Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

C++ im Unterricht

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Martin Battisti
Matrikelnummer 0325922

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Dr. Andreas Ulovec

Wien, 11.11.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Inhaltsverzeichnis

1 Spezifikation

1.1 Ziel

Ziel der Masterarbeit ist das Testen der Programmiersprache C++, inwieweit sie sich für den Unterricht eignet. Dabei werden auch verschiedene Schultypen betrachtet, in denen es Informatikunterricht gibt.

1.2 Methoden

Geachtet wird in dieser Arbeit auf drei Punkte:

- Didaktik
- Theorie
- Praxis

Dabei ist der prozentuelle Anteil der drei Themen unterschiedlich. Alle drei sind für das Testen der Eignung der Programmiersprache C++ für den Unterricht wichtig, wobei der theoretische Teil jedoch den geringsten Umfang hat. Ein großer Teil der Arbeit enthält praktische Beispiele und wird durch didaktische Beschreibungen ergänzt.

1.2.1 Didaktik

Für die Didaktik werden Themen wie:

- Wie leicht ist die Programmiersprache den Schülern erklärbar
- Wie ist sie einführbar
- Welche Konzepte sollten den Schülern erklärt werden
- Ist die Programmiersprache als Einführungsprogrammiersprache geeignet oder eher als Vertiefung der Programmierkenntnisse geeignet

näher betrachtet.

1.2.2 Theorie

In diesem Teil wird unter anderem auch die Programmiersprache selbst näher betrachtet. Wie diese entstanden ist, die Standardisierung, die Verwandtschaft mit anderen Programmiersprachen und weitere C++-spezifische Themen. Im

Gegensatz zum praktischen Teil ist der theoretische jedoch kompakter.

1.2.3 Praxis

Im praktischen Teil wird die Programmiersprache selbst anhand verschiedener Beispiele angewendet bzw. eingeführt. Dabei wird mit einem einfachen „Hello World“-Programm begonnen, während später spezifischere Programme folgen, welche den Schülern die C++-Eigenschaften näher bringen.
Zwar sind in C++ verschiedenste Programmierparadigmen vorhanden, jedoch wird das Hauptaugenmerk auf die objektorientierte Programmierung gelegt, da C++ dafür geschaffen wurde und andere Paradigmen mehrheitlich wegen der Kompatibilität existieren. Weitere wichtige Themen sind:

- GUI:
  Grafische C++-Programmierung mit Qt, gtkmm, Visual Studio
- Compiler, Entwicklungsumgebung:
  Diese sind wichtige Komponenten, welche für das Erstellen von lauffähigen Programmen verwendet werden. So wird für das Erstellen eines ausführbaren Programms neben der Kenntnis über die Programmiersprache selbst auch Wissen über den verwendeten Compiler gebraucht. Auch Entwicklungsumgebungen sind sehr hilfreich, da sie Schülern mit fortgeschrittenen Programmierkenntnissen das Programmieren sehr erleichtern können.

2 Geschichte

Die Programmiersprache C++ wurde ab 1979 von Bjarne Stroustrup bei dem Unternehmen AT&T entwickelt. Dabei verwendete er als Ausgangsprogrammiersprache C. Diese hatte durch die Verwendung im Betriebssystem Unix bereits große Beliebtheit und einen großen Bekanntheitsgrad erfahren.

Als erstes wurde C durch Klassen mit Datenkapselung erweitert, weshalb damals die Programmiersprache noch „C with Classes“ („C mit Klassen“) hieß. Danach wurden vor allem abgeleitete Klassen, ein strenges Typsystem, Inline-Funktionen und Standard-Argumente in die Sprache eingebaut. 1983 wurde „C with Classes“ in den heutigen Namen „C++“ umbenannt. Dabei wurde die Sprache um virtuelle Funktionen, das Überladen von Funktionsnamen und Operatoren, Referenzen, Konstanten, änderbare Freispeicherverwaltung und eine verbesserte Typüberprüfung erweitert. Auch der Zeilenkommentar mit der Einleitung „//“ wurde übernommen.

Die erste C++ Version die eine wichtige Referenzversion darstellte wurde 1985 veröffentlicht. Damals war diese Sprache noch nicht standardisiert. 1989 wurde schließlich die 2. Version von C++ veröffentlicht, welche um Mehrfachvererbung, abstrakte Klassen, statische und konstante Elementfunktionen und die Erweiterung des Schutzmodells um `protect` erweitert wurde.

Erst später folgten dann die Konzepte der Templates, Namensräume, neuartige Typumwandlungen und boolesche Typen. Durch die Weiterentwicklung wurde auch die Standardbibliothek gegenüber C stark erweitert. Darunter ist vor allem die Stream-I/O-Bibliothek erwähnenswert, welche C++ im Gegensatz zu C eine Standardeingabe bzw. Standardausgabe gibt. Wesentliche Teile der Standardbibliothek stammen dabei von der durch Hewlett-Packard entwickelten Standard Template Library (STL).

Es dauerte bis 1998 als schließlich die erste standardisierte C++ Version, welche durch die ISO genormt wurde, veröffentlicht wurde. Diese Version wird als ISO/IEC 14882:1998 gekennzeichnet. Bereits im Jahr 2003 folgte dann die

nächste Standardisierung durch die ISO. Diese wird als ISO/IEC 14882:2003 gekennzeichnet.
Bereits weit fortgeschritten ist die Arbeit an der nächsten Standardisierung. Behandelte Themen sind dabei unter anderem Performance-Probleme welche Rechenzeit und Speicherplatz betreffen. Darüber gibt es bereits ein *Technical Report* Namens ISO/IEC TR 18015:2006 mit Lösungsmöglichkeiten der von der ISO veröffentlicht wurde.

3 Einleitung

C++ ist eine beliebte Programmiersprache und wird sowohl in der Systemprogrammierung als auch in der Anwendungsprogrammierung eingesetzt. Typische Anwendungsfelder in der Systemprogrammierung sind Betriebssysteme, eingebettete Systeme, virtuelle Maschinen, Treiber und Signalprozessoren. Diese Bereiche brauchen vor allem performante Programmiersprachen wie C++ oder C.

In der Anwendungsprogrammierung bekommt C++ allerdings immer mehr von anderen objektorientierten Programmiersprachen wie beispielsweise Java (von Sun Microsystems) oder C# (von Microsoft) Konkurrenz. Diese sind meist einfachere Programmiersprachen (kompakter) und auch Betriebssystem unabhängig (C# durch das Mono-Projekt für Linux), weshalb vor allem Java durch die immer weitere Verbreitung des Internets schon erhebliche Marktanteile bekam. Da C++ vor allem auf Performance und Maschinennähe ausgerichtet ist, sind C++-Programme nicht Plattform-unabhängig. C++ hat wegen Kompatibilitätsgründen den gesamten Sprachumfang von C enthalten, weshalb die Einfachheit verloren geht. Meistens wird in C++ objektorientiert programmiert, jedoch kann man auch ein C Programm (Strukturierte Programmierung) oder sogar Assemblerbefehle schreiben. Weiters sind auch Modulare Programmierung und Generische Programmierung möglich.

Das von der ISO standardisierte C++ liefert durch die Konzentration auf Performance auch keine Grafische Benutzeroberfläche (GUI) mit. Es gibt jedoch einige Entwicklungsumgebungen welche jene mitliefern:

- Qt von Trolltech
- GTK+ (LGPL)
- Visual Studio von Microsoft

Mit diesen Entwicklungswerkzeugen können auf einfache Weise grafische Programme erstellt werden. Sie sind in der Schule für das Erstellen grafischer Programme in C++ sehr geeignet. Während Visual C++ für Windows und GTK+ (zwar für mehrere Plattformen erhältlich) eher für Linux geeignet ist, ist Qt auf mehreren Betriebssystemen verbreitet und durch Qtopia sogar für Handys

verfügbar. Die Firma Trolltech, welche mehrere Qt-Produkte besitzt, wurde deshalb im Jänner 2008 von Nokia übernommen.

Dadurch das C++ wie bereits beschrieben einige Vorteile, jedoch auch Nachteile besitzt ist es in einigen Bereichen Marktführend, während es in anderen Bereichen zurückgedrängt wird. Da jedoch mit einer noch länger andauernden Beliebtheit von C++ gerechnet werden kann, ist auch dessen Einsatz in der Schule prüfenswert.

In welchen Schulen und Klassen diese Programmiersprache geeignet ist und welche Voraussetzungen es braucht bzw. welche Themen für den Schulunterricht interessanter sind wird in dieser Arbeit ausführlicher beschrieben. Dabei wird mit der standardisierten C++ Version begonnen, welche keine grafische Oberfläche bietet. Erst später, wenn die wichtigsten Themen näher beschrieben und mit entsprechenden Beispielen ergänzt wurden, wird auch auf die grafische Programmierung eingegangen.

Wie bereits in der Spezifikation erwähnt, wird in dieser Masterarbeit reichlich auf Praxis Wert gelegt, wobei der didaktische Aspekt ebenso stark betrachtet wird.

Weiters werden die verschiedenen Konzepte der Sprache nicht für jede Schule einzeln untersucht, sondern es werden die wichtigsten Konzepte der Sprache allgemein behandelt. Erst danach wird näher beschrieben, welche Themen für einen bestimmten Schultyp interessant sind. Somit wird vermieden, dass ein Kapitel über C++ mehrfach beschrieben wird. Ein „Hello World“-Programm ist beispielsweise für alle Schultypen in denen C++ gelehrt wird empfehlenswert und wird somit nur einmal erwähnt.

Da diese Masterarbeit die Eignung der Programmiersprache C++ im Schulunterricht überprüft, ist diese nicht als C++ Tutorial geeignet. Es wird nämlich nicht der gesamte C++ Bereich abgedeckt, sondern nur die wichtigsten Bereiche. Die Programmiersprache C, welche von C++ 100% abgedeckt ist, wird nur für Vergleiche betrachtet. Außerdem wird die Sprache meist nicht selbst erklärt, sondern darauf geachtet, wie ein bestimmtes Konzept den Schülern am besten gelehrt werden kann.

4 Eigenschaften von C++

4.1 Abstraktion

Im Gegensatz zu anderen Programmiersprachen, wie die für den Schulunterricht geschaffene prozedurale Programmiersprache Pascal, besitzt C++ reichlich Abstraktion. Dies ist nicht nur deshalb so, weil C++ mehrere Programmierparadigmen zulässt, sondern auch weil die Konzepte von C++ bzw. jene welche von C übernommen wurden Abstraktion besitzen. Der Grund dafür ist, dass damit dem Compiler mehr Spielraum für Optimierungen gegeben wird und somit die Performance eines Programms gesteigert wird. Prozessoren können somit sehr optimal genutzt werden.

Dabei ist vor allem die Datentypdefinition erwähnenswert. Während der Integer-Datentyp in Pascal (Größe: 2 Byte, Werte: -32768 bis 32767) und Java (Größe: 4 Byte, Werte: -2147483648 bis 2147483647) eine feste Größe hat, ist das in C++ oder C nicht der Fall.

Will man den Schülern diese Tatsache erklären so wird das eigentliche Thema, das Lehren der Programmiersprache selbst, schon vernachlässigt. Wird sie jedoch nicht oder erst später erwähnt, so ist dies für die Schüler auch nicht vorteilhaft.

Jedoch sind nicht nur die Datentypengrößen selbst variabel, sondern manchmal auch die Datentypen von Funktionsergebnissen. So hat das Ergebnis der Funktion „`std::time`“ den Datentyp „`std::time_t`“. Über diesen ist nichts weiter definiert, als dass er ein arithmetischer Typ sein muss. Wie groß dieser Wertebereich ist hängt vom jeweiligen System ab. Wird das Ergebnis in den Output geschrieben, so sagt diese nichts über die aktuelle Zeit aus. Erst wenn die Zahl von anderen Standardfunktionen von C++ weiterverarbeitet wird, wird ein interpretierbares Ergebnis geliefert. Da die Schüler mit jener Zahl nicht viel anfangen können, wird wieder einige Erklärungszeit gebraucht. Für ein schnelles Erlernen der Sprache ist dies nicht förderlich.

Auch nicht initialisierte Variablen sind ein Beispiel der Abstraktion von C++, die für den Schulunterricht nicht förderlich ist. So haben deklarierte Variablen keine

Standardwerte. Während ein Integer in BASIC mit 0 initialisiert wird, wird er in C++ oft mit jenem Wert initialisiert der gerade in der Speicherzelle steht.
Will man das den Schülern näher bringen, wird auch wieder etwas Zeit gebraucht, denn sie müssen dadurch wieder einiges über die Arbeitsweise des Computers lernen. Dieses Wissen muss ihnen bereits in anderen Fächern beigebracht worden sein. Anderenfalls müssen sie es einfach akzeptieren, was jedoch wiederum nicht ideal ist.

4.2 Freiheiten

Da C++ wie C vor allem für den professionellen Einsatz und nicht für den Schulunterricht geschaffen wurde werden dem Programmierer viele Freiheiten gelassen. Für das Lehren von C++ in der Schule sind deshalb nicht alle Konzepte ideal. Es ist in C++ nämlich möglich in einer `if`-Anweisung einer Variable einen Wert zu geben:

```
if (v = 1) ...
```

Während in anderen Programmiersprachen wie auch in der Mathematik das `=` zwei Werte miteinander vergleicht, wird in C++ dabei der Variable der Wert zugewiesen. Dieses Verhalten muss den Schülern erst gelehrt werden. Da die Aktion jedoch legal ist, kompiliert der Compiler dieses Programmstück ohne das Erzeugen einer Warnung oder Fehlermeldung. In der obigen `if`-Anweisung wird in die Variable `v` der Wert `1` geschrieben. Dadurch wird die `if`-Anweisung ausgeführt, da `1` in C++ bedeutet dass die Anweisung „wahr“ ist.
Wird jedoch der Vergleichsoperator `==` verwendet, so wird die folgende `if`-Anweisung dann ausgeführt, wenn die Variable `v` den Wert `1` besitzt.

```
if (v == 1) ...
```

Während die Schüler meistens wohl die zweite Variante programmieren wollen kann es sein, dass fälschlicher Weise die 1. Variante programmiert wird. Der Fehler ist durch die fehlenden Warnungen bzw. Fehlermeldungen somit für Anfänger nicht sogleich klar ersichtlich.
Auch beim Komma-Operator liefert der C++-Compiler selten Fehlermeldungen. Dabei ist folgender Ausdruck vollkommen erlaubt:

```
double v =( 1,2 );
```

Dabei wird der Klammerausdruck von links nach rechts durchgegangen. Somit nimmt die Variable `v` den Wert `2` an, da der Komma-Operator verschiedene Ausdrücke trennt. Die Schüler welche die Sprache gerade erst lernen, werden allerdings annehmen, dass der Variable `v` der Wert `1,2` zugeordnet wird. Dies ist nämlich die übliche europäische Schreibweise, während C++ die amerikanische erwartet.

Wurde `v` bereits als `double`-Variable deklariert, so ist sogar folgender Ausdruck erlaubt:

```
v = 11,3;
```

Während Anfänger meinen könnten, dass in `v` der Wert `11,3` abgespeichert wird, ist das in der Praxis nicht so. `v` bekommt nämlich den Wert `11`, da der Komma-Operator den Ausdruck in die 2 Ausdrücke

```
v = 11;
```

und

```
3;
```

aufteilt. Während der zweite nichts bewirkt, speichert der erste in die Variable `v` den Wert `11` ab.

Will ein Schüler in eine Variable also eine Kommazahl abspeichern und verwechselt die europäische mit der amerikanischen Schreibweise, so kann es sein, dass er den Fehler erst beim lesen des Programmoutputs und nicht schon bei der Kompilation bemerkt.

Es werden auch andere Ausdrücke wie beispielsweise

```
if (v > 9);
  cout << 8 << endl;
```

akzeptiert. Dabei ist dies nicht eine einzige `if`-Anweisung, sondern zwei Ausdrücke, welche durch einen Strichpunkt voneinander getrennt sind. Dabei ist das Ergebnis des ersten Ausdrucks bei diesem Beispiel für das Programm irrelevant, während der zweite immer abgearbeitet wird und somit wird immer `8` ausgegeben, egal was die `if`-Anweisung ergibt.

Es gibt in C++ auch unterschiedliche Compiler-Verhalten. So erlauben einige Compiler Funktionsaufrufe ohne der Angabe des Aufrufoperators `()`.

```
cout << rand << endl;
```

In diesem Beispiel ergibt das `1`, wobei der GNU C++ Compiler eine Warnung

ausgibt. Andere Compiler erlauben im Gegensatz zu diesem auch die Parameterübergabe von Gleitkommazahlen anstelle von ganzen Zahlen, wobei die Nachkommastellen ohne Meldung abgeschnitten werden.
Für geübte Programmierer sind diese Freiheiten oft angenehm, da die Effekte oft gewünscht sind und der Compiler somit nicht zahlreiche Warnungen liefert. Schüler hingegen werden auf eventuelle Schreibfehler seltener aufmerksam gemacht als in Schulprogrammiersprachen wie Pascal. Diese Tatsachen sprechen nicht für C++ als Programmiersprache für den Schulunterricht. Die Schüler sich dadurch nämlich häufig mit der Fehlersuche anstatt dem Erlernen von C++ beschäftigt. Außerdem sind die Compiler-Meldungen für Schüler nicht so verständlich wie jene von einigen anderen Programmiersprachen.

4.3 Speicherverwaltung

C++ ist wie schon die Programmiersprache C auf optimale Laufzeiteffizienz ausgerichtet. Damit können C++ Programme auf vielen Systemen, vom Waschmaschinencontroller bis zum Serverbetriebssystem, mit sehr optimaler Performance laufen. Die Folge dieser Effizienz ist, dass es keine automatische Verwaltung der Belegung von Speicherbereichen gibt. Jene Verwaltung, welche auch „Garbage collector“ genannt wird, ist bei Sprachen wie Java, Perl, LISP oder VBA enthalten. Dies erleichtert, die Speicherverwaltung sehr.
Schüler müssen sich also in C++ bei der Programmierung jener Datenstrukturen nicht nur auf das Programmierziel selbst konzentrieren, sondern auch darauf, dass die Speicherverwaltung korrekt implementiert ist. Dass also eventuell besetzter Speicher wieder freigegeben wird. Bei dynamischen Typen können leicht Speicherlecks oder sonstige Speicherzugriffsverletzungen entstehen. Die Fehler müssen dabei von den Schülern mühevoll gefunden werden, was wiederum vom Programmziel ablenkt. Außerdem dient es nicht der Motivation eine Programmiersprache zu lernen. Diese Thematik wird selbst von erfahrenden Programmierern nicht immer gelobt, weil vor allem größere Projekte von diesen Fehlern betroffen sind.
Auch die Speicherverwaltung von C++ spricht also dagegen, diese Sprache für

Programmiereinsteiger zu verwenden. Diese Sprache ist nämlich auf höchste Effizienz ausgerichtet, was sicherlich seine Vorteile hat, jedoch nicht das primäre Ziel des Schulunterrichts ist. Für die Schüler ist es einfacher mit Programmiersprachen wie Pascal zu beginnen, weil es die Verhaltensweise von Programmiersprachen auf einfache Art und Weise zeigt und somit den Schülern nicht den Spaß am Programmieren nimmt. Haben die Schüler eine bestimmte Programmiererfahrung erlangt, so können immer noch „höhere“ und effizientere Programmiersprachen gewählt werden.

4.4 Umfang

C++ ist umfangreicher wie die meisten weit verbreiteten Programmiersprachen (Java, C#, ...). Wie C natürlich auch, da C++ den vollständigen Sprachumfang von C enthält. Dieser große Sprachumfang wird von einigen Experten kritisiert. Auch der Erfinder von C++ selbst, Bjarne Stroustrup, bezeichnet diese Programmiersprache als sehr umfangreich und kompliziert.

Der gesamte Sprachumfang ist so groß, dass nur ein Teil von C++ den Schülern beigebracht werden kann. Während auf Assemblerbefehle bei Programmierneulingen vollkommen verzichtet werden sollte, ist auch die C-Syntax für den Schulunterricht nicht angebracht. In diesem Fall sollte anstatt C++ lieber C gelehrt werden.

Will man C++ den Schülern wirklich als 1. Programmiersprache lehren, so muss wegen dem großen Sprachumfang sehr viel Schulzeit eingeplant werden. Will man die Sprache recht ausführlich unterrichten so reicht ein Schuljahr in diesem Fall nicht. Unter C++ wird nämlich allgemein nicht die C Syntax verwendet, sondern objektorientiert programmiert. Diese Programmierweise selbst hat schon einen größeren Sprachumfang als die prozedurale Programmierung von C. Es kommen in der objektorientierten Programmierung nämlich viele Konzepte vor, welche die Einfachheit, Portabilität, Sicherheit, Lesbarkeit, usw. eines Programms erhöhen.

Jedoch können sich verschiedene Programmierparadigmen überschneiden. In C++ wo vor allem das objektorientierte Programmierparadigma verwendet wird, kann für die Textverarbeitung die Klasse `std::string` verwendet werden.

Schon beim genaueren Erklären der Arbeitsweise der Hauptmethode `main()` kommt jedoch bereits das Konzept der Zeichenketten wie sie in C verwendet werden vor. Diese Methode erfordert nämlich einen zweidimensionalen Zeiger des Charakter-Typs (`char **`). Auch bei anderen Funktionen kann es sein, dass dieser Typ verlangt wird.
Somit muss den Schülern auch der Umgang mit C-Zeigern geläufig sein. Dabei wird durch die mehrfachen Möglichkeiten, die sich nicht immer ergänzen sondern oft Doppelstrukturen bilden, viel mehr Zeit in Anspruch genommen als bei anderen beliebten Programmiersprachen. Den Unsicherheiten welche bei den Schülern dadurch entstehen können muss deshalb vorgebeugt werden. Es muss den Schülern also der Sinn dieser Doppelstrukturen genauer erklärt werden.
C++ spezifisch ist auch die generische Programmierung. Diese Programmierweise wird bei der „vector“-Klasse angewendet.

```
vector<int> vec;
```

Dieses Thema wird jedoch später in einem eigenen Kapitel begeitet von einem Beispiel genauer erläutert.
Allgemein gilt, dass der große Sprachumfang kein Vorteil von C++ für den Einsatz in der Schule ist. Der gelehrte Sprachumfang kann zwar bewusst reduziert werden, jedoch bleibt er immer noch größer als jener anderer beliebter Programmiersprachen wie Pascal, C oder Java.

4.5 Präprozessor

Der Präprozessor ist ein weiteres Programmierparadigma von C++, welches in dieser Form kaum in einer anderen Programmiersprache verwendet wird. Häufig wird dabei die `#include`-Option gebraucht, welche sämtliche C++ Bibliotheken einbindet.

```
#include <iostream>
```

Bei der Deklaration von Konstanten kann hingegen die `#define`-Option vermieden werden, indem das Schlüsselwort `const` verwendet wird. Will man hingegen mehrfache Einbindungen von Headerdateien vermeiden, müssen den Schülern wiederum Präprozessoranweisungen erklärt werden.

```
#ifndef __HAUS__
#define __HAUS__
    int haus (int ,int);
#endif
```

Wird eine Headerdatei eingebunden mit obigen Zeilen eingebunden, so kontrolliert `#ifdef` ob das Makro `__HAUS__` bereits definiert wurde. Ist das nicht passiert wird es sogleich definiert und die Funktion `haus()` eingebunden. Anderenfalls wird sie nicht eingebunden und somit eine doppelte Definition vermieden.

Wie man sehen kann ist das nicht die eigentliche C++-Syntax sondern eine vollkommen andere. Somit wird wieder etwas Unterrichtszeit für das Erklären dieser Syntax gebraucht. Denn wie man jetzt weiß, kann dieses Konzept den Schülern nicht verschwiegen werden, auch wenn Makros möglichst vermieden werden sollten.

4.6 Grafische Oberfläche

Während C++ eine sehr umfangreiche Programmiersprache ist und viele Programmierparadigmen implementiert, wurde es für eine leichtere Portabilität auch um einige Bereiche gekürzt. Dabei gilt es vor allem die fehlende grafische Oberfläche zu erwähnen. Will man Schüler jedoch Freude am Programmieren vermitteln, so ist sicherlich auch das grafische Programmieren angebracht. Lernt man ihnen nur das reine ISO C++, so werden sie mit der Dauer etwas enttäuscht sein, da die Programme die Ergebnisse nur an die Konsole weiterleiten können. Diese Programme gelten bei den Schülern wohl eher als veraltete Programme. Auch sind jene Programme in der Praxis eher selten anzutreffen. Meistens kommen sie in professionellen Computersparten oder in Sparten mit einem hohen Sicherheitsbedarf wie beispielsweise in den Banken vor. Beim grafischen Programmieren kann auch die Kreativität der Schüler gefördert werden, was bei Textprogrammen in der Form nicht funktioniert.

Obwohl die ISO im C++-Standard keine grafischen Komponenten definiert, gibt es in der Praxis trotzdem mehrere grafische Oberflächen-Entwicklungswerkzeuge für C++:

- Qt von Trolltech (Nokia)

- GTK+ mit gtkmm
- Visual C++ von Microsoft

Jene 3 werden nun genauer betrachtet:

4.6.1 Qt

Qt ist eine Klassenbibliothek für das Plattform-unabhängige Programmieren von grafischen Oberflächen für verschiedene Programmiersprachen. Sie ist die Grundlage von KDE, eine der bekanntesten Linux-Desktopumgebungen. Qt wird von der norwegischen Firma Trolltech entwickelt, welche am 28. Jänner 2008 vom finnischen Telekommunikationskonzern Nokia übernommen wurde.
Zwar gibt es Qt-Implementierungen für mehrere Programmiersprachen doch von Trolltech selbst werden offiziell nur C++ und Java unterstützt. Die Java-Variante heißt „Qt Jambi“. Für die grafische Programmierung im Schulunterricht ist Qt sehr geeignet, da es neben der kommerziellen Variante auch eine frei verfügbare Version gibt, welche den Schülern gratis verteilt werden kann. Weiters ist sie auch für viele verschiedene Betriebssysteme wie Linux, Mac OS X, Unix-Derivate und Windows verfügbar. Es gibt sogar das Produkt „Qtopia Phone Edition“, welches die Programmiersprache C++ verwendet. Es wird für die Handysoftwareentwicklung eingesetzt.
Damit ein C++ Programm mit Qt erstellt werden kann braucht es die Qt-Klassenbibliothek (beispielsweise: „Qt Open Source Edition for Windows“) und einen unterstützten C++ Compiler. Ist dieser auf dem System noch nicht vorhanden bietet die Installation von Qt bei Windows die Möglichkeit den MinGW-Compiler nachzuinstallieren. Dieser ist eine Windowsportierung vom GCC (wird im Compiler-Kapitel genauer erklärt), welcher bei den Linux-Systemen über die Paketquellen leicht nachinstallierbar ist, falls er nicht schon bei der Installation von Linux mitinstalliert wurde.
Ein Qt-Projekt kann leicht erstellt, kompiliert und gestartet werden. Dafür braucht man vorerst einmal eine grafische Oberfläche. Diese kann mit dem Qt Designer erstellt werden. Dabei wird eine Datei mit der Erweiterung `.ui` als XML-Datei erzeugt. Weiters muss noch eine Hauptdatei namens `main.cpp` erstellt werden, welche die Hauptmethode `main()` enthält. Diese zwei Dateien

reichen auch schon für ein lauffähiges Qt Programm. Wegen der Übersichtlichkeit werden jedoch meistens noch weitere C++-Dateien und Headerdateien erzeugt.

Kompiliert wird ein Qt-Projekt folgendermaßen:

```
qmake -project
qmake
make
```

Dabei erzeugt der erste Befehl die Projektdatei welche die Dateierweiterung `.pro` hat. Der zweite Befehl erzeugt die Makefiledateien, der dritte Befehl das lauffähige Programm.

Ein einfaches „Hello World“-Programm in Qt:

main.cpp:

```
#include "helloworld.h"

#include <QtGui>
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    HelloWorld w;
    w.show();
    a.connect(&a, SIGNAL(lastWindowClosed()), &a, SLOT(quit()));
    return a.exec();
}
```

helloworld.h:

```
#ifndef HELLOWORLD_H
#define HELLOWORLD_H

#include <QtGui/QWidget>
#include "ui_helloworld.h"

class HelloWorld : public QWidget
{
    Q_OBJECT

public:
    HelloWorld(QWidget *parent = 0);
    ~HelloWorld();

private:
    Ui::HelloWorldClass ui;

private slots:
      void buttonGedrueckt();
```

```
};

#endif // HELLOWORLD_H
```

helloworld.cpp:

```
#include "helloworld.h"

HelloWorld::HelloWorld(QWidget *parent)
    : QWidget(parent)
{
      ui.setupUi(this);
      QObject::connect(ui.pushButton, SIGNAL(clicked()), this
            SLOT(buttonGedrueckt()));
}

void HelloWorld::buttonGedrueckt()
{
      ui.lineEdit->setText("Hello World!!!");
}

HelloWorld::~HelloWorld()
{

}
```

Das Layout der grafischen Oberfläche ist in diesem Beispiel in der Datei `helloworld.ui` organisiert.

Hier ein Screenshot des „Hello World“-Beispiels nach dem Klicken des Buttons namens „Klick“:

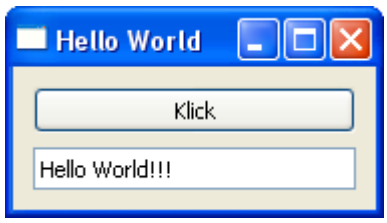

Abbildung 4.1: „Hello World“-Programm in Qt

In den oberen Quelldateien sieht man schön, dass die Definition von Klassen und Methoden, wie im normalen C++, von der Anwendung dieser Definitionen getrennt wird. Es gibt auch in Qt normale Klassen, Konstruktoren, öffentliche und private Funktionen, usw., jedoch unterscheidet sich der Quellcode schon erheblich vom standardisierten C++. Das beginnt schon mit dem `#include`-Befehl wo Qt-spezifische Headerdateien wie beispielsweise `<QtGui/QWidget>` eingebunden werden, weil es im Standard-C++ ja keine

grafische Oberfläche gibt. Auch wenn das C++ in Qt ähnlich dem ISO C++ ist, gibt es oft wesentliche Unterschiede. So gibt es zwar auch in Qt `std::string`, jedoch wird von vielen Funktionen der Qt-spezifische `QString` gebraucht. Ein `QString` muss erst konvertiert werden, wenn er einer `std::string`-Variable zugewiesen werden soll. Umgekehrt muss ein `std::string` auch konvertiert werden, bevor er beispielsweise in eine Textzeile einer Programmanwendung geschrieben werden kann.
Das heißt, dass wiederum einiges an Zeit investiert werden muss, wenn man den Schülern grafisches Programmieren auf C++ Basis beibringen will. Es muss den Schülern auch bewusst gemacht werden, dass es auch andere Klassenbibliotheken für das grafische Programmieren in C++ gibt, da es hier keinen Standard gibt.
Vorteilhaft für Qt ist sicherlich, dass die Schüler welche mit Qt umgehen können damit für die meisten Betriebssysteme Programme erstellen können. Sie können die Opensource-Variante auch daheim verwenden und dort eventuell Hausaufgaben lösen. Die Programme sind auch sehr portabel, wenn sie auf Qt-Basis programmiert wurden. Sie laufen auf allen Systemen wofür es Qt gibt, müssen allerdings neu kompiliert werden. Das ist für die Schule vorteilhaft, da sie dadurch nicht an ein bestimmtes Betriebssystem gebunden ist. Außerdem ist Qt relativ weit verbreitet und dieser Prozess dürfte sicherlich dadurch gefördert werden, dass Nokia jetzt im Besitz von Qt ist. Somit erlangen die Schüler schon Erfahrung für die Berufswelt.
Es muss jedoch auch beachtet werden, dass nicht nur das Lehren von C++ nach dem ISO-Standard, sondern auch die Einführung in das Programmieren mit Qt einiges an Zeit beansprucht.

