Es gibt in C++ die Funktion `sizeOf` mit der für eine Variable die Anzahl der von ihr belegten Speicherzellen erfragt werden kann. Da in C jedes Reihungselement in einer Reihung vom gleichen Typ ist, läßt sich aus der Größe eines beliebigen Elements und der Gesamtgröße der Reihung, die Länge einer Reihung berechnen.

CArrayLength.cpp
```
#include <iostream>
#include <cstddef>


int main(){
  int xs [] = {1,2,3};
  double ys [] = {1,2,3};

  size_t xsSize   = sizeof(xs);
  size_t xsLength = xsSize/sizeof(xs[0]);

  size_t ysSize   = sizeof(ys);
  size_t ysLength = ysSize/sizeof(ys[0]);

  std::cout << "xsSize:   " << xsSize   << std::endl;
  std::cout << "xsLength: " << xsLength << std::endl;
  std::cout << "ysSize:   " << ysSize   << std::endl;
  std::cout << "ysLength: " << ysLength << std::endl;
}
```

Starten wir dieses Programm, so sehen wir, daß tatsächlich die korrekte Länge für beide Reihungen berechnet wird. Die Elemente benötigen unterschiedlich viel Speicherplatz.

```
sep@linux:~/fh/prog3/examples/bin> ./CArrayLength
xsSize:   12
xsLength: 3
ysSize:   24
ysLength: 3
sep@linux:~/fh/prog3/examples/bin>
```

Zaunpfahlprobleme

Will man einen Zaun von 10 Meter Länge bauen und alle zwei Meter einen Zaunpfahl setzen, so benötigt man 6 und nicht 5 Pfähle. Es passiert einem leicht, sich hier um eins zu versehen. Diese typische Fehler um genau eins in einem Index, passiert bei der Programmierung von Reihungen gerne. Schuld daran dürfte sicher auch sein, daß der erste Index die 0 ist.[5] Daran sind wir aus Java bereits gewöhnt. Java wirft uns eine entsprechende Ausnahme, wenn wir mit einem zu großen Index in eine Reihung greifen:

JWrongIndex.java

```java
class WrongIndex {
  public static void main(String [] _){
    String [] xs = {"hallo", "was", "kommt", "jetzt"};
    for (int i=0;i<=xs.length;i++){
      System.out.println(xs[i]);
    }
  }
}
```

Das Programm bricht mit einer Ausnahmemeldung ab:

```
sep@linux:~/fh/prog3/examples/classes> java WrongIndex
hallo
was
kommt
jetzt
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at WrongIndex.main(JWrongIndex.java:5)
sep@linux:~/fh/prog3/examples/classes>
```

In C passiert etwas unerwartetes:

CWrongIndex.cpp

```cpp
#include <iostream>

int main(){
  std::string xs []  = {"hallo", "was", "kommt", "jetzt"};
  for (int i=0;i<=4;i++){
    std::cout <<xs[i]<< std::endl;
  }

  int ys []  = {1,2,3,4};
```

[5]Europäische Häuser starten bei der Stockwerknummerierung mit einem E für Erdgeschoss. Amerikanische Häuser numerieren das Ergeschoss bereits mit 1. Auch hier kommt es gerne zum Zaunpfahlproblem.


```cpp
  for (int i=0;i<=4;i++){
    std::cout <<ys[i]<< std::endl;
  }
}
```

Das Programm bricht nicht ab und unterrichtet uns nicht darüber, daß wir einen zu hohen Index benutzen:

```
sep@linux:~/fh/prog3/examples/bin> ./CWrongIndex
hallo
was
kommt
jetzt

1
2
3
4
1074641520
sep@linux:~/fh/prog3/examples/bin>
```

Stillschweigend greift das Programm in die nach der Reihung liegenden Speicherbereiche. Dort werden irgendwelche Werte gefunden, mit denen dann weitergerechnet wird. Dieses ist natürlich eine gefährliche Quelle für Fehler. Zusätzlich kann es sogar zu einer Sicherheitslücke werden, wenn es gelingt über einen falschen Reihungsindex auf private Daten, die im Speicher abgelegt sind, zuzugreifen.

Hier zeigt sich ganz deutlich das Entwurfskriterium von C. Es werden keine Checks während der Ausführung durchgeführt, es sei denn der Programmierer hat diese explizit vorgeschrieben. In Java steht die Robustheit und Sicherheit an oberster Stelle des Sprachentwurfs.

Reihungszuweisung

In Java haben wir uns nicht viel Gedaken um Reihungen gemacht. Wir haben Reihungen wie jeden anderen Objektypen auch behandelt. Unter anderen konnten wir Reihungsvariablen einander zuweisen und erhielten dadurch mehrere Variablen, die auf dieselbe Reihung zeigten.

JAssignAndModify.java

```java
class JAssignAndModify{
   public static void main(String [] _){
     int [] xs = {1,2,3};
     int [] ys = xs;
     ys[1]=42;
     System.out.println(xs[1]);
   }
}
```

Der Grund dafür, daß dieses in Java funktioniert, liegt in der Tatsache, daß Reihungen wie andere Objekttypen auch, Referenzen auf die entsprechenden Typen sind. Das ist in C nicht der Fall. Betrachten wir jetzt das entsprechende Programm in C:

IllegalAssign.cpp
```
#include <iostream>
int main(){
  int xs [] = {1,2,3};
  int ys [3];
  ys = xs;
  ys[1]=42;
  std::cout <<xs[1]<<std::endl;
}
```

Bereits in der Kompilierung erhalten wir einen Fehler:

```
sep@linux:~/fh/prog3/examples/src> g++ IllegalAssign.cpp
IllegalAssign.cpp: In function ‘int main()’:
IllegalAssign.cpp:5: error: ISO C++ forbids assignment of arrays
sep@linux:~/fh/prog3/examples/src>
```

Wollen wir das aus Java bekannte Verhalten simulieren, so müssen wir wieder auf Zeiger zurückgreifen. Wir benötigen einen Zeiger auf eine Reihung. Leider ist die Syntax hierfür in C etwas umständlich. Dieses liegt an der klammernden Schreibweise für Reihungstypen, die eine Variable umschließt. Der Typ läßt sich so nicht in C ohne einen Variablennamen notieren. Ein Zeiger auf eine Reihung von drei ganzzahligen Elementen schreibt sich als: `int (* pXs)[3]`.[6]

Hier nun das obige Javabeispiel über Zeiger in C realisiert:

CAssignAndModify.cpp
```
#include <iostream>
int main(){
  int xs [3] = {1,2,3};
  int (* pXs)[3]  = &xs;
  int (* pYs)[3]  = pXs;

  (*pYs)[1] =42;

  std::cout << xs[1] << std::endl;
}
```

Die Ausgabe ist wieder die beliebte Antwort `42`.

Reihungen von Zeigern

Im letzten Abschnitt haben wir Zeiger auf Reihungen realisiert. Jetzt realisieren wir eine Reihung von Zeigern. Da Zeiger ebenso Typen sind, wie alle anderen Typen auch, lassen sich von Zeigern auch Reihungen bilden:

[6] In C Fehlermeldungen und Textbüchern werden solche Typen ohne den Variablennamen geschrieben, also als `int (*)[3]`. Dieses ist in C Quelltexten so nicht erlaubt.

ArrayOfPointers.cpp
```
#include <iostream>

int main(){
  int *xs [] = {new int, new int, new int};
  for (int i = 0;i<3;i=i+1){
    *xs[i] = i;
  }

  for (int i = 0;i<3;i=i+1){
      std::cout << xs[i] << " " <<*xs[i] << std::endl;
  }
}
```

Das Programm erzeugt folgende Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> g++ -o ArrayOfPointers ArrayOfPointers.cpp
sep@linux:~/fh/prog3/examples/src> ./ArrayOfPointers
0x8049cd8 0
0x8049ce8 1
0x8049cf8 2
sep@linux:~/fh/prog3/examples/src>
```

Aufgabe 5 Implementieren sie die Funktion `compose` entsprechend folgender Header-Datei:

Compose.h
```
typedef int (* intfunction)(int);

int compose(intfunction fs [],int length,int x);
```

Sie soll eine Komposition aller Funktionen in der Reihung `fs` auf den Wert `x` anwenden. Es soll dabei gelten:

$$compose(\{f_1, \ldots f_n\}, n, x) = f_n(f_{n-1}(\ldots(f_1(x))\ldots))$$

Testen Sie Ihre Lösung mit folgender Testdatei:

ComposeTest.cpp
```
#include <iostream>
#include "Compose.h"

int f1(int x){return x*x;}
int f2(int x){return x;}
int f3(int x){return x+2;}
int f4(int x){return x*4;}
int f5(int x){return -x*x*x;}

int main(){
    intfunction fs [] = {f1,f2,f3,f4,f5};
    std::cout << compose(fs,5,2)<<std::endl;
}
```

Dynamische Reihungen

Wir haben das Schlüsselwort `new` kennengelernt, um einen Zeiger auf neu angeforderten Speicherplatz zum Speichern von Daten eines bestimmten Typs zu bekommen. Mit dem `delete`-Konstrukt konnte derart angeforderter Speicherplatz auch wieder freigegeben werden. Diese Konstrukte können ebenso auch für Reihungen angewendet werden. Auch hierbei muß gewährleistet sein, daß zur Ausführung des `new` Ausdrucks die Größe des benötigten Speicherplatzes bekannt ist, also insbesondere die Elementanzahl der Reihung.

Ebenso wie bei skalaren Typen erhalten wir durch den `new` Operator einen Zeiger auf den neu angeforderten Speicherbereich.

Beispiel:
Folgendes Programm erzeugt dynamisch nacheinander meherere Reihungen, benutzt diese kurz und gibt anschließend den durch die Reihung belegten Speicherbereich wieder frei:

NewArray.cpp

```
#include <iostream>

int main(){
  for (int i = 10; i<20;i=i+1){
    int *xs = new int[i];
    for (int j = 0; j<i;j=j+1){
      xs[j]=j;
      std::cout << xs[j] << ",";
    }
    std::cout << std::endl;;
    delete xs;
  }
}
```

Und tatsächlich können wir uns in der Programmausgabe davon überzeugen, Reihungen verschiedener Länge erzeugt zu haben:

```
sep@linux:~/fh/prog3/examples/src> ./NewArray
0,1,2,3,4,5,6,7,8,9,
0,1,2,3,4,5,6,7,8,9,10,
0,1,2,3,4,5,6,7,8,9,10,11,
0,1,2,3,4,5,6,7,8,9,10,11,12,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
sep@linux:~/fh/prog3/examples/src>
```

Reihungen als Parameter

Wir können Funktionen schreiben, die Reihungen als Parameter übergeben bekommen. Diese Reihungen lassen sich im Funktionsrumpf ganz normal als Reihungen behandeln:

AddInts.cpp

```
#include <iostream>

int add(int xs [],int argc){
  int result=0;
  for (int i=0;i<argc;i++){
    result=result+xs[i];
  }
  return result;
}

int main(){
  int xs [] = {1,2,3,4};
  std::cout << add(xs,4) << std::endl;
}
```

Wir können jetzt einmal in diesem Programm einen Fehler einbauen, um zu sehen, als was für einen Typ der Compiler unseren Reihungsparameter betrachtet:

WrongArrayParameter.cpp

```
#include <iostream>

int add(int xs [],int argc){
  int result=0;
  for (int i=0;i<argc;i++){
    result=result+xs[i];
  }
  return result;
}

int main(){
    int xs [] = {1,2,3,4};
    std::cout << add(xs) << std::endl;
}
```

Auf diese Weise erzwingen wir eine Fehlermeldung des Compilers, in der die erwarteten Parametertypen der Funktion `add` angegeben werden. Der Kompilierversuch führt zu folgender Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ -c WrongArrayParameter.cpp
WrongArrayParameter.cpp: In function ‘int main()’:
WrongArrayParameter.cpp:3: error: too few arguments to function ‘int add(int*,
   int)’
WrongArrayParameter.cpp:13: error: at this point in file
sep@linux:~/fh/prog3/examples/src>
```

Der Übersetzer ist der Meinung, der erste Parameter der Funktion `add` ist ein Zeiger auf eine ganze Zahl. Der Übersetzer macht aus einem Reihungsparameter nicht nur implizit einen Zeiger, sondern einen Zeiger auf das erste Element des Reihungsparameters. Die Schreibweise der eckigen Klammern für Reihungen, läßt sich somit als eine versteckte Zeigerarithmetik interpretieren. `xs[i]` bedeutet dann: `*(xs+i)`.

Tatsächlich können wir ohne die Syntax der eckigen Klammern mit Reihungen arbeiten. Obiges Programm läßt sich auch wie folgt formulieren:

NoBrackets.cpp

```
#include <iostream>

int add(int xs [],int argc){
  int result=0;
  for (int i=0;i<argc;i++){
    result=result + *(xs+i);
  }
  return result;
}

int main(){
  int xs [] = {1,2,3,4};
  std::cout << add(xs,4) << std::endl;
}
```

Auch diese Version hat als Ergebnis die 10.

Reihungen als Rückgabewert

Wir sind aus Java gewohnt, Reihungen mehr oder weniger ohne Ausnahme wie alle anderen Typen auch zu benutzen; sie als Parameter zu übergeben, oder sie als Ergebnis einer Funktion zurückzugeben. Im letzem Abschnitt haben wir bereits gesehen, daß bei der Parameterübergabe von Reihungen diese implizit als Zeiger auf das erste Element behandelt werden. Gleiches gilt auch für die Rückgabewerte von Funktionen. Dadurch lassen sich einige Probleme einhandeln. Versuchen wir einmal eine einfache Funktion zu schreiben, die eine neue Reihungsvariable anlegt und diese als Ergebnis zurückgibt:

WrongArrayReturn.cpp

```
#include <iostream>

int*  newArray(int length){
  int result [length];
  return result;
}

int main(){
   int* xs = newArray(5);

   for (int i=0;i<5;i++){
     std::cout << xs[i];
   }
   std::cout << std::endl;
}
```


Die Übersetzung dieser Klasse glückt zwar, aber der Übersetzer gibt uns eine ernst zu nehmende Warnung:

```
WrongArrayReturn.cpp: In function 'int* newArray(int)':
WrongArrayReturn.cpp:5: Warnung: address of local variable 'result' returned
sep@linux:~/fh/prog3/examples/src>
```

Die Warnung ist insofern ernst zu nehmen, als daß wir tatsächlich ein undefiniertes Laufzeitverhalten bekommen:

```
sep@linux:~/fh/prog3/examples/src> ./WrongArrayReturn
1-1073745324-10737454321074404108134519424
sep@linux:~/fh/prog3/examples/src>
```

Den Grund hierfür kennen wir bereits aus unseren Kapitel über Zeiger. Implizit behandelt der Übersetzer die Zeile `return result` als eine Rückgabe des Zeigers auf die lokale Variable `result`. Eine lokale Variable innerhalb einer Funktion, wird im Speicher nach Beendigung der Funktion wieder freigegeben. Zeigen wir noch auf diese Variable, so werden wir an ihrer Speicherzelle undefinierte Werte sehen.

Die einzige vernünftigen Möglichkeiten der Rückgabe von Reihungen in C sind die Rückgabe eines als Parameter überreichten Zeiger auf eine Reihung, oder aber der Zeiger auf eine dynamisch neu erzeugte Reihung:

ReturnArray.cpp

```
#include <iostream>

int*  newArray(int length){
    return new int[length];
}

int main(){
   int* xs = newArray(5);

   for (int i=0;i<5;i++){
     std::cout << xs[i];
   }
   std::cout << std::endl;
}
```

Dieses Programm übersetzt ohne Warnung und liefert die erwartete Ausgabe.

```
sep@linux:~/fh/prog3/examples/src> g++ -o ReturnArray ReturnArray.cpp
sep@linux:~/fh/prog3/examples/src> ./ReturnArray
00000
sep@linux:~/fh/prog3/examples/src>
```

Tatsächlich findet man es relativ selten in C Programmen, daß eine Funktion eine Reihung als Rückgabewert hat. Zumeist bekommen Funktionen Reihungen als Parameter übergeben und manipulieren diese.

C Strings als Reihungen von Zeichen

In der Programmiersprache C also ohne dem `++` angehängt, wurden Strings als Reihungen von Zeichen betrachtet. Dieses hat große Auswirkungen auf das Arbeiten mit Strings und unterscheidet sich vollkommen von der Arbeit mit der Klasse `String` in Java.

Eine Stringliteral kann tatsächlich als Initialisierung einer Reihung von Elementen des Typs `char` benutzt werden.

Beispiel:
Folgendes Programm legt eine Zeichenkette als Reihung an und betrachtet diese einmal über die Reihungsvariable und einmal über einen Zeiger auf das erste Element.

CString.cpp

```cpp
#include <iostream>

int main(){
  char xs [] = "hallo";
  char *ys = xs;
  ys[1] = 'e';
  std::cout << xs << std::endl;
  std::cout << ys << std::endl;
  xs[5] = '!';
  std::cout << xs << std::endl;
}
```

Die Zeichenkette besteht aus 5 Buchstaben, also ist ihr höchster Index die 4. Wenn wir am Index 5 ein neues Zeichen anspeichern, dann bekommen wir eine unerwartete Ausgabe, nämlich viele nach dem Ausrufezeichen nachfolgende Zeichen:

```
sep@linux:~/fh/prog3/examples/src> g++ -o CString CString.cpp
sep@linux:~/fh/prog3/examples/src> ./CString
hello
hello
hello!%/1?%/1#@ @@8%/1(@
sep@linux:~/fh/prog3/examples/src>
```

Intern hat jeder C-String ein Element mehr als Zeichen. In diesem letzten Element ist mit einer `0` markiert, daß der String jetzt zu Ende ist. Zeichenketten werden also intern durch ein Nullelement beendet. Dieses ist unabhängig von der eigentlichen technischen Länge der Reihung. Im obigen Beispiel haben wir diese Endemarkierung gelöscht, indem wir sie mit einem Ausrufezeichen überschrieben haben. Von da an betrachtete C alle nachfolgenden Werte im Speicher als Zeichen, die zu dem String gehören. Bei der Ausgabe werden alle diese Zeichen mit ausgegeben, obwohl die Reihung schon längst beendet ist.

Initialisiert man eine Reihung von Zeichen mit der Mengenklammeraufzählung ist der beendende Nullwert unbedingt zu berücksichtigen.

CString2.cpp

```cpp
#include <iostream>

```



```cpp
int main(){
  char hello[] =
   { 'H', 'e', 'l', 'l', 'o', '\0' };
  std::cout << hello << std::endl;
}
```

Wird die Endemarkierung bei der Initialisierung vergessen, so blickt C unkontrolliert weiter in den Speicher und betrachtet alles nachfolgende als Zeichen der Zeichenkette, bis zufällig ein Nullwert kommt. Das folgende Programm erzeugt auch eine indeterministische Ausgabe:

CString3.cpp

```cpp
#include <iostream>

int main(){
  char hello[] =
   { 'H', 'e', 'l', 'l', 'o'};
  std::cout << hello << std::endl;
}
```

Für C ist eine Zeichenkette immer genau dann beendet, wenn der abschließende Nullwert auftritt. Dieses gilt auch, wenn er inmitten der Reihung auftritt.

Im folgenden Programm ist nach der Zuweisung des Nullwerts für das Element am Index 6 für C der String nach dem Wort `hello` beendet, obwohl die Reihung länger ist und noch weitere sinnvolle Zeichen in der Reihung gespeichert sind.

CString4.cpp

```cpp
#include <iostream>

int main(){
  char hello [] = "hello world!";
  hello[5] ='\0';

  std::cout << hello << std::endl;
}
```

In traditioneller C-Programmierung gibt es typischer Weise Reihungen einer maximalen Länge, in denen unterschiedlich lange Strings gespeichert sind. Für das Programm bedeutet das, daß die Stringwerte an diesen Stellen eine bestimmte Maximallänge nicht überschreiten dürfen. Daher kommt in vielen älteren Programmen wahrscheinlich die Restriktion, daß bestimmte Namen, Pfade oder Optionen nur eine bestimmte Länge haben dürfen.

Programmieren von Strings in C ist ein mühsames Geschäft, was nur mit einer Vielzahl von bereitgestellten Bibliotheksfunktionen zu bewerkstelligen ist. In C++ steht eine umfangreiche Stringbibliothek in Form einer Stringklasse wie in Java zur Verfügung.

Kommandozeilenparameter

Auch für die Hauptmethode hat es Auswirkungen, daß C Reihungen ihre Elementanzahl nicht kennen. Unsere bisherigen Hauptmethoden hatten keine Argumente. Man kann zur

Kommandozeilenübergabe in C aber auch eine Hauptmethode schreiben, der in einer Reihung die Kommandozeilenoptionen übergeben werden. Hierzu braucht man dann aber auch ein weiteres Argument, das die Länge der übergebenen Reihung mit angibt.

