

Praktikum Informatik 1

Einführung in die C++ Programmierung (SS 2023)



Prof. Dr.-Ing. habil. Jürgen Roßmann
Institut für Mensch-Maschine-Interaktion
Ahornstr. 55, 52074 Aachen

Inhaltsverzeichnis

Inhalt	i
Vorbereitung auf das Praktikum	1
Generelles	1
Scheinkriterien	1
Erfolgreiche Praktikumsteilnahme	2
1 Datenstrukturen und Operatoren	5
1.1 Motivation	5
1.2 Qualifikationsziele	5
1.3 Einführung in Eclipse	6
1.3.1 Einrichten der Arbeitsumgebung mit Eclipse	6
1.3.2 Erstellen des ersten Projektes	9
1.3.3 Compilieren	10
1.3.4 Ausführen	10
1.4 Testen und Debuggen	11
1.4.1 Erzeugung von debugfähigem Code	11
1.4.2 Debugger mit graphischer Menüführung	11
1.4.3 Funktionen eines Debuggers	12
1.4.4 Bedienung des Debuggers in Eclipse	12
1.5 Theorie	16
1.5.1 Zielsetzung und Einordnung	16
1.5.2 Datenobjekte und Datentypen	16
1.5.3 Operatoren	18
1.5.4 Typumwandlungen	20
1.5.5 Datenstrukturen	20
1.5.6 Blöcke	26
1.5.7 Gültigkeitsbereiche	26
1.6 Aufgaben	27
1.6.1 Datentypen und Typumwandlung	27
1.6.2 Strukturen	28
2 Kontrollstrukturen, Referenzen, Zeiger und Funktionen	29
2.1 Theorie	29
2.1.1 Zielsetzung und Einordnung	29
2.1.2 Einfache Kontrollstrukturen	29
2.1.3 Weitere Kontrollstrukturen: Schleifen	32
2.1.4 Referenzen und Zeiger	36
2.1.5 Speicherbereich und Sichtbarkeit	37
2.1.6 Dynamische Speichernutzung	37
2.1.7 Funktionen	37
2.1.8 Fortgeschrittenes Debuggen	43
2.1.9 Rekursion	46
2.2 Aufgaben	48
2.2.1 Rekursive Berechnung der Fibonacci-Zahlen	48
2.2.2 Iterative Berechnung der Fibonacci-Zahlen	48
2.2.3 Felder	48

3 Komplexere Projekte und Dokumentation	51
3.1 Theorie	51
3.1.1 Zielsetzung und Einordnung	51
3.1.2 Testgetriebene Entwicklung	51
3.1.3 Übersichtlicher und verständlicher Code	51
3.1.4 Dokumentation mit Doxygen	53
3.2 Aufgaben	54
3.2.1 Vorbereitung	55
3.2.2 Basisfunktionen	56
3.2.3 Die Ablaufsteuerung	58
4 Einführung in Klassen	61
4.1 Objektorientierte Programmierung	61
4.1.1 Objekte	61
4.1.2 Klassen	62
4.2 Klassen in C++	62
4.2.1 Methoden und Attribute	63
4.2.2 Konstruktoren und Destruktoren	65
4.2.3 Der this-Zeiger	67
4.2.4 Initialisierungslisten	68
4.2.5 Zugriffsbeschränkung	70
4.3 Parameter	71
4.3.1 Konstante Parameter und Call By Reference	71
4.3.2 Defaultparameter	72
4.4 Konstante Methoden	73
4.5 Include-Wächter	73
4.6 Aufgaben	74
5 Dynamische Datenstrukturen	77
5.1 Qualifikationsziele	77
5.2 Praktische Anwendungen	77
5.2.1 Einfach verkettete Liste	77
5.2.2 Doppelt verkettete Liste	78
5.2.3 Warteschlange	79
5.2.4 Stapelspeicher	79
5.3 Aufgaben	79
5.3.1 Einfache Studentendatenbank	79
6 Standard Template Library – STL	81
6.1 Templates	81
6.2 Standard Template Library – STL	82
6.2.1 Die Klasse <i>vector</i>	83
6.3 Einfache Datei Ein- und Ausgabe	85
6.3.1 Die <i>stream</i> -Klassen	86
6.3.2 Daten in eine Datei schreiben	86
6.3.3 Daten aus einer Datei lesen	87
6.3.4 Die Datei-Flags	88
6.3.5 Fehler abfangen	89
6.4 Aufgaben	90
6.4.1 Nutzung der C++-Referenz	90
6.4.2 STL	92
7 Überladung	95
7.1 Theorie	95
7.1.1 Überladen von Funktionen	95
7.1.2 Überladen von Operatoren	95