4.6.2 GTK+ mit gtkmm

gtkmm ist eine C++ Programmbibliothek für das Erstellen von grafischen Benutzeroberflächen. Sie ist eine Anbindung der Programmiersprache C++ an GTK+ (GIMP-Toolkit) mit dem standardmäßig in C Programmiert wird. Es gibt auch Anbindungen anderer Programmiersprachen, jedoch ist gtkmm die bedeutendste. Der Name von gtkmm war usprünglich gtk--. Minus statt Plus

deshalb, weil in GTK+ bereits ein plus vorkommt. Damit gtkmm jedoch leichter von Suchmaschinen gefunden wird, wurden die Minuse in m's umgewandelt.
GTK+ ist eine freie Komponentenbibliothek und Grundlage von GNOME, eine der bekanntesten Linux-Desktopumgebungen. GTK+ ist nicht im Besitz einer Firma sondern wird von mehreren Programmieren erhalten.
GTK+ und gtkmm stehen unter der LGPL-Lizenz, welche die Verfügbarkeit des Quellcodes regelt. Sie unterstützen das Betriebssystem-unabhängige Programmieren. Sie sind für UNIX, Linux und Windows verfügbar. GTK+ für MacOS X ist in der Betaphase.
In Linux ist gtkmm bei vielen Systemen in der Paketverwaltung enthalten. Will man ein Windows-Programm mit gtkmm programmieren braucht es mehrere Komponenten. Wichtig ist das Paket gtkmm-devel, welches die Entwicklungswerkzeuge von gtkmm installiert. Dies braucht die Entwicklungswerkzeuge von GTK+, welche bei vorhandenem Internetanschluss während der Installation von gtkmm mitinstalliert werden können. Weiters braucht es auch eine eigene Konsole auf Linuxbasis. Das kann beispielsweise MSYS sein, welches auf der MinGW-Homepage erworben werden kann. Gebraucht wird auch das pkg-config Paket welches die pkg-config.exe enthält, damit ein gtkmm-Programm in der MSYS-Konsole kompiliert werden kann.
Mit gtkmm gibt es zwei verschiedene Möglichkeiten wie Programme kompiliert werden können:

1) Man schreibt ein Programm, welches aus der Hauptquelldatei und eventuell weiteren Quelldateien mit deren Headerdateien besteht. Die Hauptquelldatei, welche die Hauptmethode enthält kann einen beliebigen Namen haben. Ein Programm welches aus der Datei `prog.cc` besteht kann wie folgt kompiliert werden:

   ```
   g++ prog.cc -o prog.exe -mwindows `pkg-config gtkmm-2.4 --cflags --libs`
   ```

2) Das Design der grafischen Oberfläche kann auch separat vom Quellcode erstellt werden. So kann sie mit Hilfe von Glade, eine freie visuelle Programmierumgebung für das Erstellen von GTK+-Benutzeroberflächen, erzeugt werden. Die Datei dieser grafischen Oberfläche besitzt die Dateierweiterung `.glade`. Wählt man diese

Variante muss im Quellcode die XML-Datei dementsprechend eingebunden werden. Dadurch unterscheidet sich der Quellcode etwas von der 1. Variante. Neben der Glade-Datei wird noch die Hauptquelldatei (diese enthält die Hauptmethode, kann jedoch einen beliebigen Dateinamen haben) und eventuell weitere Quelldateien mit deren Headerdateien gebraucht. Ein Programm welches aus den Dateien `prog.cc` und `prog.glade` besteht, kann folgendermaßen kompiliert werden:

```
g++ prog.cc -o prog.exe -mwindows `pkg-config --libs --cflags
gtkmm-2.4` `pkg-config --libs --cflags libglademm-2.4`
```

Ein einfaches „Hello World“-Programm in gtkmm ohne Glade:

main.cc:

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

int main (int argc, char *argv[])
{
  Gtk::Main kit(argc, argv);
  HelloWorld helloworld;
  Gtk::Main::run(helloworld);
  return 0;
}
```

helloworld.h:

```
#ifndef GTKMM_EXAMPLE_HELLOWORLD_H
#define GTKMM_EXAMPLE_HELLOWORLD_H

#include <gtkmm/box.h>
#include <gtkmm/button.h>
#include <gtkmm/entry.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window
{
  public:
    HelloWorld();
    virtual ~HelloWorld();

  protected:
    virtual void on_button_clicked();

    Gtk::Button m_button;
    Gtk::Entry m_entry;
    Gtk::VBox m_box;
```

```
};

#endif // GTKMM_EXAMPLE_HELLOWORLD_H
```

helloworld.cc:

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

HelloWorld::HelloWorld()
  :
  m_button("Klick"),
  m_entry(),
  m_box(true, 10)

{
  set_title("Hello World");
  set_border_width(10);
  m_button.signal_clicked().connect(sigc::mem_fun(*this,
      &HelloWorld::on_button_clicked));
  add(m_box);
  m_box.pack_start(m_button);
  m_box.pack_start(m_entry);
  m_button.show();
  m_entry.show();
  m_box.show();
}

HelloWorld::~HelloWorld()
{
}

void HelloWorld::on_button_clicked()
{
  m_entry.set_text("Hello World!!!");
}
```

Hier ein Screenshot des „Hello World“-Beispiels nach dem Klicken des Buttons namens „Klick“:

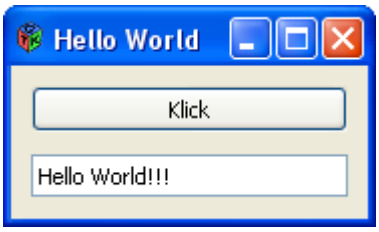

Abbildung 4.2: „Hello World“-Programm in gtkmm

Wie man am Quellcode sieht unterscheidet sich gtkmm durch die GUI-Erweiterung auch sehr vom standardisierten C++. Es wurde jedoch stark darauf geachtet, dass C++ um GUI-Funktionen erweitert wird und C++ selbst gleich

bleibt. Also ist all jenes was in gtkmm nicht die GUI betrifft ISO-C++ konform.
Dies wird den Einstieg der Schüler in diese grafische Komponentenbibliothek etwas erleichtern. Auch die Verfügbarkeit von gtkmm auf verschiedenen Plattformen ist für das Lehren dieser Sprache für den Unterricht förderlich. So ist die Schule nicht an ein Betriebssystem gebunden. Auch die Schüler können somit gtkmm auf verschiedenen Betriebssystemen einsetzen und dort ihre Hausaufgaben lösen. gtkmm welches unter der LPGL-Lizenz steht, ist frei im Internet erhältlich, weshalb es jeder Lehrer und Schüler selbst installieren und anwenden kann.
Damit in gtkmm programmierte Programme auf einem von gtkmm unterstützten Betriebssystem lauffähig sind, müssen sie auf dem entsprechenden System neu kompiliert werden. Dabei müssen im allgemeinen keine Codeänderungen vorgenommen werden, sofern der Code nicht die gtkmm-Funktionalität überschreitet. Diese Fähigkeit kann den Schülern demonstriert werden, was das Interesse am Lehrstoff sicherlich fördern kann.
Wichtig ist jedoch darauf hinzuweisen, dass gtkmm keine Standardprogrammiersprache ist, sondern dass es ein C++ mit grafischen Erweiterungen ist. Weiters ist gtkmm in der Praxis, wenn man die freiwilligen Entwickler nicht berücksichtigt, nicht sehr weit verbreitet.
Allerdings ist ist der Umstieg auf Alternativen wie beispielsweise Qt nicht sehr groß und die Nähe am standardisierten C++ spricht für gtkmm im Schulunterricht. Jedoch muss ernsthaft überlegt werden ob die Zeit für das Lehren von gtkmm reicht. Es muss nämlich eine gute Wissensbasis über das ISO C++ angenommen werden, was auch einige Unterrichtszeit beansprucht. Wenn die Zeit nicht reicht ist es besser auf Sprachen zu setzen welche die grafische Programmierung bereits ohne zusätzliche Werkzeuge unterstützen, wie beispielsweise Java.

4.6.3 Visual C++

Microsoft Visual C++ ist ein System von Microsoft für die Entwicklung von Programmen in der Programmiersprache C++. Die Entwicklungsumgebung dafür heißt Visual Studio, welches neben C++ noch weitere

Programmiersprachen unterstützt. Die aktuelle Version 9.0 (Visual Studio 2008) unterstützt die drei Programmiersprachen Visual Basic, Visual C++ und C#.
Visual C++ setzt seit einigen Jahren auf die Softwareplattform .NET auf und wird nicht mehr direkt in den Maschinencode des Zielsystems, das bisher immer Microsoft Windows war, übersetzt. Im Gegensatz zum standardisierten ISO C++ verlangsamt sich die Performance von Visual C++ also. Es entsteht allerdings die Möglichkeit der Plattform-unabhängigen Programmierung, welche durch die Offenlegung des Quellcodes auch unterstützt wird. Weiters wird die Performance durch die automatische Speicherverwaltung beeinträchtigt, was jedoch das einfachere Programmieren unterstützt. Um die C++ Entwicklergemeinde jedoch für .NET zu gewinnen wurde es von beginn an mit dem Designziel einer hohen Geschwindigkeit entwickelt, um die negativen Performance-Einschnitte zu kompensieren.
Trotz der theoretischen Plattformunabhängigkeit gibt es die .NET-Plattform in vollem Umfang momentan nur für Windows. Dies ist deshalb so, weil Microsoft .NET nicht für andere Betriebssysteme anbietet und andererseits .NET die proprietäre Lizenz *Microsoft Reference License* hat. Zwar gibt es das Mono-Projekt, eine .NET-kompatible Entwicklungs- und Laufzeitumgebung für mehrere Betriebssysteme, jedoch sehen einige die Möglichkeit von Patentansprüchen seitens Microsoft wegen ähnlichen Programmcode. Trotzdem wird die Entwicklung von Mono aktiv von Novell unterstützt.
Will man ein Programm in Visual C++ erstellen, so muss lediglich Visual Studio mit Unterstützung von C++ installiert sein. Im Internet gibt es die Option eine gratis-Version herunterzuladen. Die momentan neuste Version dieser Art ist Visual C++ 2008 Express Edition. Jedoch muss beachtet werden, dass diese Version nach 30 Tagen registriert werden muss, was jedoch gratis ist. Allerdings müssen bei der Registrierung persönliche Daten angegeben werden. Die Installation von Visual Studio setzt das Vorhandensein von .NET voraus, welches bei einer Internetverbindung während der Installation automatisch mitinstalliert werden kann.
Ist Visual Studio installiert kann bereits ein neues Projekt erzeugt werden und mit dem Entwerfen der Benutzeroberfläche und dem Programmieren begonnen

werden.

Ein einfaches „Hello World“-Programm in Visual C++ 2008:

HelloWorld.cpp:

```
#include "stdafx.h"
#include "Form1.h"

using namespace HelloWorld;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
  Application::EnableVisualStyles();
  Application::SetCompatibleTextRenderingDefault(false);

  Application::Run(gcnew Form1());
  return 0;
}
```

Form1.h:

```
#pragma once

namespace HelloWorld {

  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;

  public ref class Form1 : public System::Windows::Forms::Form
  {
  public:
    Form1(void)
    {
      InitializeComponent();
    }

  protected:
    ~Form1()
    {
      if (components)
      {
        delete components;
      }
    }
  private: System::Windows::Forms::Button^  button1;
  private: System::Windows::Forms::TextBox^  textBox1;
  protected:

  private:
    System::ComponentModel::Container ^components;
```

```
#pragma region Windows Form Designer generated code
...
#pragma endregion

  private: System::Void button1_Click(System::Object^  sender,
    System::EventArgs^  e) {
      textBox1->Text = "Hello World!!!";
    }
  };
}
```

Hier ein Screenshot des „Hello World“-Beispiels nach dem Klicken des Buttons namens „Klick“:

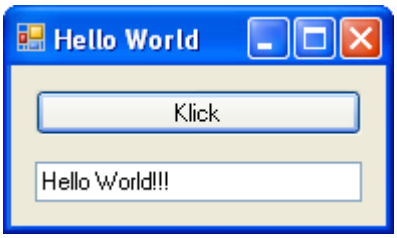

Abbildung 4.3: „Hello World“-Programm in Visual C++

Wie man an den oben angeführten Dateiinhalten erkennen kann ist, in Visual C++ die Trennung von Deklaration und Definition nicht so eingehalten wie normalerweise in C++. Ebenfalls wird von Microsoft C++ nicht nur um GUI Elemente, sondern auch um einige Befehlssätze erweitert, welche unter anderem die .NET-Programmierung vereinfachen sollen.

Diese Tatsachen bewirken, dass die Schüler eine größtenteils „neue Programmiersprache“ lernen, auch wenn sie bereits Vorkenntnisse im von der ISO standardisierten C++ haben. Außerdem müssen für den legalen Einsatz von Visual Studio entweder Lizenzen von Microsoft erworben werden, oder mit dieser Firma eine Vereinbarung für die Nutzung dieser Software in der Schule getroffen werden, will man die Anwender dieser Software nicht zur Registrierung (diese ist für die Express Edition gratis) zwingen. Weiters muss bedacht werden, dass die Schule bei der Verwendung von Visual Studio auch auf Windows setzen muss, da es jene Software offiziell nur für dieses Betriebssystem gibt.

Was für Visual C++ spricht ist, dass diese Programmiersprache im beruflichen

Leben sehr weit verbreitet ist, auch wenn einige Programmierer in gewissen Bereichen lieber auf das ebenfalls von Microsoft entwickelte C# setzen. Außerdem ist Windows momentan so weit verbreitet, dass es verkraftbar ist, wenn eine Anwendung nur für Windows funktioniert. Allerdings kann sich das schnell ändern und deshalb ist es auch für die Schule sinnvoll Plattform-unabhängiges Programmieren zu lehren.

4.6.4 Vergleich der grafischen Oberflächen

Setzt man in der Schule auf Flexibilität und möchte nicht an ein Betriebssystem gebunden sein, so eigenen sich Qt und gtkmm am besten, weil es Visual C++ offiziell nur für Windows gibt. Es gibt mit Mono zwar ein Projekt für die Lauffähigkeit von .NET-Programmen auf Linux, jedoch ist das nicht ein Projekt von Microsoft, den Besitzer von .NET. Außerdem unterstützt Mono .NET noch nicht komplett. Die Schüler müssen außerdem bei der Wahl von Qt oder gtkmm daheim nicht ein vorgegebenes System installieren.
Betrachtet man die freie Verfügbarkeit, dann sind alle 3 Produkte geeignet. Sie können alle frei vom Internet erworben werden, jedoch muss Visual Studio Express nach 30 Tagen registriert werden. Diese Registrierung ist zwar gratis, jedoch müssen private Daten preisgeben werden.
Bei der Einfachheit ist Visual Studio der Favorit. Es kann zwar sein, dass eine neuere .NET-Version von Windows nachinstalliert werden muss, jedoch sonst braucht es nur noch Visual Studio selbst. Ist dies installiert kann bereits mit dem Designen der Oberfläche bzw. dem Programmieren begonnen werden. Bei Qt muss der Benutzer schon eine Ahnung über von Qt unterstütze Compiler haben. Ist nämlich keiner vorhanden ist, muss während der Installation einer nachinstalliert werden. Danach müssen noch die Windows-Umgebungsvariablen gesetzt werden. Das Programmieren in einem normalen Texteditor ist für Fortgeschrittene jedoch nicht empfehlenswert. Will man nun wirklich eine angenehme Entwicklungsumgebung haben, kann entweder auf Visual Studio oder auf Eclipse mit jeweils dem entsprechenden Plugin gesetzt werden. Ist dies alles vollbracht, können auch in Qt auf einfache Art und Weise Programme geschrieben werden. Für gtkmm braucht es ebenfalls mehrere

Komponenten: vor allem die „gtkmm devel“- und „GTK+ devel“-Komponenten. Weiters muss noch ein unterstützter Compiler und das „pkg-config“-Paket installiert werden. Damit nun ein Programm jedoch auch kompiliert werden kann, muss entweder eine Entwicklungsumgebung, oder eine Konsole installiert werden, welche die Syntax des gtkmm Compiler-Kommandos unterstützt. Für die zwei beliebtesten Entwicklungsumgebungen Visual Studio und Eclipse gibt es auch kein Glade-Itegrations-Plugin, welches die grafische Programmierung in gtkmm vereinfachen würde. gtkmm liegt in der Einfachheit somit auf Platz 3.
Will man ein in der Praxis etabliertes Produkt einsetzen, so kann man sicherlich alle 3 Produkte erwähnen. Die weiteste Verbreitung dürfte momentan jedoch Visual C++ vor Qt haben, vor allem im Bereich der kommerziellen Programmierung.

4.7 Syntax

Die Syntax von C++ ähnelt sehr jener anderer Programmiersprachen, wie beispielsweise C, Java, PHP, C#, ... Das ist deshalb so, weil all diese Sprachen sich die Programmiersprache C als Vorbild nahmen, welche eine sehr beliebte Programmiersprache war und auch heute noch ist. Außerdem hatte C++ das Ziel C-kompatibel zu sein und enthält deshalb im Sprachumfang C komplett. Durch die Erweiterung der Sprache um objektorientierte und generische Eigenschaften kamen jedoch noch einige Schlüsselwörter hinzu. C++ erlaubt verschiedene Programmierweisen, jedoch ist das objektorientierte und generische Paradigma die häufigste Art der C++-Programmierung.
Da C++ bzw. C sehr praxisnahe Programmiersprachen sind und auch viele Operatoren besitzen ist die C++-Syntax oft sehr kurz. Das ist beim programmieren einer Anwendung sicherlich für einen Programmierer sehr praktisch. Für eine Lernprogrammiersprache ist diese Tatasche jedoch nicht sehr förderlich. In C++ werden beispielsweise Blöcke mit geschwungenen Klammern eingeleitet bzw. abgeschlossen. In Pascal hingegen beginnt ein Block mit dem Schlüsselwort „BEGIN“ und wird mit dem Schlüsselwort „END“ wieder abgeschlossen.
C++ Syntax:

```
if (zahl == 9)
{
  ...
}
```

Pascal Syntax:

```
IF zahl = 9 THEN
BEGIN
  ...
END
```

Dabei wird die Pascal-Syntax von den Schülern sicherlich schneller verinnerlicht werden, da ein in Pascal geschriebenes Programm übersichtlicher ist. Wie bereits beschrieben ist der wirtschaftliche Faktor des Programmierens in der Schule nicht der wichtigste. Wichtiger ist es den Schülern eine Programmiersprache näher zu bringen, welche sie auf einfache Art und Weise lernen können. Auf diese Wissensbasis aufbauend können dann weitere Programmiersprachen schnell gelernt werden. Sie kann auch als Grundlage für eine selbstständige Weiterbildung in der Programmierung dienen.

4.8 Schnittstellen

In C++ gibt es keine standardisierten Schnittstellen für Internet-Programmierung oder Datenbankenprogrammierung, was das leichte Erstellen solcher Programme sicherlich nicht fördert. [1]
Den Schülern können jene Konzepte sicherlich mit Programmiersprachen wie Java, welche solche standardisierten Methoden hat, leichter gelehrt werden.

4.9 Ein- und Ausgabe

Bei der Ein- und Ausgabe in C++ braucht es bereits die Operatoren „<<“ und „>>“, was schon bei einem „Hello World“-Programm einiges an Erklärungszeit fordert. Der C++-Standard bietet auch keine Möglichkeit ein Programm zu schreiben, welches auf Tastatureingaben interaktiv reagiert. Dadurch können in C++ einige einfache Lernprogramme für die Schule nicht realisiert werden.
Vorteilhaft in C++ ist sicherlich, dass die Ein- und Ausgabe standardisiert ist, während es in C mehrere unterschiedliche Funktionen dafür gibt.

5 C++-Programmierung in der Schule

Der Sprachumfang von C++ ist sehr groß, deshalb wird in der Schule nicht der gesamte Sprachumfang gelehrt werden, vor allem wenn man die grafische Programmierung miteinbezieht. Von dem abgesehen, dass in dieser Arbeit erst später genauer kontrolliert wird, ob sich C++ als Lernsprache für den Schulunterricht eignet, gibt es nicht nur eine Lösung für das Lehren der Sprache. Der Unterrichtsinhalt kann sich sicherlich von Lehrer zu Lehrer und natürlich auch zwischen den verschiedenen Schultypen unterscheiden. In einer technischen Schule wie der HTL wird beispielsweise mehr programmiert werden wie in einer Allgemeinen höheren Schule (AHS).

In dieser Arbeit wird auf einen mehrheitlich Praxis-orientierten Schulunterricht Wert gelegt. Dabei werden in den verschiedenen Kapiteln wichtige Bereiche von C++ beschrieben, welche sich für den Schulunterricht eignen und mit welchen Beispielen man sie einführen kann. Dabei wird nicht von einer bestimmten Schule ausgegangen, denn sonst müssten viele Bereiche mehrfach erklärt werden. Dabei müssen nicht alle Themen, welche hier angesprochen werden in der Schule unterrichtet werden. Je nach dem Niveau der Schule und der verfügbaren Unterrichtszeit wird sich auch der Stoffumfang verändern. Die Erklärung von Konzepten in diesem Kapitel wie man sie im Schulunterricht einführen kann und mit welchen Beispielen es erfolgen kann ist also nicht mit einem Unterrichtsplan gleichzusetzen, sondern höchstwahrscheinlich eine Übermenge. C++ wird jedoch sicherlich nicht 100%ig abgedeckt, schon alleine deswegen, weil C nicht gelehrt wird. Wie bereits beschrieben ist C in C++ enthalten. Somit können in C++ auch C-Bibliotheken eingebunden und verwendet werden. Außerdem müssen die Themen nicht immer in der Reihenfolge gelehrt werden, in welcher sie sich in dieser Masterarbeit befinden. Jeder Lehrer wird nämlich einen eigenen Lehrstil haben und nicht starr nach einem vorgegebenen Unterrichtsplan unterrichten.

Dieses Kapitel ist also wegen den vielen Beispielen in der Form der *Apponierten Übung* geschrieben [2]. Diese Form der Übung ist auch für den Unterricht sehr hilfreich, jedoch zeitlich aufwändig. Deshalb können auch

andere Übungsformen wie die *Direkte Übung* oder Mischformen für den Unterricht angewendet werden, wenn auch die erstere zu bevorzugen ist.

5.1 Erstes Programm

Das erste Programm im Unterricht ist deshalb so wichtig, weil es der erste Kontakt der Schüler mit der Programmiersprache ist. Ist es das erste Programm eines Schülers überhaupt, dann lernt der Schüler das erste mal die Funktionsweise eines Programm kennen, weshalb auch einige Zeit dafür aufgebracht werden sollte. Somit können die Schüler auch spätere Beispielprogramme schneller verstehen.
Ist das erste Programm in C++ für einen Schüler nicht die erste Begegnung mit einer Programmiersprache, dann kann für dieses weniger Zeit beansprucht werden, wenngleich es trotzdem ausführlich erklärt werden sollte. Es ist immerhin jenes Programm, welches die Schüler in C++ einführt. Je nachdem inwieweit sich C++ von der vorher gelehrten Programmiersprache unterscheidet, wird für das erste Programm unterschiedlich viel Zeit beansprucht werden.
Als erste Programme bzw. Einsteigerprogramme in der Programmierung berühmt geworden sind die so genannten „Hello World“-Programme. Diese Programme eignen sich auch für die Schüler beim Einstieg in die C++-Programmierung.
Folgendes „Hello World“-Programm schreibt den Text `!!!Hello World!!!` auf die Konsole:

„Hello World“-Programm in C++:

```
#include <iostream>
using namespace std;

int main()
{
  cout << "!!!Hello World!!!" << endl;
}
```

Ein „Hello World“-Programm hat viele Vorteile. Es ist sehr kurz, jedoch schon

ein vollständig kompilierbares und lauffähiges C++-Programm. Den Schülern kann dessen Funktionsweise bereits erläutert werden indem es den Schülern vorgeführt wird. Da das Programm relativ kurz ist kann den Schülern auch der Programmcode relativ schnell erklärt werden. Diese können somit einen ersten Eindruck vom Ablauf eines C++-Programms bekommen.

In diesem „Hello World“-Programm müssen den Schülern zwar schon einige Konzepte erklärt werden, jedoch können einige wie beispielsweise die Arbeitsweise von Funktionen und dessen Rückgabeparameter auch in späteren Programmierbeispielen näher erklärt werden.

Das kürzeste Programm, welches in C++ geschrieben werden kann ist folgendes:

```
main()
{
}
```

Reiner ISO Standard 2003 ist dies zwar nicht, jedoch lässt dieser den Compilern die Freiheit weitere Implementationen wie die obige zu akzeptieren. Diese Programme sind dadurch natürlich nicht immer beliebig portierbar.

Das kürzeste Programm, welches von allen Standard-konformen C++-Compilern kompiliert werden muss ist folgendes:

```
int main()
{
}
```

Diese Einfachheit nützt beim Erklären der Arbeitsweise von C++ jedoch nicht viel, da die Schüler kein Resultat sehen und somit das Beispiel nicht anschaulich erklärt werden kann. Sicherlich kann man die Schülern nach dem „Hello World“-Programm noch darauf hinweisen, dass es noch ein kürzeres Programm gibt. Dieses Beispiel kann auch beim näheren Erklären der Hauptmethode erwähnt werden, für ein erstes Programm, dass auch die Schüler selbst programmieren müssen, macht es jedoch nicht viel Sinn.

Der Nachteil von jenem „Hello World“-Programm ist, dass bereits ein Operator (<<) und ein Datenstrom vorhanden sind. Somit muss bereits das Konzept der Datenströme erwähnt werden, was eigentlich an einem späteren Zeitpunkt idealer wäre. Um die Ausgabe kommt man jedoch nicht herum, weil ein Einführungsprogramm den Schülern auch ein Resultat liefern muss, damit sie

das Ergebnis auch sehen. Auch wenn es also schon am Beginn den Eingabedatenstrom `cout` braucht, ist die Vertiefung des Konzeptes der Datenströme am Anfang nicht geeignet. Idealer ist die Vertiefung an einem späteren Zeitpunkt, an dem die Schüler bereits genügend Programmierkenntnisse haben.

5.2 Programmstruktur

In C++ ist die Programmstruktur, wie in vielen anderen bekannten Programmiersprachen auch, nicht starr definiert. Das bereits erwähnte „Hello World“-Programm könnte beispielsweise auch folgendermaßen geschrieben werden:

```
#include <iostream> using namespace std; int main() { cout << "!!!
Hello World!!!" << endl; }
```

Deshalb ist es sehr wichtig den Schülern bereits am Beginn des Programmierunterrichts zu erklären, warum ein wohl-strukturiertes Programm für die Lesbarkeit eines Programms unerlässlich ist. Vor allem bei Teamarbeiten und beim Programmieren an größeren Projekten ist eine ordentliche Programmstruktur wichtig. Es gibt in der Praxis jedoch sehr viele unterschiedliche Varianten von Programmierstilen, auch bei C++.

5.2.1 Einrückung

Die Einrückung betrifft vor allem die:

- Tiefe der Einrückung
- Verwendung von Leerzeichen vor `{` und `(`
- Verwendung des Tabulatorzeichen für die Einrückung

Hier gibt es mehrere Varianten die weit verbreitet sind, wobei keine als die allgemein gültige Einrückungsart gilt. Sehr oft sieht man eine Einrückung von 2, 4 oder 8, jedoch auch 3 Stellen.

Auch gibt es Programmierer, welche lieber Leerzeichen als Einrückungszeichen verwenden, während andere dafür das Tabulatorzeichen verwenden. Beide Varianten haben ihre Vorteile.

Diese Tatsache sollte den Schülern erwähnt werden. Dabei gibt es für den Schulunterricht mehrere Möglichkeiten. Es kann beispielsweise ein

Programmierstil bestimmt werden, welchen die Schüler einhalten müssen. Eine Weitere Möglichkeit ist, dass den Schülern selbst die Wahl ihres Programmierstils überlassen wird, wobei sie sich jedoch an einen bereits etablierten Programmierstil halten sollten. Wichtig ist, dass ein Schüler welcher sich für einen Programmierstil entschieden hat diesen auch beibehält und ihn nicht ständig wechselt.

Beide Möglichkeiten haben dabei ihre Vorteile. Lässt man die Schüler selbst entscheiden, welchen Programmierstil sie annehmen wollen, dann gibt es keinen Zwang und sie können sich dadurch schneller mit jenem zurechtfinden. Wird ein Programmierstil vorgeschrieben, dann kann das vorteilhaft für Gruppenarbeiten sein, weil jeder den selben verwendet. Andererseits lernen Schüler welche verschiedene Programmierstile haben sich anpassen, wenn sie an einem gemeinsamen Projekten arbeiten.

5.2.2 Positionierung umschließender Syntaxelemente

Auch bei der Positionierung der Klammern gibt es unterschiedliche Stile wie beispielsweise:

```
if {
   ...
}
```

oder

```
if
{
   ...
}
```

Auch hier bietet sich die Möglichkeit entweder den Schülern einen Stil vorzugeben oder sie selbst entscheiden lassen. Da dieser Bereich nicht so heikel ist wie das Einrücken kann man ruhig die Schüler selbst einen Stil wählen lassen.