CommandLineOption.cpp
```
#include <iostream>

int main(int argc, char * argv []){
  for (int i=0;i<argc;i++){
    std::string arg = argv[i];
    std::cout << arg << std::endl;
  }
}
```

Wie man an dem folgenden Testlauf sieht, wird als erstes Reihungselement der Name des Programms übergeben. Dieses war in Java nicht der Fall.

```
sep@linux:~/fh/prog3/examples/bin> ./CommandLineOption hallo C -v gruss
./CommandLineOption
hallo
C
-v
gruss
sep@linux:~/fh/prog3/examples/bin>
```

Gleichheit auf Reihungen

Da Reihungen Blöcke innerhalb des Speichers sind, ließe sich in C technisch eine Gleichheit auf Reihungen einfach realisieren. Für zwei Reihungen brauchen nur paarweise der Inhalt der Speicherzellen verglichen werden. C benutzt zum Arrayvergleich als Identitätsoperator ebenso wie Java die Adressidentität. Ebenso wie in Java, ist eine inhaltliche Gleichheit auszuprogrammieren:

CEQArray.cpp
```
#include <iostream>

bool arrayEq(int xs [],int xsLength, int ys [],int ysLength){
  if (xsLength!=ysLength) return false;
  for (int i=0;i<xsLength;i++){
    if (xs[i]!=ys[i]) return false;
  }
  return true;
}

std::string printBool(bool b){
  if (b) return "true"; else return "false";
}

int main(){
  int xs [] = {1,2,3,4};
  int ys [] = {1,2,3,4};
```



```
  int zs [] = {0,2,3,4};

  std::cout << printBool(xs==ys) << std::endl;
  std::cout << printBool(xs==zs) << std::endl;
  std::cout << printBool(arrayEq(xs,4,ys,4)) << std::endl;
  std::cout << printBool(arrayEq(xs,4,zs,4)) << std::endl;
}
```

Wir erhalten folgende Ausgabe:

```
sep@linux:~/fh/prog3/examples/bin> ./CEQArray
false
false
true
false
sep@linux:~/fh/prog3/examples/bin>
```

Mehrdimensionale Reihungen

In Java und in C In Java haben wir ganz generisch Reihungen von Reihungen definiert und benutzt. Hierzu brauchten wir nichts weiter zu beachten. Eine Reihung war dort ein ganz ordinärer Typ, mit dem dieselbe Konstrukte wie mit jedem anderen Typ auch gebaut werden konnten. Eine Mehrdimensionale Reihung war schlicht eine Reihung, deren Elemente wieder Reihungen waren. In C spielt eine mehrdimensionale Reihung eine kleine Sonderrolle. Die Syntax ist zunächst ähnlich der Syntax in Java:

JTwoDimArray.java
```
class JTwoDimArray{
  public static void main(String [] _){
   int xys [][] = new int [5][12];
   for (int x=0;x<5;x=x+1){
     for (int y=0;y<12;y=y+1){
       xys[x][y]=x+y;
     }
   }

   for (int x=0;x<5;x=x+1){
    for (int y=0;y<12;y=y+1){
     System.out.println(xys[x][y]);
    }
   }
  }
}
```

Und das entsprechende Programm in C:

TwoDimArray.cpp
```
#include <iostream>

```

```
int main(){
  int xys [5][12];
  for (int x=0;x<5;x=x+1){
    for (int y=0;y<12;y=y+1){
       xys[x][y]=x+y;
    }
  }

  for (int x=0;x<5;x=x+1){
    for (int y=0;y<12;y=y+1){
      std::cout << xys[x][y]<< std::endl;
     }
  }
}
```

Initialisierung durch Aufzählung In C können auch mehrdimensionale Reihungen durch die Schreibweise der Mengenklammern direkt durch eine Aufzählung der Elemente initialisiert werden. Allerdings ist hierbei zu beachten, daß C verlangt im Typnamen bereits die Länge der inneren Reihungen zu kennen. Deshalb führt das folgende Programm zu Fehlern in der Übersetzung:

WrongTwoDimArray.cpp

```
int main(){
  int xys [][]
    = {{11,12,13,14,15}
      ,{21,22,23,24,25}
      ,{31,32,33,34,35}
      ,{41,42,43,44,45}
    };
}
```

Der Gnu-Compiler erzeugt z.B. die folgende Fehlermeldung:

```
sep@linux:~/fh/prog3/examples/src> g++ WrongTwoDimArray.cpp
WrongTwoDimArray.cpp: In function ‘int main()’:
WrongTwoDimArray.cpp:3: error: declaration of ‘xys’ as multidimensional array
   must have bounds for all dimensions except the first
WrongTwoDimArray.cpp:7: error: assignment (not initialization) in declaration
sep@linux:~/fh/prog3/examples/src>
```

Geben wir hingegen in der Typdeklaration die Elementanzahl der inneren Reihungen an, so übersetzt dieses Programm.

TwoDimArray2.cpp

```
#include <iostream>

int main(){
  int xys [][5]
    = {{11,12,13,14,15}
```



```
      ,{21,22,23,24,25}
      ,{31,32,33,34,35}
      ,{41,42,43,44,45}
    };

  for (int x=0;x<4;x=x+1){
    for (int y=0;y<5;y=y+1){
      std::cout << xys[x][y]<< std::endl;
     }
  }
}
```

C verlangt eine feste Länge für die inneren Reihungen. Das war in Java nicht der Fall.

Beispiel:
Folgendes Javaprogramm erzeugt eine zweidimensionale Reihung, deren erste innere Reihung drei und deren zweite innere Reihung 4 Elemente hat. Eine solche mehrdimensionale Reihung ist in C nicht möglich.

DifferentLengths.java

```
class DifferentDims {
  public static void main(String [] _){
    int [] xs = {1,2,3};
    int [] ys = {4,5,6,7};
    int [][] xys = {xs,ys};

    for (int x=0;x<2;x=x+1){
      for (int y=0;y<xys[x].length;y=y+1){
        System.out.println(xys[x][y]);
      }
    }
  }
}

```

Zeigerzugriff In Java hat der Zugriff auf die inneren Elemente einer mehrdimensionalen Reihung eine Reihung als Ergebnis gegeben. Wir können jetzt einmal versuchen, was C uns liefert, wenn wir nur einen einfachen Index benutzen:

TwoDimArray3.cpp

```
#include <iostream>

int main(){
  int xys [][5]
    = {{11,12,13,14,15}
      ,{21,22,23,24,25}
      ,{31,32,33,34,35}
      ,{41,42,43,44,45}
    };

```

```
  for (int x=0;x<4;x=x+1){
      std::cout << xys[x]<< std::endl;
  }
}
```

Das Programm hat folgende Ausgabe:

```
sep@linux:~/fh/prog3/examples/src> ./TwoDimArray3
0xbffff1b0
0xbffff1c4
0xbffff1d8
0xbffff1ec
sep@linux:~/fh/prog3/examples/src>
```

Wir bekommen wieder Speicheradressen, nämlich die Speicheradressen für jeweils das erste Element der inneren Reihung. Alle Elemente einer mehrdimensionalen Reihung liegen anscheinend hintereinander in einem zusammenhängenden Speicherbereich. Wir können daher uns auch der Zeigerarithmetrik bedienen, um mit einer einfachen und nicht mit zwei geschachtelten Schleifen, auf alle Elemente einer mehrdimensionalen Reihung zuzugreifen:

OneLoopOnly.cpp

```
#include <iostream>

int main(){
  int xys [][5]
    = {{11,12,13,14,15}
      ,{21,22,23,24,25}
      ,{31,32,33,34,35}
      ,{41,42,43,44,45}
    };
  int *pXys = &xys[0][0];

  for (int x=0;x<20;x=x+1){
      std::cout << *(pXys+x)<< std::endl;
  }
}
```

Warnung vor einer Notation Eine kleine Warnung noch. In C gibt es die Möglichkeit innerhalb eines eckigen Klammernpaares durch Komma getrennt zwei Indizes anzugeben. Dieses liegt an einem anderen Konstrukt in C. Durch Komma getrennte Ausdrücke haben den Wert des letzten Ausdrucks in dieser Liste. Im folgenden Programm ist dieses daran zu erkennen, daß die Ausgabe Adressen sind, also nur ein Index ausgewertet wurde:

NotTheIndexYouProbablyMean.cpp

```
#include <iostream>

int main(){
  int xys [][5]
    = {{11,12,13,14,15}
      ,{21,22,23,24,25}
```



```
      ,{31,32,33,34,35}
      ,{41,42,43,44,45}
    };

  for (int x=0;x<4;x=x+1){
    for (int y=0;y<5;y=y+1){
      std::cout << (x,y) << std::endl;
      std::cout << xys[x,y]<< std::endl;
     }
  }
}
```

Die Ausgabe des Programms ist z.B.:

```
sep@linux:~/fh/prog3/examples/src> ./NotTheIndexYouProbablyMean
0
0xbffff180
1
0xbffff194
2
0xbffff1a8
3
0xbffff1bc
4
0xbffff1d0
0
0xbffff180
1
0xbffff194
2
0xbffff1a8
3
0xbffff1bc
4
0xbffff1d0
0
0xbffff180
1
0xbffff194
2
0xbffff1a8
3
0xbffff1bc
4
0xbffff1d0
0
0xbffff180
1
0xbffff194
2
0xbffff1a8
3
0xbffff1bc
4
0xbffff1d0
sep@linux:~/fh/prog3/examples/src>
```

Der Ausdruck in den eckigen Indexklammern wird immer nur zum Wert von `y` ausgewertet.

Funktionen höherer Ordnung für Reihungen

Wir haben bereits gesehen, daß Funktionen über Funktionszeigern an andere Funktionen als Parameter übergeben werden können. Im Zusammenhang mit Reihungen eröffnet dieses ein mächtiges Programmierprinzip. Fast jede Funktion, die sich mit Reihungen beschäftigt, wird einmal jedes Element der Reihung durchgehen und dazu eine `for`-Schleife benutzen. Wie wäre es denn, wenn man dieses Prinzip verallgemeinert und eine allgemeine Funktion schreibt, die einmal jedes Element einer Reihung betrachtet und etwas mit ihm macht. Was mit jedem Element zu tun ist, wird dieser Funktion als Funktionszeiger übergeben.

Funktionen, die als Parameter Funktionen übergeben bekommen, werden auch als Funktionen höherer Ordnung bezeichnet.

Beispiel:
Wir schreiben für Reihungen ganzer Zahlen die Funktion `map`, die jedes Element einer Reihung mit einer übergebenen Funktion umwandelt:

IntMap.cpp

```
#include <iostream>

void map(int xs [],int l,int (*f) (int)){
  for (int i=0;i<l;i++){
    xs[i]=f(xs[i]);
  }
}
```

Wir stellen drei Funktionen, die eine ganze Zahl als Parameter und Ergebnis haben zur Verfügung:

IntMap.cpp

```
int add5(int x){return x+5;}
int square(int x){return x*x;}
int doubleX(int x){return 2*x;}
```

Jetzt können wir die Funktion `map` verwenden, um für jedes Element einer Reihung, eine derartige Funktion anzuwenden. Zur Ausgabe von Reihungen schreiben wir eine kleine Hilfsfunktion:

IntMap.cpp

```
void printArray(int xs [],int l){
  std::cout<<"{";
  for (int i=0;i<l-1;i++){
    std::cout << xs[i]<<", ";
  }
  std::cout << xs[l-1] << "}"<<std::endl;
}

int main(){
  int xs [] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  map(xs,15,add5);
  printArray(xs,15);
  map(xs,15,square);
  printArray(xs,15);
```



```
  map(xs,15,doubleX);
  printArray(xs,15);
}
```

Und hier die Ausgabe des Programms:

```
sep@linux:~/fh/prog3/examples/src> ./IntMap
{6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
{36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
{72, 98, 128, 162, 200, 242, 288, 338, 392, 450, 512, 578, 648, 722, 800}
sep@linux:~/fh/prog3/examples/src>
```

Zunächst wird auf jedes Element der Reihung 5 aufaddiert. Dann werden die Elemente quadriert und schließlich verdoppelt.

Aufgabe 6 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester vorgestellt wurde, für Reihungen von ganzen Zahlen 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) )`

Aufgabe 7 (2 Punkte) Schreiben Sie eine Funktion `fold` mit folgender Signatur:
`int fold(int xs [],int length,int (*foldOp)(int,int),int startV)`

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

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

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

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

Testen Sie Ihre Methode `fold` mit folgenden Programm:

FoldTest.cpp

```
#include <iostream>

int add(int x,int y){return x+y;}
int mult(int x,int y){return x*y;}

int fold(int xs [],int length,int (*foldOp)(int,int),int startV);

int main(){
  int xs [] = {1,2,3,4,5};
  std::cout << fold(xs,5,add,0)<< std::endl;
  std::cout << fold(xs,5,mult,1)<< std::endl;
}
```

2.2.3 Produkte und Summen von Typen

Betrachten wir einmal Typen als die Menge aller Werte die von diesem Typ sind. So können wir aus zwei Typen einen neuen Typ konstruieren. In der theoretischen Informatik unterscheidet man hierbei zwei Konstruktionsprinzipien: Produkt und Summe.

Als das Produkt zweier Typen T_1 und T_2 bezeichnet man den Typ, der aus Paaren von Werten der beiden Grundtypen besteht. Die Bezeichnung Produkt ist in diesem Fall gerechtfertigt: habe T_1 n Elemente und T_2 m Elemente; der Produkttyp $T_1 \otimes T_2$ hat dann $n * m$ Elemente.

Als Summe zweier Typen T_1 und T_2 bezeichnet man den Typ, dessen Elemente entweder einen Wert des einen oder des anderen Typs hat. Die Bezeichnung Summe ist in diesem Fall gerechtfertigt: habe T_1 n Elemente und T_2 m Elemente; der Summentyp $T_1 \oplus T_2$ hat dann $n + m$ Elemente.

In Java haben wir ohne viel darüber nachzudenken Produkt und Summentypen gebildet. Produkttypen ergeben sich aus den Feldern einer Klasse. Folgende Klasse bildet den Produkttyp für `String` und `int`.

StringIntProduct.java

```java
class StringIntProduct{
  String typ1;
  int    typ2;
  public StringIntProduct(String e1, int e2){
    typ1=e1;
    typ2=e2;
  }
}
```

Summentypen können in Java über Vererbung gebildet werden. Bilden wir als Beispiel die Summe von `int` und `String`. Hierzu definieren wir eine Klasse, die die Summe ausdrücken soll:

StringIntSum.java

```java
class StringIntSum {}
```

Als Unterklassen dieser Summenklasse definieren wir jetzt jeweils eine Klasse für die beiden summierten Typen.

StringTyp.java

```java
class StringTyp extends StringIntSum {
  StringTyp(String v){value=v;}
  String value;
}
```

IntTyp.java

```java
class IntTyp extends StringIntSum {
  IntTyp(int v){value=v;}
  int value;
}
```

In C gibt es zwei explizite Typkonstrukte für Summen und Produkte.

Typprodukt mit Strukturen

Strukturen können als eine sehr rudimentäre Vorstufe von Klassen in C betrachtet werden. In anderen Programmiersprachen spricht man auch gerne von *records*. In C++ sind Strukturen eigentlich nicht mehr notwendig. Dort werden sie vollständig von Klassen subsumiert.

Strukturen werden ähnlich wie Klassen definiert: dem Schlüsselwort `struct` folgt der Name den man der Struktur geben will, anschließend folgt in geschweiften Klammern eingeschlossen eine Attributliste, entsprechend den Feldern einer Klasse. Ein Analogon zu Methoden in Klassen gibt es für Strukturen nicht. Ebenso keine Vererbung zwischen Strukturen. Elemente eines Strukturtyps werden durch die Aufzählung der konkreten Werte für die einzelnen Attribute der Struktur erzeugt. Diese Aufzählung steht in geschweiften Klammern und ist durch Kommas getrennt. Auf die einzelnen Attribute einer bestehenden Struktur kann durch die für Klassen bekannte Punktnotation zugegriffen werden.

Beispiel:
Im folgenden Beispiel definieren wir eine Struktur zur Speicherung von Personendaten. Eine Instanz wird gebildet, auf der mehrere Test zur internen Speicherablage von Strukturen gemacht werden.

PersonStruktur.cpp

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

struct Person
  {std::string name
  ;std::string vorname
  ;std::string strasse
  ;std::string stadt
  ;int plz
  ;int gebjahr
  ;int gebmonat
  ;int gebtag
  ;};

int main(){
  struct Person p1
    = {"Zabel", "Walther", "Hauptstrasse", "Ludwigshafen"
      , 65345, 1962, 3, 26};

  std::cout << p1.stadt << std::endl;

  std::cout << sizeof(p1) << std::endl;
  std::cout
   << &p1.name    << std::endl << &p1.vorname << std::endl
   << &p1.strasse << std::endl << &p1.stadt   << std::endl
   << &p1.plz     << std::endl << &p1.gebjahr << std::endl
   << &p1.gebmonat<< std::endl << &p1.gebtag  << std::endl;
}
```

Der Programmausgabe kann man entnehmen, daß der Übersetzer die Attribute einer Struktur direkt hintereinander im Speicher ablegt.

```
sep@linux:~/fh/prog3/examples/src> ./PersonStruktur
Ludwigshafen
32
0xbffff140
0xbffff144
0xbffff148
0xbffff14c
0xbffff150
0xbffff154
0xbffff158
0xbffff15c
sep@linux:~/fh/prog3/examples/src>
```

Zuweisung von Strukturen Für Reihungen war es nicht möglich eine Zuweisung zwischen zwei Reihungsvariablen zu machen. Der Übersetzer hat dieses bereits zurückgewiesen. Strukturvariablen können einander zugewiesen werden. Allerdings werden wir hier ein anderes Verhalten beobachten, als wir es mit unseren Javahintergrund vermutet hätten.

Beispiel:
Wir definieren eine einfache Struktur, erzeugen eine Instanz und weisen diese einer zweiten Strukturvariablen zu:

AssignStruct.cpp

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

struct S {std::string n1; std::string n2;};

int main(){
  struct S s1 = {"hallo","welt"};
  struct S s2 = s1;

  std::cout << s1.n1 << " " << s1.n2 << std::endl;
  std::cout << s2.n1 << " " << s2.n2 << std::endl;

  s2.n2 ="berlin";

  std::cout << s1.n1 << " " << s1.n2 << std::endl;
  std::cout << s2.n1 << " " << s2.n2 << std::endl;
}
```

An der Ausgabe können wir erkennen, daß die beiden Variablen `s1` und `s2` sich nicht die gleichen Daten teilen. Eine Änderung in der einen Struktur bewirkt keine Änderung in der zweiten Struktur.

```
sep@linux:~/fh/prog3/examples/src> c++ -o AssignStruct AssignStruct.cpp
sep@linux:~/fh/prog3/examples/src> ./AssignStruct
hallo welt
hallo welt
hallo welt
hallo berlin
sep@linux:~/fh/prog3/examples/src>
```


Der Grund hierfür ist, daß die Zuweisung auf Strukturen eine Kopie der Struktur auf der rechten Seite bewirkt. Wir bekommen ein Verhalten, wie wenn wir in Java explizit die Methode `clone` auf das Objekt der rechten Seite angewendet hätten.

Strukturen als Parameter Die Parameterübergabe an eine Funktion ist eine Art Zuweisung. Dem Funktionsparameter wird der Wert zugewiesen. Und auch hier gilt für Strukturen dasselbe wie für die Zuweisung: der Funktion wird eine Kopie der Instanz der Struktur übergeben.

Beispiel:
Schreiben wir für eine einfache Struktur für Personen eine Funktion, die den Namen ändern soll. Diese funktioniert nicht, wenn wir die Struktur übergeben, sondern nur wenn wir einen Zeiger auf die Struktur übergeben.

PassStruct.cpp

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

struct Person
  {std::string name
  ;std::string vorname
  ;};

void doNotChangeName(struct Person p,std::string n){
    p.name = n;
}

void changeNameWithPointer(struct Person * p,std::string n){
    (*p).name = n;
}

int main(){
  struct Person p1
    = {"Brecht", "Bertholt"};
  doNotChangeName(p1,"Schmidt");
  std::cout << p1.name << std::endl;

  changeNameWithPointer(&p1,"Schmidt");
  std::cout << p1.name << std::endl;
}
```

Die Ausgabe bestätigt uns, daß tatsächlich bei der Übergabe einer Struktur an eine Funktion, die Struktur kopiert wird.