5.2.3 Kommentare

Obwohl Kommentare vom Compiler nicht übersetzt werden sind sie als Hilfsmittel sehr wichtig. Auch das richtige Umgehen mit Kommentaren ist wichtig. Beispielsweise muss sich der Schüler entscheiden, ob er oberhalb, unterhalb oder neben der Zeile diese kommentiert. Dabei sind die ersten beiden

Varianten zu favorisieren, da sonst die Zeilen sehr lang werden könnten und manche Editoren die Zeilenbreite beschränken. Die Schüler sollten auch ein Gespür dafür bekommen, wie man kommentiert. Wird überhaupt nicht kommentiert, dann sinkt die Wartbarkeit eines Programms, vor allem wenn es sehr lang ist. Wird zu viel kommentiert kann das schnell die Lesbarkeit eines Programms beeinflussen. Es ist beispielsweise nicht sinnvoll eine Zeile zu kommentieren in der eine Hilfsvariable inkrementiert wird, da dies auch ohne Kommentar ersichtlich ist. Sinnvoll ist es jedoch die Funktionsweise von Funktionen und Klassen näher zu beschreiben.
Es gibt auch Dokumentationsprogramme wie *doxygen*, mit welchen man durch richtiges kommentieren leicht Dokumentationen von einem Programm erzeugen kann. In der Schule kann dies sicherlich erwähnt werden, jedoch ist der Einsatz wegen der zusätzlichen gebrauchten Unterrichtszeit nicht empfehlenswert. Außerdem gehört dieses Thema nicht speziell zum Sprachumfang von C++.

5.2.4 Namensgebung

Ein wichtiger Bestandteil des Programmierstils ist auch die Namensgebung, welche bei ordentlicher Anwendung auch die Lesbarkeit eines Programms sehr fördert.
Den Schülern ist beizubringen, dass Funktionsnamen, Variablennamen, Konstantennamen, Klassennamen, ... möglichst aussagekräftig sind und somit dessen Funktionsweise möglichst genau beschreiben. Natürlich muss das nicht immer sehr streng gehandhabt werden. Wenn es sich bei einer Variable beispielsweise eindeutig ersichtlich um eine Zählvariable handelt, dann reichen auch einfache Namen wie `i` oder `z`. Beachtet werden muss auch, dass ein langer Namen zwar die Funktionalität eines Objektes näher beschreiben kann, jedoch die Übersichtlichkeit eines Programms dadurch beschränkt wird. Außerdem möchte ein Programmierer nicht ständig sehr lange Objektnamen eingeben. Weiters sollten sich die Namen von Funktionen, Variablen, Konstanten, Klassen, ... voneinander unterscheiden. In C oder auch C++ ist folgende Art der Namensgebung weit verbreitet:

- Funktionsnamen und Variablen beginnen mit Kleinbuchstaben, wobei

Funktionsnamen oft auch aus mehreren Wörtern bestehen. Natürlich kann das auch bei Variablen vorkommen.

- Klassennamen beginnen mit Großbuchstaben.
- Konstanten werden ausschließlich mit Großbuchstaben geschrieben

Wichtig ist, dass dies den Schülern gleich schon gelehrt wird, damit sie später auch die Vorteile dieser Namensgebung erkennen.

5.2.5 Anwendung der objektorientierten Programmierung

In C++ ist auf die konsequente Anwendung der objektorientierten Programmierung zu achten, da sie ebenfalls die Übersichtlichkeit eines Programms fördert. Die Schüler sollten dabei selbst ein Gespür für die Anwendung dieser bekommen, beispielsweise wann es besser ist einen Programmcode in eine zusätzliche Klasse auszulagern. Dies kann durch entsprechende Beispiele anschaulich gelehrt werden.
Natürlich sind auch andere Paradigmen wie die generische Programmierung in C++ wichtig, doch ergänzen diese eher die objektorientierte Programmierung in C++ und stehen mit ihr nicht in Konkurrenz. Das Hauptparadigma ist jedoch die objektorientierte Programmierung.

5.3 Interaktionsbeispiel

Nachdem die Schüler schon ein erstes einfaches C++-Programm gesehen und größtenteils verstanden haben kann man das Interesse der Schüler mit einem Programmierbeispiel fördern, in welchem das Programm auf die unterschiedlichen Benutzereingaben reagiert. Vor allem wenn C++ die erste Programmiersprache ist, welche die die Schüler lernen, ist jenes Programm sehr geeignet. Dabei lernen die Schüler nämlich das Erstellen eines Programms, das sie durch Benutzereingaben selbst steuern können.
Das Interaktionsbeispiel in diesem Kapitel fordert den Benutzer auf, eines der 4 Grundelemente einzugeben. Wenn die Eingabe `Erde`, `Feuer`, `Wasser` oder `Luft` heißt, dann wird bestätigt, dass es sich um eines der 4 Grundelemente handelt. Anderenfalls liefert das Programm das Ergebnis, dass es sich um kein Grundelement handelt.

Interaktionsbeispiel-Programm:

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

using namespace std;

int main()
{
  string mystring;
  cout << "Geben Sie eines der 4 Elemente ein:" << endl;
  cin >> mystring;
  if ((mystring == "Erde") || (mystring == "Feuer") || (mystring ==
      "Wasser") || (mystring == "Luft"))
    cout << mystring << " ist ein Grundelement!" << endl;
  else
    cout << mystring << " ist kein Grundelement!" << endl;
}
```

Dieses Beispiel überfordert die Schüler nicht mit komplizierten Konzepten. Neu sind die Variablendeklaration, Eingabe, Ausgabe, `if`-Anweisung, der Vergleich und die oder-Verknüpfung. Diese Konzepte sind alle recht einfach und in kurzer Zeit lehrbar. Die Schüler sehen jedoch, dass ein Programm je nach Eingabe unterschiedlich reagieren kann. Dies kann das Interesse der Schüler fördern. Es wurde bei diesem ersten Eingabeprogramm bewusst der Typ `string` (ist in C++ eine Klasse) verwendet, da dabei keine fehlerhaften Eingaben erfolgen können. Wird eine Zahl erwartet so kann es zu einem Programmabsturz kommen, sofern die Eingabe im Programm nicht kontrolliert wird und ein Buchstabe eingegeben wird. Das ist nicht im Sinne eines Einführungsbeispiels.
Eine passende nächste Übung wäre, dass die Schüler das Interaktionsbeispiel passend kommentieren, wobei beispielsweise die Anweisungen innerhalb der Methode `main()` mit einem Zeilenkommentar // beschrieben werden müssen und vor der Hauptmethode das Programm mit einem mehrzeiligen Kommentar beschrieben werden muss.
In C++ ist der Programmierer nicht gezwungen auf die Ein- und Ausgabe-Datenströme `cin` und `cout` zu setzten. Diese Datenströme haben jedoch mehrere Vorteile wie Typsicherheit und sind weniger fehleranfällig. Außerdem sind es vom Programmierer erweiterbare Klassen, was dem objektorientierten Paradigma entspricht. Dadurch und weil diese Datenströme in der Praxis für die

Ein- und Ausgabe empfohlen werden, sollten Methoden wie `printf` (Eingabe) und `scanf` (Ausgabe) in der Schule nicht gelehrt werden [3]. Diese sind C-Methoden und nehmen wieder einiges an Unterrichtszeit in Anspruch. Stattdessen können andere wichtige C++-Konzepte gelehrt werden.

5.4 Interaktionsbeispiel 2

Haben sich die Schüler bereits etwas mit C++ angefreundet, so kann man ihnen nun ein Beispiel mit Zahleneingabe und einfacher Fehlerbehandlung geben.
Folgendes Programm kontrolliert, ob die Eingabe eine Zahl ist oder nicht und liefert die dementsprechende Antwort.

Beispielprogramm:

```
#include <iostream>
#include <sstream>
#include <istream>
#include <string>

using namespace std;

int main()
{
  string mein_string;
  int zahl;
  cout << "Geben Sie eine Zahl ein: ";
  cin >> mein_string;
  stringstream ss(mein_string);
  if (ss >> zahl && ss.eof())
    cout << "Zahl!" << endl;
  else
    cout << "keine Zahl!" << endl;
}
```

Dieses Programm erfordert bereits eine etwas erweiternde Einführung in das Konzept der Datenströme. Weiters ist nur noch das Thema der Fehlerbehandlung neu. Nämlich dass Datenströme auch ein Ergebnis liefern, ob die Aktion erfolgreich war oder nicht. Auch Funktionen können in C++ so wie in C Fehlerwerte liefern. In C++ gibt es jedoch wie in Java noch das Konzept der Exceptions (Ausnahmefälle), welche „geworfen“ und „abgefangen“ werden können. Es ist angebracht dieses Thema den Schülern später zu erklären, da es bereits einen gewissen Für den Anfang reicht auch das Abfragen von

Rückgabewerten.
Es ist jedoch nicht das Ziel des Schulunterrichts Programme zu schreiben, welche jede mögliche falsche Eingabe abfangen. Dies Kostet viel Zeit und hemmt den Spaß der Schüler am Programmieren. Idealer ist es, wenn den Schülern anfangs die Programmiermöglichkeiten von C++ vermittelt werden. Eine größere Programmsicherheit hat erst bei genügend vorhandenem Grundwissen über C++ eine höhere Priorität.

5.5 Typen

Das Konzept der Typen gehört in C++ zum Grundwissen der Programmiersprache, weshalb die wichtigsten (der leere Typ `void`, numerische Typen, Zeichen bzw. Zeichenketten und der Boolean-Typ) schon sehr früh erwähnt werden sollten. [4]

5.5.1 Datentyp void

Dieser Typ gibt an, dass kein Wert vorhanden ist. Beispielsweise kann man ihn in der `main`-Methode statt der Parameterliste angeben, falls dieser keine Werte übergeben werden. Das selbe gilt natürlich auch für jede andere Funktion.

5.5.2 Numerische Typen

Eine geeignete Lehrmethode ist es Unterrichtskapitel nach einer kurzen Einleitung anhand geeigneter Beispiele von den Schülern lösen zu lassen bevor sie auf Folien oder auf der Tafel näher betrachtet werden. Dadurch ist der Inhalt des Unterrichtsstoffes schon während dem Erklären den Schülern etwas vertrauter. Durch das zweite Interaktionsbeispiel haben die Schüler beispielsweise in der Praxis den Integer-Typ `int` kennen gelernt. Dieser muss den Schülern sicherlich vor diesem Beispiel gründlich erklärt werden. Da sie durch das Interaktionsbeispiel jedoch bereits den Integer-Typ kennen gelernt haben, tun sie sich bei einer anschließenden Einführung in das Konzept der numerischen Typen sicherlich um einiges leichter.
Weitere wichtige numerische C++-Typen sind dabei `char`, `short` bzw. `short int`, `unsigned int`, `long` bzw. `long int`, `float`, `double`, usw. Diese

können anhand einer Beispieltabelle, mit dem üblichen Maximum, Minimum und Speicherbedarf der Typen, den Schülern veranschaulicht werden. Wichtig ist jedoch, dass stets darauf hingewiesen wird, dass Wertebereich und Speicherbedarf der Typen auf verschiedenen Systemen verschiedene Werte haben können. Die eventuell angegebenen Werte dienen nur der Veranschaulichung.
Damit Wertebereich und Speicherbedarf festgestellt werden können, gibt es in C++ einfache Werkzeuge dafür, welche den Schülern gezeigt werden sollten. Die Byte-Größe wird mit `sizeof` bestimmt während es für das Bestimmen des Wertebereichs das Template `numeric_limits` gibt. Mit diesem können neben dem Maximum und Minimum eines Typs auch weitere Eigenschaften abgefragt werden. Beispielsweise ob der Typ `char` auf dem System als `signed char` oder `unsigned char` definiert ist. Damit `numeric_limits` überhaupt verwendet werden kann, muss das Header `<limits>` im Programmcode eingebunden werden.

Beispiel für das Bestimmen von Wertebereich und Speicherbedarf von Typen:

```
#include <iostream>
#include <limits>

using namespace std;

int main()
{
  cout << "signed char max: "
    << (int)numeric_limits<signed char>::max() << endl;
  cout << "unsigned char max: "
    << (int)numeric_limits<unsigned char>::max() << endl;
  if (numeric_limits<char>::is_signed)
    cout << "char ist als signed char definiert!" << endl;
  else
    cout << "char ist als unsigned char definiert!" << endl;
  cout << "int max: " << numeric_limits<int>::max() << endl;
  cout << "float max: " << numeric_limits<float>::max() << endl;
  cout << "float Bytegroesse: " << sizeof(float) << endl;
}
```

Dieses Beispiel ist primär für das Bestimmen des Wertebereichs und des Speicherbedarfs eines Typen gedacht und nicht als Einführungsbeispiel für Templates. Dieses Thema sollte später genauer erklärt werden. Es reicht

schon, wenn die Schüler mit Hilfe von `numeric_limits` die erforderlichen Werte erhalten können. Im Beispiel muss zweimal gecastet werden, damit der Maximale Wert von den entsprechenden `char`-Typen ausgegeben wird und nicht das ASCII-Zeichen dieses Wertes. Dies kann den Schülern näher erklärt werden, da im Laufe des Schulunterrichts sicherlich noch öfters gecastet werden muss. Vor allem weil der Typ `char` als numerischer Typ oder als Zeichen-Typ verwendet werden kann, bei der Ausgabe allerdings standardmäßig als Zeichentyp interpretiert wird. In C++ gibt es auch noch andere Möglichkeiten des Castens, welche jedoch erst nach der Einführung in die C++-Typen gelehrt werden sollten.

Wichtig bei numerischen Datentypen ist auch die Kennenlernen der Typen-Präfixe. Standardmäßig werden ganze Zahlen bei unbekanntem Datentyp als Integer und Gleitkommazahlen als Double interpretiert. Wichtige Präfixe sind: `s` bzw. S (`short`), `l` bzw. `L` (`long`), `f` bzw. `F` (`float`), `lf` bzw. `Lf` (`long double`), ... Geschrieben wird eine Floatzahl dann folgendermaßen:

```
3.14f bzw. 3.14F
```

Mit dem Präfix `u` bzw. `U` wird dem Compiler angedeutet, dass es sich um eine ganze Zahl ohne Vorzeichen handelt. Beispiel eines Long's ohne Vorzeichen:

```
3ul bzw. 3UL
```

Weiters können Zahlen auch mit den Vorzeichen + oder – beginnen. Das Exponentenzeichen e bzw. E ist nur für Gleitkommazahlen erlaubt. Beispielsweise kann 2.100.000 wie folgt geschrieben werden:

```
2.1E6
```

0,2 kann als

```
2E-1
```

dargestellt werden.

Obwohl der Wertebereich und die Abspeicherungsart der meisten numerischen Typen verschieden sind, können sie trotzdem einander zugewiesen werden. Ein `float`-Wert kann beispielsweise immer in einem `double`-Typ abgespeichert werden, da dieser genauer ist. Jedoch ist auch der umgekehrte Fall zulässig, sofern der Wert im Wertebereich von `float` liegt, beispielsweise 4.0/3.0. Natürlich ist der `float`-Wert nicht mehr so genau wie der `double`-Wert, da

einige Nachkommastellen weggeschnitten werden. Dies ist eine Tatsache, welche im Schulunterricht erwähnt werden sollte, damit sie den Schülern bewusst wird.
Noch wichtiger ist jedoch, dass die Schüler Kenntnis über die Schreibweise von Gleitkommazahlen haben. Wird nämlich einer Variable mit dem Typ `float` oder `double` der Wert `4/3` zugeordnet, so besitzt sie jeweils den Wert `1`. Die Zahlen `4` und `3` werden nämlich als Integer interpretiert. Deshalb ist auch das Ergebnis der Division ein Integer, weshalb die Nachkommastellen weggeschnitten werden. Schreibt man jedoch `4.0/3.0` bzw. `4.0f/3.0f`, dann wird das Ergebnis als `double`- bzw. `float`-Wert interpretiert. Dieses Verhalten ist C/C++-spezifisch und muss deshalb beim Lehren der Typen erwähnt werden.

5.5.3 Einzelnes Zeichen

Der Typ `char` kann zwar auch als Zahlentyp verwendet werden, wird in der Praxis jedoch meist als Zeichentyp interpretiert. Will man nämlich einen `char`-Wert als Zahl ausgeben, so muss er vorher als `int` gecastet werden, während bei der Ausgabe eines Zeichens nicht gecastet werden muss. In einer `char`-Variable sind alle ASCII-Zeichen abspeicherbar. Dieser ASCII-Zeichensatz ist normiert und auf allen Systemen gleich.
Ein Zeichen kann folgendermaßen deklariert werden:

```
char zeichen = 'E';
```

Deklaration eines Zeichens mit dem Inhalt des ASCII-Zeichens Nummer 65:

```
char zeichen = 65;
```

Die Variable `zeichen` hat nun den Zeichen-Wert `'A'`.
Den Schülern müssen diese Tatsachen über den Zeichentyp `char` ausführlich erklärt werden, da es eine wichtige Grundlage in Hinblick auf Zeichenketten darstellt. Es ist nämlich in den meisten Programmiersprachen nicht üblich, dass ein Typ für verschiedene Zwecke (Zahl und Zeichen) verwendet werden kann.

5.5.4 Zeichenketten (Strings)

Der Stringtyp `string`, der den Schülern wegen den Interaktionsbeispielen bereits bekannt sein sollte, ist in C++ eine Klasse. Wie bereits in jenen

Beispielen ersichtlich muss bei Verwendung von Strings die Header-Datei `<string>` eingebunden werden.

Stringtypen und ihre Möglichkeiten können auch schon am Anfang des Schuljahres unterrichtet werden, gleich nachdem die numerischen Typen eingeführt wurden. Strings müssen jedoch ausführlicher als die einzelnen Typen `int`, `float`, ... gelehrt werden, da die String-Klasse mehrere Programmiermöglichkeiten bietet. Das Konzept der Klassen ist eines der umfangreichsten in C++ und sollte den Schülern später gelehrt werden. Die Anwendung von Klassen (beispielsweise die String-Klasse) kann den Schülern jedoch schon während der Einführung in die C++-Typen zugetraut werden.

Da Strings nicht normale Typen wie `int`, `float`, ... sondern Klassen sind, können sie mit Hilfe vorgegebener Methoden unter anderem auch nach bestimmten Zeichen oder Zeichenketten durchsucht werden. Auch das Anhängen, Einfügen, Ersetzen und Löschen von Zeichen oder Zeichenketten ist möglich. Mit den Methoden

```
const char* data() const;
const char* c_str() const;
```

können C++-Strings auch als C-Zeichenketten umgewandelt werden, wobei die Methode `c_str()` auch die `'\0'`-Terminierung anhängt, welche von einigen Methoden in C++ erwartet wird. Weitere Informationen über C-Strings finden sich in den Kapiteln der Arrays und Zeiger.

In folgenden Zeilen wird mit der Methode `size()` die Stringlänge bestimmt, welche in diesem Beispiel den Wert `3` liefert.

```
s = "Hut";
laenge = s.size();       /* laenge == 3  */
```

Beispielprogramm für die Suche eines Teilstrings:

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

using namespace std;

int main()
{
  string s, st;
  cout << "Geben Sie ein Wort ein:" << endl;
```

```
  cin >> s;
  cout << "Geben Sie ein Teilwort ein:" << endl;
  cin >> st;

  string::size_type i = s.find(st);

  if (i != string::npos)
    cout << "\"" << st << "\" wurde an der " << i+1 << ". Stelle
        gefunden!" << endl;
  else
    cout << "Kein Fund!" << endl;
}
```

In diesem Beispielprogramm wird ein Wort und ein Teilwort eingegeben. Wird das Teilwort im Wort gefunden, so wird jene Stelle ausgegeben, ab welcher das Teilwort von links nach rechts das erste. mal gefunden wurde. Anderenfalls wird der Benutzer informiert, dass kein Fund vorliegt.
Anhand dieses Beispiels sehen die Schüler also, wie man die Klasse `string` in C++ anwenden kann. Es gibt sicherlich noch reichlich weitere Funktionen in dieser Klasse, das Prinzip ist jedoch das selbe. Man muss deshalb nicht jede einzelne Methode vorführen. Es reicht, wenn die Funktionsweise der Methoden beschrieben wird und den Schülern das Konzept der `string`-Klasse anhand eines Beispiels veranschaulicht wird. Auch `find()`-Methode bietet sich dafür an, da sie eine sehr nützliche und einfache Funktion ist, welche häufig gebraucht werden kann.
Wie bereits erwähnt gibt es in C++ auch C-Strings. Dabei sollten falls möglich erstere immer vorgezogen werden, da sie sicherer sind und der objektorientierten Programmierung entsprechen. Die Möglichkeiten von C-Strings finden sich in den Kapiteln der Arrays und Zeiger.

5.5.5 Logische Werte

In C++ gibt es auch den Datentyp `bool`, welcher die logischen Werte `true` (wahr) und `false` (falsch) annehmen kann. Diese Tatsache ist auch für Schüler mit Grundkenntnissen in C interessant. In C gibt es diesen Datentyp nämlich erst seit dem neuesten Standard. Bei Benutzung dieses Datentyps in C muss jedoch im Gegensatz zu C++ eine Bibliothek eingebunden werden.
Ein Booleantyp lässt sich folgendermaßen deklarieren und inizialisieren:

```
bool b = true;
```

5.5.6 Weitere Datentypen

Es gibt in C++ auch die Datentypen `enum`, `union` und `struct`. Dabei sind vor allem die ersten beiden hervorzuheben, `struct` ist hingegen mehr wegen der Kompatibilität zu C vorhanden. Eine Struktur ist nämlich mit einer Klasse in C++ vollständig nachbildbar. Der einzige unterschied zwischen einer Struktur und einer Klasse ist der, dass in der Struktur standardmäßig alle Elemente `public` und nicht `private` sind. Eine Struktur hat folgende Form:

```
struct Produkt
{
  string name;
  int anzahl;
};
```

Zugegriffen werden kann auf eine Variable folgendermaßen bei einer initialisierten Variable `p` vom Typ `Produkt`.:

```
int a = p.anzahl;
```

Für einfachere Datentypen wird eine Struktur jedoch häufig gegenüber Klassen bevorzugt.

Der `union`-Datentyp teilt hingeben allen seinen Membervariablen den selben Speicher zu.

```
union Test
{
  int Int;
  char bytes[4];
};
```

Im oberen Beispiel kann in *Test* ein Integer abgespeichert werden und dann mit Hilfe des `char`-Arrays auf die vier verschiedenen Bytes des Integers zugegriffen werden, weil sie den selben Speicher besetzen. Zugegriffen wird auf die selbe Art wie bei Strukturen. Im Beispiel wird angenommen, dass ein Integer auf dem System 4 Bytes groß ist.

Ein `enum`-Typ kann verschiedene selbst definierte Werte abspeichern.

```
enum farbe {ROT=7, GELB, GRUEN, BLAU};
farbe ampel = GRUEN;
if (ampel == 9)
  cout << "GRUEN" << endl;
```

Im obigen Beispiel wurde ein `enum` namens `farbe` mit den Werten `ROT`, `GELB`,

GRUEN und BLAU definiert, wobei ROT den Wert 7 bekommt, GELB den Wert 8, ... Danach wird in die Variable ampel der Wert GRUEN abgespeichert. Da dieser den Wert 9 hat, wird die if-Anweisung ausgeführt. Statt 9 könnte man genauso GRUEN schreiben.

Dies sind alles einfachere Beispiele, welche diese Typen näher erklären. Diese Typen haben in C++ zwar ihre Berechtigung, in der Schule sind sie jedoch nicht von so großer Wichtigkeit, dass sie um jeden Preis gelehrt werden müssen. Wenn dieses Thema leicht in den Unterrichtsplan passt, dann können die Typen natürlich auch unterrichtet werden. Anderenfalls können sie auch nur kurz erwähnt werden, damit sich interessierte Schüler das Wissen selbst aneignen können.

In C++ gibt es auch Bit-Felder, welche nur eine bestimmte Anzahl von Bits Speicherplatz brauchen. Dieser Datentyp ist jedoch für die Optimierung von Programmen geschaffen worden und ist für den Schulunterricht nicht von Bedeutung.

Mit dem Befehl typedef können gültigen C++-Typen eigene Namen gegeben werden. Dies kann oft sehr angenehm sein, will man diesen jedoch unterrichten, so muss genügend Unterrichtszeit zur Verfügung stehen.

5.6 Operatoren

Operatoren sind ein weiterer Bestandteil von C++, welcher den Schülern gelehrt werden muss. In C++ gibt es viele Operatoren wobei manche für den Informatikunterricht sehr wichtig, manche hingegen vernachlässigbar sind.

5.6.1 Arithmetische Operatoren

Einige dieser Operatorenzeichen, wie die Vorzeichen und die vier Grundrechenarten, sind den Schülern wohl schon vom Mathematikunterricht bekannt. Andere Zeichen wie der Modulodivisions-Operator (%) und die vier Operatoren Postfix/Präfix Inkremente/Dekremente werden ihnen näher erklärt werden müssen.

Dafür eignen sich besonders Beispiele wie folgendes:

Beispielprogramm für arithmetische Operatoren:

```
#include <iostream>

using namespace std;

int main()
{
  int a=2, b=3, c=4, erg;
  erg = b*a+++c;
  cout << "a: " << a << "  b: " << b << "  c: " << c << endl;
  cout << "erg: " << erg << endl;
  erg = ++c%a;
  cout << "a: " << a << "  b: " << b << "  c: " << c << endl;
  cout << "erg: " << erg << endl;
}
```

Dabei können den Schülern mehrere Formeln vorgegeben werden, welche sie auswerten und den Variableninhalt bestimmen müssen bevor sie das Programm erstellen und ausführen. Vorher muss natürlich die Priorität der verschiedenen arithmetischen Operatoren gelehrt werden. Beim Ausführen des Programms können sie dann die Programm-Ergebnisse mit ihren errechneten Ergebnissen vergleichen.

Bei der ersten Ausgabe ist `a=3`, `b=3`, `c=4` und `erg=10`. Die Formel ist also auch folgendermaßen darstellbar:

```
erg = b * a++ + c;
```

`a` wird erst nach der Rechenoperation um `1` erhöht.

Bei der zweiten Ausgabe ist `a=3`, `b=3`, `c=5` und `erg=2`. Die Formel kann auch so dargestellt werden:

```
erg = ++c % a;
```

In der Praxis sind die Prioritäten der arithmetischen Operationen häufig bekannt. Ist das jedoch nicht der Fall, so können sie mit dem Klammeroperator `()` explizit festgelegt werden. Für Übungszwecke sollte in der Schule jedoch bei der Einführung der arithmetischen Operatoren auf die Klammerung verzichtet werden, während sie später von den Schülern verwendet werden darf.

5.6.2 Vergleichsoperatoren

Die Vergleichsoperatoren werden den Schülern schnell erlernen, da deren Funktionsweise bereits von der Mathematik her klar ist. Die Schüler müssen

sich nur noch an deren Schreibweise gewöhnen, sofern C++ für sie die erste Programmiersprache ist oder sie noch keine Programmierkenntnisse über eine Syntax-ähnliche Programmiersprache haben. Die Vergleichsoperatoren können den Schülern anhand einer Tabelle veranschaulicht werden. Wegen der Einfachheit müssen sie aus zeitlichen Gründen nicht in einem eigenen Beispielprogramm eingeführt werden. Einige werden auch schon vor dem Lehrbeginn dieses Themas in Übungsbeispielen verwendet werden. Weitere können eventuell bei der Einführung der Schleifen verwendet werden.

5.6.3 Logische Operatoren

Es gibt drei logische Operatoren: das logische Und `&&`, logische Oder `||` und logische Nicht `!`. Es kann sicherlich vorkommen, dass sie bereits in vorigen Kapiteln verwendet wurden, wie beispielsweise das logische Oder im Interaktionsbeispiel. Falls nicht reicht ein kurzes Beispiel zur Veranschaulichung der Verwendung dieser Operatoren.
Danach kann den Schülern tabellarisch das Verhalten dieser Operatoren gezeigt werden. Haben die Schüler auch elektronische Fächer dürfte dieses Thema schnell erklärt sein.

5.6.4 Bit-Operatoren

Auch für die Bit-Operationen gibt es das Und `&`, Oder `|` und Nicht `~`. Weiters gibt es noch das bitweise Exklusive Oder `^` und den Links-Shift `<<` und Rechts-Shift `>>`. Wenn den Schülern noch keine C++-Schleifen bekannt sind, eignet sich folgendes Beispiel:

Beispiel für Bit-Operationen:

```
#include <iostream>
#include <cctype>
using namespace std;

int main (void)
{
  int zahl1, zahl2;
  char op;
  cout << "Geben Sie zwei Zahlen ein." << endl;
  cin >> zahl1;
```

```
  cin >> zahl2;
  cout << "Welche Operation wollen Sie anwenden:" << endl;
  cout << "[U]nd, [O]der, [E]xklusives Oder?" << endl;
  cin >> op;
  if (toupper(op) == 'U')
    cout << (zahl1 & zahl2) << endl;
  else
    if (toupper(op) == 'O')
      cout << (zahl1 | zahl2) << endl;
    else
      if (toupper(op) == 'E')
        cout << (zahl1 ^ zahl2) << endl;
      else
        cout << "Keine Operation gewaehlt!" << endl;
}
```

Mit diesem Programm können die Schüler selbst Ergebnisse der verschiedenen Operationen kontrollieren. Dieses Beispiel kann leicht für fächerübergreifenden Unterricht (beispielsweise mit Mathematik oder Elektronik) verwendet werden.

5.6.5 Weitere Operatoren

Neben den genannten Operatoren gibt es noch weitere. Einige dieser Operatoren sind jedoch sehr simpel und müssen deshalb nicht in einem eigenen Thema behandelt werden. Den Schülern ist ihr Verhalten sicherlich gleich geläufig, beispielsweise jenes des Zuweisungsoperators `=`. Der Sinn anderer Operatoren wird den Schülern hingegen während der Einführung in weiteren Kapiteln klar. Wird beispielsweise das Kapitel der Zeiger gelehrt, so lernen die Schüler die Operatoren `->`, `&` und `*` kennen.

Deshalb müssen diese Operatoren nicht in einem eigenen Thema erwähnt und erklärt werden.

5.7 Anweisungen und Kontrollstrukturen

Nach der Einführung der wichtigen C++-Themen „Typen“ und „Operatoren“ bietet sich als nächstes das Thema „Anweisungen und Kontrollstrukturen" an. Diese werden nämlich bei fast jedem Programm gebraucht, auch wenn es nur ein kleines ist. Mit ihnen lassen sich schon wesentlich gebräuchlichere Programme schreiben, als mit den bisher gekannten C++-Werkzeugen.