```
sep@linux:~/fh/prog3/examples/src> g++ -o PassStruct PassStruct.cpp
sep@linux:~/fh/prog3/examples/src> ./PassStruct
Brecht
Schmidt
sep@linux:~/fh/prog3/examples/src>
```

Strukturen in Strukturen Die Attribute einer Struktur können auch wieder Strukturen sein.

StructOfStruct.cpp
```
#include <string>
#include <iostream>

struct S1 {std::string n;std::string n2;std::string n3;};

struct S2 {std::string n;struct S1 s;};

int main (){
  S1 s1 = {"hello","wie","geht's"};
  S2 s2 = {"welt",s1};
  std::cout << s2.s.n << " " << s2.n << std::endl;

  std::cout << sizeof(s1) << std::endl;
  std::cout << sizeof(s2) << std::endl;
}

```

Wie man sieht, können verschiedene Strukturen gleiche Attributnahmen haben.

Der Ausgabe dieses Programms kann man entnehmen, daß bei einer Struktur tatsächlich die Attribute direkt hintereinander gespeichert werden:

```
sep@linux:~/fh/prog3/examples/src> ./StructOfStruct
hello welt
12
16
sep@linux:~/fh/prog3/examples/src>
```

In der Struktur `s2` wird kein Zeiger auf die Instanz der Struktur `s1` abgelegt, sondern die komplette Instanz der Struktur `s2`.

Rekursive Strukturen Wir haben uns im ersten Semester ausgiebig mit Listen als Beispiel einer rekursiven Datenstruktur beschäftigt. Mit Strukturen haben wir auch in C eine Möglichkeit bekommen, rekursive Datentypen zu definieren. Hierzu muß es lediglich innerhalb der Struktur ein Attribut geben, daß einen Zeiger auf eine Instanz von eben jener Struktur enthält. Damit können wir unsere Listenspezifikation aus dem ersten Semester in C++-Strukturen umsetzen. Unsere Spezifikation bestand aus 2 Konstruktoren `Cons` und `Empty`, den Selektoren `head` und `tail`, sowie einer Testfunktion `isEmpty`.

In C können wir eine entsprechende Struktur definieren. In einer Kopfdatei spezifizieren wir die für Listen relevanten Funktionen, die zwei Konstruktoren, die zwei Selektoren, die Testfunktion `isEmpty` und eine weitere Beispielfunktion auf Listen: `length`. Damit präsentieren sich Listen auch in C fast so, wie wir es bereits aus Java gewohnt sind.

IntList.h
```
struct IntList {int head;IntList *tail;bool isEmpty;};

IntList *Cons(int hd,IntList* tl);
```



```
IntList *Empty();

int head(IntList *dieses);
IntList *tail(IntList *dieses);
bool isEmpty(IntList *dieses);

int length(IntList * dieses);
```

Der große Unterschied zu der Implementierung über Klassen ist, daß alle Selektorfunktionen und Testfunktionen die Liste als Parameter benötigen. In der entsprechenden Javaklasse ist dieser Parameter implizit vorhanden durch das Objekt, auf das die Methode aufgerufen wird. Dort konnte dieser eigentlich implizite Parameter über das Schlüsselwort `this` angesprochen werden. In C Strukturen müssen wir diesen impliziten `this`-Parameter explizit machen und der Funktion übergeben.

In der Implementierungsdatei setzen wir unsere Spezifikation um. In den beiden Konstruktoren ist zunächst mit dem `new`-Operator Speicher für die zu konstruierende Liste anzufordern. Die Attribute sind entsprechend zuzuweisen und die neu erzeugte Instanz der Struktur zurückzugeben. Die Selektoren und die Testfunktion greifen jeweils nur auf eines der Attribute zu, die Funktion `length` ist identisch implementiert, wie wir es aus dem ersten Semester kennen.

IntList.cpp
```
#include "IntList.h"

IntList *Cons(int hd,IntList* tl){
  IntList * result = new IntList;
  (*result).head=hd;
  (*result).tail=tl;
  (*result).isEmpty=false;
  return result;
}

IntList *Empty(){
  IntList * result = new IntList;
  (*result).isEmpty=true;
  return result;
}

int head(IntList *dieses){
  return (*dieses).head;
}
IntList *tail(IntList *dieses){
  return (*dieses).tail;
}
bool isEmpty(IntList *dieses){
  return (*dieses).isEmpty;
}

int length(IntList * dieses){
  if (isEmpty(dieses)) return 0;
```

```
  return 1+length((*dieses).tail);
}
```

Zum Testen erzeugen wir eine kleine Liste und geben die Länge ihrer Restliste aus.

TestIntList.cpp
```
#include <iostream>
#include "IntList.h"

int main(){
  IntList *xs = Cons(1,Cons(2,Cons(3,Cons(4,Empty()))));
  std::cout << length(tail(xs))<<std::endl;
}
```

```
sep@linux:~/fh/prog3/examples/src> g++ -c IntList.cpp
sep@linux:~/fh/prog3/examples/src> g++ -c TestIntList.cpp
sep@linux:~/fh/prog3/examples/src> g++ -o TestIntList IntList.o TestIntList.o
sep@linux:~/fh/prog3/examples/src> ./TestIntList
3
sep@linux:~/fh/prog3/examples/src>
```

Benutzen wir die obige Listenimplementierung aus C genauso wie wir es in Java gewohnt waren, so stoßen wir irgendwann auf ein ernstes Speicherproblem. Jeder Aufruf einer der beiden Konstruktorfunktionen `Cons` und `Empty` fordert neuen Speicherplatz an. Wenn wir Listen in unserem Programm erzeugt haben, die nicht mehr benutzt werden, für die es sogar keine Möglichkeit mehr gibt, sie zu benutzen, weil keine Variable, oder auch keine Verzeigerung mehr auf ihren Speicherbereich zeigt, so werden diese trotzdem weiterhin im Speicher stehen. In Java wurden solche Daten irgendwann automatisch aufgeräumt. Javas virtuelle Maschine hat eine automatische Speicherverwaltung, die nicht mehr benutze Datenübereste im Speicher aufräumt. C hat dieses nicht. In C müssen wir solche Speicherbereiche, wie wir bereits gesehen haben, explizit mit dem Operator `delete` wieder freigeben.

Und jetzt einmal ganz in C Die Listenimplementierung im letzten Abschnitt benutzt zwar keine objektorientierten Eigenschaften von C++, benutzt aber eine Vielzahl bequemer Notationen und Konstrukte aus C++, die in C noch mühsam von Hand kodiert werden müssen. Wie wir bereits wissen, gibt es keine bool'schen Werten in C. Wir müssen uns hierzu eine eigene Aufzählung definieren. Desweiteren existiert nicht der bequeme Operator `<<` zur Ausgabe auf Ströme in C. Bei einer genauen Betrachtung des obigen C++ Programms, kann man sehen, daß wir oft auf das Schlüsselwort `struct` beim Typ `IntList` verzichtet haben. Der C++-Übersetzer hat aus dem Typnamen allein abgeleitet, daß es sich um eine Struktur handelt. In C geht dieses nicht.

Diese kleinen Unterschiede spiegeln sich schon in der Kopfdatei der C Implementierung wieder:

CIntList.h
```
enum bool {false,true};

struct IntList{
  int head;
  struct IntList* tail;
```



```
  enum bool empty;
};

struct IntList *Cons(int hd,struct IntList* tl);
struct IntList *Empty();

int head(struct IntList *dieses);
struct IntList *tail(struct IntList *dieses);
enum bool isEmpty(struct IntList *dieses);

int length(struct IntList* dieses);
```

Aber zu allervorderst gibt es keinen Operator `new` in C. In C wird Speicher angefordert, indem man lediglich spezifiziert, wieviel Speicherplatz man benötigt, für das, was zu speichern ist. Hierzu gibt es in C die Funktion `void* malloc(size_t n);`. Ihr ist als Argument mitzugeben, wieviel Speicher angefordert wird. Ihr Ergebnis ist ein Zeiger auf den neuen Speicherbereich. Der Typ für die Daten, auf die der Zeiger zeigt, ist `void`. Es ist ja noch nicht angegeben worden, was für Daten dort zu speichern sind. Durch eine Typzusicherung mit der gleichen Notation wie aus Java bekannt, werden diese Daten dann erst als Daten eines bestimmten Typs interpretiert. In unserer C Implementierung werden entsprechend die `new`-Ausdrücke durch Aufrufe an `malloc` ersetzt. Damit sieht die C Implementierung wie folgt aus.

CIntList.c
```
#include "CIntList.h"

struct IntList *Cons(int hd,struct IntList* tl){
  struct IntList * result
     = (struct IntList*) malloc(sizeof(struct IntList));
  (*result).head=hd;
  (*result).tail=tl;
  (*result).empty=false;
  return result;
}

struct IntList *Empty(){
  struct IntList * result
    = (struct IntList*) malloc(sizeof(struct IntList));
  (*result).empty=true;
  return result;
}

int head(struct IntList *dieses){
  return (*dieses).head;
}
struct IntList *tail(struct IntList *dieses){
  return (*dieses).tail;
}

enum bool isEmpty(struct IntList *dieses){
```

```
  return (*dieses).empty;
}

int length(struct IntList* dieses){
  if (isEmpty(dieses)) return 0;
  return 1+length((*dieses).tail);
}
```

Zum Testen ist jetzt schließlich lediglich zu beachten, daß statt des C++ Operators `<<` ein Funktionsaufruf auf eine Standardausgabefunktion aus C zu erfolgen hat:

TestCIntList.c

```
#include "CIntList.h"

int main(){
  struct IntList *xs
    = Cons(1,Cons(2,Cons(3,Cons(4,Empty()))));
  printf("%i\n",length(tail(xs)));
}
```

Der Test liefert wieder das zu erwartende Ergebnis:

```
sep@linux:~/fh/prog3/examples/src> gcc -c TestCIntList.c
sep@linux:~/fh/prog3/examples/src> gcc -c CIntList.c
sep@linux:~/fh/prog3/examples/src> gcc -o TestCIntList TestCIntList.o CIntList.o
sep@linux:~/fh/prog3/examples/src> ./TestCIntList
3
sep@linux:~/fh/prog3/examples/src>
```

Aufgabe 8 Implementieren Sie für unsere in C++ implementierten Listenstruktur die folgenden Funktionen (nach der Spezifikation aus dem ersten Semester):

a) `int last(struct IntList * dieses)`

b) `struct IntList * concat(struct IntList * xs,struct IntList * ys)`

c) `int elementAt(struct IntList * dieses,int i)`

d) `int fold(struct IntList * dieses,int start,int(*foldOp)(int,int))`

Die Pfeilnotation Im letzten Beispiel haben wir oft mit Zeigern auf Strukturen zu tun gehabt. Um auf die Attribute dieser Strukturen zuzugreifen muß erst der Zeiger aufgelöst werden, um dann schließlich mit der Punktnotation auf die referenzierte Struktur zuzugreifen. Wir erhalten dann einen Ausdruck wie z.B.: `(*dieses).head;`. Ausdrücke dieser Art kommen sehr häufig beim Arbeiten mit C Strukturen und auch mit C++ Klassen vor. Daher gibt es eine abkürzende Schreibweise hierfür: den Operator `->`, der einen Pfeil darstellen soll. Es läßt sich also `dieses->head` statt `(*dieses).head;` schreiben.

Beispiel:
Ein kleines C Beispiel zum Gebrauch der Pfeilnotation.

Arrow.c

```
struct Person
  {char* name
  ;char* vorname
  ;};

void setName(struct Person* p,char* newName){
  p->name=newName;
}

int main(){
  struct Person p1 = {"Hotzenplotz", "Walther"};
  struct Person * pP1 = &p1;
  printf("%s\n",p1.name);
  printf("%s\n",(*pP1).name);
  printf("%s\n",pP1->name);
  setName(pP1,"Zabel");
  printf("%s\n",p1.name);
  printf("%s\n",(*pP1).name);
  printf("%s\n",pP1->name);
}
```

Die drei Arten auf den Namen der Struktur zuzugreifen, ergeben jeweils dasselbe Ergebnis.

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

- Objektmethoden
- Vererbeung
- 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.

Zunächst einmal gibt es ein 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

```
// pseudo class for persons
struct CPerson
```

```
  {char* name
  ;char* vorname
  ;char* strasse
  ;char* stadt
  ;int plz
  ;char* (*getFullNameFunction)(struct CPerson*)
  ;};

//pseudo instance method for some this object
char* getFullName(struct CPerson * this);

//constructor for CPerson
struct CPerson* newPerson
  (char* n, char* v,char* str,char* st,int p);
```

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 können. Es reicht somit aus, ein Attribut für eine Instanz der Pseudosuperklasse vorzusehen und danach die zusätzlichen Attribute zu definieren.

CPerson.h

```
// pseudo subclass of CPerson for students
struct CStudent
  {struct CPerson super
  ;int matrNr;
  };

//constructor for CStudent
struct CStudent* newStudent
  (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

```
#include "CPerson.h"

int main(){
  //create a person object
  struct CPerson* p1
   = newPerson("Schmidt","Harald","studio449","koeln",54566);

  //create a student object, but store it in a Person variable
  struct CPerson* p2
```



```
   = newStudent("Andrak","Manuel","studio449","koeln",54566
               ,565456);

  //print the full names of the two person objects
  printf("die erste Person ist: %s\n",getFullName(p1));
  printf("der zweite Person ist: %s\n",getFullName(p2));

  //make a cast of the second person to student
  printf("mit Matrikel-Nr.: %i\n",((struct CStudent*)p2)->matrNr);
}
```

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

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

char* getFullPersonName(struct CPerson* this){
  size_t nameLength = strlen (this->name);
  size_t vornameLength = strlen (this->vorname);
  char* result
   = malloc((nameLength + vornameLength+3) * sizeof(char));
  char* end = result;
  end=(char*)stpcpy(end,this->name);
  end=(char*)stpcpy(end,", ");
  end=(char*)stpcpy(end,this->vorname);
  return result;
}

char* getFullStudentName(struct CPerson* this){
  size_t nameLength = strlen (this->name);
  size_t vornameLength = strlen (this->vorname);
  char* result
   = malloc((nameLength + vornameLength+3) * sizeof(char));
  char* end = result;
  end=(char*)stpcpy(end,this->vorname);
  end=(char*)stpcpy(end," ");
  end=(char*)stpcpy(end,this->name);
  return result;
}
```

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
```
void initPerson(struct CPerson* result
               ,char* n, char* v,char* str,char* st,int p){
  result->name = n;
  result->vorname = v;
  result->strasse = str;
  result->stadt = st;
  result->plz = p;
  result->getFullNameFunction = &getFullPersonName;
}
```

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
```
struct CPerson* newPerson
   (char* n, char* v,char* str,char* st,int p){
  struct CPerson* result
    = (struct CPerson*) malloc(sizeof(struct CPerson));
  initPerson(result, n,  v, str, st, p);
  return result;
}
```

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 Java 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
```
struct CStudent* newStudent
   (char* n, char* v,char* str,char* st,int p,int ma){
  struct CStudent* result
    = (struct CStudent*) malloc(sizeof(struct CStudent));
  initPerson(&(result->super), n,  v, str, st, p);

  (result->super).getFullNameFunction = &getFullStudentName;

  result->matrNr = ma;

  return result;
}
```


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

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.

Typsumme mit Unions

Im letzen Abschnitt haben wir mittels Strukturen Produkttypen zwischen mehreren Typen definiert. Die zweite aus der theoretischen Informatik bekannte Typbildung ist die Summe von Typen. Auch hierfür stellt C ein eigenes Konstrukt zur Verfügung: die Vereinigung von Typen, die `union`.

Syntaktisch sieht eine Typvereinigung einer Struktur sehr ähnlich. Lediglich das Schlüsselwort `struct` ist durch das Schlüsselwort `union` zu ersetzen. Somit läßt sich eine Vereinigung der Typen `int` und `char*` definieren als:
`union stringOrInt {int i;char *s;};`.

Daten des Typs `union stringOrInt` sind nun entweder eine Zahl, die über das Attribut `i` angesprochen werden kann, oder der Zeiger auf das erstes Element einer Zeichenkette, der über das Attribut `s` angesprochen werden kann.

Beispiel:
Im folgenden kleinem Programm wird eine Typsumme definiert und beispielhaft in einer Hauptmethode benutzt.

Union1.cpp
```
#include <iostream>

union stringOrInt {int i;char *s;};

int main(){
  union stringOrInt sOrI;
```

```
  sOrI.s="hallo da draussen";
  std::cout << sOrI.s << std::endl;

  sOrI.i=42;
  std::cout << sOrI.i << std::endl;
}
```

Wie man der Ausgabe entnehmen kann, wird in der Variablen des Typs `stringOrInt` entweder ein `int` oder eine Zeichenkettenreferenz gespeichert:

```
sep@linux:~/fh/prog3/examples/src> g++ -o Union1 Union1.cpp
sep@linux:~/fh/prog3/examples/src> ./Union1
hallo da draussen
42
sep@linux:~/fh/prog3/examples/src>
```

Im letzten Beispiel haben wir die Typsumme gutartig benutzt. Die Typsumme drückt aus, daß die Daten entweder von einem oder vom anderen Typ sind. Im Beispiel haben wir uns gemerkt, welchen Typ wir als letztes in der entsprechenden Variablen gespeichert haben und nur auf diesen Typ zugegriffen. C bewahrt uns aber nicht davor, dieses immer korrekt zu tun.

Beispiel:
In diesem Beispiel benutzen wir den selben Summentypen wie im vorhergehenden Beispiel. Jetzt interpretieren wir aber jeweils die Daten auch als die andere Typalternative.

Union2.cpp
```
#include <iostream>

union stringOrInt {int i;char *s;};

int main(){
  union stringOrInt sOrI;
  sOrI.s="hallo da draussen";
  std::cout << sOrI.s << std::endl;
  std::cout << sOrI.i << std::endl;

  sOrI.i=42;
  std::cout << sOrI.i << std::endl;
  std::cout << sOrI.s << std::endl;
}
```

Wenn wir jetzt einen Adresse auf einen Zeichenkette in der Variablen `sOrI` gespeichert haben und auf die Variable mit ihrem `int`-Attribut `i` zugreifen, so bekommen wir die gespeicherte Adresse als Zahl geliefert.

Wenn wir umgekehrt eine Zahl gespeichert haben und diese nun als Adresse einer Zeichenkette lesen wollen, so kommt es in der Regel zu einem Fehler, weil wir höchstwahrscheinlich versuchen eine Adresse im Speicher zu lesen, auf die wir nicht zugreifen können.


```
sep@linux:~/fh/prog3/examples/src> ./Union2
hallo da draussen
134514920
42
Speicherzugriffsfehler
sep@linux:~/fh/prog3/examples/src>
```

Um Fehler, wie im letzem Beispiel zu vermeiden, ist es ratsam Strukturen und Typsummen geschachtelt zu verwenden. Hierzu kapselt man die Typsumme in eine Struktur. In dieser Struktur gibt ein zweites Attribut an, um welchen Typ es sich gerade in der Typsumme handelt. Für das Attribut, daß den Typ kennzeichnet, bietet sich an, einen Aufzählungstypen zu verwenden.

Beispiel:
In diesem Beispiel definieren wir eine Typsumme, die in einer Struktur eingebettet ist. Dazu definieren wir drei Typen: eine Aufzählung, eine Typsumme und eine Struktur.

Union3.cpp
```
#include <iostream>

enum TypeMarker {INT,STRING};
union stringOrInt {int i;char * s;};

struct strOrInt {TypeMarker type;union stringOrInt v;};
```

Jetzt können wir uns zwei Funktionen schreiben, die jeweils eine der beiden Varianten unseres Summentypens konstruieren:

Union3.cpp
```
struct strOrInt forInt(int i){
  strOrInt result;
  result.type=INT;
  result.v.i=i;
  return result;
}

struct strOrInt forString(char * s){
  strOrInt result;
  result.type=STRING;
  result.v.s=s;
  return result;
}
```

Schließlich ein Beispiel für eine Funktion, die den neuen Typen benutzt. Hierzu ist zunächst zu fragen, von was für einen Typ denn die gespeichert sind, um dann entsprechend mit den Daten zu verfahren:

Union3.cpp
```
char * toString(struct strOrInt si){
  char* result;
  if (si.type==INT){
    sprintf(result,"%d",si.v.i);
```

```
  }else if (si.type==STRING){
    result=si.v.s;
  }
  return result;
}
```

Abschließend ein kleiner Test in einer Hauptmethode:

Union3.cpp
```
int main(){
  strOrInt test = forString("hallo da draussen");
  std::cout << toString(test)<<std::endl;
  test = forInt(42);
  std::cout << toString(test)<<std::endl;
}
```

In der Ausgabe können wir uns davon überzeugen, daß tatsächlich immer die Daten der Typsumme korrekt interpretiert werden:

```
sep@linux:~/fh/prog3/examples/src> g++ -o Union3 Union3.cpp
sep@linux:~/fh/prog3/examples/src> ./Union3
hallo da draussen
42
sep@linux:~/fh/prog3/examples/src>
```

Schließlich ist noch interessant, wieviel Speicher C für eine Typsumme reserviert. Auch dieses können wir experimentel erfragen:

Union4.cpp
```
#include <iostream>

union U1 {char a1 [15]; char a2 [200];};
union U2 {int a1; char a2 [16];};

int main(){
  U1 u1;
  U2 u2;
  std::cout << sizeof(u1) << std::endl;
  std::cout << sizeof(u2) << std::endl;
 }
```

An der Ausgabe ist zu erkennen, daß der maximal notwendige Speicher für einen der Typen in der Summe angefordert wird.

```
sep@linux:~/fh/prog3/examples/src> ./Union4
200
16
sep@linux:~/fh/prog3/examples/src>
```

Kapitel 3

Generische und Überladene Funktionen

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
```
template <typename a>
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.

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

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

Trace.cpp
```
#include <iostream>
#include "Trace.h"

template <typename a>
a trace(a x){
  std::cout << ">>>trace: " << x << std::endl;
  return x;
}
```

Jetzt können wir die Funktion für einfache Beispiele einmal ausprobieren:

Trace.cpp
```
int main(){
  int x = 5 + trace(21*2);
  char *str = "hallo";
  trace(str);
  x = trace(6 + trace(18*2));
}
```

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 Typvariabel `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
```
#include <iostream>
#include "Trace.h"

```



```
template <typename a>
a trace(a x){
  std::cout <<">>>trace: "<< x << std::endl;
  return x;
}

struct S {int x;char* s;};

int main(){
  S u = {42,"hallo"};
  u = trace(u);
}
```

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 wird dieses möglich sein.

Aufgabe 9 Schreiben Sie eine Funktion zur Funktionakomposition mit folgender Signatur:

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

Sie sei spezifiziert durch die Gelichung: $\mathit{composition}(f_2, f_1, x) = f_2(f_1(x))$. 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
```
template <typename a>
a trace(a x);
```

Und die entsprechende Implementierungsdatei:

T.cpp
```
#include <iostream>
#include "T.h"

template <typename a>
a trace(a x){
  std::cout << ">>>trace: " << x << std::endl;
  return x;
}
```

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

UseT.cpp
```
#include "T.h"

int main(){
  trace(2*21);
}
```

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 Expizite 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
```
#include "T.cpp"

int main(){
  trace<float>(2*21656567);
  trace<int>(2*21656567);
}
```

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.

Wir haben im Kapitel über Reihungen eine Funktion `map` geschrieben mit folgender Signatur:
`void map(int xs [],int l,int (*f) (int));`
Hier wurde auf jedes Reihungselement eine Funktion angewendet. Die Funktion `map` war so aber nur für Reihungen, deren Elemente ganze Zahlen sind, definiert. Mit generischen Fuktionen können wir diese Funktion veralgemeinern. 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

```
#include <iostream>

template <typename a, typename b>
void map(a xs [],b ys [],int l,b (*f) (a)){
  for (int i=0;i<l;i++){
    ys[i]=f(xs[i]);
  }
}
```

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

```
template <typename a>
void printArray(a xs [],int l){
  std::cout<<"{";
  for (int i=0;i<l-1;i++){
    std::cout << xs[i]<<", ";
  }
  std::cout << xs[l-1] <<"}"<<std::endl;
}
```

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

GenMap.cpp

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

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

GenMap.cpp

```
int main(){
  int xs [] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  int ys [15];
  map(xs,ys,15,&add5);
  printArray(ys,15);

  std::string us  []= {"hallo","jetzt","wirds","cool"};
  char vs [4];
  map(us,vs,4,&charAt2);
  printArray(vs,4);

  std::string ws [4];
  map(us,ws,4,&doubleString);
  printArray(ws,4);
}
```

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 10 Schreiben Sie jetzt eine generische Fuktion `fold` auf Reihungen und testen Sie diese auf verschiedenen Typen aus.