Jede den Schülern noch nicht bekannte Anweisung bzw. Kontrollstruktur sollte in einem eigenen Beispiel veranschaulicht werden, da die meisten eine recht

unterschiedliche Funktion haben und jede einzelne zum Grundwissen von C++ gehört.

5.7.1 if-Anweisung

Diese Anweisung wird sicherlich schon vor dem Beginn dieses Themas verwendet werden. In diesem Thema braucht sie deshalb nicht näher betrachtet werden. Nur die verschachtelte `if`-Anweisung sollte näher betrachtet werden, wenn sie noch nicht verwendet wurde.

5.7.2 switch-Anweisung

Haben die Schüler das oben erwähnte *Beispiel für Bit-Operationen* bereits gelöst, so bietet sich an, dieses Programm nun statt verketteten `if`-Anweisungen mit einer `switch`-Anweisung zu programmieren.

Bit-Operationen-Beispiel mit switch-Anweisung:

```
#include <iostream>
#include <cctype>
using namespace std;

int main (void)
{
  int zahl1, zahl2;
  char op;
  cout << "Geben Sie zwei Zahlen ein." << endl;
  cin >> zahl1;
  cin >> zahl2;
  cout << "Welche Operation wollen Sie anwenden:" << endl;
  cout << "[U]nd, [O]der, [E]xklusives Oder?" << endl;
  cin >> op;
  switch (toupper(op))
  {
    case 'U':
      cout << (zahl1 & zahl2) << endl;
      break;
    case 'O':
      cout << (zahl1 | zahl2) << endl;
      break;
    case 'E':
      cout << (zahl1 ^ zahl2) << endl;
      break;
    default:
      cout << "Keine Operation gewaehlt!" << endl;
  }
}
```

5.7.3 Kontrollstrukturen

Es gibt in C++ die `while`-Schleife, `do-while`-Schleife und die `for`-Schleife. Jede dieser drei ist sehr wichtig und sollte gründlich erklärt werden.
Nachdem die Schüler bereits Bekanntschaft mit den Bit-Operatoren gemacht haben und spätestens seitdem wissen müssten was binäre Zahlen sind, bietet sich auch folgendes Beispiel an:

while-Schleifen-Beispiel:

```
#include <iostream>
#include <string>
using namespace std;

int main (void)
{
  int zahl;
  string bin = "";
  cout << "Geben Sie eine positive Zahl ein:" << endl;
  cin >> zahl;
  while (zahl > 1)
  {
    short bit = zahl % 2;
    zahl = zahl / 2;
    if (bit == 0)
      bin = "0" + bin;
    else
      bin = "1" + bin;
  }
  if (zahl == 1)
    bin = "1" + bin;
  else
    bin = "0" + bin;
  cout << bin << endl;
}
```

In diesem Beispiel lernen die Schüler die Funktionalität einer `while`-Schleife in der Praxis kennen. Dabei wird eine positive Zahl eingegeben, welche vom Programm in eine binäre Zahl umgewandelt und ausgegeben wird.

5.7.4 Sprunganweisungen

In C++ gibt es mehrere Sprunganweisungen: `goto`, `break`, `continue`, `return` und `exit`. Diese müssen nicht in einem eigenen Kapitel erwähnt werden. Es ist angebracht `break` und `continue` im Kapitel der Schleifen bzw.

der `switch`-Anweisung näher zu betrachten und `return` und `exit` bei der Einführung der Funktionen in C++.
`goto` entspricht weder der objektorientierten Programmierung, noch der strukturierten oder generischen Programmierung. Da diese Anweisung auch Kompilierprobleme und bei falscher Verwendung falsche Variableninitialisierungen verursachen kann, ist sie für das Programmieren im Schulunterricht nicht geeignet.
Es gibt zwar auch positive Meinungen bezüglich der `goto`-Anweisung, jedoch sind die Vorteile gegenüber den Nachteilen zu klein, damit diese Anweisung für den Schulunterricht unerlässlich wäre [5].

5.8 Funktionen

Das Kapitel der Funktionen ist ein aufwendiges jedoch unerlässliches Thema für den Informatikunterricht mit C++. Besonders der Funktionskopf muss den Schülern genau erklärt werden, da dort mehrere Konzepte verstanden werden müssen:

- Typ des Rückgabewertes
- Übergabeparameter
- Speicherklassen (Bsp: `static`)

Dabei sollte wiederum der C++ Standard eingehalten werden. Nach dem ISO Standard vom Jahr 2003 werden Funktionen, denen keine Variablen übergeben werden, parameterlos geschrieben. Der Datentyp `void` wird also nicht in die Parameterliste geschrieben, was von einigen Compilern akzeptiert wird.
Im Funktionsrumpf gibt es hingegen nicht viel Neues. Das einzige Neue ist die Rückgabe von Variablenwerten und dass `return` öfters als einmal aufgerufen werden kann. Die Rückgabe von Werten ist bereits von der `main`-Methode bekannt. Im Unterschied zu dieser ist die `return`-Anweisung nicht optional, sondern sie muss angegeben werden. Die Ausnahme gilt, wenn die Methode keinen Wert zurückliefert, also den Rückgabetyp `void` besitzt.
Ein weiteres Wichtiges Thema das im Funktions-Kapitel nicht fehlen darf ist der Funktionsaufruf und wie Daten zwischen der aufgerufenen Funktion und dem Programmblock, der die Funktion aufruft, ausgetauscht werden.

Wurde das Kapitel der Funktionen von den Schülern angemessen verstanden, dann sollte jenes mit dem Konzept der rekursiven Funktionsaufrufe abgeschlossen werden. Die Rekursion ist ein wichtiger Bestandteil der C++-Programmierung mit Funktionen, weil sie viele Programme vereinfacht und übersichtlicher gestaltet. Sie kann einfach anhand von mathematischen Beispielen, wie der Berechnung der Fibonaccizahlen oder der Fakultäten, eingeführt werden.

Rekursionsbeispiel welches die Fibonaccizahlen berechnet:

```
#include <iostream>
using namespace std;

int fibonacci (int f)
{
  if (f < 3)
    return 1;
  else
    return fibonacci(f-2) + fibonacci(f-1);
}

int main()
{
  int fibzahl;
  cout << "Die wievielte Fibonacci-Zahl wollen sie bestimmen:"
      << endl;
  cin >> fibzahl;
  if (fibzahl < 1)
    cout << "Die eingegebene Zahl muss mindestens 1 betragen!"
        << endl;
  else
    cout << "Das Ergebnis ist: " << fibonacci(fibzahl) << endl;
}
```

5.9 Eindimensionale Arrays (statische Felder)

Nachdem die Schüler nun reichlich Erfahrung mit den einfachen Datentypen in C++ haben, sollten schließlich auch die zusammengesetzten Datentypen gelehrt werden. Die einfachsten davon sind sicherlich die eindimensionalen Arrays. Ob es sinnvoll ist diese bald nach den einfachen Datentypen zu lehren oder wie in dieser Arbeit vorher einige andere Konzepte zu erläutern ist sicherlich je nach Lehrperson und Klasse verschieden. Wichtig ist jedoch, dass die einfachen Datentypen vorher von den Schülern gut beherrscht werden. Sicherlich müssen Konzepte wie die Schleifen vor den eindimensionalen Arrays

gelehrt werden, da es für den Zugriff auf die einzelnen Array-Elemente sehr häufig gebraucht wird.
Ein wichtiges Thema der Arrays ist bereits die Deklaration. Ein Integer-Array wird folgendermaßen deklariert:

```
int zahlen1[3];
int zahlen2[3*3];
int const i = 3;
int zahlen3[i];
...
```

Bei der Einführung wie ein Array deklariert werden kann ist es sehr wichtig, dass der C++ Standard eingehalten wird. Dieser besagt, dass Arrays eine Konstante Länge haben müssen. Es können auch Berechnungen wie `3*3` als Dimension angegeben werden, dessen Ergebnis muss jedoch bereits während der Kompilezeit bekannt sein. Einige Compiler unterstützen auch variable Arraydimensionen, was jedoch nicht dem ISO Standard entspricht.
Beim Thema der Felder ist auch die Funktionsübergabe von Arrays wichtig, was den Schülern keinesfalls vorenthalten werden darf. Weil Arrays mit einer konstanten Größe deklariert werden müssen ist diese stets bekannt, wenn man die Feldgrenze bei einer Funktionsübergabe oder in einer Schleife braucht. Den Schülern kann jedoch auch gezeigt werden, wie dieser Wert mittels `sizeof` bestimmt werden kann.
Wie normale Datentypen auch, können auch Arrays bei der Deklaration oder erst später initialisiert werden. Das Konzept stellt nicht viel neues dar und wird deshalb in relativ kurzer Zeit erklärt sein. Beispiel einer Initialisierung:

```
zahlen[5] = {1, 8, 7, 3, 5};
```

Neu für die Schüler wird sicherlich sein, dass die Array-Nummerierung bei `0` und nicht bei `1` beginnt, was sich jedoch in kurzer Zeit einprägen lässt. Auch der Zugriff auf die Array-Elemente ist schnell erklärbar. Dabei wird ein Element auf die selbe Weise wie bei der Initialisierung verwendet, mit dem Unterschied, dass ein Lesezugriff statt einem Schreibzugriff erfolgt.
Ein geeignetes Einführungsbeispiel ist ein Sortierprogramm, da beim Umgang mit mehreren Zahlen ein Array eingesetzt werden kann. Kennen die Schüler bereits verschiedene Sortieralgorithmen, so kann die Wahl der Sortierart den

Schülern überlassen werden. Anderenfalls kann ein einfacher Algorithmus vorgeben werden und falls es in den Zeitplan passt, andere Algorithmen später gelehrt werden.

Sortierbeispiel mit eindimensionalen Arrays:

```
#include <iostream>
using namespace std;

int main (void)
{
  int const max = 10;
  int feld[max], pos, kleinstes;

  cout << "Geben Sie " << max << " Zahlen ein die sortiert werden:"
       << endl;
  for (int i=0; i<max; i++)
    cin >> feld[i];

  for (int i=0; i<max; i++)
  {
    kleinstes = feld[i];
    pos = i;
    for (int j=i+1; j<max; j++)
      if (feld[j]<kleinstes)
      {
        kleinstes = feld[j];
        pos = j;
      }
    feld[pos] = feld[i];
    feld[i]=kleinstes;
  }

  for (int i=0; i<max-1; i++)
    cout << feld[i] << " ";
  cout << feld[max-1] << endl;
}
```

In dem Sortierbeispiel wird das unsortierte Feld, welches die eingegebenen Elemente beinhaltet von vorne nach hinten durchgegangen wird. Als erstes wird das erste Element des Arrays mit dem kleinsten Element des Arrays vertauscht, wenn es nicht bereits das kleinste ist. Da dieses Element nun sortiert ist, wird es bei den nächsten Sortierungen nicht mehr beachtet. So wird als nächstes das zweite Element mit dem kleinsten Element ab dieser zweiten Position vertauscht. Das geht solange weiter bis alle Elemente sortiert sind. Dieser Algorithmus entspricht praktisch dem *Selectionsort* bzw. dem *Minsort* (so wird

der *Selectionsort* bezeichnet bei dem immer das kleinste Element vertauscht wird).

Da in den C++-Methoden auch C-Strings anzutreffen sind, können diese im Unterricht den Schülern nicht vorenthalten werden. Sie sollten allerdings möglichst vermieden werden und statt dessen die C++-String-Klasse verwendet werden. Diese ist durch das objektorientierte Programmierparadigma nämlich Fehler-resistenter. Dabei ist vor allem interessant wie C-String in Arrays deklariert werden:

```
char cstr[] = "Heute ist ein wunderbarer Tag";
char cstr2[]  = { 'P', 'r', 'i', 'm', 'a', '\0' };
char cstr3[3];
cstr3[0] = 'E';
cstr3[1] = 'i';
cstr3[2] = '\0';
```

und wie sie und deren Länge ausgegeben werden können:

```
cout << "\"" << cstr << "\" hat die Laenge " << strlen(cstr) << endl;
```

Dabei muss vor allem darauf hingewiesen werden, dass C-Strings mit dem Zeichen `'\0'` aufhören müssen und dafür auch ein zusätzliches `char`-Element abspeichern müssen. Wird ein C-String so initialisiert wie die Variable `cstr`, dann wird jenes Zeichen automatisch an die Zeichenkette angehängt. Dies sind wichtige Programmierfeinheiten von C-Strings.

Weitere Konzepte wie das Abschneiden eines C-Strings durch das Einfügen des C-String-Abschlusszeichens `'\0'`, sollten im Thema der Zeiger gelehrt werden, da Zeiger dynamischer und dafür geeigneter sind.

5.10 Mehrdimensionale Arrays

Das Thema der mehrdimensionalen Arrays kann kürzer gefasst werden, da vieles ähnlich ist wie bei den eindimensionalen Arrays und C-Strings bereits bekannt sind. Beispielsweise sind Deklaration von mehrdimensionalen Arrays und Zugriff auf mehrdimensionale Arrays nahezu identisch mit jenen eindimensionaler Felder. Der Unterschied ist, dass mehrdimensionale Arrays eben entsprechend mehrere Dimensionen haben. Die Initialisierung hingegen weist einige Neuigkeiten auf. So sind alle folgenden Initialisierungen gültig:

```
int marray[2][3] = {1, 2, 3, 4, 5, 6};
int marray2[2][3] = {{1, 1, 1}, {2, 2, 2}};
int marray3[][3] = {2, 3, 4, 5, 6, 7};
```

Auch folgende Zeilen sind gültig:

```
char namen[][10] = {"Anton", "Fritz", "Gerda"};
for (int i=0; i<3; i++)
  cout << namen[i] << endl;
```

Bei der Funktionsübergabe gilt es den Schülern jedoch zu zeigen, dass zwar die 1. Dimension, wie bei eindimensionale Arrays, im Funktionskopf der Funktionsdeklaration nicht angegeben werden muss, die weiteren Dimensionen jedoch schon, damit der Programmcode korrekt kompiliert werden kann.

5.11 Zeiger

Dieses Kapitel ist sehr wichtig für den Schulunterricht und darf keinesfalls fehlen. In C++ gibt es zwar auch verschiedene C++-Container-Klassen (`vector`, `list`, ...), welche das dynamische Verwalten von Daten vereinfachen. Jedoch sind Zeiger im C++-Standard noch weit verbreitet und finden sich bereits in der Hauptmethode `main` wieder.

Der für Neulinge eher kompliziertere Teil der Speicherplatzanforderung mittels `malloc` oder ähnlichen Funktionen sollte jedoch weggelassen werden [6]. Dies ist nämlich eine C-Funktion, welche von C++ wegen Kompatibilitätsgründen übernommen wurde. Außerdem gibt es in C++ ersatzweise Klassen wie `vector`, `list`, ..., welche vieles vereinfachen und der Programmsicherheit dienen. Die Alternativen von `malloc` und verwandten C-Methoden sind dynamische Arrays, welche später noch näher beschrieben werden.

5.11.1 Einfache Zeiger

Es ist sicherlich sinnvoll anfangs mit den einfachen Zeigern zu beginnen, also jenen Zeigern, welche automatisch schon den erforderlichen Speicherplatz reservieren. Das kann beispielsweise die Deklaration und Initialisierung eines Zeigers auf einen Integer sein. Weiters ist es auch wichtig, den Schülern den Umgang mit Zeigeradressen zu lehren.

Beispiel von einfachen Zeigern:

```
#include <iostream>
using namespace std;
```

```
int main ()
{
  int v_int1 = 5, v_int2 = 15, * p_int1, * p_int2;

  p_int1 = &v_int1;
  p_int2 = &v_int2;
  *p_int1 = 10;
  *p_int2 = *p_int1;
  p_int1 = p_int2;
  *p_int1 = *p_int1 + 11;

  cout << endl << "Zahl 1 ist " << v_int1 << endl;
  cout << "Zahl 2 ist " << v_int2 << endl;
  cout << endl << "Pointer 1 ist " << p_int1 << endl;
  cout << "Inhalt Pointer 1 ist " << *p_int1 << endl;
}
```

Im oberen Zeiger-Beispiel lernen Schüler das Verhalten von Zeigeradressen und von Zeigerinhalten kennen und wie man Zeiger korrekt ausgibt.
Eine Neuigkeit für die Schüler ist auch die Übergabe von Zeigern. Dieses Thema dürfte jedoch schnell verstanden werden.

5.11.2 Dynamische Speicherverwaltung

Oft möchten man jedoch, wie in Arrays möglich, mehrere Daten abspeichern. Der Vorteil der Zeiger gegenüber Arrays ist nun, dass die Daten nicht statisch sind. Der Speicherplatz kann nämlich wieder freigegeben werden und bei Bedarf wieder neuer reserviert werden. Dabei sollte den Schülern beigebracht werden den besetzten Speicherplatz konsequent wieder freizugeben. Dies ist nämlich in C++ sehr wichtig, weil es keine „garbage collection“ wie in Java gibt, die automatisch nicht mehr verwendeten Speicherplatz wieder freigibt.
Wie bereits im Kapitel 5.11.1 *Einfache Zeiger* beschrieben, sollte der C-Stil mit den Funktionen `malloc`, `calloc`, ... vermieden werden. Dafür gibt es in C++ den Operator `new`. Will man beispielsweise ein Objekt anlegen, so ruft `new` dessen Konstruktor auf. `delete`, das den von `new` angelegten Speicherplatz wieder freigibt, ruft wiederum den Destruktor des betreffenden Objektes auf. Außerdem kann man den `new`-Operator bei Bedarf überladen. Im Gegensatz zu `malloc` ist dieser auch Typ-sicher. Er liefert nicht einen Zeiger auf `void` zurück, wie `malloc`, sondern einen Zeiger auf den Datentyp für welchen der

Speicherplatz reserviert wurde [5]. Weiters kann dieses Konzept den Schülern leichter erklärt werden.

Speicherplatz für eine bestimmte Anzahl von Integern kann folgendermaßen reserviert werden:

```
cin >> anz;
int * zahlen = new int[anz];
```

Wie man sieht muss im Unterschied zu normalen Arrays die Größe von dynamischen Arrays dem Compiler während der Kompilezeit noch nicht bekannt sein. Sie kann auch während der Laufzeit des Programms bestimmt werden.

Eine mit `new` initialisierte Variable kann wie reine Zeiger (wie sie in C verwendet werden) behandelt werden. Zeiger können also auch addiert, subtrahiert, inkrementiert oder dekrementiert werden. Weiters kann als Übung ein String durch das Zeichen `'\0'` in mehrere Teilstrings aufgeteilt werden:

```
char text[] = "Wort1 Wort2";
char * t=text+6, * h=text+5;
*h = '\0';
```

Somit lernen die Schüler auch den Umgang mit C-Strings (welche in C++ auch vorkommen) näher kennen. Folgende Zeile würde für das Beispiel nicht funktionieren, da der String nur lesbar wäre (der Zeiger selbst ist zwar variabel, der Zeigerinhalt jedoch konstant, weil es im C++-Standard so definiert ist):

```
char * text = "Wort1 Wort2";
```

Die Funktionsübergabe geschieht so wie bei den einfachen Zeigern. Die einzige Neuigkeit für die Schüler ist das Zeichen `&` vor Variablen im Funktionskopf. Dieses Zeichen bewirkt, dass nicht nur der Inhalt der Objekte auf die der Zeiger verweist geändert werden kann, sondern dass der Inhalt der Zeigervariablen (die Adresse) selbst geändert werden kann.

```
void erzeuge_dynamisch(int n, double * & v)
{
  v = new double[n];
  v[0] = 2.1;
  v[1] = 2.2;
}
```

`v = new double[n];` bewirkt, dass `v` ein neuer Speicherbereich zugeordnet wird. Dieser neue Speicherbereich bleibt für Operationen außerhalb der Funktion nur bestehen, wenn im Funktionskopf vor dem `v` ein `&` steht.

Die Variablen `v[0],...,v[n]` können jedoch auch ohne dem `&`-Zeichen, für Operationen außerhalb der Funktion, von dieser geändert werden. Zeigervariablen werden nämlich von der Funktion als globale Variablen interpretiert.

Nach dem Aufruf folgender Funktion

```
void erzeuge_dynamisch2(int n, double * v)
{
  v[0] = 3.1;
  v[1] = 3.2;
  v = new double[n];
  v[0] = 4.1;
  v[1] = 4.2;
}
```

hat `v[0]` und `v[1]` den Inhalt `3.1` und `3.2`. Das Verstehen dieser Tatsache ist sehr wichtig, damit das Prinzip der lokalen und globalen Variablen richtig umgesetzt wird.

Die Funktionalität von `delete` wird den Schülern schnell klar sein. Nämlich den von `new` angelegten Speicherplatz wieder freizugeben. Dabei gilt es nur auf eines zu achten. Ob man `delete` oder `delete[]` verwendet. Der erste Befehl wird bei einem von `new` angelegten Typ verwendet während der zweite von einem mit `new[]` angelegten Array verwendet wird. Beispielsweise wird `v` vom vorigen Beispiel folgendermaßen freigegeben:

```
delete[] v;
v = 0;
```

Wichtig ist dabei den Schülern zu vermitteln freigegebene Zeiger immer auf `0` zu setzen. Ist eine Zeigervariable nämlich `0`, so bedeutet das, dass sie keine Daten enthält. Das selbe gilt für neu deklarierte Zeiger, welche nicht mit sinnvollen Daten initialisiert werden. Somit wird sichergestellt, dass das Programm fehlerfrei funktioniert. Es kann dadurch nämlich immer abgefragt werden, ob ein Zeiger Daten enthält oder nicht. Oft werden die Daten mit `delete` nämlich nicht gelöscht sondern nur für neue Speicheranforderungen freigegeben. Dadurch kann auf die Daten noch zugegriffen werden, obwohl sie im Programm eigentlich freigegeben wurden. Wird der Zeiger nach `delete` jedoch auch `0` gesetzt, so stürzt das Programm bei einem illegalen Datenzugriff ab. So kann das fehlerhafte Verhalten des Programms oft viel schneller

korrigiert werden.

Ein Nachteil von `new`/`delete` gegenüber C-Funktionen ist, dass es keinen Ersatz für `realloc` gibt. Diese Funktion kann die Größe des besetzten Speichers eines Zeigers verändern ohne die enthaltenen Daten zu ändern. Mit `new`/`delete` funktioniert das nur, wenn eine neue Variable mit `new` angelegt wird, die Daten in diese Variable kopiert werden und die alte Variable mit `delete` gelöscht wird. Für solche Fälle gibt es in C++ jedoch sehr flexible Klassen wie: `list`, `vector`, ... Sie sollten den Zeigern immer vorgezogen werden, wenn die Lösung auch mit diesen Klassen erzielt werden kann.

5.11.3 Weitere Themen im Zeiger-Kapitel

Das Kapitel der Zeiger ist eines der aufwendigsten. Dieses Kapitel beinhaltet sehr viele Konzepte, je nach verfügbarer Unterrichtszeit wird es unterschiedlich vertieft werden. Die vorher beschriebenen Themen sollten jedoch keinesfalls fehlen. Neben diesen sind auch noch Funktionszeiger und das Schlüsselwort `const` in Verbindung mit Zeigern wichtige Themen.

Funktionszeiger gibt es nämlich nicht in vielen Programmiersprachen. Mit dem Schlüsselwort `const` lassen sich hingegen sehr unterschiedliche Zeiger definieren:

```
char* p1;
char const* p2;
const char* p3;
char* const p4;
char const* const p5;
const char* const p6;
```

Der erste Zeiger `p1` zeigt auf den Datentyp `char`. `p2` und `p3` sind äquivalent; beide zeigen auf ein konstantes `char`. `p4` ist ein konstanter Zeiger auf ein `char`. `p5` und `p6` sind äquivalent; beide sind konstante Zeiger auf ein konstantes `char`.

5.12 Klassen

5.12.1 Einleitung

Das Kapitel der Klassen ist neben das der Zeiger wohl das längste. Gleichzeitig

ist es auch ein sehr wichtiges, das beim Unterrichten von C++ nicht fehlen darf. Klassen, die dem objektorientierten Paradigma entsprechen, wurden vor allem wegen der Programmsicherheit eingeführt. Sie kapseln Daten, wodurch größere Übersichtlichkeit entsteht, während gleichzeitig private Daten geschützt werden. Weiters können während dem Anlegen und Löschen von Klassenobjekten durch Konstruktoren und Destruktoren benutzerdefinierte Operationen ausgeführt werden.

Einführungsbeispiel von Klassen:

```
#include <iostream>
using namespace std;

class Rechnungen
{
    int a, b;
  public:
    void setze_zahlen(int, int);
    int summe();
    int differenz();
};

void Rechnungen::setze_zahlen(int a, int b)
{
  Rechnungen::a = a;
  Rechnungen::b = b;
}

int Rechnungen::summe()
{
  return a+b;
}

int Rechnungen::differenz()
{
  return a-b;
}

int main()
{
  Rechnungen r;
  r.setze_zahlen(23, 15);
  cout << "Summe: " << r.summe() << endl;
  cout << "Differenz: " << r.differenz() << endl;
}
```

5.12.2 Wichtige Einzelheiten

Es ist in C++ auch das Deklarieren von Zeigern auf Objekte möglich. Da beide

Konzepte in C++ sehr wichtig sind muss auch diese Verbindung im Unterrichtsplan vorkommen. Die Schüler sollen dabei auch lernen wie auf Memberfunktionen dieser Objekte zugegriffen wird. Beispielsweise:

```
var = objekt->funktion(param1, param2);
```

Es gibt weitere wichtige Details für den Schulunterricht. Eines ist das Schlüsselwort `this`. Dieses kann in Funktionen eines Objektes verwendet werden. Dabei ist `this` ein Zeiger auf dieses Objekt. Er wird im Beispiel *Einführungsbeispiel von Klassen* verwendet. Dort hat ein Parameter in der Funktion `setze_zahlen()` den selben Namen wie eine private Objektvariable. Will man nun die Objektvariable innerhalb der Funktion ansprechen, so muss vor diese Objektvariable der Klassennamen gefolgt von zwei Doppelpunkten geschrieben werden. Anderenfalls würde die Parametervariable angesprochen werden.

Das Schlüsselwort `static` dürfte den Schülern bereits bekannt sein. In Objekten spielt es eine wichtige Rolle. Dort kann den Schülern gezeigt werden wie eine statische Variable als Zählvariable verwendet werden kann, da es statische Variablen in Objekten nur einmal gibt, egal wie viele Objekte erzeugt werden. Später kann das Wissen der Schüler auf statische Funktionen erweitert werden.

5.12.3 Überladung von Operatoren

Ein wichtiges Thema ist auch die Operatoren-Überladung. Klassen können beispielsweise nicht summiert, multipliziert, ... werden. Da es in C++ jedoch die Operatoren-Überladung gibt können diese Operationen selbst programmiert werden. Dafür muss die Funktion den Namen `operator` gefolgt vom zu überladenen Operator-Zeichen haben. Wird der Plus-Operator überschrieben, dann heißt die Funktion `operator+`.

Beispiel Operatorenüberladung:

```
#include <iostream>
using namespace std;

class Zeit
```

```
{
  public:
    int sec, min, h;
    Zeit() {};
    Zeit(int, int, int);
    Zeit operator+(Zeit);
};

Zeit::Zeit(int h, int min, int sec)
{
  Zeit::h = h;
  Zeit::min = min;
  Zeit::sec = sec;
}

Zeit Zeit::operator+(Zeit z)
{
  Zeit tmp;
  int s_tmp = sec + z.sec;
  tmp.sec = s_tmp % 60;
  int m_tmp = min + z.min + s_tmp / 60;
  tmp.min = m_tmp % 60;
  tmp.h = h + z.h + m_tmp / 60;
  return tmp;
}

int main()
{
  Zeit z1(3, 15, 45);
  Zeit z2(1, 45, 31);
  Zeit z_summe;
  z_summe = z1 + z2;
  cout << "h: " << z_summe.h << "  min: " << z_summe.min << "  sec: "
      << z_summe.sec << endl;
}
```

Das obere Beispiel zeigt wie der Operator „+“ überladen werden kann. Es werden dabei Sekunden, Minuten und Stunden in Objekten abgespeichert und addiert, wobei die Sekunden und Minuten in einem Objekt nicht größer als 59 sein können.

5.12.4 Friends (Freunde)

Dieses Thema entspricht nicht der objektorientierten Programmierung und muss deshalb nicht bzw. nicht ausführlich gelehrt werden. Vor allem, wenn die Unterrichtszeit schon für andere Kapitel verwendet werden muss.

Friends sind Funktionen bzw. Klassen die mit dem Schlüsselwort `friend` als Freunde einer Klasse deklariert werden. Diese sind zwar nicht Teil der Freund-Klasse, können jedoch auf all deren Elementen zugreifen, also auch auf

Elemente die als `private` und `protected` deklariert wurden. Im Gegensatz zu Elementen die als `public` deklariert werden, könnten jene sonst nicht außerhalb einer Klasse angesprochen werden.

Es gibt in C++ `friend`-Funktionen und `friend`-Klassen. Eine `friend`-Funktion kann folgendermaßen deklariert werden:

```
class MeineKlasse {
    int var1, var2;
  public:
    void setze_werte (int, int);
  friend int friend_funktion (int);
};

int friend_funktion (int param)
{
  MeineKlasse mk;
  mk.var1 = param*2;
  ...
}
```

Wie man sieht, ist die Funktion `friend_funktion()` kein Element der Klasse `MeineKlasse`, da vorher nicht das Wort `MeineKlasse::` geschrieben steht. Diese Tatsache wird von den Schülern recht schnell verstanden werden. Allerdings sollte sie explizit erwähnt werden, weil es nicht auf den ersten Blick ersichtlich ist. Weiters müssen Friends nicht im öffentlichen Bereich der Klasse deklariert werden, da sie automatisch als öffentlich gelten. Werden sie jedoch in diesem Bereich deklariert so bietet das mehr Übersichtlichkeit, da die öffentlichen Elemente dadurch nicht getrennt werden.

`friend`-Klassen funktionieren nach dem selben Prinzip wie `friend`-Funktionen. Die Klassen selbst werden normal deklariert. Nur die Klasse, welche eine Freund-Klasse haben will, muss dies dementsprechend vermerken. Das kann so aussehen:

```
class Klasse
{
  ...
  friend class Freund;
};

class Freund
{

};
```

Dabei kann wiederum die Klasse `Freund` auf alle Elemente der Klasse `Klasse` zugreifen, egal ob sie `private`, `protected` oder `public` sind. Den Schülern

gilt es mitzuteilen, dass das nur für eine Richtung gilt. Also kann die Klasse `Klasse` nicht automatisch auf alle Elemente der Klasse `Freund` zugreifen. Natürlich kann realisiert werden, dass beide Klassen untereinander Freunde sind, jedoch muss das dementsprechend programmiert werden.

5.12.5 Vererbung

Ein weiterer wichtiger Bestandteil der Klassen, der für den Schulunterricht wichtig ist, ist die Vererbung. Durch dieses Konzept werden Programme viel einfacher und können übersichtlicher gestaltet werden. Die Vererbung ist ein wichtiger Bestandteil von zahlreichen objektorientierten Programmiersprachen. Vererbung in C++ bedeutet, dass abgeleitete Klassen die Funktionalität von Elternklassen erben. Für eine einfache Einführung in dieses Konzept gibt es zahlreiche Möglichkeiten.

Beispiel von Vererbung:

```
#include <iostream>
using namespace std;

class Werte
{
  protected:
    int wert1, wert2, wert3;
  public:
    void setzeWerte(int, int, int);
};

void Werte::setzeWerte(int wert1, int wert2, int wert3)
{
  Werte::wert1 = wert1;
  Werte::wert2 = wert2;
  Werte::wert3 = wert3;
}

class Formel1: public Werte
{
  public:
    int rechne();
};

int Formel1::rechne()
{
  return wert1 * wert2 / wert3;
}

class Formel2: public Werte
```

```
{
  public:
    int rechne();
};

int Formel2::rechne()
{
  return wert1 * wert3 / wert2;
}

int main()
{
  Formel1 f1;
  Formel2 f2;
  f1.setzeWerte(12, 4, 3);
  f2.setzeWerte(12, 4, 3);
  cout << f1.rechne() << endl;
  cout << f2.rechne() << endl;
}
```

Anhand dieses Beispiel sehen die Schüler, dass die abgeleitete Klassen `Formel1` und `Formel2` die Funktion `setzeWerte()` und die drei Variablen von der Klasse `Werte` geerbt haben.

Das Beispiel kann danach um eine einfache Klasse erweitert werden, welche nur das Ergebnis ausgibt:

```
class Ausgabe
{
  public:
    void schreibe(int);
};

void Ausgabe::schreibe(int ergebnis)
{
  cout << ergebnis << endl;
}
```

Somit wird den Schülern auf einfache Weise bereits die Mehrfachvererbung gezeigt. Der Klassen-Kopf der Klasse `Formel1` würde dann so aussehen:

```
class Formel1: public Werte, public Ausgabe
{
  ...
}
```

Die Ausgabe hat folgende Form:

```
f1.schreibe(f1.rechne());
```

5.12.6 Polymorphie

Dieses Thema enthält mehrere verschiedene Konzepte über das

objektorientierte Programmieren mit Klassen. Dabei sollte im Unterricht besonders viel Wert auf Verständlichkeit gelegt werden, da dieses Thema sehr wichtig ist und auch sehr viele Neuigkeiten für die Schüler enthält. Passend ist es also das Thema den Schülern Schritt für Schritt beizubringen. So verstehen sie auch besser warum in C++ etwas genauso realisiert wurde wie es vorgefunden wird. Am Schluss des Kapitels sollte dann erwähnt werden, welche Vorteile Polymorphie im Allgemeinen hat. Das geht am besten mit einem Beispiel, das ohne Polymorphie in der Form nicht realisierbar wäre.

Im Kapitel der Polymorphie kann den Schülern als erster Schritt ein Beispiel gezeigt werden, wo ein Zeiger vom Typ der Basisklasse auf ein Objekt der abgeleiteten Klasse zeigt:

```
Formel1 f1;
Werte * w1 = &f1;
w1->setzeWerte(12, 4, 3);
cout << f1.rechne() << endl;
```

So sehen sie bereits den ersten Vorteil der Polymorphie. Gleichzeitig werden auch die Grenzen dieser Methode aufgezeigt, da die Methode `rechne()`, welche erst in den abgeleiteten Klassen vorkommt, vom Zeiger nicht angesprochen werden kann.

Der nächste logische Schritt ist die Einführung des Schlüsselwortes `virtual` und das dadurch realisierbare Konzept der Überladung. Passend ist es dabei das bereits begonnene Beispiel zu erweitern. Dabei wird folgende virtuelle Methode in die Klasse `Werte` eingefügt:

```
virtual int rechne();
```

Diese kann dann einen bestimmten Wert returnieren, der später den Schülern beim Löschen des Schlüsselwortes `virtual` den Unterschied im Ergebnis zeigt. Die folgende Methode zeigt den Schülern einen weiteren Vorteil der Polymorphie:

```
void Ausgabe(Werte * w)
{
  w->setzeWerte(12, 4, 3);
  cout << w->rechne() << endl;
}
```

Der Programmcode der sonst äquivalent wäre muss durch das Konzept der Polymorphie nur einmal geschrieben werden. Beispiele dieser Art, welche die Vorteile der Polymorphie Schritt für Schritt zeigen, sind für den Schulunterricht

sehr wichtig, da sie den Schülern die Sinnhaftigkeit dieses Konzepts zeigen. Sie wird vor allem in Beziehung mit dynamischen Objekten oft eingesetzt.
Folgender Code gibt schließlich die zwei verschiedenen Formelergebnisse ohne zusätzlichen Programmieraufwand aus:

```
Formel1 f1;
Formel2 f2;
Ausgabe(&f1);
Ausgabe(&f2);
```

Zu den oberen vier Zeilen äquivalent und oft der geeignetere Stil ist:

```
Werte * f1 = new Formel1();
Werte * f2 = new Formel2();
Ausgabe(f1);
Ausgabe(f2);
```

Das Beispiel ist jedoch noch nicht vollständig. Geeignet ist es, dass die Klasse `Werte` abstrakt deklarieren wird, weshalb nun die Abstraktion wunderbar im Schulunterricht eingeführt werden kann. Die bisherige Methode `rechne()` der Klasse `Werte` liefert nämlich immer `0` zurück, was nicht viel Sinn macht. Sie musste mit dem bisherigen Unterrichtswissen jedoch so deklariert werden, damit die Vorteile der Polymorphie zum tragen kommen.
Wird nun die Abstraktion angewendet so wird die virtuelle Methode `rechne()` folgendermaßen deklariert:

```
virtual int rechne() = 0;
```

Diese Funktion hat also keine Implementation und wird als *pure virtuelle Funktion* bezeichnet. Diesen Funktionen wird immer ein "`= 0`" angehängt. Neben diesem Anhang und dem Schlüsselwort `virtual` haben sie die selbe Form wie Klassenfunktionen, welche innerhalb dieser deklariert werden. Klassen welche eine *pure virtuelle Funktion* besitzen werden als abstrakte Basisklassen oder einfacher als abstrakte Klassen bezeichnet. Von abstrakten Klassen lassen sich keine Objekte definieren, sondern dienen als Basisklasse für spätere Vererbungen. Dabei muss den Schülern auch gezeigt werden, dass die bisherige Funktion mit dem Kopf `void Ausgabe(Werte * w)` weiterhin funktioniert, weil hier kein `Werte`-Objekt angelegt wird. Es handelt sich hier nämlich um einen `Werte`-Zeiger. Zwar ist `Werte` eine abstrakte Klasse, jedoch dürfen davon beliebig viele Zeiger deklariert werden, solange sie nicht ein `Werte`-Objekt instanzieren. Der `Werte`-Zeiger kann auf sämtliche Objekte

zeigen, welche eine Instanz einer von der Klasse `Werte` abgeleiteten Klasse sind.

5.12.7 Konstruktoren/Destruktoren

Ein Einführungsbeispiel für Konstruktoren und Destruktoren kann folgende Form haben:

```
Rechnungen::Rechnungen()
{
  cout << "Konstruktor" << endl;
}

Rechnungen::~Rechnungen()
{
  cout << "Destruktor" << endl;
}
```

Dabei sollte den Schülern auch das Überladen von Konstruktoren gezeigt werden. Es können nämlich mehrere Konstruktoren gleichzeitig erstellt werden die unterschiedliche Typen und Parameteranzahl haben. Dabei wird immer automatisch der richtige Konstruktor aufgerufen, also jener welcher die korrekte Parameteranzahl hat und bei dem gleichzeitig alle Parametertypen mit jenen des Ausdrucks, der den Konstruktor aufruft, übereinstimmen.

Während bei Objekten immer die entsprechenden Konstruktoren aufgerufen werden ist das bei Destruktoren nicht so. Will man, dass auch der Destruktor des instanzierten Objektes aufgerufen wird, so müssen die Destruktoren der Basisklassen als `virtual` deklariert werden. Normalerweise wird jeder Destruktor immer als `virtual` deklariert.

Im Schulunterricht bietet es sich an virtuelle Destruktoren gemeinsam mit der Polymorphie zu lehren, da beide das Schlüsselwort `virtual` verwenden. Wichtig ist es jedoch den Schülern den Unterschied zu zeigen. Während der Destruktor vom definierten Objekt bis hin zu jenem der Basisklasse nacheinander aufgerufen wird, ist das bei virtuellen Funktionen nicht der Fall. Dort wird immer nur eine Funktion aufgerufen, wie bei überladenen Konstruktoren, welche jedoch nicht das Schlüsselwort `virtual` verwenden. Diese Unterschiede müssen genau erklärt werden, damit die Schüler dem Unterricht auch folgen können.

5.13 Typ-Casting

Haben die Schüler die bisherigen Kapitel bereits gelernt, so haben sie bereits genug Grundwissen für die Vertiefung des Casting-Konzepts. Sicherlich kann dies auch früher erfolgen, jedoch sind vorige Kapitel für das Grundwissen von C++ im Allgemeinen wichtiger. Vor allem, weil den Schülern eine folgender zwei Casting-Arten schon während der Einführung in die C++-Typen vermittelt werden sollte:

```
double d1 = 1.1;
double d2 = 2.2;
int i1 = (int) d1;
int i2 = int (d2);
```

In diesem Beispiel werden die Nachkommastellen einfach abgeschnitten, da die Variablen `i1` und `i2` vom Typ `int` sind und deshalb keine Nachkommastellen abspeichern können. Durch das Casten ist sichergestellt, dass der Compiler diese Operation erlaubt. Es können auch beide Casting-Möglichkeiten vermischt werden, indem Typ und der Ausdruck umklammert werden, da diese Klammern nichts am eigentlichen Sinn ändern. Diese zwei Castingarten können auch als C-Casts bezeichnet werden, da sie von dieser Programmiersprache übernommen wurden. Zwar wird in C++ wegen Sicherheitsgründen von C-Casts abgeraten, jedoch sind diese für Programmieranfänger sicherlich verständlicher.

Durch diese C-Casts ist es in C++ erlaubt Zeiger beliebig nach anderen Zeigern zu casten, weswegen Laufzeitfehler auftreten können, obwohl das Programm vorher fehlerfrei kompiliert wurde. Deshalb gibt es in C++ eine weitere Möglichkeit des Castens mit vier verschiedenen Cast-Operationen. Diese sollten den Schülern gezeigt werden, sobald sie fortgeschrittene Programmierkenntnisse in C++ haben, da diese die Programm-Sicherheit erheblich erhöhen. Ab diesem Zeitpunkt sollten die Schüler diese Casts den C-Casts vorziehen. Die vier C++-Casts haben folgende Form:

```
dynamic_cast <neuer_Typ> (Ausdruck)
reinterpret_cast <neuer_Typ> (Ausdruck)
static_cast <neuer_Typ> (Ausdruck)
const_cast <neuer_Typ> (Ausdruck)
```

5.13.1 dynamic_cast

`dynamic_cast` wird für Zeiger und Referenzen auf Objekte verwendet. Er kontrolliert, ob das Objekt ordentlich gecastet wird. Eine praktische Anwendung des `dynamic_cast` wird in folgenden Zeilen gezeigt:

```
class Haupt { };
class Ableitung: public Haupt { };

Haupt h; Haupt* zh;
Ableitung a; Ableitung* za;

zh = dynamic_cast<Haupt*>(&a);
za = dynamic_cast<Ableitung*>(&h);  // ergibt Compiler-Fehlermeldung
```

Dabei ergibt die letzte Zeile eine Compiler-Fehlermeldung, da eine Basisklasse nicht nach einer abgeleiteten Klasse gecastet werden kann, wenn die Basisklasse nicht polymorphisch ist. Folgendes Beispiel wird mit polymorphischen Klassen realisiert:

Dynamischer Cast mit polimorphischen Klassen:

```
#include <iostream>
#include <exception>
using namespace std;

class Haupt { virtual void dummy() {} };
class Ableitung: public Haupt { };

int main()
{
  try
  {
    Haupt * zha = new Ableitung;
    Haupt * zhb = new Haupt;
    Ableitung * za;
    za = dynamic_cast<Ableitung*>(zha);
    if (za==0) cout << "Null Zeiger bei erstem Typ-Cast" << endl;
    za = dynamic_cast<Ableitung*>(zhb);
    if (za==0) cout << "Null Zeiger bei zweitem Typ-Cast" << endl;
  }
  catch (exception& e)
  {
    cout << "Exception: " << e.what();
  }
}
```

In diesem Beispiel wird anfangs der Zeiger `zha` nach den Zeiger `za` gecastet. Die Variable `zha` ist vom Typ `Haupt` und enthält eine Instanz der Klasse

`Ableitung`, welche von der Klasse `Haupt` abgeleitet ist, was legal ist. Da `za` jedoch einen anderen Typ als `zha` hat, muss das Objekt vorher als `Ableitung`-Objekt gecastet werden. Da dieses Objekt eigentlich auch ein `Ableitung`-Objekt ist und die Zielvariable `za` vom Typ `Ableitung` ist, wird die Operation erfolgreich umgesetzt. Die Variable `zhb` ist vom Typ `Haupt` und speichert ein `Haupt`-Objekt ab. Das `Haupt`-Objekt wird zwar nicht in ein `Ableitung`-Objekt umgewandelt, da diese Aktion nicht legal ist, allerdings wird das Programm ordnungsgemäß kompiliert. Da der Cast nicht durchgeführt wird, liefert `dynamic_cast` einen Null-Zeiger zurück. Auf diese weise kann kontrolliert werden, ob der Cast erfolgreich durchgeführt wird. `dynamic_cast` kann Variablen auch nach Referenz-Typen casten. Gelingt dies nicht, so wird eine `bad_cast`-Exception geworfen.

5.13.2 static_cast

Mit `static_cast` können auch Zeiger gecastet werden, jedoch ist dafür der dynamische Cast geeigneter. `static_cast` wird für das Casten von Typen verwendet, welche zur Compilezeit bekannt sind.
Vor allem statt dem statischen Cast werden häufig die von C bekannten Casts verwendet. Es wird jedoch empfohlen `static_cast` zu verwenden, da dieser sicherer ist.

```
int a;
int *b;
a = (int)b;
a = static_cast<int>(b); // Compiler-Fehlermeldung
```

In diesem Beispiel würde die dritte Zeile kompilieren, was jedoch meistens nicht sinnvoll ist. Es wird nämlich die Zeiger-Adresse und nicht der Zeigerinhalt gecastet. Wegen dem statischen Cast kompiliert die vierte Zeile jedoch nicht. So kann dieser Fehler von Programmierern schnell gefunden werden. Soll die Aktion der dritten Zeile wirklich realisiert werden, so kann dafür der `reinterpret_cast` verwendet werden, welcher im folgenden Kapitel näher beschrieben wird.

5.13.3 reinterpret_cast

Der `reinterpret_cast` ist der vielseitigste Cast. Mit ihm können Zeiger-Typen nach beliebig anderen Zeiger-Typen gecastet werden. Dabei ist das Resultat eine einfache Binär-Kopie. Bei diesem Cast-Typ hat der Programmierer besondere Verantwortung für die Programmsicherheit. Mit Hilfe des `reinterpret_cast` lässt sich das Beispiel vom `static_cast`-Kapitel nun realisieren:

```
int a;
int *b;
a = reinterpret_cast<int>(b);
```

Die Variable `a` enthält nach diesen drei Zeilen die Adresse der Variable `b`. Dies ist zwar ziemlich Hardware-nahes programmieren, kann jedoch in der Praxis manchmal gebraucht werden. Für den Schulunterricht ist dieser Cast nicht sehr interessant. Er wird nämlich häufig bei besonderen Einzelfällen gebraucht, wo der Programmierer genau wissen muss, was er tut. Es kann im Unterricht erwähnt werden, dass es ihn gibt, genauer gelehrt werden muss dieser Cast-Typ jedoch nicht.

5.13.4 const_cast

const_cast ist der einfachste Cast. Er wird immer dann gebraucht, wenn die Konstanz einer Variable entfernt oder gesetzt werden muss.

Beispiel für das Entfernen der Konstanz:

```
#include <iostream>
using namespace std;

void print(int * i)
{
  *i = 2;
  cout << *i << endl;
}

int main()
{
  const int i=1;
  const int * pi = &i;
  print(const_cast<int *> (pi));
}
```

Da der Funktion `print()` keine konstante Variable übergeben werden kann, wird diese Konstanz bei der Übergabe entfernt. Jene Variable kann jedoch nur innerhalb der Funktion geändert werden.
`const_cast` wird in der Schule eher selten gebraucht werden. Es reicht also, dass dieser Cast-Typ nur erwähnt wird, solange er in den Übungseinheiten nicht gebraucht wird.

5.14 Exception Handling

Dieses wichtige C++-Thema eignet sich nach dem Zeiger-Kapitel eingeführt zu werden. Es muss jedoch nicht direkt danach gelehrt werden, vor allem wenn den Schülern noch nicht das Konzept der Klassen in C++ bekannt ist. Nach dem Kapitel der Zeiger deshalb, weil ein Zeiger der mit `new` initialisiert wird, eine Exception auslösen kann. Den Schülern kann dabei erklärt werden, wie solche Ausnahmefälle erfolgreich abgefangen werden können. Für das Abfragen von Ausnahmefällen gibt es in C++ zwei Möglichkeiten. Eine ist das Verwenden von `nothrow`:

```
int * i_feld = new (nothrow) int[3];
if (!i_feld)
  exit (1);
```

Durch das Verwenden von `nothrow` wird in `i_feld` die Nulladresse abgespeichert, falls Speicherprobleme auftreten. Dies ist zwar standardisiertes C++, sollte jedoch vermieden werden, da es nicht objektorientierter Programmierung entspricht.
Die 2. Möglichkeit ist das Abfangen von Exceptions mit der `try-catch`-Anweisung:

```
try
{
  int * i_feld = new int[3];
}
catch (bad_alloc& bae)
{
  cerr << bae.what() << endl;
}
```

Diese entspricht dem objektorientierten Programmierstil und sollte von den Schülern auch verwendet werden. Wie man sieht, können dabei auch mehr Informationen als bei der ersten Methode erhalten werden (beispielsweise

durch `bae.what()`). Der Bezeichner `&` nach `bad_alloc` besagt, dass auch die von `bad_alloc` abgeleiteten Exceptions von dieser `try-catch`-Anweisung abgefangen werden. Weil `bad_alloc` von der `exception`-Klasse abgeleitet ist könnte man deshalb auch `exception&` schreiben. Damit alle Exceptions abfangen werden, muss `catch (...)` geschrieben werden.
Am Schluss dieses Kapitels sollten die Schüler auch eigene Exception-Klassen definieren und „werfen“ können, was im folgenden Beispiel realisiert wird:

Exception-Handling-Beispiel:

```
#include <iostream>
#include <exception>
using namespace std;

class meineexception: public exception
{
  public:
    virtual const char* what() const throw()
    {
      return "Meine Exception";
    }
} meineecpn;

int main()
{
  try
  {
    throw meineecpn;
  }
  catch (meineexception me)
  {
    cerr << me.what() << endl;
  }
}
```

Für das Werfen von Exceptions müssen jedoch nicht neue Exception-Klassen definiert werden. Es können auch einfach Zahlen, Zeichenketten, ... als Exception geworfen werden. Beispielsweise: `throw 20;`. Dementsprechend kann der Inhalt von `catch` so aussehen: `catch (int e)`.
Es gibt noch weitere Möglichkeiten, beispielsweise die Verschachtelung von `try-catch`-Anweisungen. Dabei kann ein innerer `catch`-Block die Exception(s) mit dem Aufruf von `throw;` an die äußere `try-catch`-Anweisung

weiterleiten. Diese Möglichkeiten gehören jedoch zum erweiterten Verständnis des Exception-Konzepts und müssen wegen der begrenzten Unterrichtszeit nicht in praktischen Beispielen erklärt werden.

5.15 Templates

Templates sind ein wichtiges Kapitel in C++, da es ein Konzept vieler moderner Programmiersprachen ist und in C nicht vorkommt. Es empfiehlt sich, dieses Kapitel vor der Einführung in Listen, Vektoren und anderen vordefinierten Templates zu unterrichten, damit das Hintergrundwissen von Listen und Vektoren bereits vorhanden ist.

Es gibt in C++ zwei Arten von Templates. Funktions-Templates sind eine Art davon:

```
template <class T>
T bekommeGroesseres (T a, T b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Diese Funktion hat einen Typ `T` der später beim Aufruf dieser implizit oder explizit bestimmt wird. Für jeden passenden Typ returniert diese Funktion den größeren Parameter. Das Schlüsselwort `class` kann auch durch `typename` ersetzt werden, da beide die selbe Bedeutung haben. Aufgerufen wird die Funktion auf folgende Weise:

```
k=bekommeGroesseres<int>(7,1);
```

bzw.

```
k=bekommeGroesseres(7,1);
```

Da der Compiler die Typen selbstständig bestimmen kann, ist auch der zweite Aufruf erlaubt. Es wird in diesem Fall angenommen, dass es sich bei den Zahlen um Integer-Werte handelt, da sie ganzzahlig sind.

Die Klassen-Templates sind die zweite Art der Templates:

```
#include <iostream>
using namespace std;

template <class T>
class Rechteck
{
  private:
```

```
    T a, b;
  public:
    Rechteck(T, T);
    T flaeche();
    T umfang();
};

template <class T>
Rechteck<T>::Rechteck(T a, T b)
{
  Rechteck::a = a;
  Rechteck::b = b;
}

template <class T>
T Rechteck<T>::flaeche()
{
  return a*b;
}

template <class T>
T Rechteck<T>::umfang()
{
  return 2*a+2*b;
}

int main()
{
  Rechteck<int> rechteck(11, 5);
  cout << rechteck.flaeche() << endl;
  cout << rechteck.umfang() << endl;
  Rechteck<double> rechteck2(11.4, 5.4);
  cout << rechteck2.flaeche() << endl;
  cout << rechteck2.umfang() << endl;
}
```

In diesem Beispiel lernen die Schüler wiederum mehrere Neuheiten kennen. Einerseits die Definition von Klassen-Templates und andererseits wie dessen Funktionen außerhalb dieser implementiert werden.

Es gibt bei Templates auch Spezialisierungen. Beispielsweise kann ein Template definiert werden, welches beispielsweise für alle Typen außer dem `char`-Typ gleich funktioniert. Da es sich um keine Überladung handelt muss der Code, der im Template und der Template-Spezialisierung des `char`-Typs gleich ist, doppelt programmiert werden. Template-Spezialisierungen gehören jedoch nicht zum Grundwissen von C++, weshalb sie im Schulunterricht nicht von besonderer Bedeutung sind.

Templates können auch „normale“ Parametertypen haben.

```
template <class T, int N>
class Rechteck
{
```

```
  ...
};
```

Die Integervariable kann dabei wie normale Variablen verwendet werden. Weiters können die Template-Parameter Standardwerte haben:

```
template <class T=char, int N=10>
class Rechteck
{
  ...
};
```

Definiert wird das Template `Rechteck` dann folgendermaßen:

```
Rechteck<> rechteck;
```

Eine wichtige theoretische Tatsache für die Schüler ist dabei, dass Templates nicht normale Funktionen bzw. Klassen sind. Sie werden erst kompiliert, wenn sie im Programmcode auch wirklich implementiert wurden. Dabei werden die Typparameter durch die entsprechenden Typen ersetzt. Das Resultat davon ist, dass Deklaration und Implementation von Templates nicht auf verschiedene Dateien aufgeteilt werden dürfen, wie es sonst bei größeren Projekten üblich ist. Natürlich dürfen Template-Variablen selbst in anderen Dateien deklariert, initialisiert und benützt werden.

5.16 Dateien verwalten

In C++ können natürlich auch Dateien verwaltet werden. Sie können für Lese- und Schreiboperationen geöffnet werden. Dieses Thema ist für die Schüler sicherlich interessant, da sie dabei lernen wie Dateiinhalte erstellt, verändert und gelöscht werden können. Es ist deshalb so interessant, weil die Resultate auch für spätere Programmaufrufe erhalten bleiben. So können nun Spielzwischenstände und andere zwischengespeicherte Daten wiederverwendet werden.

Für die Dateiverwaltung in C++ gibt es drei Datenströme, welche die Schüler kennen lernen müssen:

- `ofstream`: Datenstrom für das Schreiben von Dateidaten
- `ifstream`: Datenstrom für das Lesen von Dateidaten
- `fstream`: Datenstrom für Lesen und Schreiben von Dateidaten

Daten können auf zwei Arten geschrieben werden, binär oder textlich. Für eine anschauliche Einführung in dieses Kapitel eignen sich im Unterricht vor allem

Textdateien. In Textdateien kann auf die selbe Art geschrieben werden wie in die Standardausgabe, welche den Datenstrom `cout` verwendet.

```
ofstream datei;
datei.open ("datei.txt");
datei << "Das ist der Dateiinhalt" << endl;
datei.close();
```

Gelesen kann eine Datei wie folgt werden:

```
string zeile;
ifstream datei ("datei.txt");
if (datei.is_open())
{
  while (!datei.eof())
  {
    getline (datei,zeile);
    cout << zeile << endl;
  }
  datei.close();
}
else
  cout << "Datei konnte nicht geoeffnet werden";
```

Im letzten Beispiel wird die Datei zeilenweise eingelesen, was beim Lesen von Textdateien üblich ist. Existiert jedoch eine Datei die beispielsweise mehrere durch Zwischenraumzeichen (Leerzeichen, Tabulator-Zeichen, ...) getrennte ganze Zahlen beinhaltet, so können jene auch mit dem >>-Operator eingelesen werden. Angenommen `i` ist ein Integer, kann dies folgendermaßen realisiert werden:

```
datei >> i;
```

Wichtig ist auch das Vermitteln des konsequenten Schließen von Dateien. So ist sichergestellt, dass Dateien fehlerfrei wiederverwendbar sind.

Mit `datei.is_open()` wird geprüft, ob die Datei erfolgreich geöffnet werden konnte. Dies ist vor allem beim Lesen von Dateien wichtig, da die Datei nicht immer existieren muss. Mit `datei.eof()` wird überprüft, ob das Dateiende erreicht wurde. Durch `datei.fail()` kann fehlerhaftes Verhalten abgefangen werden. Dabei ist den Schülern zu zeigen, dass diese Statusflags mit `datei.clear()` zurückgesetzt werden können. Das ist wichtig, wenn eine Datei wieder geöffnet werden muss. Konnte eine Datei nämlich mit bestimmten Optionen nicht geöffnet werden, so kann mit `clear()` das `fail()`-Flag zurückgesetzt und die Datei mit anderen Optionen geöffnet werden.

Mögliche Optionen sind:

- `ios::in`: Datei lesend öffnen
- `ios::out`: Datei schreibend öffnen
- `ios::binary`: Datei binär öffnen
- `ios::ate`: Position des Lese- und Schreibzeigers der Datei ist am Dateiende
- `ios::app`: Geschriebene Daten werden an das Dateiende angehängt. Damit dieses Flag benutzt werden kann darf die Datei ausschließlich für Schreiboperationen geöffnet sein.
- `ios::trunc`: Wenn die Datei auf welche geschrieben wird schon existiert, dann wird der vorige Dateiinhalt gelöscht und ersetzt.

Viele Optionen können auch kombiniert werden:

```
ofstream datei;
datei.open ("datei.bin", ios::out | ios::app | ios::binary);
```

Dabei sollten wichtige Kombinationen in Übungsbeispielen vorkommen, damit die Schüler verschiedene mögliche Kombinationen kennen lernen. Die Flags `ios::ate` und `ios::app` haben etwa auf den ersten Blick die selbe Aufgabe. Mit `ios::app`, das wie bereits beschrieben nicht mit Lesezugriffen auf Dateien aufgerufen werden darf, werden Daten immer ans Dateiende angehängt. `ios::ate` ist jedoch nur sinnvoll in Kombination mit Datei-Leserechten. In diesem Fall werden eventuell einzufügende Daten auch ans Dateiende angehängt. Der Unterschied beim Verwenden von `ios::ate` ist, dass die Positionen des Lese- und Schreibzeigers geändert werden können. Dabei bestimmt der Lesezeiger die Position ab der die Daten gelesen werden, der Schreibzeiger ab welcher die Daten geschrieben werden. Dabei ist es wichtig den Schülern zu erläutern, dass beim Schreiben die Daten nicht nur eingefügt werden. Befindet sich der Schreibzeiger nämlich nicht am Dateiende, werden ab jener Position vorhandene Daten überschrieben. Die Position des Lesezeigers wird mit `tellg()`, jene des Schreibzeigers mit `tellp()` erhalten. Mit den entsprechenden seek-Zeigern `seekg()` und `seekp()` können jene Positionen manuell geändert werden.

Wie bereits ersichtlich wurde, können Daten auch binär gelesen und geschrieben werden. Der Unterschied zwischen Binärdateien und Textdateien

ist der, dass Binärdateien auch nicht-alphabetische Zeichen abspeichern können. Es kann also jeder beliebige Bytewert vorkommen. Sie werden In C++ wie Textdateien deklariert, jedoch mit der zusätzlichen Option `ios::binary`. Die Lese- und Schreibfunktionen sind wie folgt definiert:

```
read (daten, groesse);
write (daten, groesse);
```

Dabei sind die Daten vom Typ `char*` und `groesse` vom Typ `pos_type`. Auf diesen Typ kann im Datenstrom `ifstream` mittels `ifstream::pos_type` zugegriffen werden. Jedoch kann dieser Typ auch mittels anderer Datenströme erhalten werden.

Dateienverwaltungs-Beispiel:

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char wahl;
  string const dateiname = "Kerze.dat";
  int brenndauer;
  string beschreibung, farbe;
  fstream datei;
  do
  {
    cout << "Wahlen Sie die Kerzen-Operation:" << endl;
    cout << "[E]ingeben, [L]iste loeschen, a[U]flisten, [A]bbrechen"
        << endl;
    cin >> wahl;

    switch(wahl)
    {
      case 'E':
        cout << "Geben Sie den Kerzennamen oder eine passende
            Beschreibung ein:"
          << endl;
        cin >> beschreibung;
        cout << "Geben Sie die Farbe der Kerze ein:" << endl;
        cin >> farbe;
        cout << "Geben Sie die Brenndauer der Kerze in Minuten ein:"
            << endl;
        cin >> brenndauer;
        datei.open(dateiname.c_str(),ios::out|ios::app);
        datei.seekg (0, ios::end);
        cout << endl;
        if ((long)datei.tellg() == 0)
          datei << beschreibung << endl << farbe << endl
```

```
            << brenndauer;
        else
          datei << endl << beschreibung << endl << farbe << endl
              << brenndauer;
        datei.close();
        break;
      case 'U':
        datei.open(dateiname.c_str(),ios::in);
        if (datei.is_open())
        {
          string zeile;
          while (!datei.eof())
          {
            getline(datei,zeile);
            cout << "Beschreibung: " << zeile << endl;
            getline(datei,zeile);
            cout << "Farbe: " << zeile << endl;
            getline(datei,zeile);
            cout << "Brenndauer: " << zeile << endl << endl;
          }
          datei.clear();
          datei.close();
        }
        else
        {
          datei.clear();
          cout << "Liste ist leer\n" << endl;
        }
        break;
      case 'L':
        remove(dateiname.c_str());
        cout << endl;
        break;
    }
  } while (wahl != 'A');
}
```

In diesem Beispiel können Daten von Kerzen an das Dateiende angehängt werden und die Daten aufgelistet bzw. gelöscht werden. Weiters kann das Programm auch abgebrochen werden und die Daten beim nächsten Programmstart weiterverarbeitet werden.