Aufgabe 11 Verallgemeinern Sie jetzt Ihre Funktion `bubbleSortBy` zu einer generischen Funktion, mit der sie Reihungen beliebiger Typen sortieren können.

3.2 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. Dieses Prinzip kennen wir bereits aus Java.

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

OverloadedTrace.cpp

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

struct Person
  {std::string name
  ;std::string vorname
  ;};

Person trace(Person& p){
  std::cout << p.vorname << " " << p.name<< std::endl;
  return p;
}

int trace (int i){
  std::cout << i << std::endl;return i;
}

float trace (float i){
  std::cout << i << std::endl;return i;
}

std::string trace (std::string i){
  std::cout << i << std::endl;return i;
}

int main(){
  Person p1 = {"Zabel", "Walther"};
  Person p2 = trace(p1);
  trace (456456456);
  trace (425456455.0f);
  std::string s1 = "hallo";
  std::string s2 = trace (s1);
}
```

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>
```

3.3 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

```
#include <iostream>

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

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

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, so wie wir es bereits aus Java gewohnt sind, 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:~>
```

Kapitel 4

Objektorientierte Programmierung in C++

Eine der wichtigsten Erweiterungen von C nach C++ ist die Einführung der objektorientierten Programmierung. Hierzu wurden in C++ Strukturen erweitert, so daß mit ihnen folgende aus Java bereits bekannte Konzepte ausgedrückt werden können:

- Objektmethoden
- Konstruktoren
- Vererbung
- späte Bindung
- Erreichbarkeiten

Verwirrend für den Javaprogrammierer mag zunächst sein, daß es keine allgemeine Oberklasse `Object` gibt. Auch das aus Java bekannte Konzept von Schnitstellen gibt es in C++ nicht. Hingegen kann eine Klasse mehrere Oberklassen haben.

4.1 Klassen

Klassen sind eine natürliche Erweiterung von Strukturen. Die Syntax ist entsprechend dieselbe. Lediglich statt des Schlüsselworts `struct` ist das Schlüsselwort `class` zu benutzen.

4.1.1 Klassendeklaration

In einer Klasse gibt es wie in Java Felder und Methode. Anders als in Java sind in C++ standardmäßig alle Eigenschaften einer Klasse als `private` deklariert, d.h. sie dürfen nur innerhalb der Klasse benutzt werden. Mit dem Schlüsselwort `public` gefolgt von einem Doppelpunkt werden alle folgenden Eigenschaften der Klasse als öffentlich für jeden auf den Objekten zugreifenden Benutzer deklariert. Dieses kann mit dem Schlüsselwort `private` gefolgt von einem Doppelpunkt für weitere Eigenschaften wieder zurückgesetzt werden.

Wie wir für Strukturen bereits gesehen haben, bekommen wir auch für Klassen nicht automatisch wie in Java Referenzen auf Objekte, sondern die Objekte direkt im Speicher. Die Zuweisung bewirkt auch für Objekte daher eine Kopierung des Objekts. Die Initialisierung von Objekten kann wie bei Strukturen auch durch eine Aufzählung der Felder in geschweiften Klammern erfolgen.

Beispiel:
Unsere erste C++ Klasse zur Darstellung von Personen:

P1.cpp
```
#include <iostream>

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

int main(){
  Person p1;
  p1.nachname="Zabel";
  p1.vorname="Walther";
  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;
  Person p3 = {"Shakespeare","William"};
  std::cout << p3.getFullName() << std::endl;
}
```

An der Ausgabe kann man wieder deutlich erkennen, daß eine Zuweisung ein Objekt kopiert.

```
sep@linux:~/fh/prog3/examples/src> ./P1
Walther Zabel
Walther Zabel
Walther Zabel
Walther Hotzenplotz
William Shakespeare
sep@linux:~/fh/prog3/examples/src>
```

4.1.2 Header-Dateien 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.

P2.h
```
#include <string>

class Person {

public:
  std::string nachname;
  std::string vorname;
  std::string getFullName();
};
```

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;}
```

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

TestP2.cpp
```
#include "P2.h"
#include <iostream>

int main(){
  Person p1;
  p1.nachname="Zabel";
  p1.vorname="Walther";
  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;
  Person p3 = {"Shakespeare","William"};
  std::cout << p3.getFullName() << std::endl;
}
```

Das Programm verhält sich wieder wie die vorherige Version ohne Kopfdatei:

```
sep@linux:~/fh/prog3/examples/src> g++ -c TesteP2.cpp
sep@linux:~/fh/prog3/examples/src> g++ -c P2.cpp
sep@linux:~/fh/prog3/examples/src> g++ -c TestP2.cpp
sep@linux:~/fh/prog3/examples/src> g++ -o TestP2 P2.o TestP2.o
sep@linux:~/fh/prog3/examples/src> ./TestP2
```



```
Walther Zabel
Walther Zabel
Walther Zabel
Walther Hotzenplotz
William Shakespeare
sep@linux:~/fh/prog3/examples/src>
```

4.1.3 Konstruktoren

Aus Java kennen wir Konstruktoren. Auch in C++-Klasse können Konstruktoren definiert werden. Syntaktisch wird dieses ebenso wie in Java als Methoden ohne Rückgabetyp mit dem Namen der Klasse notiert. Ebenso wie in Java gibt es in C++ auch den `this`-Zeiger, der als Zeiger auf das Objekt dient, für das eine Methode oder ein Konstruktor aufgerufen wurde. Da `this` eine Zeigervariable ist, wird für sie nicht mit dem Punktoperator sondern mit dem Pfeiloperator auf die einzelnen Eigenschaften zugegriffen.

Beispiel:
Wir können jetzt eine Klasse `Person` schreiben, die so wie wir es aus Java gewohnt sind, einem Konstruktor hat, in dem alle relevanten Felder der Klasse explizit gesetzt werden:

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

class Person {

  private:
   std::string nachname;
   std::string vorname;
   std::string strasse;
   int hausnummer;
   int plz;
   std::string ort;

  public:
    Person(std::string nachname ,std::string vorname
          ,std::string strasse  ,int hausnummer
          ,int plz              ,std::string ort);

    std::string getFullName();
    std::string getAddress();
    void setStrasse(std::string strasse);
};
```

Person.cpp
```
#include <string>
#include <iostream>
#include "Person.h"

Person::Person(std::string nachname,std::string vorname
              ,std::string strasse,int hausnummer
```

```
              ,int plz,std::string ort){
  this->nachname=nachname;
  this->vorname=vorname;
  this->strasse=strasse;
  this->hausnummer=hausnummer;
  this->plz=plz;
  this->ort=ort;
}

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

std::string Person::getAddress(){
  char hausnr[20];
  sprintf(hausnr,"%d",hausnummer);

  char plznr[20];
  sprintf(plznr,"%d",plz);

  return strasse+" "+hausnr +"\n"+" "+plznr+ort;
}

void Person::setStrasse(std::string strasse){
  this->strasse = strasse;
}
```

Bei der Benutzung der Klasse `Person` ist jetzt zu beachten, daß sobald eine Variable als vom Typ `Person` deklariert wird, C++ einen Konstruktor von dieser Klasse aufrufen will. Anders als in Java, wo Variablen nur Referenzen auf die entsprechenden Objekte dargestellt haben, gilt in C++, daß Variablen direkt die Daten bezeichnen. Daher führt eine Variablendeklaration direkt zum Aufruf eines Konstruktors. Wenn ein Konstruktor mit Parametern benutzt werden soll, so sind die konkreten Parameter in runden Klammern dem Variablennamen direkt folgend anzugeben.

TestePerson1.cpp

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

int main(){
  Person p1("Schmidt","Helmut","Brüderstrasse"
           ,82,20121,"Hamburg");
  Person p2("Kohl","Helmut","Saumagenstrasse"
           ,98,60121,"Ludwigshafen");

  std::cout << p1.getFullName() << std::endl
            << p1.getAddress()  << std::endl;
  std::cout << p2.getFullName() << std::endl
            << p2.getAddress()  << std::endl;

```



```
  p1.setStrasse("Sonnenstrasse");
  std::cout << p1.getFullName() << std::endl
            << p1.getAddress()  << std::endl;
}
```

Hieraus folgt insbesondere, daß, sobald es keinen Konstruktor ohne Parameter gibt, keine Variable ohne die Angabe der Parameter für den Konstruktor mehr deklariert werden kann. C++ will nämlich in diesem Fall den leeren Konstruktor benutzen, den es nicht gibt.

Folgendes Programm gibt daher einen Übersetzungsfehler.

PersonKonstruktorError.cpp

```
#include "Person.h"

int main(){
  Person p;
}
```

Der C-Compiler beschwert sich über einen fehlenden Konstruktor:

```
sep@linux:~/fh/prog3/examples/src> g++ PersonKonstruktorError.cpp
PersonKonstruktorError.cpp: In function ‘int main()’:
PersonKonstruktorError.cpp:4: error: no matching function for call to ‘Person::
   Person()’
Person.cpp:4: error: candidates are: Person::Person(const Person&)
Person.cpp:21: 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> >, std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >, int, int,
   std::basic_string<char, std::char_traits<char>, std::allocator<char> >)
sep@linux:~/fh/prog3/examples/src>
```

Konstruktion mit new

Wir kennen schon den Operator `new` zum Erzeugen von Zeigern auf Daten eines bestimmten Typs. In Zusammenhang mit Klassen bekommt dieser Operator die aus Java bekannte Semantik. Dem Operator `new` folgt der Name des Typs, für den ein neuer Speicherbereich reserviert werden soll; in unserem Fall jetzt ist dies die Klasse. Dem Klassennamen folgen die konkreten Parametern für einen der Konstruktoren. Das Ergebnis ist ein Zeiger auf ein neues Objekt der Klasse. Dieses wird auch deutlich im Typ. Eine Variable die mit dem Operator `new` für die Klasse `Person` erzeugt wird, ist nicht vom Typ `Person` sondern `Person*`.

Beispiel:
Jetzt benutzen wir die Klasse `Person`, indem wir die Objekte über den Operator `new` erzeugen. Wir speichern das Ergebnis in Variablen des Zeigertyps `Person*`. Zum Benutzen der Eigenschaften dieses Zeigertyps müssen wir den Pfeiloperator bedienen und nicht wie zuvor den Punktoperator.

TestePerson2.cpp

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

```

```
int main(){
  Person* p1 = new Person("Schmidt","Helmut","Brüderstrasse"
                         ,82,20121,"Hamburg");
  Person* p2 = new Person("Kohl","Helmut","Saumagenstrasse"
                         ,98,60121,"Ludwigshafen");

  std::cout << p1->getFullName() << std::endl
            << p1->getAddress()  << std::endl;
  std::cout << p2->getFullName() << std::endl
            << p2->getAddress()  << std::endl;

  p1->setStrasse("Sonnenstrasse");
  std::cout << p1->getFullName() << std::endl
            << p1->getAddress() << std::endl;
}
```

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:

```
this->nachname=nachname;
```

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.

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

PersonAlternativ.cpp

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

Person::Person(std::string nachname,std::string vorname
              ,std::string strasse,int hausnummer
              ,int plz,std::string ort):
   nachname(nachname),vorname(vorname),strasse(strasse)
  ,hausnummer(hausnummer),plz(plz),ort(ort){}
```

4.1.4 Destruktoren

Im Zusammenhang mit der dynamischen Speicherverwaltung haben wir den zum Operator `new` dualen Operator `delete` kennengelernt. Ein Pendant zu `delete` gab es in Java nicht. Die virtuelle Maschine gibt nicht mehr benutzte Speicherzellen automatisch wieder frei. In C++ muß der Programmierer die dynamische Speicherverwaltung selbst vornehmen. Als Faustregel ist zu merken, daß für jedes `new` im Programmdurchlauf irgendwann einmal ein `delete` ausgeführt werden muß.

Betrachten wir einmal genau die dynamische Speicherverwaltung für ein Programm. Hierzu bedienen wir uns der einfachen verketteten Listenimplementierung, wie wir sie für Java implementiert hatten. Listen seien wieder durch die zwei Konstruktoren, den zwei Selektoren und einer Testmethode spezifiziert. Zusätzlich wollen wir eine Funktion implementieren, die zwei Listen konkateniert. Um ein zu dem entsprechenden Javaprogramm analoge Funktionalität zu haben, müssen wir explizit mit Zeigern auf die Restliste arbeiten.

StringList.h

```
#include <string>

class StringList {
  private:
    bool empty;
    std::string hd;
    StringList*  tl;

  public:
    StringList();
    StringList(std::string hd,StringList* tl);

    bool isEmpty();
    std::string head();
    StringList* tail();

    StringList* concat(StringList* other);
};
```

Die Funktionen lassen sich relativ einfach implementieren.

StringList.cpp

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

StringList::StringList():empty(true){};
StringList::StringList(std::string hd,StringList* tl):
   empty(false),hd(hd),tl(tl){}

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

StringList* StringList::concat(StringList* other){
  if (isEmpty()) return other;
  return new StringList(head(),tail()->concat(other));
}
```

Folgendes kleines Testprogramm wollen wir in seiner Ausführung genau betrachten. Wir erzeugen zwei Listen, die in lokalen Variablen gespeichert werden. Dann konkatenieren wir diese beiden Listen und speichern die Ergenisliste in einer weiteren lokalen Variablen. Anschließend rufen wir den Operator `delete` für die erste Liste auf und weisen allen lokalen Variablen das Ergebnis der Konkatenation zu.

ConcatTest.cpp

```
1  #include <iostream>
2  #include "StringList.h"
3
4  int main(){
5    StringList* xs = new StringList("friends"
6                     ,new StringList("romans"
7                     ,new StringList("contrymen"
8                     ,new StringList())));
9
10   StringList* ys = new StringList("lend"
11                    ,new StringList("me"
12                    ,new StringList("your"
13                    ,new StringList("ears"
14                    ,new StringList()))));
15
16   StringList* zs = xs->concat(ys);
17
18   delete xs;
19   ys=zs;
20   xs=zs;
21 }
```

Betrachten wir in einer vereinfachten Darstellung, wie der Speicher aussieht, nachdem Zeile 16 des Programms ausgeführt wurde. Der Speicher teilt sich in zwei Bereiche, den sogenannten Keller *(stack)*, auf dem lokale Variablen und Parameterwerte abgespeichert werden und dem *heap* zur Speicherung dynamisch erzeugter Daten. Für die lokalen Variablen der Methode `main` ist Speicher auf dem *stack* angelegt worden. In diesen Speicherzellen, stehen für die drei lokalen Zeigervariablen `xs`, `ys` und `zs` Adressen aus dem Speicher. Der *stack* sieht also in etwa wie folgt aus:

zs	22
ys	13
xs	1

Diese Adressen wurden durch die Ausführung des Operators `new` während der Laufzeit des Programms errechnet. Insgesamt wurde 12 mal während des Programmdurchlaufs der Operator `new` ausgeführt. Viermal beim Erzeugen der Liste `xs`, fünfmal beim Erzeugen der Liste `ys` und schließlich dreimal bei der Ausführung der Funktion `concat`. (`concat` erzeugt für jeden nichtleeren Listenknoten des `this`-Objekts einen neuen Listenknoten.) Jedesmal wurde Speicher mit `new` angefordert, um ein Objekt der Klasse `StringList` abzuspeichern. In

unserer vereinfachten Form gehen wir davon aus, daß für jedes Feld der Klasse `StringList` eine Adresse zum Speichern benutzt wird. Ein `new` für `StringList` reserviert in unserer Darstellung einen Block von drei Speicherzellen. Insgesamt werden also $12*3$ Speicherzellen im *heap* reserviert. In der ersten Zelle dieser Tripel steht der bool'sche Wert, der angibt, ob es sich um eine leere Liste handelt, in der zweiten Zelle der Kopf des Listenknotens und in der dritten Zelle die Adresse der Schwanzliste. Man beachte, daß auch für leere Listen die Zellen für Kopf und Schwanz existieren, auch wenn dort undefinierte Werte drinstehen.

Die folgende Tabelle gibt in diesem vereinfachten Modell ein Abbild des `heap` nach Ausführung der Listenkonkatenation:

1	false
2	"friends"
3	4
4	false
5	"romans"
6	7
7	false
8	countrymen
9	10
10	true
11	
12	
13	false
14	"lend"
15	16
16	false
17	"me"
18	19
19	false
20	"your"
21	22
22	false
23	"ears"
24	25
25	true
26	
27	
28	false
29	"friends"
30	31
31	false
32	"romans"
33	34
34	false
35	"countrymen"
36	13

36 Speicherzellen wurden reserviert. Auf die Speicherzelle 13 wird zweimal verwiesen. Einmal von Zelle 36 aus und einmal vom *stack* aus. In Zeile 28 des Programms fordern wir jetzt C++ auf, den Speicher auf dem die Variable `xs` zeigt im Speicher wieder frei zu geben. C++ weiß, daß es sich um einen Zeiger auf ein Objekt des Typs `StringList` handelt und solche Objekte drei Speicherzellen belegen. Daher gibt C++ die Speicherzellen 1, 2 und 3 wieder frei. Die Speicherzellen 4 bis 12, die auch zur Liste `xs` gehören bleiben hingegen erhalten. Der `delete`-Operator löscht zunächst nur genau das eine Objekt auf dem gezeigt wird. Es werden in diesem Objekt durch Zeigervariablen weiter referenzierte Objekte nicht mitgelöscht. Führen wir schließlich noch die letzten zwei Befehle des Programms aus, so ergibt sich auf dem *stack* folgende Situation:

zs	28
ys	28
xs	28

Im *heap* existieren jetzt mit den Zellen 4 bis 12 Speicherzellen, die nirgends mehr referenziert werden. Wir haben ein Speicherleck. Es wird an Speicher festgehalten, auf den nicht mehr zugegriffen werden kann. Wir hätten statt nur den Knoten, auf den `xs` zeigt, zu löschen, auch jeweils die Schwanzlisten für alle Knoten die über `xs` zu erreichen sind, löschen müssen. Statt `delete xs` hätte man also eine Funktion benötigt, die rekursiv alle Schwanzknoten löscht, bevor der Knoten selbst gelöscht wird:

```
void deleteList(StringList* xs){
  if (!xs->isEmpty()){
    deleteList(xs->tail());
  }
  delete xs;
}
```

Genau hierfür gibt es in C++ ein Konstrukt, den Destruktor. Ein Destruktor hat keinen Parameter. Der Destruktor enthält den Code, der für ein Objekt ausgeführt wird, wenn für einen Zeiger auf dieses Objekt der Operator `delete` aufgerufen wird, bevor das Objekt selbst aus dem Speicher entfernt wird. Der Destruktor wird in C++ durch den Klassennamen mit vorangestellter Tilde ˜ bezeichnet. Ebenso wie bei Konstruktoren wird, wenn kein Destruktor im Programm definiert wird, von C++ ein leerer Destruktor generiert.

Beispiel:
Jetzt ergänzen wir die Klasse `StringList` um eine Destruktor, der dafür sorgt, daß mit `delete` nicht nur der ersten Knoten einer Liste aus dem Speicher entfernt wird, sondern die komplette Liste.