5.17 Namensräume (Namespaces)

Das Konzept der Namensräume in C++ ist in der Praxis sehr nützlich. Vor allem für größere Projekte. Da es für das Erlernen der bisherigen Konzepte nicht von großer Bedeutung ist, kann auf dieses Thema im Schulunterricht verzichtet werden, wenn die Unterrichtszeit bereits für die anderen Themen gebraucht wird. Ist jedoch noch freie Unterrichtszeit vorhanden sollten Namensräume

natürlich gelehrt werden. Vor allem weil schon seit dem „Hello World“-Programm der Namensraum `std` verwendet wird. Ein Namensraum kann mehrere Klassen, Objekte und Funktionen beinhalten. Eine einfache Deklaration ist:

```
namespace Namensraum
{
  const int a=1, b=2;
}
```

Verwendet können Namensrauminhalte dann folgendermaßen werden:

```
cout << Namensraum::a << endl;
cout << Namensraum::b << endl;
```

Namensräume erlauben es nun, dass mehrere Objekte den selben Namen haben können, solange sie sich nicht im selben Namensraum befinden.

Namensraumbeispiel:

```
#include <iostream>
using namespace std;

namespace n_eins
{
  int var = 5;
}

namespace n_zwei
{
  double var = 3.1416;
}

int main () {
  const char var = 'E';
  cout << n_eins::var << endl;
  cout << n_zwei::var << endl;
  cout << var << endl;
}
```

Wie in diesem Beispiel ersichtlich, wird die Variable `var` einmal in der Hauptmethode selbst und zweimal in verschiedenen Namensräumen deklariert. Je nachdem Zugriffsart, wird die entsprechende Variable verwendet. In einem Namensraum können auch weitere Namensräume deklariert werden.
Mit `using namespace <Namensraum>;` kann ein bereits deklarierter Namensraum verwendet werden. Dadurch muss vor der Variable nicht mehr der Namensraum stehen. Dies dürfte den Schülern schon wegen dem in

zahlreichen Programmen verwendeten `using namespace std;` bekannt sein. Ohne diese Anweisung müsste beispielsweise statt

```
cout << var << endl;
```

die Zeile

```
std::cout << var << std::endl;
```

geschrieben werden. Da der `std`-Namensraum häufig gebraucht wird, ist es angebracht diesen einzubinden damit weniger Schreibarbeit beansprucht werden muss. Namensräume werden nur dort eingebunden, wo die Einbindung sichtbar ist. Werden Namensräume beispielsweise in einem Block eingebunden, gilt die Einbindung nur für diesen Block. Werden mehrere Namensräume eingebunden, so muss darauf geachtet werden, dass deren Inhalt eindeutig ist. Es kann nämlich vorkommen das zwei Namensräume jeweils eine Variable mit den selben Variablennamen enthalten. In diesem Fall können zwar beide Namensräume normal eingebunden werden, die Variable muss jedoch mit dem entsprechenden Namensraumnamen aufgerufen werden, da der Compiler sonst eine Fehlermeldung liefert.

Werden im Namensraumbeispiel die zwei Namensräume `n_eins` und `n_zwei` eingebunden werden, so ist die Zeile

```
cout << var << endl;
```

immer noch gültig. In diesem Fall wird einfach die in der Hauptmethode `main` deklarierte Variable `var` in die Standardausgabe geschrieben. Die beiden Variablen `var` der Namensräume `n_eins` und `n_zwei` müssen mit dem entsprechenden Namensraumnamen aufgerufen werden. In diesem Beispiel würde die Einbindung der beiden Namensräume `n_eins` und `n_zwei` also keinen Vorteil bringen.

Programme mit Namensräumen haben mehr Flexibilität, weil sie schnell anpassbar sind. In C gab es beispielsweise lange Zeit keinen Boolean-Typ. So wurde dieser oft von Programmieren manuell erzeugt. In C++ könnten Typen, die im Standard noch nicht definiert sind, in selbst definierten Namensräumen erstellt werden. Falls ein Typ in einem neuen C++-Standard tatsächlich eingeführt wird und den selben Namen hat, kann der Programmcode ohne große Änderungen immer noch weiter benützt werden.

Namensräume sind auch für externe Bibliotheken sehr hilfreich. Werden nämlich mehrere eingebunden, so kommt es zu keinen Namensüberschneidungen.

5.18 Makefiles

Makefiles sind vor allem hinsichtlich größerer Projekte wichtig, also wenn mehrere Programmdateien verwaltet werden müssen. Dieses Konzept sollte den Schülern beigebracht werden, bevor sie größere Projekte alleine oder im Team programmieren müssen.
Makefiles sind ein Kapitel für sich, es kann darüber sehr viel gelehrt werden. Für den Schulunterricht ist jedoch nur ein kleiner Teil der Möglichkeiten von Makefiles sehr wichtig. Diese werden hier näher beschrieben. Es gibt in der Praxis mehrere verschiedene Umsetzungen von Makefiles. In diesem Kapitel wird jene von GNU näher beschrieben, da es sehr weit verbreitet ist. *GNU Make* ist für jeden frei verfügbar und wurde auf mehrere Betriebssysteme portiert. Das ist für die Schule vorteilhaft, da die Software dadurch nicht an ein bestimmtes Betriebssystem gebunden ist.
Aufgerufen wird ein Makefile im Allgemeinen mit `make`. Ohne Parameterangabe wird geschaut ob ein Makefile namens `Makefile` oder `makefile` vorhanden ist, welches anschließend interpretiert wird. Hat es einen anderen Namen muss es nach der `make`-Option `-f` angegeben werden. Der Inhalt von Makefiles hat allgemein die folgende Form:

```
Ziel: Abhängigkeiten
[Tabulator] Systemkommando
```

Das Wichtige dabei ist, dass vor dem Systemkommando ein Tabulator steht. Wird nach dem `make`-Befehl kein Ziel angegeben so wird das erste Ziel im Makefile angesprungen. Anderenfalls wird das angegebene Ziel angesprungen. Das nächste wichtige Konzept von Makefiles für den Schulunterricht ist das Angeben mehrerer Ziele, welche meist voneinander abhängen. Diese Möglichkeit hat den Vorteil, dass wenn eine Datei verändert wurde, nur die davon abhängigen Dateien neu compiliert werden müssen. Je mehr Dateien in einem Projekt vorhanden sind, umso schneller ist diese Variante.

Makefile-Beispiel:

```
all: prog

prog: main.o auto.o
      g++ main.o auto.o -o auto

main.o: main.cpp
      g++ -Wall -ansi -pedantic -c main.cpp

auto.o: auto.cpp
      g++ -Wall -ansi -pedantic -c auto.cpp

clean:
      rm -f auto *.o
```

Wie man am oberen Beispiel sieht hängt, kein Ziel vom Ziel `clean` ab. Dieses muss also mit dem Befehl `make clean` aufgerufen werden.

In Makefiles können auch Kommentare geschrieben werden. Diese werden durch das Zeichen `#` am Zeilenanfang gekennzeichnet. Eine weitere Möglichkeit ist das Erstellten von Variablen. Dieses nützliche Konzept ist das letzte, welches bei der Einführung in Makefiles noch dabei sein muss. Jede andere Möglichkeit von Makefiles geht über die Einführung in Makefiles hinaus und ist wohl nur bei reichlich vorhandener Unterrichtszeit empfohlen. Variablen haben als Inhalt oft den Compilerbefehl und die Compileroptionen.

```
CC=g++
CFLAGS=-Wall -ansi -pedantic -c
```

Diese Zeilen definieren die Variablen `CC` und `CFLAGS` und initialisiert sie mit dem Compilerbefehl `g++` bzw. mit den Compileroptionen `-Wall -ansi -pedantic -c`.

Das dritte Ziel im oberen Makefile-Beispiel würde durch diese Variablen folgende Form haben:

```
main.o: main.cpp
      $(CC) $(CFLAGS) main.cpp
```

Die Variablen werden also aufgerufen indem sie von runden Klammern umgeben und mit einem `$`-Zeichen eingeleitet werden.

5.19 Programmieren im Team

Wie bei den meisten Programmiersprachen ist es auch in C++ möglich den

Quelltext auf mehrere Dateien aufzuteilen. Das ist vor allem bei großen Projekten wichtig, wo mehrere Programmierer an einem Projekt beteiligt sind. Ein Programm wird dadurch viel übersichtlicher und deshalb auch sicherer. Mögliche Änderungen können schneller bewerkstelligt werden. So bekommen die Schüler anhand einem einfacheren Beispiel einen ersten Einblick in die Programmierung im Team.

Dieses Thema spielt auch im Informatikunterricht eine wesentliche Rolle. Dabei wird die Deklaration von der Definition getrennt. Während die Deklaration von Klassen in der Header-Datei stattfindet (hat meistens die Dateierweiterung `.h`) findet die Definition in der Quelltext-Datei statt, welche meistens die Erweiterung `.cpp` hat. Werden in einer Quellcodedatei benutzerdefinierte Header-Dateien eingebunden, so erfolgt dies in folgender Form:

```
#include "ort_der_Header_Datei.h"
```

Dabei ist ersichtlich, dass bei benutzerdefinierten Header-Dateien Anführungszeichen statt den spitzen Klammern verwendet werden. Der Ort der Header-Datei sollte dabei nicht ein kompletter Pfad sein damit das Programm auf verschiedenen Rechnern bzw. Betriebssystemen kompiliert werden kann. Deshalb gibt es die Möglichkeit relative Pfade anzugeben. Je nach dem wo sich die Dateien befinden, werden sie beim Kompilieren entweder automatisch gefunden oder müssen über einen Parameter eingebunden werden. Beispielsweise:

```
g++ ... -I/usr/local/include
```

Als nächstes sollte den Schülern auch klar gemacht werden, dass Header-Dateien mehrmals eingebunden werden können, was zu vermeiden ist. Dieses mehrfache einbinden wird mit entsprechenden Präprozessor-Befehlen verhindert. Deshalb hat eine Header-Datei im allgemeinen die Form:

```
#ifndef HEADERDATEINAME_H
#define HEADERDATEINAME_H

Deklaration...

#endif
```

Wurde ein bestimmter Header-Dateiname also noch nicht definiert, wird er dies und der Header-Dateiinhalt deklariert. Anderenfalls wird eine doppelte Deklaration vermieden.

Beispielprogramm für das Programmieren im Team:

Im folgenden Programm müssen Schüler eine Autoverwaltung programmieren. Dabei werden Gruppen von 2 bis 3 Schülern gebildet wobei die `Auto`-Klasse und `MeineListe`-Klasse von verschieden Schülern programmiert werden müssen. Bei drei Schülern programmiert ein weiter Schüler alleine das Hauptprogramm. Geeigneter sind jedoch 2er-Gruppen, da die `Auto`-Klasse um einiges kürzer ist als die `MeineListe`-Klasse, was durch das Hauptprogramm kompensiert werden kann. Bei 3er-Gruppen kann eventuell jener Schüler, welcher die `Auto`-Klasse programmiert auch das Makefile erstellen.

Für das Projekt gibt es mehrere Vorgaben:

- Autos müssen mit dem Operator == vergleichbar sein
- Die Klasse `MeineListe` muss Autos:
  - ✔ finden
  - ✔ einfügen
  - ✔ löschen
  - ✔ und auflisten können
  - ✔ Autos in einer manuell erstellten einfach verketteten Liste abspeichern
- Im Hauptprogramm müssen alle `MeineListe`-Operationen beliebig oft getätigt werden können, bis das Programm bewusst vom Benutzer abgebrochen wird.

Bevor das Projekt begonnen wird, müssen folgende Kapitel bereits unterrichtet worden sein:

- Klassen mit dem Konzept der Polymorphie
- Operatoren-Überladung
- Datenstrukturen
- Makefile

Natürlich müssen auch jene Konzepte bereits unterrichtet worden sein, welche zum Grundwissen von C++ gehören, wie beispielsweise Operatoren, Typen, Funktionen, usw.

auto.h:

```
#ifndef AUTO_H
#define AUTO_H

#include <string>

using namespace std;

class Auto
{
  private:
    string marke, modell, besitzer;
  public:
    Auto(string, string, string);
    Auto();
    bool operator==(Auto);
    string bekomme_marke();
    string bekomme_modell();
    string bekomme_besitzer();
    Auto * n_a;
};

#endif
```

auto.cpp:

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

using namespace std;

Auto::Auto(string marke, string modell, string besitzer)
{
  Auto::marke = marke;
  Auto::modell = modell;
  Auto::besitzer = besitzer;
}

Auto::Auto()
{
  Auto::marke = "";
  Auto::modell = "";
  Auto::besitzer = -1;
}

bool Auto::operator==(Auto a2)
{
  if ((this->marke == a2.marke) && (this->modell == a2.modell)
    && (this->besitzer == a2.besitzer))
    return true;
  else
    return false;
}

string Auto::bekomme_marke()
{
  return marke;
```

```
}

string Auto::bekomme_modell()
{
  return modell;
}

string Auto::bekomme_besitzer()
{
  return besitzer;
}
```

autoliste.h:

```
#ifndef AUTOLISTE_H
#define AUTOLISTE_H

#include "auto.h"

struct AutoListe
{
  Auto * liste;
  int anz;
};

#endif
```

meineliste.h:

```
#ifndef MEINELISTE_H
#define MEINELISTE_H

#include "autoliste.h"

class MeineListe
{
  private:
    Auto * autos;
    int anz;
  public:
    MeineListe();
    int finden(Auto);
    void einfuegen(Auto);
    void loeschen(Auto);
    AutoListe * bekomme_liste();
};

#endif
```

meineliste.cpp:

```
#include "auto.h"
#include "meineliste.h"

MeineListe::MeineListe()
{
  autos = 0;
```

```
  anz = 0;
}

int MeineListe::finden(Auto f_a)
{
  Auto * h_a = autos;
  int pos = 0;
  while (h_a != 0)
  {
    pos++;
    if (*h_a == f_a)
      return pos;
    h_a = h_a->n_a;
  }
  return 0;
}

void MeineListe::einfuegen(Auto e_a)
{
  if (finden(e_a) != 0)
    throw 1; // Element bereits vorhanden
  Auto * h_a = new Auto();
  *h_a = e_a;
  if (autos == 0)
  {
    autos = h_a;
    autos->n_a = 0;
    h_a = 0;
    anz++;
  }
  else
  {
    h_a->n_a = autos;
    autos = h_a;
    h_a = 0;
    anz++;
  }
}

void MeineListe::loeschen(Auto l_a)
{
  if (autos == 0)
    throw 2; // Lösch-Element nicht gefunden
  else
  {
    Auto * h_a = autos;
    if (*autos == l_a)
    {
      autos = autos->n_a;
      h_a->n_a = 0;
      delete h_a;
      anz--;
    }
    else
    {
      while ((h_a->n_a) != 0)
      {
        if (*(h_a->n_a) == l_a)
        {
```

```
          Auto * h_l_a = h_a->n_a;
          h_a->n_a = h_a->n_a->n_a;
          h_l_a->n_a = 0;
          delete h_l_a;
          anz--;
          return;
        }
        else
          h_a = h_a->n_a;
      }
      throw 2; // Lösch-Element nicht gefunden
    }
  }
}

AutoListe * MeineListe::bekomme_liste()
{
  Auto * a = autos, * h_autos;
  AutoListe * ret = new AutoListe;
  int a_pos = 0;
  if (anz < 1)
  {
    ret->liste = 0;
    ret->anz = anz;
    return ret;
  }
  else
  {
    a = new Auto[anz];
    a[a_pos] = *autos;
    a[a_pos].n_a = 0;
    h_autos = autos;
    while (h_autos->n_a != 0)
    {
      a[++a_pos] = *(h_autos->n_a);
      h_autos = h_autos->n_a;
      a[a_pos].n_a = 0;
    }
  }
  ret->liste = a;
  ret->anz = anz;
  return ret;
}
```

main.cpp:

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

using namespace std;

Auto eingabe()
{
  string marke, modell;
  string besitzer;
  cout << "Automarke: ";
  cin >> marke;
```

```
  cout << "Automodell: ";
  cin >> modell;
  cout << "Maximale Geschwindigkeit: ";
  cin >> besitzer;
  cout << endl;
  return Auto(marke, modell, besitzer);
}

int main()
{
  MeineListe * meine_liste = new MeineListe();
  char wahl;
  cout << endl;
  do
  {
    cout << "Wollen Sie Autos:" << endl;
    cout << "[E]infuegen, [F]inden, [L]oeschen, a[U]flisten oder die
        Aktion [A]bbrechen ..." << endl;

    cin >> wahl;
    try
    {
      int pos;
      AutoListe * auto_liste;
      switch (wahl)
      {
        case 'E':
          meine_liste->einfuegen(eingabe());
          cout << "Element eingefügt!" << endl;
          break;
        case 'F':
          pos = meine_liste->finden(eingabe());
          if (pos < 1)
            cout << "Element wurde nicht gefunden!" << endl;
          else
            cout << "Element an Position " << pos << " gefunden!"
                << endl;
          break;
        case 'L':
          meine_liste->loeschen(eingabe());
          cout << "Element gelöscht!" << endl;
          break;
        case 'U':
          auto_liste = meine_liste->bekomme_liste();
          if (auto_liste == 0)
          {
            cout << endl << "Liste ist leer!" << endl;
            break;
          }
          if (auto_liste->liste != 0)
          {
            for (int i=0; i<auto_liste->anz; i++)
            {
              cout << endl << "*** Auto Pos. " << (i+1) << " ***"
                  << endl;
              cout << "Marke: "
                  << auto_liste->liste[i].bekomme_marke() << endl;
              cout << "Modell: "
                  << auto_liste->liste[i].bekomme_modell() << endl;
```

```
              cout << "Besitzer: "
                  << auto_liste->liste[i].bekomme_besitzer() << endl;
            }
            delete [] (auto_liste->liste);
            auto_liste->liste = 0;
          }
          else
            cout << endl << "Liste ist leer!" << endl;
          delete auto_liste;
          auto_liste = 0;
      }
    }
    catch (int i)
    {
      switch (i)
      {
        // eingefügtes Element existiert bereits
        case 1:
          cout << "Element bereits vorhanden!" << endl;
          break;
        // Lösch-Element nicht gefunden
        case 2:
          cout << "Löschelement nicht gefunden!" << endl;
          break;
        default:
          cout << "Throw-ID: " << i << endl;
      }
    }
    cout << endl;
  } while (wahl != 'A');
}
```

makefile:

```
CC=g++
CFLAGS=-Wall -ansi -pedantic -c

all: auto

auto: main.o auto.o meineliste.o
      $(CC) -o auto main.o auto.o meineliste.o

main.o: main.cpp meineliste.h auto.h autoliste.h
      $(CC) $(CFLAGS) main.cpp

auto.o: auto.cpp auto.h
      $(CC) $(CFLAGS) auto.cpp

meineliste.o: meineliste.cpp meineliste.h auto.h autoliste.h
      $(CC) $(CFLAGS) meineliste.cpp

clean:
      rm -f auto *.o
```

In C++ gibt es zwar bereits Listen mit denen Daten einfach abgespeichert und verwaltet werden können. Trotzdem ist das manuelle Erstellen von

Datenstrukturen im Schulunterricht sehr angebracht, damit die Schüler verstehen wie jene Daten abgespeichert werden können. Weiters lernen sie die Vorteile einer bestimmten Datenstruktur anschaulich kennen, womit sie sich später in einem Programm leichter für eine passende Datenstruktur entscheiden können.

5.20 Compiler

Für C++ sind mehrere Compiler verfügbar. Für die Kommandozeilen-Programme in dieser Magisterarbeit wird der GCC Compiler von GNU verwendet. Dieser ist frei im Internet verfügbar und ist Plattform-unabhängig. Für Windows ist dabei MinGW erhältlich, was einer Windows-Portierung des GCC-Compilers entspricht. Jenes ist eine Open-Source-Entwicklungswerkzeug, welches es erlaubt Programme zu schreiben die unter Windows lauffähig sind, ohne Software dritter zu installieren. Hält man sich an den C++-Standard der ISO, so sind Programme, welche in Linux mit dem GCC kompiliert werden auch in Windows mit dem MinGW-Entwicklungswerkzeug kompilierbar. Der umgekehrte Fall gilt natürlich auch.
MinGW kann einfach auf der Internetseite http://www.mingw.org heruntergeladen werden. Dabei können entweder die einzelnen Pakete heruntergeladen und manuell entpackt werden oder einfach die binäre Installationsdatei heruntergeladen werden. Bei dieser Installationsart können die entsprechenden Komponenten gewählt werden, welche je nach Option nur heruntergeladen oder auch installiert werden.

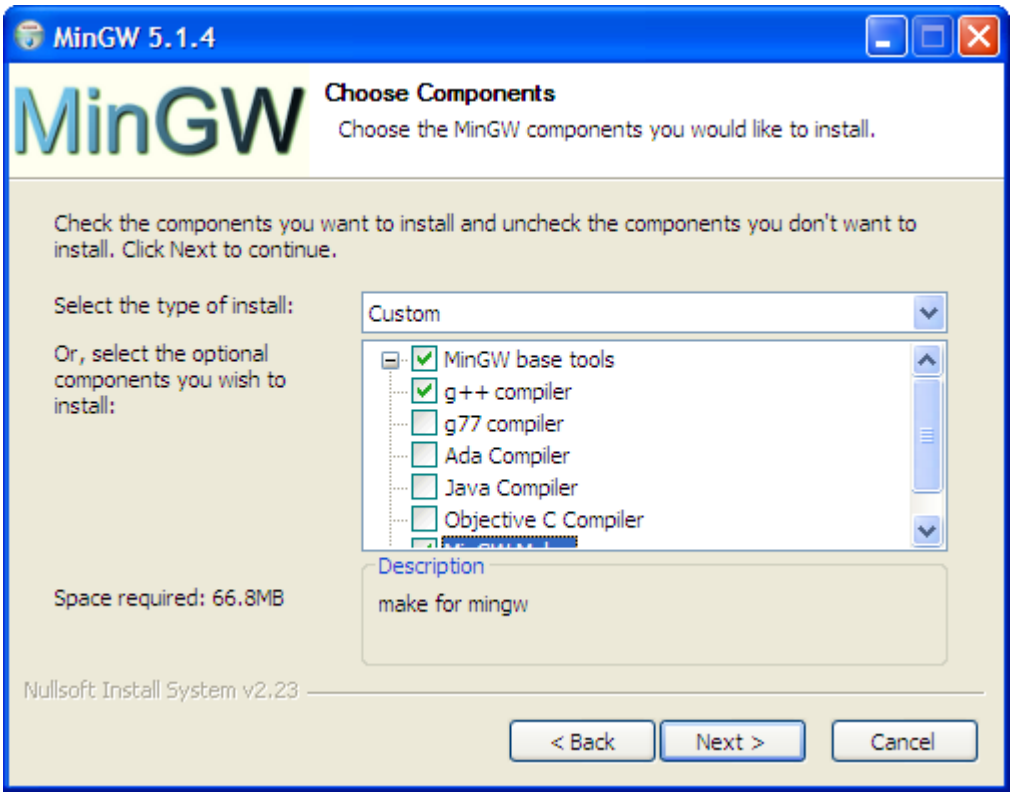

Abbildung 5.1: Komponentenwahl in MinGW-Installation

Dabei werden die *MinGW base tools* und, da in der Programmiersprache C++ programmiert wird, auch der *g++ compiler* gebraucht. Empfehlenswert ist auch die Komponente *MinGW Make* damit Makefiles ausgeführt werden können. Damit in der Windows-Kommandozeile nach der Installation die Compiler-Befehle wie in der Linux-Shell aufgerufen werden können, muss noch die Windows-Umgebungsvariable *Path* gesetzt werden. Diese muss auf den *bin*-Ordner von MinGW zeigen, beispielsweise „C:\Programme\MinGW\bin“.

Eine Quelldatei mit den Dateinamen `bsp.cpp` kann dann wie in Linux mit einen folgender zwei Befehle kompiliert werden:

```
g++ bsp.cpp
```

bzw.

```
g++ -o Bsp bsp.cpp
```

Beim ersten Befehl wird das Programm `a.exe` erstellt, während es beim zweiten `Bsp.exe` heißt.

Eine Objekt-Datei wird ebenso wie in Linux mit

```
g++ -c bsp.cpp
```

erstellt.

Damit ein Programm nach dem ISO-C++-Standard kompiliert wird, werden weitere Parameter angegeben:

```
g++ -Wall -ansi -pedantic bsp.cpp
```

Auch das Makefile wird wie in Linux mit `make` aufgerufen, wobei dieser Befehl unter Windows den äquivalenten Befehl `mingw32-make` startet.

Beim arbeiten auf verschiedenen Betriebssystemen muss auf den Inhalt des Makefiles geachtet werden. Während in Linux Dateien mit `rm` gelöscht werden geschieht dies in Windows mit dem Befehl `del`. Oft unterstützt Linux jedoch auch Windows-Kommandozeilen-Befehle. Weiters gibt es für Windows auch Projekte, wie *GnuWin32* auf http://gnuwin32.sourceforge.net. Dieses bietet zahlreiche Pakete an, welche unterschiedliche „Befehls“-Programme für die Kommandozeile enthalten. Diese sind nativ unter Windows lauffähig und haben die selbe Syntax wie unter Linux. Der Löschbefehl `rm` ist beispielsweise im Paket *coreutils* enthalten. Damit die Befehle unter Windows anwendbar sind muss der *bin*-Ordner der *GnuWin32*-Komponenten in der Windows-Umgebungsvariable *Path* eingebunden werden.

So ist nicht nur der C++-Quellcode Betriebssystem-unabhängig sondern auch die Makefiles. Das kann für die Schüler sehr nützlich sein, wenn sie in der Schule und daheim zwei verschiedene Betriebssysteme verwenden. Für dieses Thema sollte jedoch nicht sehr viel Unterrichtszeit beansprucht werden.

5.21 Entwicklungsumgebung

Grundsätzlich sollte Schülern, welche noch keine bzw. geringe Programmiererfahrung haben keine größere Entwicklungsumgebung aufgezwungen werden, weil sie überfordert werden könnten. Da sie sich vor allem auf das Erlernen der Programmiersprache konzentrieren müssen, ist eine gleichzeitige Einschulung in eine größere Entwicklungsumgebung nicht förderlich. Für C++ ist deshalb anfangs ein einfacher Texteditor am idealsten. Die Schüler kommen dadurch häufiger mit den Quelldateien selbst in Verbindung. Sie erkennen deshalb besser, dass C++-Quelldateien nichts anderes als normale Textdateien mit einer anderen Erweiterung (beispielsweise `.cpp`) sind. Außerdem müssen sie die Programme somit in der Kommandozeile

kompilieren. Die Schüler haben deshalb über wichtige Bereiche eine bessere Übersicht, als mit vielen Entwicklungsumgebungen von denen vieles automatisch vollbracht wird.

Hat ein Schüler eine Programmiererfahrung von 1 bis 2 Semestern, so überwiegen die Vorteile einer größeren Entwicklungsumgebung immer stärker. Eine die sich für den Schulunterricht besonders eignet ist Eclipse. Diese ist frei im Internet erhältlich, Plattform-unabhängig, sehr weit verbreitet und von der Industrie stark gefördert.

Eclipse unterstützt mehrere Programmiersprachen, auch C++. Will man in dieser Programmiersprache programmieren, so muss das *eclipse-cpp*-Paket heruntergeladen werden oder nachträglich Eclipse um das CDT-Plugin erweitert werden.

Weiters braucht Eclipse auch einen Compiler. Dafür bietet sich wie bereits im Compiler-Kapitel beschrieben MinGW an. Ist alles ordentlich eingerichtet so kann der Schüler bereits mit dem Programmieren beginnen. Er kann sich unter „File->New“ für ein neues C++-Projekt entscheiden. Nachher muss ein Projektname eingegeben werden und im Knoten „Executable“ gibt es unter anderem die Möglichkeiten mit einem leeren Projekt oder einem „Hello World“-Projekt zu beginnen. Weiters muss der Compiler (beispielsweise: MinGW) bestimmt werden.

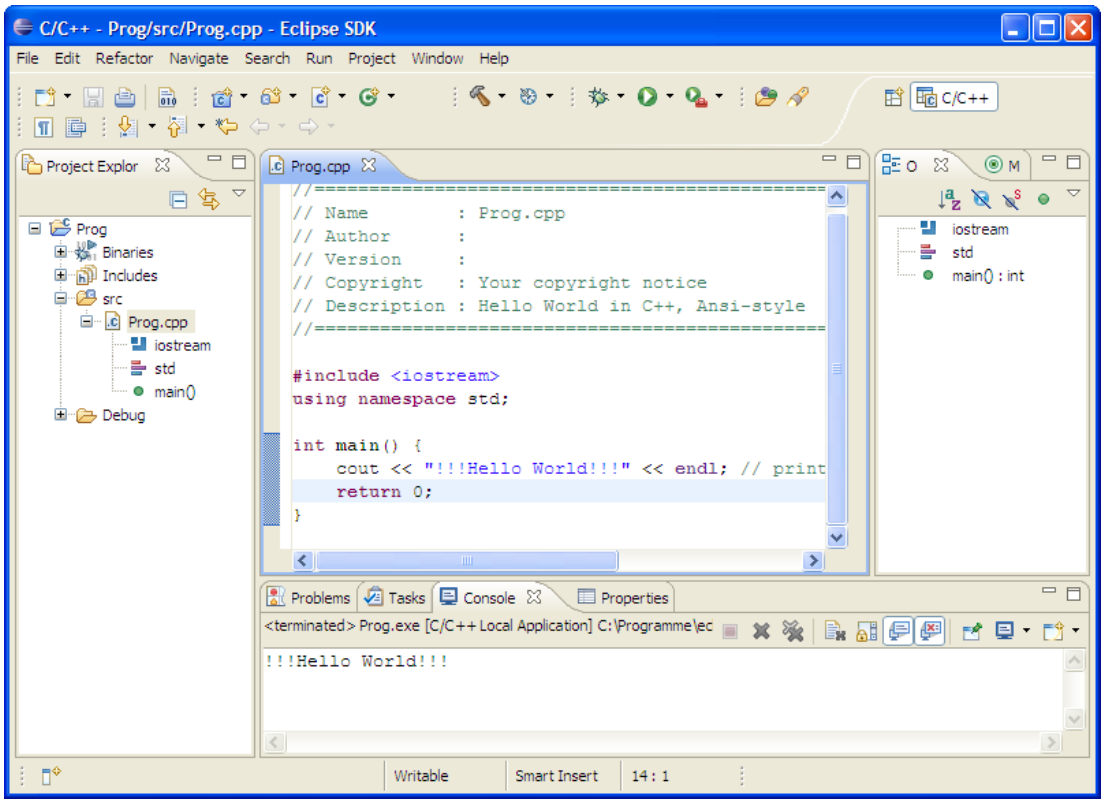

Abbildung 5.2: „Hello World“-Programm in Eclipse programmiert

In diesem Bild ist ein in Eclipse erstelltes „Hello World“-Programm in C++ sichtbar. Das Programm ist mit einem Klick kompilierbar und startbar. In der Konsolen-Ansicht sieht man die Ausgabe `!!!Hello World!!!`. Weiters kann Eclipse auch das Verwalten eines Makefiles überlassen werden und die Compiler-Parameter angepasst werden. Es wird auch viel mehr Übersichtlichkeit geboten, als in einem normalen Texteditor. Die einzelnen Teilfenster können beispielsweise minimiert werden, sobald sie nicht mehr gebraucht werden, damit andere Fenster mehr Platz haben.

Bei entsprechender Programmiererfahrung kann mit einer Entwicklungsumgebung also viel produktiver programmiert werden als mit einem reinen Texteditor. Im Schulunterricht sollte jedoch hauptsächlich das Programmieren selbst und nicht die Entwicklungsumgebung gelehrt werden. Es können beispielsweise nur die wichtigsten Konzepte gelehrt werden oder Folien verteilt werden in denen diese anschaulich erklärt werden. Weitere Erkenntnisse über die Entwicklungsumgebung können interessierte Schüler selbst erlangen.

5.22 Grafische Programmierung

Auch die grafische Programmierung ist ein wichtiger Bestandteil beim Lehren von C++. Es gibt zwar wie bereits erwähnt keine standardisierte C++-GUI-Bibliotheken, doch einige Möglichkeiten grafische Programme auf C++-Basis zu programmieren. Bevor mit der grafischen Programmierung begonnen wird sollten bereits gute Programmierkenntnisse des standardisierten C++ vorhanden sein. Das hat folgende Gründe:

- Schüler könnten überfordert sein, wenn das erste Programm bereits eine grafische Benutzeroberfläche besitzt, da ihnen wesentlich mehr Konzepte erklärt werden müssen als bei Konsolenprogrammen.
- Es gibt mehrere Entwicklungswerkzeuge für grafische Programme auf C++-Basis. Da diese C++ nicht immer erweitern, sondern daran auch Änderungen vornehmen, ist es für die Schüler vorteilhaft vorher standardisiertes C++ kennen zu lernen.

Eine Einführung in die grafische Programmierung ist deshalb frühestens nach einem Jahr C++-Programmierunterricht (bei ca. 4 Stunden pro Woche) empfehlenswert. Für etwas umfangreichere Programme ist frühestens das 4. Semester des C++-Programmierunterrichts (bei ca. 4 Stunden pro Woche) geeignet.

Beispielprogramm für grafische Programmierung:

In diesem Beispiel wird ein Taschenrechner beschrieben, welcher die vier Grundrechenoperationen beherrscht. Es gilt Punktrechnung vor Strichrechnung und es können auch Klammern in der Rechnung vorkommen, welche eine Teilrechnung bestimmen. Zahlen können Kommastellen und ein Minus-Vorzeichen besitzen. Mit „C“ wird eine Rechnung gelöscht und mit „R“ können Ziffern oder das Komma einer aktuell eingegebenen Zahl gelöscht werden. Schließlich wird mit dem „=“-Button das Ergebnis berechnet. Für die einfachere Eingabe wurden auch Tastaturkürzel programmiert. Diese sind für das Schulbeispiel natürlich optional.

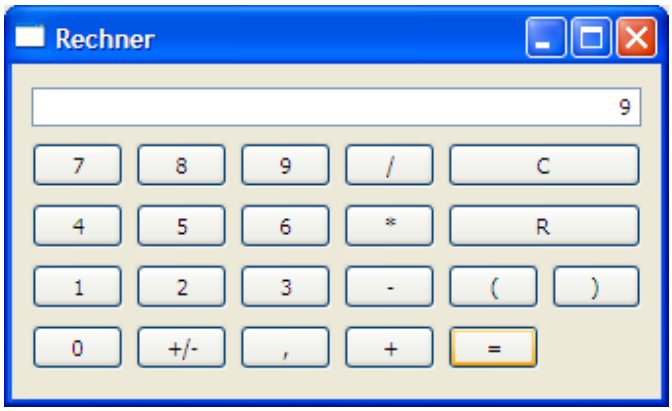

Abbildung 5.3: Taschenrechner in Windows XP

Sicherlich kann das Beispiel für den Schulunterricht noch vereinfacht werden (beispielsweise das Programmieren eines Taschenrechners ohne Klammer-Operationen), da es bereits eine bestimmte Größe besitzt.

Für das Beispiel wurde die Klassenbibliothek Qt verwendet. Sie ist frei im Internet erhältlich, Plattform-unabhängig und besitzt reichlich Funktionalität. In der Praxis wird sie auch von einigen Programmen (momentan vor allem unter Linux) verwendet. Für die Entwicklungsumgebungen Eclipse (für mehrere Plattformen verfügbar) und Visual Studio gibt es ein Qt-Integrations-Plugin das Qt in diese Entwicklungsumgebungen integriert. Qt ist deshalb für den Schulunterricht sehr geeignet, da es in Verbindung mit Eclipse auf mehreren Plattformen verfügbar ist. Qt kann natürlich auch ohne Eclipse verwendet werden, jedoch vereinfacht es das Programmieren mit Qt um einiges.

Nachfolgend wird der Quellcode des Taschenrechner-Programms aufgelistet. Der Inhalt der `ui`-Datei (Fenster-Layout) ist nicht dabei, da sie eine reine XML-Datei ist. Stattdessen sind Screenshots (Abbildung 5.3 bzw. 5.4) vorhanden.

main.cpp:

```
#include "rechner.h"

#include <QtGui>
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Rechner w;
    w.show();
    return a.exec();
}
```

rechner.h:

```
#ifndef RECHNER_H
#define RECHNER_H

#include <QtGui/QWidget>
#include "ui_rechner.h"
#include "offeneKlammer.h"
#include "zahl.h"
#include "operation.h"
#include <QTextStream>
#include <limits>


class Rechner : public QWidget
{
    Q_OBJECT

public:
    Rechner(QWidget *parent = 0);
    ~Rechner();

private:
    Ui::RechnerClass ui;
    QVector<QString> speicher_2;
    QVector<Elemente*> speicher;
    bool za;
    QString aktuell;
    void fuegeZifferEin(QChar);
    const static int vorkommastellen =
        (int)std::numeric_limits<double>::digits10*3/5;
    const static int nachkommastellen =
        (int)std::numeric_limits<double>::digits10*2/5;
    const static char komma = ',';
    void schreibe_meldung(QString);
    void strichOperationsErgebnis(double, bool, bool);
    int anzK, anzG;
    void double_nach_aktuell(double);
    void loescheNachkommaNullen(QString &, bool);
    char ziffer_event;
    char op_event;
    char rechen_event;

protected:
    void keyPressEvent(QKeyEvent *event);

private slots:
    void ziffer_gedrueckt();
    void komma_gedrueckt();
    void clear_gedrueckt();
    void r_gedrueckt();
    void vorzeichen_gedrueckt();
    void operation_gedrueckt();
    void berechnen_gedrueckt();
    void offene_Klammer_gedrueckt();
};

#endif
```

rechner.cpp:

```
#include "rechner.h"
#include <QKeyEvent>
#include <QDesktopWidget>

Rechner::Rechner(QWidget *parent)
    : QWidget(parent)
{
      ui.setupUi(this);
      // Ziffern
      QObject::connect(ui.pushButton, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_2, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_3, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_6, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_7, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_8, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_10, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_11, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_13, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      QObject::connect(ui.pushButton_14, SIGNAL(clicked()), this,
            SLOT(ziffer_gedrueckt()));
      // Komma
      QObject::connect(ui.pushButton_19, SIGNAL(clicked()), this,
            SLOT(komma_gedrueckt()));
      // Clear
      QObject::connect(ui.pushButton_5, SIGNAL(clicked()), this,
            SLOT(clear_gedrueckt()));
      // R
      QObject::connect(ui.pushButton_9, SIGNAL(clicked()), this,
            SLOT(r_gedrueckt()));
      // +/-
      QObject::connect(ui.pushButton_18, SIGNAL(clicked()), this,
            SLOT(vorzeichen_gedrueckt()));
      // Operationen
      QObject::connect(ui.pushButton_4, SIGNAL(clicked()), this,
            SLOT(operation_gedrueckt()));
      QObject::connect(ui.pushButton_12, SIGNAL(clicked()), this,
            SLOT(operation_gedrueckt()));
      QObject::connect(ui.pushButton_15, SIGNAL(clicked()), this,
            SLOT(operation_gedrueckt()));
      QObject::connect(ui.pushButton_20, SIGNAL(clicked()), this,
            SLOT(operation_gedrueckt()));
      // offene Klammer ( -> K
      QObject::connect(ui.pushButton_16, SIGNAL(clicked()), this,
            SLOT(offene_Klammer_gedrueckt()));
      // geschlossene Klammer ) -> G
      QObject::connect(ui.pushButton_17, SIGNAL(clicked()), this,
            SLOT(berechnen_gedrueckt()));
      // =
```

```
	QObject::connect(ui.pushButton_21, SIGNAL(clicked()), this,
		SLOT(berechnen_gedrueckt()));

	speicher_2.push_back("B");
	aktuell = "0";
	za = false;
	ui.lineEdit->setText("0");
	OffeneKlammer * beginn = new OffeneKlammer();
	beginn->setzeName("B");
	// beginn wird in den Vektor eingefügt, jedoch nicht kopiert
	speicher.push_back(beginn);
	beginn = 0;
	anzK = anzG = 0;
	// Fenster im Bildschirm zentrieren
	QRect qRect(QApplication::desktop()->screenGeometry());
	int iXpos=qRect.width()/2-this->width()/2;
	int iYpos=qRect.height()/2-this->height()/2;
	move(iXpos,iYpos);
}

Rechner::~Rechner()
{

}

void Rechner::ziffer_gedrueckt()
{
	QPushButton *zahlButton = qobject_cast<QPushButton *>(sender());
	QChar ziffer;
	if (zahlButton != 0)
		ziffer = zahlButton->text()[0];
	else
		// Tastatureingabe
		ziffer = ziffer_event;
	Elemente * element = speicher.last();
	ui.lineEdit->insert(element->bekommeName());
	// sind alle Zahlen abgeschlossen und das letzte Speicherelement
	// eine Operation, offene Klammer oder der Beginn
	if ((za == true) && ((element->bekommeName()[0] == 'P')
		|| (element->bekommeName()=="K")
			|| (element->bekommeName() == "B")))
	{
		aktuell = ziffer;
		ui.lineEdit->setText(ziffer);
		// neue Zahl als nicht abgeschlossene Zahl kennzeichnen
		za = false;
	}
	else
		// Zahl noch nicht abgeschlossen
		if (za == false)
		{
			fuegeZifferEin(ziffer);
			ui.lineEdit->setText(aktuell);
		}
		else
			// ist letzte abgeschlossene Zahl ein Resultat
			if ((za == true) && (element->bekommeName() == "ZE"))
			{
				speicher.pop_back();
```

```
                        aktuell = ziffer;
                        ui.lineEdit->setText(ziffer);
                        za = false;
                  }
      element = 0;
}

void Rechner::fuegeZifferEin(QChar ziffer)
{
      if ((aktuell == "0") || (aktuell == ""))
      {
            aktuell = ziffer;
            return;
      }
      int pos_komma = aktuell.indexOf(komma);
      // Anzahl von Vorkommastellen kontrollieren
      // und eventuell Ziffer einfügen
      if (pos_komma < 0)
      {
            char vorzeichen = 0;
            if (aktuell[0] == '-')
                  vorzeichen = 1;
            if (aktuell.length() < vorkommastellen + vorzeichen)
                  aktuell.push_back(ziffer);
      }
      // Anzahl von Nachkommastellen kontrollieren
      // und eventuell Ziffer einfügen
      else
      {
            if ((aktuell.length()-(pos_komma+1)) < nachkommastellen)
                  if (ziffer != '0')
                        aktuell.push_back(ziffer);
                  else
                        if ((aktuell.length() - (aktuell.indexOf(komma)
                              + 1)) < (nachkommastellen - 1))
                              aktuell.push_back(ziffer);
      }
}

void Rechner::komma_gedrueckt()
{
      if ((za == false) && (aktuell.indexOf(komma) < 0))
      {
            aktuell.push_back(komma);
            ui.lineEdit->insert(QChar(komma));
      }
}

void Rechner::clear_gedrueckt()
{
      speicher.clear();
      OffeneKlammer * beginn = new OffeneKlammer();
      beginn->setzeName("B");
      speicher.push_back(beginn);
      beginn = 0;
      aktuell = "0";
      za = false;
      anzK = anzG = 0;
      ui.lineEdit->setText("0");
```

```
}

void Rechner::r_gedrueckt()
{
	if (za == false)
	{
		if (aktuell == "")
			return;
		// letztes Zeichen löschen
		aktuell.remove(aktuell.length()-1, 1);
		if ((aktuell == "") || (aktuell == "-"))
			aktuell = "0";
		ui.lineEdit->setText(aktuell);
	}
}

void Rechner::vorzeichen_gedrueckt()
{
	// Ist zahl nicht selbst eingegeben so wird Vorzeichen nicht
	// gewechselt
	if (za == true)
		return;
	char komma_gesetzt = 0;
	if (aktuell.indexOf(komma) > -1)
		komma_gesetzt = 1;
	if (aktuell.count('0') < (aktuell.length() - komma_gesetzt))
	{
		if (aktuell[0] == '-')
		{
			aktuell.remove(0, 1);
			ui.lineEdit->setText(aktuell);
		}
		else
		{
			aktuell.push_front('-');
			ui.lineEdit->setText(aktuell);
		}
	}
}

void Rechner::operation_gedrueckt()
{
	// Sende-Button abspeichern
	QPushButton *operationsButton =
		qobject_cast<QPushButton*>(sender());
	// Bestimmen ob Button eine Strich- oder Punktoperation ist
	char akt_operation;
	if (operationsButton != 0)
		akt_operation = operationsButton->text()[0].toAscii();
	else
		// Tastatureingabe
		akt_operation = op_event;
	bool strich_operation = ((akt_operation == '+')
		|| (akt_operation == '-'));

	Elemente * el = speicher.last();
	// Zahl noch nicht abgeschlossen
	if ((za == false) || (za == true)
		&& (el->bekommeName()[0] == 'Z'))
```

```
{
	Zahl * z = 0;
	if (za == false)
	{
		if (aktuell.indexOf(komma) > -1)
			// Zahl richtig formatieren
			while (aktuell.endsWith('0'))
				aktuell.remove(aktuell.length()-1, 1);
		if (aktuell == "")
			aktuell = "0";
		else
			if (aktuell.endsWith(komma))
				// Komma löschen
				aktuell.remove(aktuell.length()-1, 1);
		z = new Zahl();
		z->setzeName("Z");
		bool kont;
		QString tmp = aktuell;
		tmp.replace(komma, '.');
		z->setzeZahl(tmp.toDouble(&kont));
		// konnte die Zahl nicht von QString nach double
		// konvertiert werden
		if (!kont)
		{
			schreibe_meldung("Konvertion Error!");
			return;
		}
		ui.lineEdit->setText(aktuell);
	}
	else
	{
		z = dynamic_cast<Zahl *>(el);
		speicher.pop_back();
	}

	el = speicher.last();
	// Ist bereits eingetragenes Element eine Punkt-Operation
	// oder ist bereits eingetragenes Element eine beliebige
	// Operation wobei
	// die momentan gedrückte Operationstaste eine
	// Strichoperation sein muss
	if ((el->bekommeName() == "PP") || (el->bekommeName()[0] ==
		'P') && strich_operation)
	{
		// Punkt-Operation
		Operation * p = dynamic_cast<Operation *>(el);
		QChar gespeicherte_op = p->bekommeOperation();
		// Punkt-Operatoin vom Speicher löschen
		speicher.pop_back();
		delete p;
		p = 0;
		// Zweite Zahl für Punktoperation vom Speicher lesen
		Zahl * z2 = dynamic_cast<Zahl *>(speicher.last());
		// und dann vom Speicher löschen
		speicher.pop_back();
		double aktuell_d;
		// Punktoperatoin durchführen
		switch (gespeicherte_op.toAscii())
		{
```

```
                case '+':
                      aktuell_d = z2->bekommeZahl() +
                            z->bekommeZahl();
                      break;
                case '-':
                      aktuell_d = z2->bekommeZahl() -
                            z->bekommeZahl();
                      break;
                case '*':
                      aktuell_d = z2->bekommeZahl() *
                            z->bekommeZahl();
                      break;
                case '/':
                      if (z->bekommeZahl() == 0.0)
                      {
                            schreibe_meldung("Division durch 0!");
                            return;
                      }
                      aktuell_d = z2->bekommeZahl() /
                            z->bekommeZahl();
                      break;
                default:
                      schreibe_meldung("keine gültige Operation!");
                      return;
                }

                // Mögliche weitere Strichoperationen vollbringen
                strichOperationsErgebnis(aktuell_d, false, true);
                ui.lineEdit->setText(aktuell);
                delete z;
          }
          // ist if-Bedingung nicht geschehen so wird die Zahl wieder
          // (die vom Speicher geholte Zahl) bzw. neu (die vorher
          // nicht abgeschlossene Zahl) eingetragen
          else
          {
                speicher.push_back(z);
                z = 0;
          }
          Operation * op = new Operation();
          if (strich_operation)
                op->setzeName("PS");
          else
                op->setzeName("PP");
          op->setzeOperation(akt_operation);
          speicher.push_back(op);
          op = 0;
          za = true;
    }
    // keine Zahl
    else
    {
          Elemente * el = speicher.last();
          // ist letztes Element eine Operation so wird sie durch
          // eine neue ersetzt
          if ((el->bekommeName()[0] == 'P'))
          {
                speicher.pop_back();
```

```
                    // Operation einfügen
                    Operation * op = new Operation();
                    if (strich_operation)
                          op->setzeName("PS");
                    else
                          op->setzeName("PP");
                    op->setzeOperation(akt_operation);
                    speicher.push_back(op);
                    op = 0;
              }
        }
}

// Schreibt double nach den String "aktuell"
void Rechner::double_nach_aktuell(double d)
{
        if (d == 0.0)
        {
              aktuell = "0";
              return;
        }
        QString aktuell_hilf;
        aktuell_hilf.setNum(d, 'E', 14);
        aktuell_hilf.replace('.', komma);
        int index_von_E =
              aktuell_hilf.indexOf('E', 0, Qt::CaseInsensitive);
        QString zs = aktuell_hilf.left(index_von_E);
        QString hochzahl_s = aktuell_hilf.mid(index_von_E + 1);
        short hochzahl = hochzahl_s.toShort();
        // hat die Zahl Vorkommastellen
        // und können alle Vorkommastellen als normale Zahl dargestellt
        // werden
        if ((hochzahl >= 0) && (hochzahl < vorkommastellen))
        {
              // Nachkommastellen der Mantisse
              short z_nachkomma = zs.length() - (zs.indexOf(komma) + 1);
              // Sind die Nachkommastellen der Mentisse kleiner gleich
              // der Hochzahl so wird das Komma gelöscht und die
              // Differenz bestimmt wieviele
              // Nullen an den String angehängt werden
              if (z_nachkomma <= hochzahl)
              {
                    short nullen = hochzahl - z_nachkomma;
                    zs.remove(zs.indexOf(komma), 1);
                    while (nullen > 0)
                    {
                          zs.append('0');
                          nullen--;
                    }
              }
              // Hat die Mentisse mehr Nachkommastellen als die Hochzahl
              // gross ist so wird das Komma um "Hochzahl" Stellen
              // verschoben und eventuell die Nachkommastellen auf
              // "nachkommastellen" Stellen gekürzt
              else
              {
                    short komma_pos = zs.indexOf(komma);
                    zs.remove(zs.indexOf(komma), 1);
                    komma_pos = komma_pos + hochzahl;
```

```
                zs.insert(komma_pos, komma);
                short l = komma_pos + nachkommastellen;
                loescheNachkommaNullen(zs, true);
                if (l < zs.length() - 1)
                        zs.resize(l + 1);
        }
        aktuell = zs;
}
else
        // hat die Zahl nur Nachkommaziffern
        if (hochzahl < 0)
        {
                loescheNachkommaNullen(zs, false);
                short komma_pos = zs.indexOf(komma);
                short anz_nz = zs.length() - (komma_pos + 1);
                // sind alle Nachkommastellen darstellbar
                if ((anz_nz - nachkommastellen) <= hochzahl)
                        // Solange "hochzahl" kleiner "0" ist
                        // wird die Hochzahl erhöht und eine Nachkomma-
                        // null eingefügt
                        while (hochzahl < 0)
                        {
                                zs.remove(komma_pos, 1);
                                komma_pos--;
                                QString n_und_k = "0";
                                n_und_k += komma;
                                zs.insert(komma_pos, n_und_k);
                                anz_nz = zs.length() - (++komma_pos + 1);
                                hochzahl++;
                        }
                // Sind mehr Nachkommastellen enthalten als angezeigt
                // werden sollen
                // so werden jene abgeschnitten
                short l = komma_pos + nachkommastellen;
                if (l < (zs.length() - 1))
                        zs.resize(l + 1);
                // ist ein Komma am Schluss so wird es abgeschnitten
                if (zs.endsWith(komma))
                        zs.remove(zs.length() - 1, 1);
                // ist "hochzahl" immer noch kleiner als 0 so wird in
                // der Anzeige
                // dies durch die Exponentenschreibweise angezeigt
                if (hochzahl < 0)
                {
                        zs += "E";
                        QString s_hochzahl;
                        s_hochzahl.setNum(hochzahl);
                        zs += s_hochzahl;
                        if (hochzahl >= -9)
                                zs.insert(zs.length()-1, '0');
                }
                aktuell = zs;
        }
        // hat die Zahl Vorkommastellen und ist sie nur in der
        // Exponentenschreibweise darstellbar
        else
        {
                aktuell.setNum(d, 'E', nachkommastellen);
                aktuell.replace('.', komma);
```

```
                while (aktuell[aktuell.indexOf('E', 0,
                      Qt::CaseInsensitive)-1] == '0')
                      aktuell.remove(aktuell.indexOf('E', 0,
                            Qt::CaseInsensitive)-1, 1);
                if (aktuell[aktuell.indexOf('E', 0,
                      Qt::CaseInsensitive)-1] == komma)
                      aktuell.remove(aktuell.indexOf('E', 0,
                            Qt::CaseInsensitive)-1, 1);
          }
}

// 1. Parameter: Zahlstring
// 2. Parameter: Bei true Komma auch entfernen, wenn es am Schluss
//               steht
void Rechner::loescheNachkommaNullen(QString & zahl,
      bool loesche_komma)
{
      while (zahl.endsWith('0'))
            zahl.remove(zahl.length()-1, 1);
      if (loesche_komma && zahl.endsWith(komma))
            zahl.remove(zahl.length()-1, 1);
}

// Liefert Strichpunkt-Berechnung
// Möglichkeiten:
// Von hinten bis Beginn -> Ereignis durch Berechnung gestartet
// von hinten bis einschliesslich offener Klammer -> Ereignis durch
// geschlossene Klammer gestartet
// von hinten bis vor offeneer Klammer -> Ereignis wird durch
// Operation gestartet
void Rechner::strichOperationsErgebnis(double ersteZahl,
      bool abschliessen, bool durch_op_aufgerufen)
{
      double z_nach_zahl = ersteZahl;
      double z_nach_op = 0.0;
      Elemente * el;
      bool zahl_mom = false;
      do
      {
            el = speicher.last();
            // Strich-Operation
            if (el->bekommeName() == "PS")
            {
                  if (zahl_mom)
                  {
                        schreibe_meldung("Operatorenreihenfolge-
                              Fehler");
                        return;
                  }
                  // Strich-Operation
                  Operation * ps = dynamic_cast<Operation *>(el);
                  if (ps->bekommeOperation() == '+')
                        z_nach_op = z_nach_op + z_nach_zahl;
                  else
                        z_nach_op = z_nach_op - z_nach_zahl;
                  // Strich-Operatoin vom Speicher löschen
                  speicher.pop_back();
                  ps = 0;
                  zahl_mom = true;
```

```
            }
            else
                  if (el->bekommeName()[0] == 'Z')
                  {
                        if (!zahl_mom)
                        {
                              schreibe_meldung("Zahlenreihenfolge-
                                    Fehler");
                              return;
                        }
                        Zahl * z =
                              dynamic_cast<Zahl *>(speicher.last());
                        speicher.pop_back();
                        z_nach_zahl = z->bekommeZahl();
                        z = 0;
                        zahl_mom = false;
                  }
                  else
                        if ((el->bekommeName() == "B")
                              || (el->bekommeName() == "K"))
                        {
                              if (!durch_op_aufgerufen
                                    && (el->bekommeName() == "K"))
                                    speicher.pop_back();
                              z_nach_op = z_nach_op + z_nach_zahl;
                              Zahl * z = new Zahl();
                              if (abschliessen)
                                    z->setzeName("ZE");
                              else
                                    z->setzeName("Z");
                              z->setzeZahl(z_nach_op);
                              speicher.push_back(z);
                              za = true;
                              double_nach_aktuell(z_nach_op);
                              return;
                        }
                        else
                        {
                              schreibe_meldung("Rechnung nicht
                                    ausführbar!");
                              return;
                        }
      } while (el->bekommeName() != "B"
              && (el->bekommeName() != "K"));
}

// Button 17 -> )
// Button 21 -> =
void Rechner::berechnen_gedrueckt()
{
      QPushButton * rechenButton =
            qobject_cast<QPushButton *>(sender());
      char akt_berechnung;
      if (rechenButton != 0)
            if (rechenButton == ui.pushButton_21)
                  akt_berechnung = '=';
            else
                  akt_berechnung = ')';
      else
```

```
      // Tastatureingabe
      akt_berechnung = rechen_event;
if (akt_berechnung == '=')
{
      if (anzK > 0)
      {
            schreibe_meldung("noch offene Klammern enthalten");
            return;
      }
}
else
      if (anzK <= 0)
      {
            schreibe_meldung("es konnte keine Klammer geschlossen
                  werden");
            return;
      }

double ersteZahl;
// ist Zahl noch nicht abgeschlossen
if (za == false)
{
      za = true;
      // Zahl richtig formatieren
      if (aktuell.endsWith(komma))
            // Komma löschen
            aktuell.remove(aktuell.length()-1, 1);
      bool kont;
      QString tmp = aktuell;
      tmp.replace(komma, '.');
      ersteZahl = tmp.toDouble(&kont);
      // konnte die Zahl nicht von QString nach double
      // konvertiert werden
      if (!kont)
      {
            schreibe_meldung("Konvertion Error!");
            return;
      }
}
else
{
      Elemente * el_z = speicher.last();
      if (el_z->bekommeName()[0] == 'Z')
      {
            Zahl * z_h = dynamic_cast<Zahl *>(el_z);
            speicher.pop_back();
            ersteZahl = z_h->bekommeZahl();
            delete z_h;
            z_h = 0;
      }
      else
      {
            schreibe_meldung("letztes Element ist keine Zahl");
            return;
      }
      el_z = 0;
}

Elemente * el = speicher.last();
```

```
	// Ist letzte Operation eine Punkt-Operation
	if (el->bekommeName() == "PP")
	{
		// Punkt-Operation
		Operation * pp = dynamic_cast<Operation *>(el);
		bool multiplikation = true;
		if (pp->bekommeOperation() != '*')
			multiplikation = false;
		// Punkt-Operatoin vom Speicher löschen
		speicher.pop_back();
		delete pp;
		pp = 0;
		// Zweite Zahl für Punktoperation vom Speicher lesen
		Zahl * z2 = dynamic_cast<Zahl *>(speicher.last());
		// und dann vom Speicher löschen
		speicher.pop_back();
		double aktuell_d;
		// Punktoperatoin durchführen
		if (multiplikation)
			aktuell_d = z2->bekommeZahl() * ersteZahl;
		else
		{
			if (ersteZahl == 0.0)
			{
				schreibe_meldung("Division durch 0!");
				return;
			}
			aktuell_d = z2->bekommeZahl() / ersteZahl;
		}
		delete z2;
		z2 = 0;
		if (akt_berechnung == '=')
			strichOperationsErgebnis(aktuell_d, true, false);
		else
			strichOperationsErgebnis(aktuell_d, false, false);
	}
	else
		if (akt_berechnung == '=')
				strichOperationsErgebnis(ersteZahl, true,
					false);
			else
				strichOperationsErgebnis(ersteZahl, false,
					false);

	ui.lineEdit->setText(aktuell);
	if (akt_berechnung == ')')
		anzK--;
}

// Schreibt Meldung auf Anzeige
// Erneuert wieder die Einstellungen
// "za" wird auf "true" gesetzt
// dadurch wird bei korrekter Eingabe eine neue Rechnung gestartet
void Rechner::schreibe_meldung(QString text)
{
	speicher.clear();
	OffeneKlammer * beginn = new OffeneKlammer();
	beginn->setzeName("B");
	speicher.push_back(beginn);
```

```
	beginn = 0;
	aktuell = "0";
	za = true;
	ui.lineEdit->setText(text);
	anzK = 0;
}

void Rechner::offene_Klammer_gedrueckt()
{
	Elemente * el = speicher.last();
	if ((el->bekommeName() == "B") || (el->bekommeName()[0] == 'P')
		|| (el->bekommeName() == "K"))
	{
		if ((el->bekommeName() == "B") && (aktuell != "0"))
			return;
		// Klammer als erstes Element nach dem Beginn setzen
		za = true;
		anzK++;
		OffeneKlammer * ok = new OffeneKlammer();
		ok->setzeName("K");
		speicher.push_back(ok);
		ui.lineEdit->setText("(");
	}
}

// Tastatureingabe
void Rechner::keyPressEvent(QKeyEvent *event)
{
	switch (event->key())
	{
	case Qt::Key_0:
		ziffer_event = '0';
		ziffer_gedrueckt();
		break;
	case Qt::Key_1:
		ziffer_event = '1';
		ziffer_gedrueckt();
		break;
	case Qt::Key_2:
		ziffer_event = '2';
		ziffer_gedrueckt();
		break;
	case Qt::Key_3:
		ziffer_event = '3';
		ziffer_gedrueckt();
		break;
	case Qt::Key_4:
		ziffer_event = '4';
		ziffer_gedrueckt();
		break;
	case Qt::Key_5:
		ziffer_event = '5';
		ziffer_gedrueckt();
		break;
	case Qt::Key_6:
		ziffer_event = '6';
		ziffer_gedrueckt();
		break;
	case Qt::Key_7:
```

```
            ziffer_event = '7';
            ziffer_gedrueckt();
            break;
        case Qt::Key_8:
            ziffer_event = '8';
            ziffer_gedrueckt();
            break;
        case Qt::Key_9:
            ziffer_event = '9';
            ziffer_gedrueckt();
            break;
        case Qt::Key_Escape:
        case Qt::Key_C:
            clear_gedrueckt();
            break;
        case Qt::Key_Backspace:
        case Qt::Key_R:
            r_gedrueckt();
            break;
        case Qt::Key_V:
            vorzeichen_gedrueckt();
            break;
        case Qt::Key_Plus:
            op_event = '+';
            operation_gedrueckt();
            break;
        case Qt::Key_Minus:
            op_event = '-';
            operation_gedrueckt();
            break;
        case Qt::Key_Asterisk:
            op_event = '*';
            operation_gedrueckt();
            break;
        case Qt::Key_Slash:
            op_event = '/';
            operation_gedrueckt();
            break;
        case Qt::Key_Return:
        case Qt::Key_Equal:
            rechen_event = '=';
            berechnen_gedrueckt();
            break;
        case Qt::Key_K:
        case Qt::Key_ParenLeft:
            offene_Klammer_gedrueckt();
            break;
        case Qt::Key_G:
        case Qt::Key_ParenRight:
            rechen_event = ')';
            berechnen_gedrueckt();
            break;
        case Qt::Key_Comma:
        case Qt::Key_Period:
            komma_gedrueckt();
            break;
        }
}
```