StringListDestructed.cpp

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

class StringList {
  private:
```



```
    bool empty;
    std::string hd;
    StringList*  tl;

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

    StringList(){
      empty= true;
    }

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

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

    StringList* concat(StringList* other){
      if (isEmpty()) return other;
      return new StringList(head(),tail()->concat(other));
    }
};

int main(){
  StringList* xs = new StringList("friends"
                  ,new StringList("romans"
                  ,new StringList("contrymen"
                  ,new StringList())));

  StringList* ys = new StringList("lend"
                  ,new StringList("me"
                  ,new StringList("your"
                  ,new StringList("ears"
                  ,new StringList()))));

  StringList* zs = xs->concat(ys);

  delete xs;
  ys=zs;
  xs=zs;
}
```

Was soll ich löschen?

Die `delete`-Operation kann schwierig sein. Wann genau darf ein Objekt aus dem Speicher gelöscht werden. Man mag geneigt sein im obigen Beispiel auch einen Aufruf `delete ys` einzufügen. Dann würden mittels unserers Destruktors auch die Speicherzellen 13 bis 27 gelöscht werden. Das wäre nicht gut, denn ausgehend über die Speicherzelle 28 für die Liste `zs` kann man eine Zeigerkette bilden, die Schließlich in Adressfeld 36 auf die Adresse 13 verweist. Wenn wir mit `delete ys` diesen Speicherbereich wieder freigegeben hätten, dann würden wir jetzt auf undefinierte Werte stoßen. Der Grund dafür ist, daß wir in der Funktion `concat` keine Kopie der zweiten Liste erzeugen, sondern ein zweites Mal auf den Speicherbereich dieser Liste verweisen. Da dieses in der Funktion `concat` passiert, können wir das nicht unbedingt wissen, wenn die Funktion `concat` nicht ganz genau darüber dokumentiert ist. Für die Programmierung und Entwicklung vobn Bibliotheken in C++ ist daher ganz wichtig, nur in Ausnahmefällen und gut dokumentiert Zeiger auf bestimmte Speicherbereiche zu kopieren, so daß über mehrere Zeiger auf den gleichen Bereich verwiesen wird. Ansonsten ist die explizite Speicherverwaltung entsprechend schwierig und schwer zu handhaben.

Wie macht Java das?

In Java haben wir uns nie Gedanken darüber machen müssen, ob in Speicher noch irgendwelcher Datenreste von Objekten liegen, die wir nicht mehr benötigen. Java hat eine eigene Müllentsorgung, die solche nicht mehr benötigten Objekte aus dem Speicher löschen. Wie macht Java das?

Bleiben wir bei unseren obigen Beispiel. Wie wir uns überzeugt haben können die Speicherzellen 1 bis 12 nach Abarbeitung der Methode nicht mehr erreicht werden. Turnusmäßig, wenn der Speicherbereich im *heap* knapp wird, startet Java die Müllentsorgung. Hierzu folgt Java allen Zeigern, die auf dem *stack* liegen und führt Buch darüber, welche Adressen im *heap* damit erreicht werden. In unserem Beispiel wird vom *stack* aus nur noch die Speicherzelle 28 erreicht. Hier liegt ein Objekt, das in Zelle 30 auf die Zelle 31 verweist. Auch dieses vermerkt sich Java. Java bildet den transitiven Abschluß der Zeigerketten und erhält damit die Menge aller noch erreichbaren Speicherzellen. Diese Speicherzellen kopiert sich Java in einen neuen frischen Speicherbereich. Die Zeigervariablen müssen dabei auf die Werte in den neuen Speicherbereich umdefiniert werden. Ist dieser Prozess abgeschlossen, so hat Java die noch benötigt Daten aus dem Speicher herausgezogen. Alles andere ist Müll gewesen. Java gibt komplett den alten *heap* zur Neuverwendung wieder frei. Das Bild der Müllentsorgung ist nach diesem Prinzip etwas ungenau. Java schmeißt nicht den Müll weg, sondern rettet die noch benötigten Daten und schmeißt dann alles weg.

Sprachen mit automatischer Speicherverwaltung gehen ursprünglich auf List zurück. C hat diese automatische Speicherverwaltung nicht, es gibt aber C Umgebungen, die eine solche automatische Verwaltung anbieten. Hier muß ein C Programmierer sich darauf beschränken keine Zeigerarithmetik zu verwenden, sprich Speicheradressen auf die zugegriffen wird, erst wieder zu berechnen. Alle noch benötigten Speicheradressen müssen explizit im Speicher liegen.

4.1.5 Konstantendeklarationen

Ähnlich wie in Java Variablen und Parameter als mit dem Schlüsselwort `final` als konstant deklariert werden können, lassen sich auch in C++ Werte als unveränderbar deklarieren.

Hierzu dient in C das Schlüsselwort `const` ;

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

```cpp
#include <iostream>

class Const {
public:
  int i;

  void printValue() const {
    std::cout << i << std::endl;
  }
};

int main(){
  Const test;
  test.i=42;
  test.printValue();
}
```

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

```cpp
#include <iostream>

class Const {
public:
  int i;
  void setValue(int j) const {
    i=j;
  }

  void printValue() const {
    setValue(42);
    std::cout << i << std::endl;
  }
};
```

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

```cpp
#include <iostream>

class Const {
public:
  int i;
  void setValue(int j) {
    i=j;
  }

  void printValue() const {
    setValue(42);
    std::cout << i << std::endl;
  }
};
```

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>
```

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

```cpp
#include <iostream>
class Const {
  public:
    int i;
    void Const::printValue() const;
};

void doNotModify(const Const* c){
  std::cout << "in doNotModify" << std::endl;
}

void Const::printValue() const {
  doNotModify(this);
  std::cout << i << std::endl;
}

int main(){
  Const c;
  c.i=42;
  c.printValue();
}
```

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

ConstParamError.cpp

```cpp
#include <iostream>
class Const {
public:
  int i;
  void Const::printValue() const;
};

void doNotModify(Const* c){
  std::cout << "in doNotModify" << std::endl;
}

void Const::printValue() const {
  doNotModify(this);
  std::cout << i << std::endl;
}
```

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

```
#include <iostream>
class Const {
public:
  int i;
  void Const::printValue() const;
};

void doModify(Const* c){
  std::cout << "in doModify" << std::endl;
}

int main(){
  const Const c();
  doModify(&c);
}
```

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>
```

Wie man sieht, geht das Konzept zur Deklaration von konstanten Werten um einiges über das hinaus, was in Java über das Attribut `final` ausgedrückt werden kann. Stellen wir einmal gegenüber, inwieweit in Java das Attribut `final` und inwieweit in C++ das Attribut `const` Objekte vor Modifikationen schützt.

Zunächst testen wir dieses anhang einer einfachen Containerklasse in Java, die wir der Kürze halber mit Javas generischen Typen, die es seit Java 1.5 gibt, ausdrücken:

Box.java

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

  public static void main(String [] _){
    final Box<Box<String>> bbs
        = new Box<Box<String>>(new Box<String>("hallo"));

    //allowed
    bbs.contents.contents = "welt";

    //allowed
    bbs.contents = new Box<String>("world");

    //not allowed
```



```
    //bbs = new Box<Box<String>>(new Box<String>("hello"));
  }
}
```

Der Javaübersetzer verhindert nur, daß einer als `final` attributierten Variablen ein neues Objekt zugewisen wird. Das über die Variable gespeicherte Objekt kann beliebig verändert werden

Jetzt versuchen wir dasselbe in C++. Da wir noch nicht generische Klassen in C++ kennengelernt haben, benötigen wir zwei Klassen `Box`. Eine um einen String zu kapseln, und eine um eine Box für einen String zu kapseln:

BoxBox.cpp

```
#include <string>

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

class BoxBoxString{
  public:
    BoxString* contents;
    BoxBoxString(BoxString* c) {contents = c;}
};
```

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

BoxBox.cpp

```
int main(){
  const BoxBoxString* pbbs
   = new BoxBoxString(new BoxString("hallo"));
  const BoxBoxString 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

```
  //allowed
  pbbs->contents->contents = "welt";

  //not allowed
  //pbbs->contents = new BoxString("world");

  //allowed
  pbbs = new BoxBoxString(new BoxString("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

```
  //allowed
  bbs.contents->contents = "welt";

  //not allowed
  //bbs.contents = new BoxString("world");

  //not allowed
  //bbs = *pbbs;
}
```

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.

4.1.6 Ableiten

Ebenso wie Java kennt C++ das Ableiten von Klassen. Es können Unterklassen zu einer Klasse definiert werden, die diese Unterklasse um weitere Eigenschaften erweitern oder bestehende Eigenschaften überschreiben. Statt des aus Java bekannten Schlüsselworts `extends` wird die Oberklasse zu einer Klasse mit einem Doppelpunkt vom Klassennamen abgetrennt. Wenn alle Eigenschaften der Oberklasse ohne weitere Einschränkungen benutzt werden sollen, so ist die Oberklasse als `public` geerbt mit anzugeben. Eine Objekt der Unterklasse ist damit auch ein Objekt der Oberklasse:

FirstSubClass.cpp

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

class Upper{};

class Lower: public Upper{};

int main(){
  Upper* up = new Lower();
}
```

Für Konstruktoren gilt wieder dasselbe wie in Java. Bevor im Konstruktor der Unterklasse weitere für das Objekt der Unterklasse notwendige Initialisierungen bewerkstelligt werden können, ist zunächst einmal über einen Konstruktor der Oberklasse das Objekt vollständig zu initialisieren.

Hierzu gibt es nicht wie in Java das Schlüsselwort `super` sondern der Superkonstruktoraufruf wird vor dem eigentlichen Rumpf des Konstruktors mit dem Namen der Superklasse notiert.

Student.cpp

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

class Student: public Person {

  private:
   int matrNr;

  public:
    Student(std::string nachname ,std::string vorname
          ,std::string strasse  ,int hausnummer
          ,int plz              ,std::string ort
          ,int mNr)
     :Person(nachname,vorname,strasse,hausnummer,plz,ort){
      this->matrNr = mNr;
    }
};

int main(){
  Person* pP
   = new Student("Andrak","Manuel","schmidtstr"
                ,449,54654,"Koeln",723454);

  std::cout << pP->getFullName() << std::endl;
}
```

Anders als in Java braucht die Ableitungshierarchie in C++ nicht streng hierarchisch zu sein; eine Klasse kann mehrere Oberklassen haben. Die verschiedene Oberklassen einer Klasse werden durch Komma getrennt angegeben. Es kann dabei zu einen neuen Problem kommen: wie ist zu verfahren, wenn eine Klasse von zwei Oberklassen ein und dieselbe Methode erben. Welche der beiden Methoden ist im Zweifelsfall auszuführen? Wie wir es aus Java kennen, können wir Methoden in einer Unterklasse überschreiben (eng. *override*). Allerdings kennt C++ zwei Arten überschriebener Methoden:

- virtuelle Methoden, für die das Prinzip der späten Bindung auf Referenzen realisiert ist.
- Methoden ohne späte Bindung.

Machen wir zunächst einen einfachen Test der Überschreibung einer Methode. Wir definieren uns eine Ober- und eine davon abgelittene Unterklasse. In der Unterklasse wird die Methode `getDescription` überschrieben.

UpperLowerTest1.cpp

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

```

```
  class Upper{
    public:
      std::string getDescription(){return "Obere Klasse";}
  };

  class Lower: public Upper{
    public:
      std::string getDescription(){return "Untere Klasse";}
  };
```

In einem Test legen wir je ein Objekt dieser Klasse an und betrachten dieses einmal über eine Zeigervariable und einmal direkt. Die Variablen sind jeweils vom Typ der Oberklasse.

UpperLowerTest1.cpp

```
int main(){
  Upper* upP = new Upper();
  Upper* loP = new Lower();

  Upper up = *upP;
  Upper lo = *loP;

  std::cout << "upP->getDecription()"
            << upP->getDescription() << std::endl;
  std::cout << "loP->getDecription()"
            << loP->getDescription() << std::endl;

  std::cout << "up.getDecription()"
            << up.getDescription() << std::endl;
  std::cout << "lo.getDecription()"
            << lo.getDescription() << std::endl;
}
```

Wir betrachten das Ergebnis der Methode `getDescription` für alle die vier definierten Variablen. Als Ergebnis wird immer die Methode der Oberklasse ausgeführt.

```
sep@linux:~/fh/prog3/examples/src> ./UpperLowerTest
upP->getDecription()Obere Klasse
loP->getDecription()Obere Klasse
up.getDecription()Obere Klasse
lo.getDecription()Obere Klasse
sep@linux:~/fh/prog3/examples/src>
```

Wir haben also keine späte Bindung beobachten können. Obwohl in den Variablen `loP` und `lo` auf ein Objekt zugegriffen wird, daß einmal von der Klasse `Lower` erzeugt wurde, wird doch jeweils die Methode der Klasse `Upper` ausgeführt.

Jetzt machen wir einmal denselben Test, ändern aber nur ein Wort: wir stellen in der Klasse `Upper` der Methode `getDescription` das Attribut `virtual` voran:

UpperLowerTest2.cpp

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



```

  class Upper{
    public:
      virtual std::string getDescription(){return "Obere Klasse";}
  };

  class Lower : public Upper{
    public:
     std::string getDescription(){return "Untere Klasse";}
  };
```

Nun lassen wir ein und denselben Test noch einmal laufen:

UpperLowerTest2.cpp

```
#include <string>

int main(){
  Upper* upP = new Upper();
  Upper* loP = new Lower();

  Upper up = *upP;
  Upper lo = *loP;

  std::cout << "upP->getDecription()"
            << upP->getDescription() << std::endl;
  std::cout << "loP->getDecription()"
            << loP->getDescription() << std::endl;

  std::cout << "up.getDecription()"
            << up.getDescription() << std::endl;
  std::cout << "lo.getDecription()"
            << lo.getDescription() << std::endl;
}
```

Jetzt stellen wir fest, daß der Zugriff auf das für die Klasse `Lower` erzeugte Objekt per Zeigervariable, zur späten Bindung der Methode `getDescription` führt:

```
sep@linux:~/fh/prog3/examples/src> ./UpperLowerTest2
upP->getDecription()Obere Klasse
loP->getDecription()Untere Klasse
up.getDecription()Obere Klasse
lo.getDecription()Obere Klasse
sep@linux:~/fh/prog3/examples/src>
```

Zu beachten ist, daß dieses nur über die Zeigervariable funktioniert. Der Aufruf der Methode `getDescription` über die Variable `lo` führt zur Ausführung der entsprechenden Methode aus der Klasse `Upper`.

Virtueller Destruktor Destruktoren sollen dafür sorgen, daß ein Objekt regelgerecht aus dem Speicher wieder entfernt wird. Hier ist späte Bindung wünschenswert. Es soll nicht davon abhängen, ob wir über eine Zeigervariable einer Oberklasse auf ein Objekt zugreifen, bei einem `delete` ist das Objekt der Unterklasse vollständig zu entfernen. Daher sind Destruktoren in der Regel virtuelle Methoden. Mindestens dann, wenn es mindestens eine virtuelle Methode in einer Klasse gibt, ist der Destruktor auch als virtuell zu deklarieren.

4.1.7 protected

Auch C++ kennt noch ein drittes Sichtbarkeitsattribut mit dem gleichen Namen wie in Java, nämlich `protected`. Mit `protected` wird Unterklassen einer Klasse erlaubt auf eine Eigenschaft zuzugreifen.

4.1.8 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

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

class Complex {
  public:
    float im;
    float re;

    Complex(float re,float im);

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

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

Die Operatoren lassen sich wie reguläre Funktionen implementieren:

Complex.cpp

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

```



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

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

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

float Complex::operator()()const{
  return re*re+im*im;
}
std::string Complex::toString()const{
  std::string result="";
  std::stringstream ss;
  ss << re;
  ss >> result;
  result=result+"+";
  ss << im;
  ss >> result;
  result=result+"i";
  return result;
}
```

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

TestComplex.cpp

```
#include "Complex.cpp"

int main(){
  Complex c1(31,40);
  Complex c2(0.5,2);
  std::cout << (c1+c2).toString() << std::endl;
  std::cout << (c1*c2).toString() << std::endl;
  std::cout << c2() << std::endl;
}
```

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

```
#include <string>

class StringLi {
```

```
  private:
    std::string hd;
    const StringLi* tl;
    const bool empty;

  public:
    StringLi(std::string hd,const StringLi* tl);
    StringLi();
    virtual ~StringLi();

    bool isEmpty()const;
    std::string head()const;
    const StringLi* tail()const;

    StringLi* clone()const;

    const StringLi* operator+(const StringLi* other) const;

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

Im folgenden die naheliegende Implementierung dieser Listen.

StringLi.cpp

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

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

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

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

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

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

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



```
}

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

Listen lassen sich jetzt durch die Addition konkatenieren:

StringLiTest.cpp

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

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


  const StringLi* ys
   = new StringLi("lend",
     new StringLi("me",
     new StringLi("your",
     new StringLi("ears",
     new StringLi()))));

  const StringLi* zs = *xs+ys;
  delete xs;
  delete ys;

  std::cout << zs->toString() << std::endl;
}
```

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 4.1.

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

Abbildung 4.1: C++ Operatoren, die überladen werden können.

Aufgabe 12 **(3 Punkte)** 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.

4.1.9 Generische Formularklassen

Wir haben im Zusammenhang mit dem Attribut `const` bereits ein Beispiel dafür gesehen, daß wir zweimal fast dieselbe Klasse geschrieben haben. Lediglich in einem bestimmten Typnamen haben sich die beiden Klassen unterschieden. Einmal hatten wir eine Klasse zum kapseln eines Stringwertes und einmal eine analoge Klasse zum Speichern eines Objektes der ersten Klasse erhalten. 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 <class 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>{
  public:
    sndType snd;
    Pair(fstType c,sndType snd):Box<fstType>(c),snd(snd) {}

    std::string toString(){
        return "("+contents+","+snd+")";
    }
};
```

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

TestPair.cpp
```
#include "Pair.h"
#include <iostream>
#include <string>

int main(){
  Pair<std::string,int> p("hallo",42);
  std::cout << p.contents << std::endl;
  std::cout << p.snd << std::endl;
}
```

Wir können z.B. eine Paarklasse definieren, die nur noch eine Typvariable hat.

UniPair.h
```
#include "Pair.h"

template <class fstType>
class UniPair: public Pair<fstType,fstType>{
  public:
    UniPair(fstType c,fstType snd):Pair<fstType,fstType>(c,snd) {}
    void swap(){fstType tmp=contents;contents=snd;snd=tmp;}
};
```

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

TestUniPair.cpp
```
#include "UniPair.h"
#include <iostream>
#include <string>

int main(){
  UniPair<std::string> p("welt","hallo");
  p.swap();
  std::cout << p.contents << std::endl;
  std::cout << p.snd << std::endl;
}
```

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
```
#include <string>
#include <sstream>

template <typename X>
bool equals(X x,X y){return x==y;}

std::string callToString(std::string x ){return x;}
std::string callToString(int x ){

  std::stringstream ss;
  std::string result;

  ss << x;
  ss >> result;

  return result;
}

template <typename X>
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
```
template <class A>
class Li {
  private:
    A hd;
    const Li<A>* tl;
    const bool empty;

  public:
    Li(A hd,const Li<A>* tl):empty(false){
      this->hd=hd;
      this->tl=tl;
    }

    Li():empty(true){}


    bool isEmpty()const{return empty;}
    A head()const{return hd;}
    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
```
    virtual ~Li(){
      if (!isEmpty()) delete tl;
    }
```

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

Li.h
```
    Li<A>* clone()const{
      if (isEmpty()) return new Li<A>();
      return new Li<A>(head(),tail()->clone());
    }

    const Li<A>* operator+(const Li<A>* other) const{
      if (isEmpty()) return other->clone();
      return new Li<A>(head(),(*tail())+other);
    }

    int size() const {
      if (isEmpty()) return 0;
      return 1+tail()->size();
    }
```

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
```
    bool contains(A el, bool (*eq)(A,A))const{
      if (isEmpty())return false;
      if (eq(head(),el)) return true;
      return tail()->contains(el,eq);
    }

    bool contains(A el)const{
      return contains(el,equals);
    }
```

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

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

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

```
int callSize(std::string s){return s.size();}

std::string getContents(UniPair<std::string>* p){
 return p->contents;
}

int main(){
  Li<std::string>* xs
    = new Li<std::string>("Schimmelpfennig",
      new Li<std::string>("Moliere",
      new Li<std::string>("Shakespeare",
      new Li<std::string>("Bernhard",
      new Li<std::string>("Brecht",
      new Li<std::string>("Salhi",
      new Li<std::string>("Achternbusch",
      new Li<std::string>("Horvath",
      new Li<std::string>("Sophokles",
      new Li<std::string>("Calderon",
      new Li<std::string>()))))))))));

  std::cout << xs->toString() << std::endl;
  std::cout << xs->contains("Brecht") << std::endl;
  std::cout << xs->contains("Zabel") << std::endl;

  Li<int>* is = xs->map(callSize);
  delete xs;

  std::cout << is->toString() << std::endl;
  delete is;

  Li<UniPair<std::string>* >* ys =
  new Li<UniPair<std::string>* >(new UniPair<std::string>("A","1")
 ,new Li<UniPair<std::string>* >(new UniPair<std::string>("B","2")
 ,new Li<UniPair<std::string>* >(new UniPair<std::string>("C","3")
 ,new Li<UniPair<std::string>* >())));

  std::cout << ys->toString() << std::endl;

  xs=ys->map(getContents);
  std::cout << xs->toString() << std::endl;
}
```

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>
```

Aufgabe 13 (Alternative zur letzten Punkteaufgabe 3 Punkte) In dieser Aufgabe ist eine Klassen für HashMengen zu schreiben:

a) Schreiben Sie eine Klasse, die eine Hashmenge realisiert. Ihre Klasse soll der folgenden Kopfdatei entsprechend Funktionalität bereitstellen

```
HashSet.h
#include "Li.h"

template <class elementType>
class HashSet {
  protected:
    int capacity;
    int (*hashFunction)(elementType);
    Li<elementType>* hashArray [];

  public:
    HashSet(int c, int (*hashF)(elementType));
    virtual ~HashSet();
    void add(elementType el);

    bool contains(elementType el);
    std::string toString();
};
```

b) Schreiben Sie Testfälle und betrachten Sie, wie gut in diesen Tests die Elemente gestreut liegen.

Aufgabe 14 Schreiben Sie jetzt analog zu Ihrer Lösung aus der letzten Aufgabe, eine generische Klasse `HashMap`, die eine endliche Abbildung von Schlüsseln auf Werte realisiert. Benutzen Sie hierzu auch die bereits geschrieben Klasse `Pair`, um die Schlüssel-Wert-Paare in Listen zu speichern.