elemente.h:

```
#ifndef ELEMENTE_H_
#define ELEMENTE_H_

#include <QtGui/QWidget>

class Elemente
{
private:
	QString name;
public:
	virtual void setzeName(QString);
	virtual QString bekommeName();
	virtual ~Elemente(){};
};

#endif /*ELEMENTE_H_*/
```

elemente.cpp:

```
#include "elemente.h"

void Elemente::setzeName(QString name)
{
	Elemente::name = name;
}

QString Elemente::bekommeName()
{
	return name;
}
```

offeneKlammer.h:

```
#ifndef OFFENEKLAMMER_H_
#define OFFENEKLAMMER_H_
#include "elemente.h"

class OffeneKlammer: public Elemente
{
private:
	bool punktRechnung;
public:
	void setzePunktRechnung(bool punktRechnung);
	bool bekommePunktRechnung();
	OffeneKlammer();
};

#endif /*OFFENEKLAMMER_H_*/
```

offeneklammer.cpp:

```
#include "offeneKlammer.h"

void OffeneKlammer::setzePunktRechnung(bool punktRechnung)
{
```

```
	OffeneKlammer::punktRechnung = punktRechnung;
}

bool OffeneKlammer::bekommePunktRechnung()
{
	return punktRechnung;
}

OffeneKlammer::OffeneKlammer()
{
	punktRechnung = false;
}
```

operation.h:

```
#ifndef OPERATION_H_
#define OPERATION_H_

#include "elemente.h"

class Operation: public Elemente
{
private:
	QChar op;
public:
	void setzeOperation(QChar);
	QChar bekommeOperation();
};

#endif /*OPERATION_H_*/
```

operation.cpp:

```
#include "operation.h"

void Operation::setzeOperation(QChar op)
{
	Operation::op = op;
}

QChar Operation::bekommeOperation()
{
	return op;
}
```

zahl.h:

```
#ifndef ZAHL_H_
#define ZAHL_H_

#include "elemente.h"

class Zahl: public Elemente
{
private:
	double zahl;
public:
```

```
	void setzeZahl(double);
	double bekommeZahl();
};

#endif /*ZAHL_H_*/
```

zahl.cpp:

```
#include "zahl.h"

void Zahl::setzeZahl(double zahl)
{
	Zahl::zahl = zahl;
}

double Zahl::bekommeZahl()
{
	return zahl;
}
```

Funktionsweise des Rechners: Je nach Benutzereingabe werden verschiedene Elemente in einem Vektorfeld in der Eingabereihenfolge abgespeichert. Die Elemente besitzen alle einen Namen:

- "B" steht für den Beginn
- "K" bezeichnet eine offene Klammer
- "Z" bezeichnet eine Zahl
- "ZE" bezeichnet eine Ergebniszahl (Ergebniszahl einer abgeschlossenen Rechnung)
- "PS" bezeichnet eine Strichoperation
- "PP" bezeichnet eine Punktoperation

Die Zahlen und Operationen brauchen eine weitere Variable in welcher deren entsprechender Inhalt (Zahl bzw. Operation) abgespeichert wird.
Für die Reihenfolge der Elemente gilt folgendes(P ist entweder PS oder PP, Z ist Z oder ZE):

- K kann nach B, P, K sein
- P kann nach Z sein
- Z kann nach B, K, P sein
- Eine Berechnung kann nur nach einer Zahl durchgeführt werden (auch eine geschlossene Klammer bewirkt eine Berechnung)

Wird eine Operation eingegeben, so wird die letzte Punktrechnung im Vektor

berechnet:

```
Vektor: 1 * 2
Eingabe: +
neuer Vektor: 2 +

Vektor: 1 * 2
Eingabe: *
neuer Vektor: 2 *
```

In diesen Beispielen werden für ein besseres Verständnis nicht die genauen Einträge eines Vektors dargestellt. Beispielsweise würde der Vektor der die Rechnung 1 * 2 beinhaltet folgende Elemente enthalten:

```
E1:  name: "B"
E2:  name: "Z", zahl: 1.0
E3:  name: "PP", op: "*"
E4:  name: "Z", zahl: 2.0
```

Eventuell werden weitere noch existierende Strichrechnungen berechnet:

```
Vektor: 1 + 2 * 3
Eingabe: +
neuer Vektor: 6 +

Vektor: 1 + 2
Eingabe: +
neuer Vektor: 3 +
```

Eine Strichrechnung wird erst berechnet, wenn sie nicht mehr die letzte Strichrechnung in der aktuellen Teilrechnung ist, da eine Punktrechnung Vorrang hat.

Das selbe geschieht in Klammern: Es wird überprüft, ob die letzte Rechnung eine Punktrechnung ist. Trifft dies zu wird sie berechnet. Gibt es restliche Operationen, so können diese nur mehr Strichrechnungen sein (jede Punktrechnung wird immer schnellstmöglich berechnet). Schließlich wird der Klammerausdruck berechnet und im Vektor durch die Ergebniszahl ersetzt.

```
Vektor: ( 1 + 2 * 1
Eingabe: )
neuer Vektor: 3
```

Der Ausdruck `2 * 1` kann erst mit der Eingabe der Klammer berechnet werden, da die Zahl 1 erst bei Eingabe dieser abgeschlossen ist. Vor dem Abschließen der Zahl 1 könnte der Benutzer beispielsweise noch weitere Ziffern eingeben, sodass der Ausdruck `2 * 11` heißen könnte.

Das selbe wie bei Eingabe der geschlossenen Klammer geschieht beim Drücken der Berechnen-Taste „=“:

```
Vektor: 1 + 2 * 3
```

```
Eingabe: „=“-Taste
neuer Vektor: 7
```

Mit der boolschen Variable `za` wird bestimmt, ob eine vom Benutzer veränderbare Zahl vorherrscht oder nicht. D.h. ob sie vom Benutzer noch gelöscht bzw. geändert werden kann. Ist sie veränderbar, so können Ziffern, Vorzeichen, usw. eingegeben werden und auch die Lösch-Taste „R“ kann betätigt werden.

Die Variable `aktuell` speichert die momentan noch nicht abgeschlossene Zahl ab. Sie ist nur relevant, wenn `za` den Wert `false` hat. Sie ist eine String-Variable, da diese für Benutzereingaben leichter verwaltbar ist. Ist die Zahl abgeschlossen (beispielsweise durch Eingabe einer Operation) so wird sie als `double`-Wert in einem Zahlen-Objekt abgespeichert und in den Vektor eingefügt.

Die Variable `anzK` zählt die offenen Klammern, die noch nicht von einer schließenden Klammer ergänzt wurden. Somit wird sichergestellt, dass beim Abschluss der Rechnung die Klammeranzahl stimmt.

Wird die Clear-Taste „C“ gedrückt, so wird der Vektor wieder neu initialisiert, die aktuelle Zahl auf `0` gesetzt und als vom Benutzer änderbar gekennzeichnet.

Wird das Programm mit Qt-Bibliotheken programmiert so ist das in Windows programmierte Rechner-Programm leicht auf Linux portierbar:

Abbildung 5.4: Taschenrechner in openSUSE 11.0

Dieser Portierungsvorgang kann den Schülern näher erklärt werden. Dabei müssen einige von Qt in Windows automatisch erzeugte Dateien bei der Portierung nach Linux gelöscht werden. Alternativ können auch nur die brauchbaren Dateien kopiert werden. Diese Dateien sind die Quelldateien,

deren Header-Dateien, sowie die `ui`-Datei (grafisches Oberflächendesign) und die Projektdatei. Danach kann das Projekt bereits mit Eclipse als Qt-Projekt importiert werden. Der einzige Unterschied zu neu geschriebenen Programmen ist der, dass das Projekt nicht in den *workspace*-Ordner (Standard-Ordner für neu erstellte Projekte in Eclipse) von Eclipse importiert wird. Will man dies erreichen, so muss das Projekt in Eclipse wieder entfernt werden. Dabei muss darauf geachtet werden, dass Eclipse die Projekt-Dateien auf der Festplatte belässt. Dies wird vor dem Löschen des Projektes abgefragt. Jetzt kann das Projekt als „bereits existierendes Projekt“ in den Eclipse-Projekt-Ordner importiert werden. Diese Prozedur kann sicherlich von Eclipse noch vereinfacht werden. Dass sich ein Projekt im *workspace*-Ordner von Eclipse befinden soll hat mehr ordnungsspezifische als technische Gründe. Will sich ein Schüler die Prozedur des erneuten Importierens ersparen, kann er das Projekt auch an dem Ort belassen, wo es in Linux erstellt wurde.

Nach der Portierung sehen die Schüler, dass es auf einfache Art möglich ist ein grafisches C++-Programm für mehrere Plattformen zu schreiben. Das Rechner-Programm ist in kürzester Zeit portiert und muss nicht auf jeder Plattform neu geschrieben werden.

6 C++ in der Schule

6.1 Ansätze für Lehrpersonen

6.1.1 Strömungen der Lernpsychologie

Es ist sinnvoll, dass sich Lehrer über Unterrichtsmethoden informieren und sich dadurch weiterbilden. Andererseits muss jeder Lehrer deren Unterricht auf ihre Schüler spezialisieren. Bereits frühe Strömungen der Lernpsychologie wie der Behaviourismus und Kognitivismus haben einige sinnvolle Ergebnisse gebracht. Der Behaviourismus beispielsweise kommt anhand von empirischen Versuchen zum Schluss, dass Bestrafung im Unterricht nicht zum Erfolg führt. Er verlangt deshalb unter anderem das Schaffen von angenehmen Lernumgebungen, die Förderung der Aufmerksamkeit von Schülern und das Vermeiden von Bestrafungen. Der Kognitivismus fordert Maßnahmen wie die Angabe eines Lehrziels, die Sinnhaftigkeit des Lehrstoffs und das Näherbringen des Lehrstoffs an die Schüler auf möglichst einfache Art und Weise. [7]

6.1.2 Motivierung

Ohne den durch eine angemessene Motivierung erzeugten Lernwillen büßt der Unterricht viel an Effizienz ein. Deshalb ist Motivierung ein sehr wichtiges Ziel didaktischen Handelns. Dabei ist auch das Vermitteln der Freude am Tun und das Berücksichtigen von spontanen Interessen seitens der Schüler wichtig. Motivierung kann auch dadurch gefördert werden, indem auf Fragen der Schüler eingegangen wird, das Tempo des Unterrichts die Schüler nicht unter- bzw. überfordert und Teilerfolge möglichst früh vermittelt werden. Auch Erfolgserlebnisse fördern die Motivierung bei den Schülern. Eines wird bereits beim „Hello World“-Beispiel vorgefunden, weil dieses Beispiel ihnen ein sichtbares Ergebnis liefert. [8]

6.1.3 Kreativitätsförderung

Kreativität kann vielseitig interpretiert werden. *Beer* hat den Begriff als Fähigkeit

beschrieben, welche unsere heutige Gesellschaft zu originellen bisher nie da gewesenen Leistungen und Werken beflügelt und so diese bereichert und den Fortschritt der Kultur garantiert.
Kreativität ist die Voraussetzung für die Neukonstruktion von Wissen. Was für Unterrichtsfächer wie Kunst selbstverständlich ist, hat sicherlich auch im Informatikunterricht seine Daseinsberechtigung. Sie kann dadurch gefördert werden, indem Übungsaufgaben nicht starre Bedingungen haben, sondern dass diese jeder Schüler auf eigene Weise lösen kann. Sicherlich muss die Aufgabe so gestellt werden, dass das soeben Gelernte im Übungsbeispiel angewendet werden muss. [9]

6.1.4 Übung

Es gibt verschiedene Übungsformen wie die Apponierte, Direkte, Disponierte und Latente Übung. Dabei muss sicherlich jeder Lehrer selbst entscheiden, welche Übungsform für die Schulklasse am geeignetsten ist. Es kann auch eine Mischform sein. Wichtig beim Lehren der Programmiersprache C++ ist jedoch, wie bei anderen Programmiersprachen auch, die Anwendung von viel Praxis. [10]
Informatik hat eine eigenständige Fachdidaktik. Es geht bei den Lerninhalten meist nicht um Faktenwissen, sondern um Fertigkeiten und um die Umsetzung in vorzeigbare Produkte. Die Schüler müssen im Rahmen des Informatikunterrichts immer wieder die Chance bekommen, eigene Erfahrungen zu sammeln. Die Freude am Lernen und die Erfolgserlebnisse kommen vom selbstständigen Lösen der Aufgaben. Ein guter Informatiklehrer gibt seinen Schülern daher auch herausfordernde Aufgaben und genügend Zeit zur selbstständigen Lösung. Er zeigt Lösungsmöglichkeiten, aber nicht immer gleich die Lösung auf. [11]

6.1.5 Veranschaulichung

Die Veranschaulichung ist ein sehr wichtiger Part im Unterricht. Sie erleichtert den Schülern das Verstehen des Gelehrten. *Seibert* und *Serve* beschreiben Veranschaulichung wie folgt:

> Veranschaulichung ist das Bemühen des Lehrenden, einen Lerninhalt so aufzubereiten, dass bei aller Wahrung der Sachgemäßheit die Vorstellungsfähigkeit des Lernenden unterstützt wird, um zur intendierten Begriffsbildung zu gelangen. Die Anschaulichkeit der Unterrichtsgestaltung zielt auf Anschauung. Dies ist ein aktiver Prozess, der nur vom Lernenden vollzogen werden kann.

Veranschaulichung kann in vielerlei Hinsicht erreicht werden. In den Unterrichtsstunden in denen das Informatiklabor nicht beansprucht wird kann der Informatiklehrer beispielsweise einen Laptop zur Beispielvorführung mitnehmen, anstatt alles auf der Tafel aufzuzeichnen. Weiters können verschiedene Konzepte bildlich erklärt werden. Beispielweise die Veranschaulichung eines C-Strings.

Abbildung 6.1: Veranschaulichung eines C-Strings

So kann sich der Schüler besser vorstellen, wie das Wort "Hund" als C-String abgespeichert wird.

Das nächste Beispiel veranschaulicht einen dynamischen Zeiger `v` der auf ein Array mit drei Elementen zeigt. Das Array enthält die Integer-Zahlen `4`, `1` und `6`.

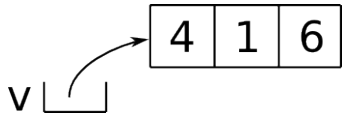

Abbildung 6.2: Veranschaulichung eines dynamischen Zeigers

Es ist empfohlen bereits existierende Regeln zu übernehmen. Zeiger werden nämlich häufig wie in der Abbildung 6.2 dargestellt. Somit ist beispielsweise sichergestellt, dass von einem anderen Autor übernommene Zeichnungen nicht völlig anders aussehen als jene des Lehrers.

Als Programms-Veranschaulichung kann ein Programm mit grafischer Benutzeroberfläche bezeichnet werden. Deshalb ist es wichtig den Schülern die grafische Programmierung beizubringen. Die Schüler werden nämlich privat noch wenig Erfahrung mit Konsolen-Programmen erlangt haben. In heutigen Betriebssystem erfolgt im Anwendungsbereich fast alles grafisch. Konsolen-Programme sind bei Programmieren oder Administratoren noch ein Begriff. Die Schüler wünschen sich deshalb sicherlich auch das Erstellen von grafischen

Programmen, da diese als "richtige" Programme angesehen werden. Grafisch kann in C++ im Schulunterricht jedoch erst programmiert werden, wenn die Schüler bereits reichlich Erfahrung in der textbasierten C++-Programmierung haben. Dafür wird reichlich Unterrichtszeit beansprucht. [12]

6.1.6 Variabilität, Flexibilität

Damit die Schüler dem Schulunterricht interessiert folgen, sie vom Lehrer nach ihren Fähigkeiten beschäftigt werden und der Lehrer auf verschiedenste Situationen spontan reagieren kann, muss der Schulunterricht so variabel und flexibel wie möglich gestaltet werden.
Ein Beispiel wie alle Schüler angemessen beschäftigt werden können ist fortgeschrittenen und interessierten Schülern weiterführende Literatur zu geben, mit der sie ihr Wissen vertiefen können. Mit weiterführender Literatur sind Blätter, Folien oder andere Medien gemeint, welche den gerade gelehrten Unterrichtsstoff vertiefen und nicht den Unterrichtsstoff von zukünftigen Kapiteln darstellen. Damit wird vermieden, dass fortgeschrittene Schüler sich später langweilen, wenn für sie ein Unterrichtskapitel bereits bekannt ist. [13]

6.2 Objektorientierte Programmierung

Das Konzept der Objektorientierten Programmierung ist erfunden worden um eine Programmiersprache programmtechnisch besser in Griff zu bekommen und sicherer zu machen. Deshalb kommt in der Objektorientierung ein Netzwerk von Begriffen vor, das erfordern kann, dass mehrere Konzepte gleichzeitig gelehrt werden müssen. Der Lehrer muss dabei darauf achten, dass die Schüler angemessen gefordert werden. Im Kapitel 5 *C++-Programmierung in der Schule* wird dabei eine mögliche Reihenfolge des Lehrens von Konzepten der Programmiersprache C++ beschrieben. [14]

6.3 Schultypen

6.3.1 Allgemeinbildende höhere Schule

In der allgemeinbildenden höheren Schule (AHS) ist Informatik in der 5. Klasse

Oberstufe ein zweistündiges Pflichtfach. In der 6. bis 8. Klasse AHS-Oberstufe kann Informatik als zweistündiges Wahlpflichtfach gewählt werden [15].
Will man C++ in dieser Schule einführen so muss im Informatikunterricht ein Jahr alleine diese Programmiersprache unterrichtet werden, damit die wichtigsten Konzepte erklärt werden können. Wird C++ in der 5. Klasse eingeführt so müssen viele wesentliche Bereiche [16] der Informatik der Programmierung weichen. Dies ist für diese Schule nicht ideal, weil es keine rein technische Schule ist. Wie bereits beschrieben ist Informatik in der 6. bis 8. Klasse der AHS ein Wahlpflichtfach. Dabei werden reichlich Programmiereinheiten angeboten [17]. Da C++ für die 5. Klasse nicht geeignet ist, muss beim Unterrichten von C++ im Wahlpflichtfach darauf geachtet werden, dass keine Programmierkenntnisse vorausgesetzt werden. Ohne Programmierkenntnisse müssen für das Lehren von C++ bei zwei Wochenstunden wohl mindestens ein bis eineinhalb Jahre beansprucht werden. Dabei ist das grafische Programmieren noch nicht miteinberechnet. Für recht einfache grafische Programme muss mindestens noch ein weiteres halbes bis ganzes Jahr für die C++-Programmierung beansprucht werden. Da der Wahlpflichtfächer-Katalog der AHS recht modular aufgebaut ist und die Fächer meist nur die Dauer eines Semesters haben, muss der Schüler entweder auf die grafische Programmierung verzichten oder sich fast ausschließlich für Programmierfächer entscheiden.
Für die AHS ist die Programmiersprache C++ deshalb nicht geeignet. Es gibt heute einige Programmiersprachen welche sich in der Schule etablieren, die einfacher lehrbar sind als C++. Je nachdem in welchem Bereich programmiert wird sind es Programmiersprachen wie Java, PHP oder Python. Es lassen sich damit schnell größere ansehnliche Programme schreiben.

6.3.2 Berufsbildende höhere Schule

Einige berufsbildende höhere Schulen wie beispielsweise die HTL sind auf Technik spezialisierte Schulen. Dabei gibt es sicherlich Unterschiede in Umfang des Informatikunterrichts und welche Programmiersprache verwendet wird. Technische Schulen die viel Informatik oder Elektronik unterrichten, setzen im

Allgemeinen keine Lehrprogrammiersprachen ein, sondern für die Lehrveranstaltung nützliche Programmiersprachen. Vor allem wenn Schulen Informatik und Elektronik unterrichten bieten sich dabei Programmiersprachen wie C oder C++ an, da sie sehr hardwarenahe Entwicklung erlauben. Die Schüler sind wohl auch größtenteils technisch interessiert, da sie sich für eine technische Schule entschieden haben. Dass sie also in C++ einige Zeit oder sogar die gesamte Schulzeit nur Konsolenprogramme schreiben wird für diese Schüler verkraftbar sein. Trotzdem muss überlegt werden, ob sich das Lehren von C++ lohnt oder stattdessen C gelehrt wird.
Da in Schulen wie beispielsweise im Ausbildungszweig *Mobile Computing / Software Engineering* (Abteilung Elektronik) der HTL Braunau in Braunau am Inn in Oberösterreich C++ gelehrt wird, hat C++ in technischen Schulen sicherlich seine Daseinsberechtigung.

6.3.3 Weitere Schulen

In jeder nicht technischen Schule ist schon wie in der AHS die C++-Programmierung nicht empfehlenswert. Falls es in manchen dieser Schulen überhaupt die Möglichkeit des Programmierens gibt, sind andere Programmiersprachen C++ vorzuziehen.

7 Schlusswort

C++ ist eine sehr beliebte Programmiersprache und in der Praxis in vielen Bereichen einsetzbar, da sie sehr vielseitig und sehr anpassbar ist. C++-Programme sind deshalb auch sehr leistungsstarke Programme.

Das Ziel der Schule ist jedoch nicht die Erzeugung fertiger Entwickler, welche besondere Fähigkeiten in einen bestimmten Programmierbereich haben, sondern Schülern das Konzept des Programmierens auf einfache Art und Weise zu vermitteln. Programmiersprachen, welche in den Schulen gelehrt werden, müssen deshalb nicht unbedingt in der Praxis weit verbreitet sein.

Wenn C++ im Unterricht gelehrt wird, so muss es als Einstiegsprogrammiersprache verwendet werden, da die Unterrichtszeit sonst nicht reicht. Im Schulunterricht ist C++ jedoch im Allgemeinen nicht als Einstiegsprogrammiersprache geeignet. Viele Gründe werden bereits im Kapitel 4 *Eigenschaften von C++* beschrieben. So haben beispielsweise Typen in C++ nicht auf jeder Rechenarchitektur die selbe Größe. Diese und weitere Eigenschaften sprechen in der Praxis oft für C++. In der Schule werden jedoch prinzipiell leicht lernbare Programmiersprachen verwendet. Eine speziell für den Unterricht erstellte Programmiersprache ist beispielsweise Pascal. In vielen Schulen werden heute außerdem die Programmiersprachen Java und PHP gelehrt. Auch Python wird als einfache Programmiersprache bezeichnet und erfreut sich momentan im Schulunterricht wachsender Beliebtheit [18].

In nicht-technischen Schulen ist das Lehren von C++ nicht geeignet. Die Gründe werden bereits im Kapitel 6.3.1 *Allgemeinbildende höhere Schule* genannt.

In technische Schulen, wie der HTL, kann der Einsatz von C++ sicherlich überprüft werden. Da es auch vorkommt, dass in einigen dieser Schulen C++ gelehrt wird hat diese Programmiersprache dort sicher seine Daseinsberechtigung. Vor allem weil es eine Hardware-nahe Programmiersprache ist, welche vor allem in der Elektronik eingesetzt werden können. Jedoch wird in diesen Schulen in C++ nicht grafisch programmiert werden, was technisch interessierte Schüler sicherlich verkraften können.

Allerdings muss in diesen Schulen überlegt werden, ob es Sinn macht C++ statt C zu lehren, wenn nicht grafisch programmiert wird. Zwar hat C++ sicherlich viele neuartige Konzepte, allerdings ist der Sprachumfang von C++ auch wesentlich größer als jener von C.
Die allgemeine Schlussfolgerung ist, dass obwohl C++ in der beruflichen Welt eine beliebte Programmiersprache ist, sie in der Schule seltenen für den Informatikunterricht geeignet ist.

8 Abkürzungsverzeichnis:

Abkürzung	Vollständige Bezeichnung	Beschreibung
AHS	Allgemeinbildende höhere Schule	Schule in Österreich
AT&T	American Telephone & Telegraph Corporation	Telekommunikationskonzern
CDT	C/C++ Development Tooling	Plugin der Entwicklungsumgebung Eclipse für C/C++-Programmierung
GCC	GNU Compiler Collection	Compiler
GIMP	GNU Image Manipulation Program	Bildbearbeitungsprogramm
GNU	GNU's Not Unix	Freies Betriebssystem
GTK bzw. GTK+	GIMP-Toolkit	Freie Komponentenbibliothek
gtkmm	gtk minus minus	C++-Programmbibliothek
GUI	Graphical User Interface	Grafische Benutzeroberfläche
HTL	Höhere Technische Lehranstalten	Schule in Österreich
ISO	isos (von gr. „gleich“)	Internationale Organisation für Normung
LGPL	GNU Lesser General Public License	Freie Lizenz
MinGW	Minimalist GNU for Windows	Programm-Entwicklungswerkzeuge
MSYS	Minimal SYStem	Softwareportierung der Unix-Shell auf die Windows-Plattform
Qt	Quasar toolkit	Klassenbibliothek
XML	Extensible Markup Language	Auszeichnungssprache für den Datenaustausch zwischen Computer-Systemen

9 Literaturverzeichnis

[1] http://www.purl.org/stefan_ram/pub/c++_lernsprache_de
letzter Aufruf: 21. April 2008

[2] P. Hubwieser: *Übung*. In: *Didaktik der Informatik*, 2000, S. 19

[3] http://www.parashift.com/c++-faq-lite/input-output.html#faq-15.1
letzter Aufruf: 23. April 2008

[4] http://www.math.uni-bayreuth.de/~rbaier/lectures/c_ss2003/html/node33.html
letzter Aufruf: 25. April 2008

[5] http://www.old.uni-bayreuth.de/departments/math/~rbaier/lectures/c/html/node107.html
letzter Aufruf: 2. Mai 2008

[6] http://www.parashift.com/c++-faq-lite/freestore-mgmt.html#faq-16.4
letzter Aufruf: 6. Mai 2008

[7] P. Hubwieser: *Lernpsychologische Fundierung*. In: *Didaktik der Informatik*, 2000, S. 3-5

[8] P. Hubwieser: *Prinzipien didaktischen Handelns*. In: *Didaktik der Informatik*, 2000, S. 15-17

[9] P. Hubwieser: *Prinzipien didaktischen Handelns*. In: *Didaktik der Informatik*, 2000, S. 17-18

[10] P. Hubwieser: *Prinzipien didaktischen Handelns*. In: *Didaktik der Informatik*, 2000, S. 19

[11] A. Reiter, G. Scheidl, H. Strohmer, L. Tittler, M. Weissenböck: *Didaktik der Informatik*. In: *Schulinformatik in Österreich*, 2003, S. 381

[12] P. Hubwieser: *Prinzipien didaktischen Handelns*. In: *Didaktik der Informatik*, 2000, S. 20-21

[13] P. Hubwieser: *Prinzipien didaktischen Handelns*. In: *Didaktik der Informatik*, 2000, S. 22

[14] A. Reiter, G. Scheidl, H. Strohmer, L. Tittler, M. Weissenböck: *Wie kann Objektorientierung in der Schule gelehrt werden?.* In: *Schulinformatik in Österreich*, 2003, S. 204-205

[15] http://www.peraugym.at/aktiv/it.html
letzter Aufruf: 16. Juli 2008

[16] http://www.peraugym.at/aktiv/informatik-5.html
letzter Aufruf: 16. Juli 2008

[17] http://www.ahs-rahlgasse.at/images/stories/schulprofil/WPF/inf.pdf
letzter Aufruf: 16. Juli 2008

[18] A. Reiter, G. Scheidl, H. Strohmer, L. Tittler, M. Weissenböck: *Sprachen.* In: *Schulinformatik in Österreich*, 2003, S. 373