4.1.10 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
```
template <class A>
class Closure {

  private:
    A value;

  public:
    Closure <A> (A a) {
      this->value=a;
    };

    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
```
    virtual  A operator()() {
      return value;
    };
};
```

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

TestSimpleClosure.cpp
```
#include "Closure.h"
#include <iostream>

int main(){
   Closure<int> f1 = Closure<int>(42);
   std::cout << f1() << std::endl;
}
```

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

```



```
template <class aT,class rT>
class ApplyFun : public Closure<rT> {
```

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

ApplyFun.h
```
  private:
    rT (*f)(aT);
    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
```
    rT result;
    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
```
  public:
    ApplyFun <aT,rT> (rT(*f)(aT),aT a) {
      this->f=f;
      this->arg=a;
      resultIsEvaluated=false;
    };

    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
```
    virtual  rT operator()() {
      if (!resultIsEvaluated){
        resultIsEvaluated = true;
        result=f(arg);
      }
      return result;
    };
};
```

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

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

TestClosure.cpp
```
  std::cout << (*(xs->tail()->tail()->head()))() << std::endl;
  delete xs;
```

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

TestClosure.cpp
```
  Closure<int>*  c5
    = new ApplyFun<std::string,int>(getSize,"witzelbritz");
  std::cout << (*c5)() << std::endl;
}
```

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[**?**] 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

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

template <class A>
class LazyList {
  private:
    A hd;
```

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

LazyList.h

```
    Closure<LazyList<A>*>* tl;
    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

```
  public:
    LazyList(A hd, Closure<LazyList<A>*>* tl):empty(false){
      this->hd=hd;
      this->tl=tl;
    }

    LazyList(A hd,LazyList<A>* t):empty(false){
      this->hd=hd;
      this->tl= new Closure<LazyList<A> >(t);
    }
```

Der Konstruktor für leer Listen fehlt natürlich nicht:

LazyList.h

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

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

LazyList.h

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

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

LazyList.h

```
    LazyList<A>* tail(){
      return (*tl)();
    }
};
```


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.

Unsere unendlichen Listen realisieren ein Konzept, das uns schon lange bekannt ist: *Iteratoren*. Die Methode `isEmpty` entspricht der negierten Methode `hasNext`. Über die Methode `tail` erhalten wir in dieser Analogie einen neuen Iterator, dessen nächster Wert über die Methode `head` erfragt werden kann. Ebenso wie bei Iteratoren in der Regel nicht alle Werte, die der Iterator enthält, komplett zu einer Zeit im Speicher liegen, werden von Objekten der Klasse `LazyList` auch die Elemente erst nach Bedarf im Speicher angelegt. Und natürlich können Iteratoren auch unendlich viele Werte liefern. Hierzu haben wir im ersten Semester bereits ein Beispiel gesehen, als wir die obiger Liste analoge Klasse `From` programmiert haben.

Aufgabe 15 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
```
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);
    };
};
```

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[1].

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

template <class a,class b,class c>
class Curry : public Lambda<a,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.cpp
```
  public:
    b i;
    a (*f)(b x,c y);

    Curry(a(*f)(b x, c y),b i):i(i),f(f){}
```

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

Curry.cpp
```
    virtual a operator()(c y){ return f(i,y);}
};
```

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:

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

Curry.cpp
```
int add(int x,int y){return x+y;}

int main(){
  Lambda<int,int>* add5 = new Curry<int,int,int>(add,5);
  int i = (*add5)(37);
  std::cout << i << std::endl;
}
```

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:

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

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

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

4.2 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 eine Klasse für Strings, ähnlich wie wir sie aus Java kennen, 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.

4.2.1 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.

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
```
#include <algorithm>
#include <iostream>

void doppel(int& x){x=2*x;}

int main(){
  int  ar[3] = {17,28,21};
  std::for_each(ar,ar+3,doppel);

  for (int i= 0;i<3;i=i+1){
    std::cout << ar[i] << std::endl;
  }
}
```

Anwendung auf Vektoren

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

ForEachVector.cpp
```
#include <vector>
#include <algorithm>
#include <iostream>

void doppel(int& x){x=2*x;}

int main(){
  std::vector<int> v = std::vector<int>(3);
  v[0] = 17; v[1]=28; v[2] = 21;
  std::for_each(v.begin(),v.begin()+3,doppel);

  for (int i= 0;i<3;i=i+1){
      std::cout << v[i] << std::endl;
```

```
  }
}
```

Wie man sieht, ist die Klasse `vector` generisch über ihren Elementtyp[2]. 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.

4.2.2 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 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`:

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each
 (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.

In Java würde man Konzepte durch Schnittstellen beschreiben.

4.2.3 Iteratoren

Eines der wichtigsten Konzepte der STL sind Iteratoren. Iteratoren in der STL sind kaum zu vergleichen mit den über eine Schnittstelle definierten Iteratoren aus Java. In der STL

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

stehen Operatoren vielmehr 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.

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`.

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

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

template <typename InputIterator,typename Element>
Element* maximum(InputIterator begin,InputIterator end){
  Element* result = begin;
  for (InputIterator it = begin+1;it<end;it++){
```

```
    if ((*result) < (*it)){ result=it;}
  }
  return result;
}

int main(){
  int  ar[3] = {17,28,21};
  std::cout << * (maximum<int*,int>(ar,ar+3)) << std::endl;

  std::string  sr[3] = {"shakespeare","calderon","moliere"};
  std::cout << *(maximum<std::string*,std::string>(sr,sr+3))
            << std::endl;
}
```

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.

4.2.4 Behälterklassen

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

Sequenzen

Sequenzen sind Behältertypen, die die klassischen Listenfunktionalität zur Verfügung stellen:

- sie können dynamische beliebig lang wachsen
- die gespeicherten Elemente haben eine Reihenfolge und können über einen Index angesprochen werden.

Als Implementierung für Sequenzen stehen in der STL die folgenden fünf Klassen zur Verfügung:

- `vector`: ermöglicht Elementzugriff über den Index in konstanter Zeit, Einfügen und Löschen am Ende der Sequenz in konstanter Zeit, Löschen und Einfügen an beliebigen Indexstellen in linearer Zeit. Diese Klasse entspricht am ehesten der Klasse `java.util.ArrayList` aus Java.
- `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

```
#include <vector>
#include <list>
#include <string>
#include <iostream>

template <typename Iterator>
void doPrint(Iterator begin,Iterator end){
  std::cout << "[";
  for (Iterator it = begin;it!=end;){
    std::cout << *it;
    it++;
    if (it != end) std::cout << ",";
  }
  std::cout << "]"<<std::endl;
}

int main (){
  std::string words [] ={ "the","world","is","my","oyster"};
  std::vector<std::string> sv;
  std::list<std::string> sl;
  std::cout << sv.capacity() << std::endl;

  for (int i=0;i<5;i++){
    sv.insert(sv.end(),words[i]);
    sl.insert(sl.end(),words[i]);
  }
  std::cout << sv.capacity() << std::endl;

  doPrint(sv.begin(),sv.end());
  sl.reverse();
  doPrint(sl.begin(),sl.end());
}
```

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

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

class Address {

  public:
    std::string strasse;
    int hausnummer;
    Address(){}
    Address(std::string strasse,int hasunummer)
        :strasse(strasse),hausnummer(hausnummer){}
};


int main(){
  std::map<std::string, Address,std::less<std::string> >
    addressBook;
  addressBook.insert(
   std::pair<std::string,Address>
    ("Hoffmann",Address("Gendarmenmarkt",42)));
  addressBook.insert(
   std::pair<std::string,Address>
    ("Schmidt",Address("Bertholt-Brecht-Platz",1)));

  std::cout << addressBook["Hoffmann"].strasse << std::endl;

  addressBook["Tierse"] = Address("Pariser Platz",1);

  std::cout << addressBook["Tierse"].strasse << std::endl;
```



```
  std::cout << addressBook["Andrak"].strasse << std::endl;
}
```

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

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;
}
```

4.2.5 Algorithmen

Wir haben bereits gesehen, daß die STL nicht durchgängig objektorientiert entworfen wurde. Nur Basisfunktionalität ist in den einzelnen Behäterklassen 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.

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
```
#include <functional>
#include <iostream>

int main(){
  std::cout <<bind1st(std::plus<int>(),17)(25) <<std::endl;
}
```

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
```
#include <algorithm>
#include <functional>
```



```
#include <iostream>

using namespace std;

void print (int a){cout << a <<",";}

int main(){
  int xs [] = {1,2,3,4,5};
  int ys [] = {6,7,8,9,10};
  int zs [5];
  transform(xs,xs+5,ys,zs,plus<int>());
  for_each(zs,zs+5,print);
  cout << endl;
}
```

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
```
#include <vector>
#include <list>
#include <string>
#include <iostream>
```

Zunächst definieren wir eine Klasse, für die kleiner Relation, die sich nur auf die Länge von Strings bezieht.

SortTest.cpp
```
class ShorterString{
  public:
    bool operator()(std::string s1,std::string s2){
      return s1.length()<s2.length();
    }
};
```

Die nächste Kleinerrelation vergleicht zwei Strings rückwärts gelesen in der lexikographischen Ordnung:

SortTest.cpp
```
bool reverseOrder(std::string s1,std::string s2){
  reverse(s1.begin(),s1.end());
  reverse(s2.begin(),s2.end());
  return s1<s2;
}
```

Mit folgenden Hilfsmethoden werden wir die Behälterobjekte auf dem Bildschirm ausgeben:

SortTest.cpp
```
void print(std::string p){
  std::cout << p << ",";
}

template <typename Iterator>
void printSequence(Iterator begin,Iterator end){
  if (begin==end) {
    std::cout << "[]" << std::endl;return;
  }
  std::cout << "[";
  for_each(begin,end-1,print);
  std::cout << *(end-1) << "]"<<std::endl;
}
```

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
```
int main (){
  int l = 8;
  std::string words []
   ={"rasiert","schweinelende","passiert","legende"
    ,"schwestern","verlustiert","gestern","stern"};

  std::sort(words,words+l,ShorterString());
  printSequence(words,words+l);

  std::sort(words,words+l,reverseOrder);
  printSequence(words,words+l);

  std::sort(words,words+l);
  printSequence(words,words+l);
}
```

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 16 (2 Punkte) Sie haben bereits zweimal eine Funktion `fold` implementiert. Einmal auf Reihungen und einmal auf unseren selbstgeschriebenen einfach verketteten Listen. 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.

4.3 Ausnahmen

Das Konzept der Ausnahmen, die geworfen und gefangen werden können, kennt C++ in ähnlicher Weise, wie wir es in Java kennengelernt haben. Es gibt den `throw`-Befehl und die `try` und `catch` Blöcke. Hauptunterschiede sind:

- es gibt keine gesonderte Ausnahmeklassen. Alles kann geworfen werden, sogar Strukturen, Zeiger oder primitive Typen.
- Funktionen brauchen und können nicht deklarieren, welche Ausnahmen bei ihrem Aufruf geworfen werden. Es gibt keine `throws`-Klausel.
- Es gibt kein `finally`.

Beispiel:

ExceptionTest.cpp
```
#include <string>
#include <iostream>

int fac(int x){
  if (x<0) {
    std::string msg = "negative Zahl in Fakultaet";
    throw msg;
  }
  if (x==0) return 1;
  return x*fac(x-1);
}


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

4.4 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
```
#include <iostream>

using namespace std;

namespace paket1 {
  void drucke() { cout << "paket1" << endl;
  }
}

namespace paket2 {
  void drucke() { cout << "paket2" << endl;
  }
  namespace paket3{
    void drucke() { cout << "paket3" << endl;
    }
  }
}
```

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

NamespaceTest.cpp
```
int main(){
  {
    using namespace paket2::paket3;
    paket2::drucke();
    drucke();
    paket1::drucke();
  }
  {
    using namespace paket2;
    drucke();
    paket3::drucke();
    paket1::drucke();
  }
  {
    using namespace paket1;
    paket2::drucke();
    paket2::paket3::drucke();
```

```
    drucke();
  }
}
```

Kapitel 5

Graphik und GUI Programmierung

5.1 3 dimensionale Graphiken mit OpenGL

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].

5.1.1 Eine Zustandsmaschine

5.1.2 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
```
#include <GL/glut.h>

```

```
void display(void){
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glFlush();
}

int main (int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutCreateWindow("Hello Window");
  glutDisplayFunc(display);
  glutMainLoop();
}
```

Glut bietet über das Öffnen von Fenster eine ganze Reihe weiterer Bibliotheksfunktionen an. Insbesondere auch die Möglichkeit auf Tastatureingaben zu reagieren.

5.1.3 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

```
#include <GL/glut.h>
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

```
#include "SomePoints.h"
int shape=GL_POINTS;

void points(){
   glVertex3f(0.2,-0.4,0);
   glVertex3f(0.46,-0.26,0);
   glVertex3f(0.6,0,0);
   glVertex3f(0.6,0.2,0);
   glVertex3f(0.46,0.46,0);
```

[1] `moin` nehme ich als Norddeutscher gerne für Funktionen, die fast die Methode `main` sind, lediglich noch mehr Parameter haben.


```
   glVertex3f(0.2,0.6,0);
   glVertex3f(0.0,0.6,0);
   glVertex3f(-0.26,0.46,0);
   glVertex3f(-0.4,0.2,0);
   glVertex3f(-0.4,0,0);
   glVertex3f(-0.26,-0.26,0);
   glVertex3f(0,-0.4,0);
}
```

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

```
void pointsAsShape(){
  glClear(GL_COLOR_BUFFER_BIT );
  glColor3f (0.0, 0.0, 0.0);
  glBegin(shape);
    points();
  glEnd();
  glFlush();
}
```

Und schließlich noch die Pseudohauptmethode, die noch zusätzlich die zu erzeugende Figur als Parameter enthält

SomePoints.cpp

```
void moin(int argc, char **argv,int s){
  shape=s;
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutCreateWindow("Hello");
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glutDisplayFunc(pointsAsShape);
  glutMainLoop();
}
```

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);}
```


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
```
#include <GL/glut.h>

float (*f) (float);
float square(float x) {return 3*x*x*x;};

void display(void){
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);

  const int steps =100;
  const float step = 2/(float)100;
  glBegin(GL_LINE_STRIP);
  for (int i = 0; i < steps; i=i+1){
    const float x = -1+i*step;
    const float y = f(x);
    glVertex2f(x,y);
  }
  glEnd();
  glFlush();
}

int main (int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutCreateWindow("Function Graph Window");
  f = square;
  glutDisplayFunc(display);
  glutMainLoop();
}
```

Aufgabe 17 **(1 Punkt)** Schreiben Sie eine OpenGL-Funktion
void circle(int radius),
mit der Sie einen Kreis zeichnen können. Benutzen Sie hierzu die Figur GL_POLYGON.

5.1.4 Nah und Fern

5.1.5 Vordefinierte Figuren

Im letzen 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.

- `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
```
#include <GL/glut.h>

void display(void){
  glClear(GL_COLOR_BUFFER_BIT );
  glColor3f (1.0, 0.0, 0.0);
  glutSolidSphere(0.8,10,10);
  glFlush();
}

int main (int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutCreateWindow("Hello Cube");
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glutDisplayFunc(display);
  glutMainLoop();
}
```

Das Ergebnis einer solchen dreidimensionalen Figur ist auf dem Bildschirm zunächst recht enttäuschend. Man sieht nur einen Kreis.

[2] Entspricht in meiner installierten OpenGL Bibliothek nicht der Dokumentation.?

5.1.6 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
```
#include <GL/glut.h>
int moin (int argc, char **argv,void(*display)(),char* title);
```

RenderShape.cpp
```
#include "RenderShape.h"

void (*shapeFun)();

void display(){
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  shapeFun();
  glFlush();
}

int moin (int argc, char **argv,void(*shape)(),char* title){
  shapeFun=shape;
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH);

  glutCreateWindow(title);

  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, -2.5, 0.0 };

  glShadeModel (GL_SMOOTH);

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);


  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);

  glClearColor(1.0, 0,0,0);
```

```
  glutDisplayFunc(display);
  glutMainLoop();
}
```

Und jetzt lassen sich auf einfache Weise Tests für alle Figuren erzeugen:

WireSphere.cpp
```
#include "RenderShape.h"
void f(){glutWireSphere(1,10,10); }
int main (int argc,char **argv){moin(argc,argv,f,"WireSphere");}
```

SolidSphere.cpp
```
#include "RenderShape.h"
void f(){glutSolidSphere(1,10,10); }
int main (int argc,char **argv){moin(argc,argv,f,"SolidSphere");}
```

WireCube.cpp
```
#include "RenderShape.h"
void f(){glutWireCube(1); }
int main (int argc,char **argv){moin(argc,argv,f,"WireCube");}
```

SolidCube.cpp
```
#include "RenderShape.h"
void f(){glutSolidCube(1); }
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 5.3 bzw. 5.4.

5.1.7 Transformationen

5.1.8 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);
  glutCreateWindow("The Cube World");
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutReshapeFunc(resize);
  glutMainLoop();
}

void red()  {glColor3f(1.0,0.0,0.0);}
void blue() {glColor3f(0.0,1.0,0.0);}
void green(){glColor3f(0.0,0.0,1.0);}
void grey() {glColor3f(0.8, 0.8, 0.8);}

void drawcube(int x_offset, int z_offset, void (*color)()){
  color();
  glPushMatrix();
    glTranslatef(x_offset,0.0,z_offset);
    glutSolidCube(10);
  glPopMatrix();
}

void display(void){
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  /* draw the floor */
  glPushMatrix();
    glTranslatef(0,-5,20);
    glScalef(30,0.1,30);
    drawcube(0, 0, grey);
  glPopMatrix();
```



```

  /* draw 12 cubes with different colors */
  drawcube(75, 57, red);
  drawcube(-65, -12, green);
  drawcube(50, -50, red);
  drawcube(-56, 17, blue);
  drawcube(67, 12, green);
  drawcube(-87, 32, red);
  drawcube(-26, 75, blue);
  drawcube(57, 82, green);
  drawcube(-3, 12, red);
  drawcube(46, 35, blue);
  drawcube(37, -2, green);

  glutSwapBuffers();
}

void keyboard(unsigned char key, int _, int __){
  switch (key){
    case 'a': case 'A': glTranslatef(5.0, 0.0, 0.0); break;
    case 'd': case 'D': glTranslatef(-5.0, 0.0, 0.0);break;
    case 'w': case 'W': glTranslatef(0.0, 0.0, 5.0); break;
    case 's': case 'S': glTranslatef(0.0, 0.0, -5.0);break;
  }
  display();
}

void resize(int width, int height){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, width / height, 1.0, 400.0);
  glTranslatef(0.0, -5.0, -150.0);
  glMatrixMode(GL_MODELVIEW);
}
```

Die Klötzchenwelt ist in Abbildung 5.5 zu bewundern.

5.1.9 Beispiel: Tux der Pinguin

In einem abschließenden Beispiel benutzen wir alle oben vorgestellten OpenGL Konstrukte, um das Linuxmaskottchen *Tux* zu zeichnen.

Tux.cpp
```
#include <GL/glut.h>
#include <math.h>

float zPos = -1;

void display();
void keyboard(unsigned char, int, int);
```

```
void resize(int, int);

int main (int argc, char **argv){
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutCreateWindow("Tux");
  glClearColor(1.0, 0.0, 0.0, 0.0);

  glEnable(GL_LIGHTING);
  glEnable(GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);

  GLfloat lights[] = { 1, 1, 1, 1 };
  GLfloat light_position[] = { 1, 0.4, 0.8, 1 };
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);

  glLightfv(GL_LIGHT0, GL_AMBIENT, lights);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, lights);
  glLightfv(GL_LIGHT0, GL_SPECULAR, lights);
  glEnable(GL_LIGHT0);

  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutReshapeFunc(resize);
  glutMainLoop();
}

void keyboard(unsigned char key, int _, int __){
  switch (key){
    case ’+’: zPos=zPos+0.1;break;
    case ’-’: zPos=zPos-0.1;break;
  }
  display();
}

void resize(int w, int h){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  float near   = 0.001;
  float    far    = 40;
  float    fov    = 90;
  float    ang    = (fov*3.14)/(360);
  float    top    = near / ( cos(ang) / sin(ang) );
  float    aspect = w/h;
  float    right = top*aspect;
  glFrustum(-right, right, -top, top, near, far);
  glMatrixMode(GL_MODELVIEW);
}

void sphere(float r, float xs, float ys,float  zs){
```



```
  glScalef(xs, ys, zs);
  glutSolidSphere(r,100,100);
}

void rotateZ(float a){glRotatef(a,0,0,1);}
void rotateY(float a){glRotatef(a,0,1,0);}
void rotateX(float a){glRotatef(a,1,0,0);}

void crMat(float rd,float gd,float bd
          ,float rs,float gs,float bs,float exp){
  float diffuse [] ={rd, gd, bd,  1.0};
  glMaterialfv(GL_FRONT,GL_DIFFUSE, diffuse);
  glMaterialfv(GL_FRONT,GL_AMBIENT, diffuse);
  float specular [] ={rs, gs, bs,  1.0};
  glMaterialfv(GL_FRONT,GL_SPECULAR, specular);
  float shine [] ={exp};
  glMaterialfv(GL_FRONT,GL_SHININESS,shine);
}

void whitePenguin() {
  crMat(0.58, 0.58, 0.58, 0.2, 0.2, 0.2, 50.0);
}
void blackPenguin() {crMat(0.1, 0.1, 0.1,0.5, 0.5, 0.5, 20.0);}
void beakColour()   {crMat(0.64, 0.54, 0.06,0.4, 0.4, 0.4, 5);}
void nostrilColour(){
  crMat(0.48039, 0.318627, 0.033725,0.0,0.0,0.0, 1);
}
void irisColour()   {crMat(0.01, 0.01, 0.01,0.4, 0.4, 0.4, 90.0);}

void makeBody(){
  glPushMatrix();
    blackPenguin();
    sphere(1, 0.95, 1.0, 0.8);
  glPopMatrix();
  glPushMatrix();
    whitePenguin();
    glTranslatef( 0, 0, 0.17);
    sphere(1, 0.8, 0.9, 0.7);
  glPopMatrix();
}

void createTorso(){
  glPushMatrix();
    glScalef( 0.9, 0.9, 0.9);
    makeBody();
  glPopMatrix();
}


void leftHand(){
```

```
  glPushMatrix();
    glTranslatef( -0.24, 0, 0);
    rotateZ(20);
    rotateX(90);
    blackPenguin();
    sphere( 0.5, 0.12, 0.05, 0.12);
  glPopMatrix();
}


void leftForeArm(){
  glPushMatrix();
    glTranslatef(-0.23, 0, 0);
    rotateZ( 20);
    rotateX( 90);
    leftHand();
    blackPenguin();
    sphere( 0.66, 0.3, 0.07, 0.15);
  glPopMatrix();
}

void rightForeArm(){leftForeArm();}

void leftArm(){
  glPushMatrix();
    rotateY(180);
    glTranslatef( -0.56, 0.3, 0);
    rotateZ( 45);
    rotateX( 90);
    leftForeArm();
    blackPenguin();
    sphere( 0.66, 0.34, 0.1, 0.2);
  glPopMatrix();
}

void rightArm(){
  glPushMatrix();
    glTranslatef(-0.56, 0.3, 0);
    rotateZ( 45);
    rotateX(-90);
    rightForeArm();
    blackPenguin();
    sphere( 0.66, 0.34, 0.1, 0.2);
  glPopMatrix();
}

void createShoulders(){
  glPushMatrix();
    glTranslatef( 0, 0.4, 0.05);
    leftArm();
```



```
    rightArm();
    glScalef( 0.72, 0.72, 0.72);
    makeBody();
  glPopMatrix();
}


void createBeak(){
  glPushMatrix();
    glTranslatef( 0,-0.205, 0.3);
    rotateX(10);
    beakColour();
    sphere( 0.8, 0.23, 0.12, 0.4);
  glPopMatrix();
  glPushMatrix();
    beakColour();
    glTranslatef( 0, -0.23, 0.3);
    rotateX( 10);
    sphere( 0.66, 0.21, 0.17, 0.38);
  glPopMatrix();
}


void leftIris(){
  glPushMatrix();
    glTranslatef( 0.12,-0.045, 0.4);
    rotateY( 18);
    rotateZ( 5);
    rotateX( 5);
    irisColour();
    sphere( 0.66, 0.055, 0.07, 0.03);
  glPopMatrix();
}


void rightIris(){
  glPushMatrix();
    glTranslatef( -0.12,-0.045, 0.4);
    rotateY( -18);
    rotateZ( -5);
    rotateX( 5);
    irisColour();
    sphere( 0.66, 0.055, 0.07, 0.03);
  glPopMatrix();
}

void leftEye(){
  glPushMatrix();
    glTranslatef( 0.13,-0.03, 0.38);
    rotateY( 18);
```

```
    rotateZ( 5);
    rotateX( 5);
    whitePenguin();
    sphere( 0.66, 0.1, 0.13, 0.03);
  glPopMatrix();
}

void rightEye(){
  glPushMatrix();
    glTranslatef( -0.13,-0.03, 0.38);
    rotateY( -18);
    rotateZ( -5);
    rotateX( 5);
    whitePenguin();
    sphere( 0.66, 0.1, 0.13, 0.03);
  glPopMatrix();
}

void createEyes(){
  leftEye();
  leftIris();
  rightEye();
  rightIris();
}

void createHead(){
  glPushMatrix();
    glTranslatef( 0, 0.3, 0.07);
    createBeak();
    createEyes();
    rotateY( 90);
    blackPenguin();
    sphere( 1, 0.42, 0.5, 0.42);
  glPopMatrix();
}

void createNeck(){
  glPushMatrix();
    glTranslatef( 0, 0.9, 0.07);
    createHead();
    rotateY(90);
    blackPenguin();
    sphere( 0.8, 0.45, 0.5, 0.45);
    glTranslatef( 0, -0.08, 0.35);
    whitePenguin();
    sphere( 0.66, 0.8, 0.9, 0.7);
  glPopMatrix();
}

void footBase(){
```



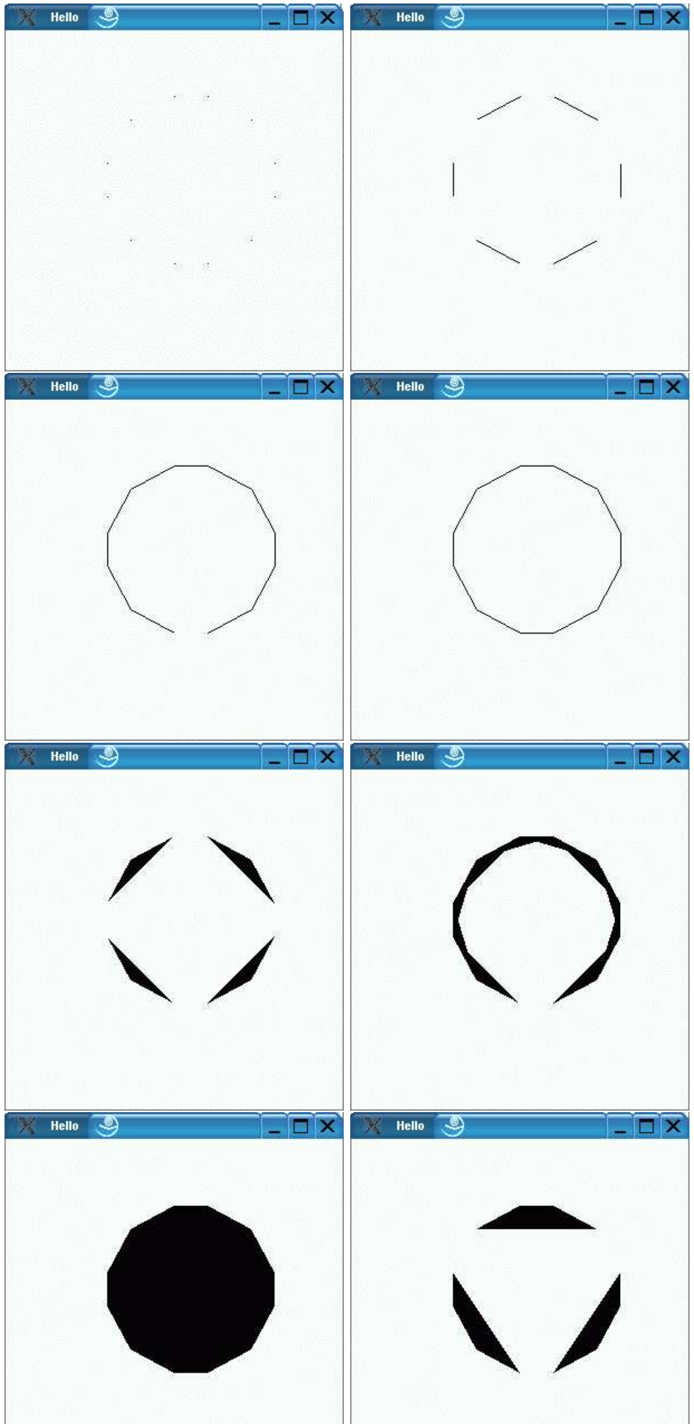
```
  glPushMatrix();
    sphere( 0.66, 0.25, 0.08, 0.18);
  glPopMatrix();
}


void toe1(){
  glPushMatrix();
    glTranslatef(-0.07, 0, 0.1);
    rotateY( 30);
    sphere(0.66, 0.27, 0.07, 0.11);
  glPopMatrix();
}

void toe2(){
  glPushMatrix();
    glTranslatef(-0.07, 0, -0.1);
    rotateY (-30);
    sphere( 0.66,  0.27, 0.07, 0.11);
  glPopMatrix();
}

void toe3(){
  glPushMatrix();
    glTranslatef(-0.08, 0, 0);
    sphere( 0.66,  0.27, 0.07, 0.10);
  glPopMatrix();
}

void foot(){
  glPushMatrix();
    glScalef(  1.1, 1.0, 1.3);
    beakColour();
    footBase();
    toe1();
    toe2();
    toe3();
  glPopMatrix();
}

void rightFoot(){
  glPushMatrix();
    glTranslatef( 0,-0.09, 0);
    rotateY( 180);
    foot();
  glPopMatrix();
}

void leftFoot(){
  glPushMatrix();
```

```
    glTranslatef( 0,-0.09, 0);
    rotateY( 100);
    foot();
  glPopMatrix();
}


void leftCalf(){
  glPushMatrix();
    glTranslatef( 0,-0.21, 0);
    rotateY( 90);
    leftFoot();
    beakColour();
    sphere( 0.5, 0.06, 0.18, 0.06);
  glPopMatrix();
}

void rightCalf(){
  glPushMatrix();
    glTranslatef( 0,-0.21, 0);
    rightFoot();
    beakColour();
    sphere( 0.5, 0.06, 0.18, 0.06);
  glPopMatrix();
}

void hipBall(){
  glPushMatrix();
    blackPenguin();
    sphere( 0.5, 0.09, 0.18, 0.09);
  glPopMatrix();
}

void leftTigh(){
  glPushMatrix();
    rotateY( 180);
    glTranslatef(-0.28,-0.8, 0);
    rotateY( 110);
    hipBall();
    leftCalf();

    rotateY (-110);
    glTranslatef( 0,-0.1, 0);
    beakColour();
    sphere( 0.5, 0.07, 0.3, 0.07);
  glPopMatrix();
}

void rightTigh(){
  glPushMatrix();
```



```
    glTranslatef(-0.28,-0.8, 0);
    rotateY( -110);
    hipBall();
    rightCalf();

    glTranslatef( 0,-0.1, 0);
    beakColour();
    sphere( 0.5, 0.07, 0.3, 0.07);
  glPopMatrix();
}

void createTail(){
  glPushMatrix();
    glTranslatef( 0,-0.4,-0.5);
    rotateX (-60);
    glTranslatef( 0, 0.15, 0);
    blackPenguin();
    sphere( 0.5, 0.2, 0.3, 0.1);
  glPopMatrix();
}

void tux(){
  glPushMatrix();
    glScalef( 0.35, 0.35, 0.35);
    rotateY (-180);
    createTorso();
    createShoulders();
    createNeck();
    leftTigh();
    rightTigh();
    createTail();
  glPopMatrix();
}

void display(){
  glLoadIdentity();
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  gluLookAt(0,0,zPos,0,0,0,0,1,0);
  tux();
  glutSwapBuffers();
}
```

Der so gezeichnete kleine Kerl ist in Abbildung 5.6 zu sehen.



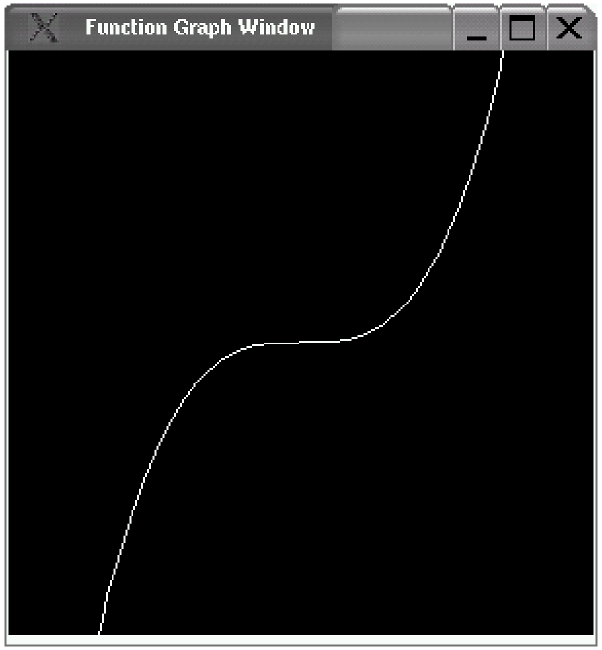
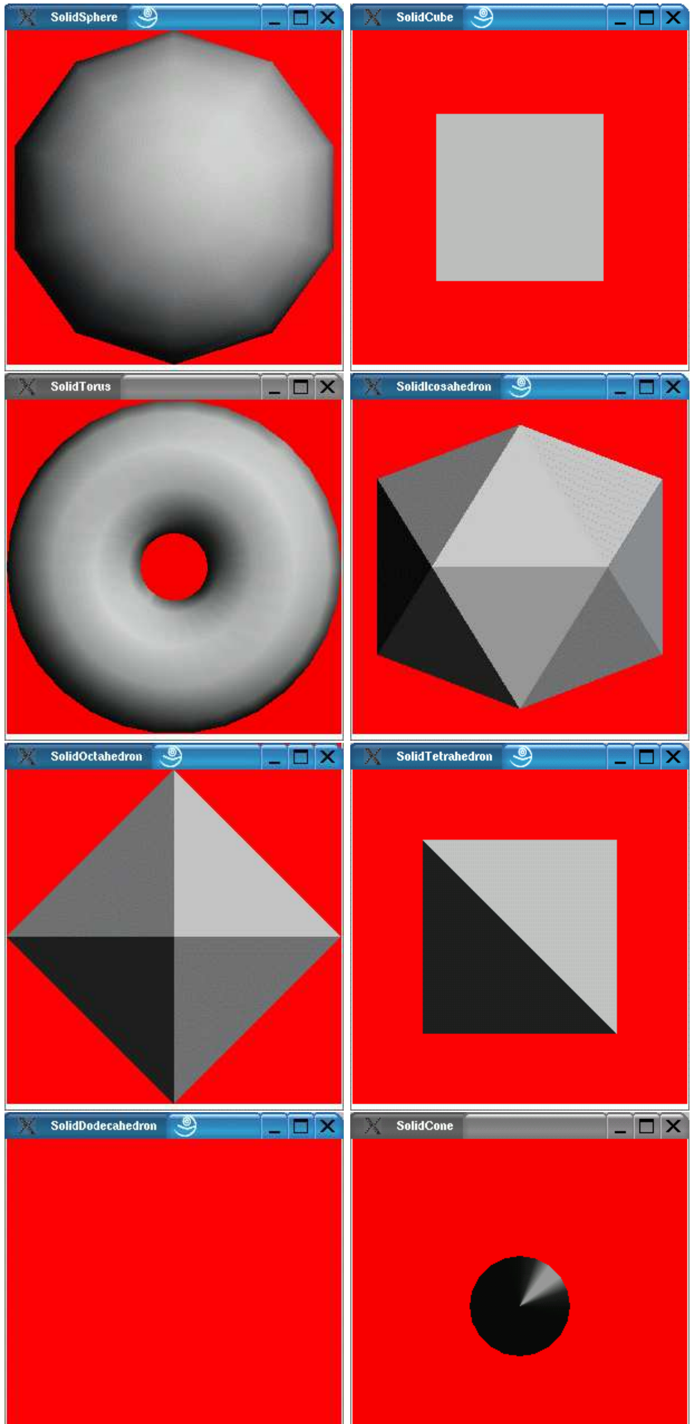
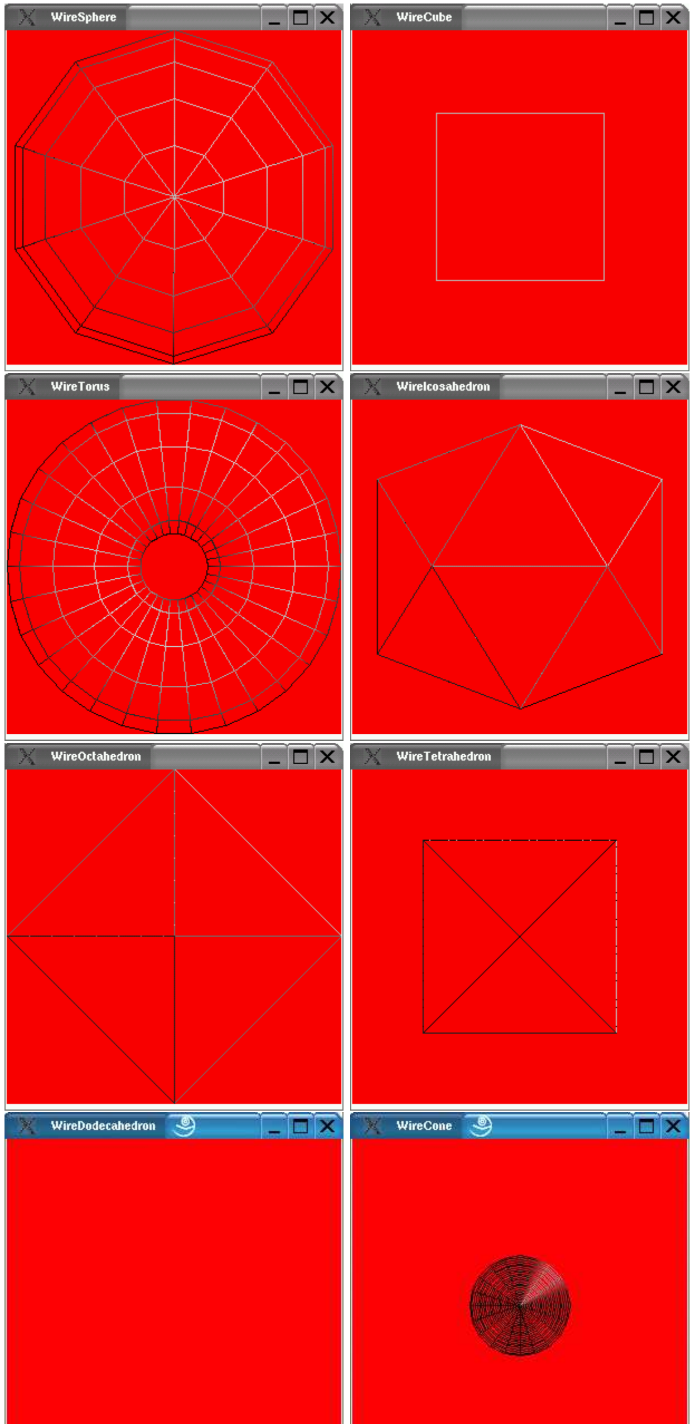


Abbildung 5.2: Der Graph einer Funktion.


SolidSphere
SolidCube
SolidTorus
SolidIcosahedron
SolidOctahedron
SolidTetrahedron
SolidDodecahedron
SolidCone


WireSphere
WireCube
WireTorus
WireIcosahedron
WireOctahedron
WireTetrahedron
WireDodecahedron
WireCone

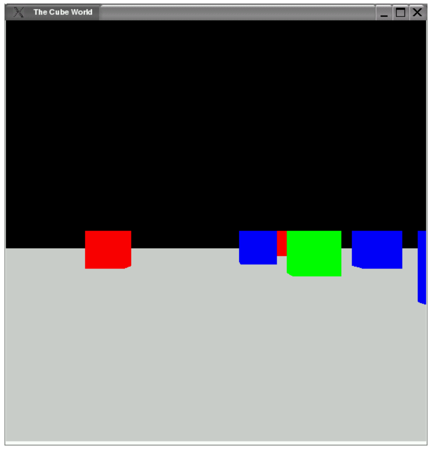


Abbildung 5.5: Dreidimensionale Klötzchenwelt.

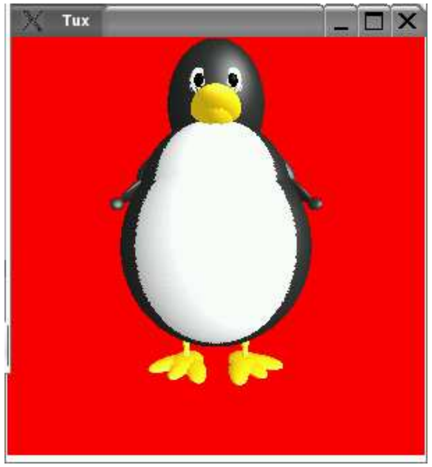


Abbildung 5.6: Tux der Pinguin.

Anhang A

Gtk für C++ Beispielprogramme

A.1 Ein einfacher Zähler

Die wahrscheinlichst einfachste interaktive GUI-Anwendung stellt ein Zähler da. Es wird in einem Fenster in einemn Label eine Zahl dargestellt, die sich über einen Knopf erhöhen läßt. Wir sehen eine Klasse für diese GUI-Anwendung vor.

A.1.1 Die Kopfdatei

In der Kopfdatei werden die Eigenscvhaften der Klasse festgelegt. Sie erweitert die Fensterklasse aus GTK, enthält den aktuellen Zählerstand in einem `int` Feld und drei graphisches Unterkomponenten. In der Methode `on_button_clicked()` wird das Verhalten auf die Aktion eines Knopfdrucks spezifiziert.

Counter.h

```
#include <gtkmm/button.h>
#include <gtkmm/window.h>
#include <gtkmm/label.h>
#include <gtkmm/box.h>

class Counter : public Gtk::Window{

public:
  Counter();
  virtual ~Counter();

protected:
  virtual void on_button_clicked();

  Gtk::Button button;
  Gtk::Label  label;
  Gtk::HBox   box;
```

```
  int counter;
};
```

A.1.2 Implementierung

In der Implementierungsdatei werden die graphischen Elemente initialisiert. Über die Methode `signal_clicked()` kann der Knopfkomponente die entsprechende Funktion für die Knopfdruckaktion zugewiesen werden.

Counter.cpp

```
#include "Counter.h"
#include <iostream>
#include <sstream>
#include <string>

Counter::Counter()
:  button("DrÃ¼ck Mich")
  ,label("0"){
  counter=0;
  set_border_width(10);

  button
    .signal_clicked()
    .connect(SigC::slot
              (*this, &Counter::on_button_clicked));

  box.add(button);
  box.add(label);
  add(box);

  button.show();
  label.show();
  box.show();
}

Counter::~Counter(){}

void Counter::on_button_clicked(){
  std::stringstream ss;
  std::string zahl;
  counter=counter+1;
  ss << counter;
  ss >> zahl;
  label.set_text(zahl);
}
```


A.1.3 Beispielaufruf

Zur Benutzung dieser GUI-Komponente ist lediglich die GTK zu starten und das Objekt der Methode `run` von GTK übergeben werden.

testcounter.cpp

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

int main (int argc, char *args[]){
  Gtk::Main kit(argc,args);

  Counter counter;
  Gtk::Main::run(counter);
}
```

Abbildung A.1: Counter in Aktion.

A.2 Einfaches Zeichnen

Strichkreis.cpp

```
#include <gtkmm/drawingarea.h>
#include <gdkmm/colormap.h>
#include <gdkmm/window.h>
#include <gtkmm/box.h>
#include <gtkmm/window.h>
#include <gtkmm/main.h>

#include <cmath>

const double DEG2RAD = 3.1415928 / 180.0;

class StrichKreis : public Gtk::DrawingArea
```

```
{
public:
  StrichKreis(int r);
  ~StrichKreis();

protected:
  virtual void on_realize();
  virtual bool on_expose_event(GdkEventExpose* event);

  Glib::RefPtr<Gdk::GC> gc_;
  Gdk::Color black_, white_;
};

StrichKreis::StrichKreis(int r){
  Glib::RefPtr<Gdk::Colormap> colormap = get_default_colormap();
  black_ = Gdk::Color("black");
  white_ = Gdk::Color("white");

  colormap->alloc_color(black_);
  colormap->alloc_color(white_);
  set_size_request(r, r);
}

StrichKreis::~StrichKreis(){}

void StrichKreis::on_realize(){
  // We need to call the base on_realize()
  Gtk::DrawingArea::on_realize();

  // Now we can allocate any additional resources we need
  Glib::RefPtr<Gdk::Window> window = get_window();
  gc_ = Gdk::GC::create(window);
  window->set_background(black_);
}

bool StrichKreis::on_expose_event(GdkEventExpose*){
  Glib::RefPtr<Gdk::Window> window = get_window();

  // window geometry: x, y, width, height, depth
  int winx, winy, winw, winh, wind;
  window->get_geometry(winx, winy, winw, winh, wind);

  window->clear();
  int radius = winw/2;
  if (winh<winw) radius=winh/2;

  gc_->set_foreground(white_);

  for (int i = 0; i<=360; i=i+1){
      int x1,y1,x2,y2;
```



```
      x1=winw/2;
      y1=winh/2;
      x2 = x1 + (int) (radius * cos(DEG2RAD * i));
      y2 = y1 + (int) (radius * sin(DEG2RAD * i));
      window->draw_line(gc_, x1,y1,x2,y2 );
  }

  return true;
}


int main(int argc, char** argv){
  Gtk::Main main_instance (argc, argv);
  Gtk::Window* window = new Gtk::Window();
  Gtk::HBox* m_box0 = new Gtk::HBox(false,0);

  StrichKreis* kreis1 = new StrichKreis(400);
  StrichKreis* kreis2 = new StrichKreis(180);
  StrichKreis* kreis3 = new StrichKreis(50);
  StrichKreis* kreis4 = new StrichKreis(100);
  StrichKreis* kreis5 = new StrichKreis(20);

  m_box0->pack_start(*kreis1, Gtk::PACK_EXPAND_WIDGET, 0);
  m_box0->pack_start(*kreis2, Gtk::PACK_EXPAND_WIDGET, 0);
  m_box0->pack_start(*kreis3, Gtk::PACK_EXPAND_WIDGET, 0);
  m_box0->pack_start(*kreis4, Gtk::PACK_EXPAND_WIDGET, 0);
  m_box0->pack_start(*kreis5, Gtk::PACK_EXPAND_WIDGET, 0);

  window->add(*m_box0);
  window->show_all();

  Gtk::Main::run(*window);
  return 0;
}
```

A.3 Das Telespiel: Ping

Movable.h

```
#include <gtkmm/window.h>
#ifndef __MOVABLE_H

#define __MOVABLE_H ;

class Movable {
  int _xPos;
  int _yPos;
  int _xDir;
  int _yDir;
```

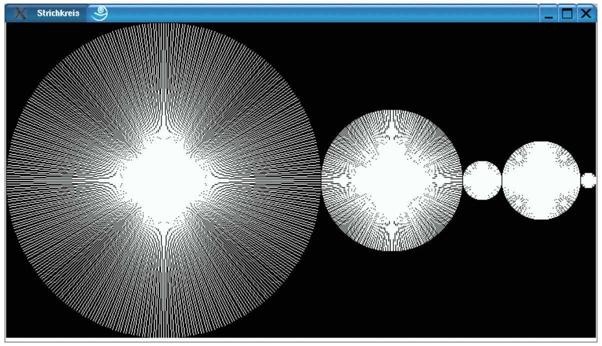


Abbildung A.2: Strichkreis in Aktion.

```

  int _width;
  int _height;

public:
  Movable();
  Movable(const int xPosVal,const int yPosVal);
  virtual ~Movable();

  void setXPos(const int val);
  void setYPos(const int val);
  void setXDir(const int val);
  void setYDir(const int val);
  void setWidth(const int val);
  void setHeight(const int val);

  int getXPos(){return _xPos;}
  int getYPos(){return _yPos;}
  int getXDir(){return _xDir;}
  int getYDir(){return _yDir;}
  int getWidth(){return _width;}
  int getHeight(){return _height;}

  bool touches(Movable &other);

  virtual void move(const int maxX,const int maxY);
  virtual void paint(Glib::RefPtr<Gdk::Window> window
                    ,Glib::RefPtr<Gdk::GC> gc_);

private:
  bool touchesHorizontal(Movable &other);
  bool touchesVertical(Movable &other);
};
```



```

#endif /* __MOVABLE_H */
```

Movable.cpp

```


#include "Movable.h"


Movable::Movable(){
  _xPos=0;
  _yPos=0;
  _xDir=1;
  _yDir=1;

  _width=10;
  _height=10;
}

Movable::Movable(const int xPosVal,const int yPosVal){
  _xPos=xPosVal;
  _yPos=yPosVal;
  _xDir=10;
  _yDir=10;

  _width=10;
  _height=10;
}

Movable::~Movable(){}

void Movable::setXPos(const int val){_xPos=val;}
void Movable::setYPos(const int val){_yPos=val;}
void Movable::setXDir(const int val){_xDir=val;}
void Movable::setYDir(const int val){_yDir=val;}
void Movable::setWidth(const int val){_width=val;}
void Movable::setHeight(const int val){_height=val;}

void Movable::move(const int maxX,const int maxY){
  _xPos = _xPos + _xDir;
  _yPos = _yPos + _yDir;
}

void Movable::paint
    (Glib::RefPtr<Gdk::Window> window,Glib::RefPtr<Gdk::GC> gc_){
}

bool Movable::touches(Movable &other){
    return (   touchesHorizontal(other)
            && touchesVertical(other));
```

```
  }

bool Movable::touchesHorizontal(Movable &other){
    const int otherRight = other._xPos+other._width;
    const int thisRight  = _xPos+_width;

    return (_xPos<=other._xPos  && other._xPos<=thisRight)
        || (_xPos<=otherRight   && otherRight<=thisRight)
        || (_xPos>=other._xPos  && otherRight>=_xPos);

}

bool Movable::touchesVertical(Movable &other){
    const int otherUpper = other._yPos+other._height;
    const int thisUpper  = _yPos+_height;

    return   (_yPos<=other._yPos    && other._yPos<=thisUpper)
          || (_yPos<=otherUpper     && otherUpper<=thisUpper)
          || (_yPos>=other._yPos    && otherUpper>=thisUpper);
}
```

Ball.h

```
#include "Movable.h"

class Ball  : public Movable{
public:
  Ball();
  ~Ball();

  virtual void move(const int maxX,const int maxY);
  virtual void paint(Glib::RefPtr<Gdk::Window> window
                    ,Glib::RefPtr<Gdk::GC> gc_);
};
```

Ball.cpp

```
#include <gdkmm/window.h>
#include <gtkmm/window.h>
#include "Ball.h"

Ball::Ball(){}
Ball::~Ball(){}

void Ball::move(const int maxX,const int maxY){
  Movable::move(maxX,maxY);

  if (getXPos()>=maxX+80) setXPos(30);
  if (getXPos()<=-80) setXPos(maxX-40);

  if (getYPos()>=maxY-getHeight() || getYPos()<=0)
    setYDir(-getYDir());
```



```
}

void Ball::paint(Glib::RefPtr<Gdk::Window> window
                ,Glib::RefPtr<Gdk::GC> gc_){
  window->draw_arc(gc_, 1,getXPos(),getYPos()
                  ,getWidth(),getHeight(),0,360*64);
}
```

Paddle.h

```
#include "Movable.h"

class Paddle  : public Movable{
public:
  Paddle(const int xPosVal,const int boardHeight);
  ~Paddle();

  virtual void move(const int maxX,const int maxY);
  virtual void paint(Glib::RefPtr<Gdk::Window> window
                    ,Glib::RefPtr<Gdk::GC> gc_);

  static const int yMoveVal =11;
};
```

Paddle.cpp

```
#include <gdkmm/window.h>
#include <gtkmm/window.h>
#include "Paddle.h"

Paddle::Paddle(const int xPosVal,const int boardHeight)
  :Movable(xPosVal,boardHeight-50){
  setWidth(10);
  setHeight(50);
  setXDir(0);
  setYDir(0);
}

Paddle::~Paddle(){}

void Paddle::move(const int maxX,const int maxY)
{ Movable::move(maxX,maxY);
  if (getYPos()>=maxY-getHeight()) setYPos(maxY-getHeight());

  if (getYPos()<0) setYPos(0);
}

void Paddle::paint(Glib::RefPtr<Gdk::Window> window
                  ,Glib::RefPtr<Gdk::GC> gc_){
  window->draw_rectangle(gc_,1,getXPos()
                        ,getYPos(),getWidth(),getHeight());
}
```

GameCanvas.h

```
#include <gtkmm/drawingarea.h>
#include <gdkmm/window.h>
#include <gtkmm/window.h>

class GameCanvas : public Gtk::DrawingArea{
public:
  GameCanvas();
  virtual ˜GameCanvas();
  bool timer_callback();

  virtual void moveObjects(const int maxX,const int maxY);
  virtual void paintObjects(Glib::RefPtr<Gdk::Window> window
                          ,Glib::RefPtr<Gdk::GC> gc_);
  virtual void doChecks();


  //Overridden default signal handlers:
  virtual void on_realize();
  virtual bool on_expose_event(GdkEventExpose* event);

  Glib::RefPtr<Gdk::GC> gc_;
  Gdk::Color black_, white_;
};
```

GameCanvas.cpp

```
#include <gtkmm/drawingarea.h>
#include <gdkmm/colormap.h>
#include <gdkmm/window.h>
#include <gtkmm/window.h>

#include "GameCanvas.h"


GameCanvas::GameCanvas(){
  Glib::RefPtr<Gdk::Colormap> colormap = get_default_colormap();
  black_ = Gdk::Color("black");
  white_ = Gdk::Color("white");

  colormap->alloc_color(black_);
  colormap->alloc_color(white_);
  set_size_request(400, 200);

  Glib::signal_timeout()
   .connect(SigC::slot(*this,&GameCanvas::timer_callback), 3);

  add_events(Gdk::EXPOSURE_MASK);
}

GameCanvas::˜GameCanvas(){}
```



```

void GameCanvas::on_realize(){
  // We need to call the base on_realize()
  Gtk::DrawingArea::on_realize();

  // Now we can allocate any additional resources we need
  Glib::RefPtr<Gdk::Window> window = get_window();
  gc_ = Gdk::GC::create(window);
  window->set_background(black_);
}

bool GameCanvas::on_expose_event(GdkEventExpose*){
  // we need a ref to the gdkmm window
  Glib::RefPtr<Gdk::Window> window = get_window();

  window->clear();

  gc_->set_foreground(white_);

  paintObjects(window,gc_);
  return true;
}

bool GameCanvas::timer_callback(){
  int winx, winy, winw, winh, wind;
  Glib::RefPtr<Gdk::Window> window = get_window();
  window->get_geometry(winx, winy, winw, winh, wind);

  moveObjects(winw,winh);
  doChecks();

  queue_draw();

  // never disconnect timer handler
  return 1;
}

void GameCanvas::moveObjects(const int maxX,const int maxY){}

void GameCanvas::paintObjects(Glib::RefPtr<Gdk::Window> window
                             ,Glib::RefPtr<Gdk::GC> gc_){}

void GameCanvas::doChecks(){}
```

Ping.cpp

```
#include <gtkmm/drawingarea.h>
#include <gdkmm/colormap.h>
#include <gdkmm/window.h>
#include <gtkmm/box.h>
#include <gtkmm/main.h>
```

```
#include <iostream>
#include "Ball.h"
#include "Paddle.h"
#include "GameCanvas.h"

using namespace std;

class Ping : public GameCanvas{
public:
  Ping();
  ~Ping();
  bool timer_callback();

  int pointsLeftPlayer;
  int pointsRightPlayer;
  bool newBall;

  Ball ball;
  Paddle leftPaddel;
  Paddle rightPaddel;

  virtual void moveObjects(const int maxX,const int maxY);
  virtual void paintObjects(Glib::RefPtr<Gdk::Window> window
                           ,Glib::RefPtr<Gdk::GC> gc_);

  virtual void doChecks();

  virtual bool on_key_press_event(GdkEventKey* event);
  virtual bool on_key_release_event(GdkEventKey* event);
};

Ping::Ping():GameCanvas(),leftPaddel(10,200),rightPaddel(380,200){
  set_flags(Gtk::CAN_FOCUS);
  pointsLeftPlayer = 0;
  pointsRightPlayer = 0;
}


Ping::~Ping(){}

void Ping::paintObjects(Glib::RefPtr<Gdk::Window> window
                       ,Glib::RefPtr<Gdk::GC> gc_){
  ball.paint(window,gc_);
  leftPaddel.paint(window,gc_);
  rightPaddel.paint(window,gc_);
}

void Ping::moveObjects(const int maxX,const int maxY){
  this->ball.move(maxX,maxY);
  this->leftPaddel.move(maxX,maxY);
```



```
  this->rightPaddel.move(maxX,maxY);
}

void Ping::doChecks(){
  if (ball.touches(leftPaddel) || ball.touches(rightPaddel))
    ball.setXDir(-ball.getXDir());

  if (ball.getXPos()<0&& newBall){
    pointsRightPlayer++;newBall=false;
  }else if (ball.getXPos()>400&&newBall){
    pointsLeftPlayer++;newBall=false;
  }else if (!newBall && ball.getXPos()>40
                    && ball.getXPos()<80) {
    newBall=true;
  }
}

bool Ping::on_key_press_event(GdkEventKey* event){
  if(event->keyval == 'l')
    rightPaddel.setYDir(- Paddle::yMoveVal);
  else if(event->keyval == 'a')
    leftPaddel.setYDir(- Paddle::yMoveVal);
}

bool Ping::on_key_release_event(GdkEventKey* event){
  if(event->keyval == 'l')
    rightPaddel.setYDir(Paddle::yMoveVal);
  else if(event->keyval == 'a')
    leftPaddel.setYDir(Paddle::yMoveVal);
  else cout << event->keyval << endl;
}

int main(int argc, char** argv){
  Gtk::Main main_instance (argc, argv);
  Gtk::Window* window = new Gtk::Window();
  Gtk::HBox m_box0(false,0);

  Ping* p = new Ping();
  m_box0.pack_start(*p, Gtk::PACK_EXPAND_WIDGET, 0);
  window->add(m_box0);
  window->show_all();

  Gtk::Main::run(*window);
  delete p;
  delete window;
  return 0;
}
```

Anhang B

Gesammelte Aufgaben

Aufgabe 1 Übersetzen Sie die Programme aus diesem Abschnitt und lassen Sie sie laufen.

Aufgabe 2 Schreiben Sie die rekursive Funktion der Fakultät. Hierzu brauchen Sie den `if`-Befehl, wie Sie ihn aus Java kennen. Testen Sie die Funktion mit ein paar Werten.

Aufgabe 3 Schreiben Sie jetzt eine Version der Fakultätsfunktion, in der ein Zeiger auf die Variable, in der das Ergebnis akkumuliert wird, als zweites Argument übergeben wird:

FakAcc.h
```
void fak(int x,int* result);
```

Schreiben Sie Testfälle für diese Funktion.

Aufgabe 4 **(2 Punkte)** Schreiben Sie eine Funktion, die mit einer Intervallschachtellung für eine stetige Funktion eine Nullstelle approximiert.

Nullstellen.h
```
float nullstelle(float (*f) (float),float mi,float ma);
```

Testen Sie Ihre Implementierung mit folgendem Programm:

TesteNullstellen.cpp
```
#include "Nullstellen.h"
#include <iostream>

float f1(float x){return 2*x*x*x+5;}
float f2(float x){return 2*x +1 + x*x*x/4 ;}
float f3(float x){return 10*x+5;}
float f4(float x){return 0;}
float f5(float x){return x*x;}

void teste(float (*f)(float)){
```



```
  const float fn = nullstelle(f,-10,11);
  std::cout << "n = "<<fn<< ", f(n) = " << f(fn) << std::endl;
}

int main(){
  teste(f1);
  teste(f2);
  teste(f3);
  teste(f4);
  teste(f5);
}
```

Aufgabe 5 Implementieren sie die Funktion `compose` entsprechend folgender Header-Datei:

Compose.h
```
typedef int (* intfunction)(int);

int compose(intfunction fs [],int length,int x);
```

Sie soll eine Komposition aller Funktionen in der Reihung `fs` auf den Wert `x` anwenden.

Es soll dabei gelten:

$$compose(\{f_1, \ldots f_n\}, n, x) = f_n(f_{n-1}(\ldots(f_1(x))\ldots))$$

Testen Sie Ihre Lösung mit folgender Testdatei:

ComposeTest.cpp
```
#include <iostream>
#include "Compose.h"

int f1(int x){return x*x;}
int f2(int x){return x;}
int f3(int x){return x+2;}
int f4(int x){return x*4;}
int f5(int x){return -x*x*x;}

int main(){
    intfunction fs [] = {f1,f2,f3,f4,f5};
    std::cout << compose(fs,5,2)<<std::endl;
}
```

Aufgabe 6 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester vorgestellt wurde, für Reihungen von ganzen Zahlen 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) )`

Aufgabe 7 **(2 Punkte)** Schreiben Sie eine Funktion `fold` mit folgender Signatur:
`int fold(int xs [],int length,int (*foldOp)(int,int),int startV)`

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

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

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

```
#include <iostream>

int add(int x,int y){return x+y;}
int mult(int x,int y){return x*y;}

int fold(int xs [],int length,int (*foldOp)(int,int),int startV);

int main(){
  int xs [] = {1,2,3,4,5};
  std::cout << fold(xs,5,add,0)<< std::endl;
  std::cout << fold(xs,5,mult,1)<< std::endl;
}
```

Aufgabe 8 Implementieren Sie für unsere in C++ implementierten Listenstruktur die folgenden Funktionen (nach der Spezifikation aus dem ersten Semester):

a) `int last(struct IntList * dieses)`

b) `struct IntList * concat(struct IntList * xs,struct IntList * ys)`

c) `int elementAt(struct IntList * dieses,int i)`

d) `int fold(struct IntList * dieses,int start,int(*foldOp)(int,int))`

Aufgabe 9 Schreiben Sie eine Funktion zur Funktionakomposition mit folgender Signatur:

```
template <typename a, typename b, typename c>
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 10 Schreiben Sie jetzt eine generische Fuktion `fold` auf Reihungen und testen Sie diese auf verschiedenen Typen aus.

Aufgabe 11 Verallgemeinern Sie jetzt Ihre Funktion `bubbleSortBy` zu einer generischen Funktion, mit der sie Reihungen beliebiger Typen sortieren können.

Aufgabe 12 **(3 Punkte)** 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 13 **(Alternative zur letzten Punkteaufgabe 3 Punkte)** In dieser Aufgabe ist eine Klassen für HashMengen zu schreiben:

a) Schreiben Sie eine Klasse, die eine Hashmenge realisiert. Ihre Klasse soll der folgenden Kopfdatei entsprechend Funktionalität bereitstellen

HashSet.h

```
#include "Li.h"

template <class elementType>
class HashSet {
  protected:
    int capacity;
    int (*hashFunction)(elementType);
    Li<elementType>* hashArray [];

  public:
    HashSet(int c, int (*hashF)(elementType));
    virtual ~HashSet();
    void add(elementType el);

    bool contains(elementType el);
    std::string toString();
};
```

b) Schreiben Sie Testfälle und betrachten Sie, wie gut in diesen Tests die Elemente gestreut liegen.

Aufgabe 14 Schreiben Sie jetzt analog zu Ihrer Lösung aus der letzten Aufgabe, eine generische Klasse `HashMap`, die eine endliche Abbildung von Schlüsseln auf Werte realisiert. Benutzen Sie hierzu auch die bereits geschrieben Klasse `Pair`, um die Schlüssel-Wert-Paare in Listen zu speichern.

Aufgabe 15 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:
$z_{n+1} = (91 * z_n + 1) \bmod 347$

Aufgabe 16 **(2 Punkte)** Sie haben bereits zweimal eine Funktion `fold` implementiert. Einmal auf Reihungen und einmal auf unseren selbstgeschriebenen einfach verketteten Listen. 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 17 **(1 Punkt)** 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

[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. `ftp://ftp.sgi.com/opengl/doc/opengl1.2/glu1.3.ps`, 1998.

[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). `http://www.opengl.org/developers/documentation/glut/glut-3.spec.ps`, 1996.

[NH02] Alexander Niemann and Stefan Heitsiek. *Das Einsteigerseminar: C++ Objektorientierte Programmierung*. verlag moderne industrie Buch AG & Co KG, Bonn, 2002.

[Pan02] Sven Eric Panitz. Programmieren I. `www.panitz.name/prog1/index.html`, 2002. Skript zur Vorlesung, TFH Berlin.

[Pan03a] Sven Eric Panitz. HOpenGL – 3D Graphics with Haskell, A small Tutorial. `www.panitz.name/hopengl/index.html`, 2003. Online Script, TFH Berlin.

[Pan03b] Sven Eric Panitz. Programmieren II. `www.panitz.name/prog2/index.html`, 2003. Skript zur Vorlesung, TFH Berlin.

[RL03] Heike Ripphausen-Lipa. Programmieren II (TI). `www.tfh-berlin.de/~ripphaus/Prg2/Prg2.htm`, 2003. Skript, TFH Berlin.

[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. `www.mcp.com/sams`, 1999.