7.1.3	Algorithmen der STL	97
7.1.4	Überladung der Streamoperatoren	98
7.2	Aufgaben	100
7.2.1	Operatorenüberladung	100
7.2.2	STL - Funktionalität	100
8	Vererbung und Polymorphie	101
8.1	Vererbung — Hierarchien im Klassenkonzept	101
8.2	Polymorphismus	103
8.2.1	Späte Bindung	104
8.3	Initialisierungslisten	106
8.4	Abstrakte Klassen	106
8.5	Virtuelle Destruktoren	108
8.6	Zugriffsspezifizierer protected	108
8.7	Aufgaben	109
8.7.1	Erstellen der Unterklassen	110
8.7.2	Einführung von Ausleihbeschränkungen	111
8.7.3	Einführen einer rein virtuellen Funktion	112
8.7.4	Operatorüberladung zur Ausgabe aller ausgeliehenen Medien	112
8.7.5	Beispielhafte Ausgaben	112
9	GUI-Programmierung mit Qt	115
9.1	Qt	115
9.1.1	Eine erste Applikation	116
9.2	Klassen und Funktionen in Qt	117
9.2.1	Signals und Slots	117
9.2.2	QApplication	117
9.2.3	QDebug	117
9.2.4	QString	118
9.2.5	QMessageBox	118
9.2.6	QPushButton	119
9.2.7	QLabel	119
9.2.8	QLineEdit	120
9.2.9	QGraphicsView und QGraphicsScene	120
9.2.10	QMainWindow	121
9.2.11	QDialog	121
9.3	Statische Attribute und Methoden	122
9.4	Aufgaben	124
9.4.1	Eine neue Applikation	124
9.4.2	Datenstruktur und Darstellung	128
9.4.3	Karte bearbeiten	130
9.4.4	Karte einlesen	130
9.4.5	Einen Weg suchen	131
9.4.6	Wahlpflichtaufgaben	131
9.5	Zusammenfassung	133
Anhang		135
Literaturverzeichnis		137

Vorbereitung auf das Praktikum

Generelles

Dieses Praktikum soll Sie qualifizieren, programmiersprachenunabhängige, einfache Problemstellungen oder Verfahren (Algorithmen) auf Lösungsstrukturen abzubilden, deren Elemente die Programmiersprache C++ vorgibt. Dabei sollen Sie anhand der gewählten Versuche und Teilaufgaben lernen und erklären können, welchen Einfluss die Programmiersprache auf die Qualität der Lösung ausübt (u.a. Genauigkeit, Einschränkungen). Im Laufe dieses Praktikums werden Sie auch Techniken wie Operatorenüberladung, Vererbung innerhalb von Klassen oder den Gebrauch der *Standard Template Library* kennenlernen, die für effiziente Lösungen erforderlich sind. Um Ihnen die Installation der Entwicklungsumgebungen auf privaten Rechnern zu erleichtern, finden Sie im RWTMoodle eine Installationsanleitung sowie ein Testverfahren zur Verifikation, ob die Installation der einzelnen Komponenten gelungen ist. Bitte beachten Sie, dass aus organisatorischen Gründen nur während der ersten Praktikumswoche Hilfestellungen zur Installation der Entwicklungsumgebung geleistet werden können.

Noch vor dem ersten Versuch, und damit auch vor dem ersten Termin zum betreuten Programmieren, sollten Sie sich mit der Entwicklungsumgebung vertraut machen und sicherstellen, dass diese wie gewünscht funktioniert.

Scheinkriterien

Zur Bestätigung der erfolgreichen Praktikumsteilnahme ist die Bestätigung über die erfolgreiche Durchführung von so genannten Testaten notwendig. Während des Praktikums wird es zwei Termine geben, an denen Sie von einem Praktikumsbetreuer mündlich über die Inhalte des vorigen Versuchsblocks geprüft werden. Hierbei stellen die Versuche 1 bis 5 den ersten Versuchsblock dar, die Versuche 6 bis 9 den zweiten Versuchsblock. Werden die Fragen während eines Testats korrekt beantwortet, erhalten Sie hierüber eine Bestätigung. Die Qualität der Antworten und der Implementierung entscheidet dabei über die Bewertung des Testats. Das Ergebnis wird auf der Testatkarte, welche Ihnen während des ersten Testats ausgehändigt wird, sowie zusätzlich in einer zentralen Liste festgehalten.

Anmeldung zum Praktikum

Um am Praktikum Informatik 1 teilnehmen zu können, müssen Sie zwei verschiedene Anmeldungen über RWTHonline durchführen. Erstens, Sie müssen sich zu einem Termin des Praktikums anmelden (Mo1, Mo2, Mo3, Di1, Di2, Di3, Mi1, Mi2, Mi3). Dieser Termin legt fest, zu welchem Termin Sie zu den Testaten erscheinen müssen. Zweitens, Sie müssen sich zur Prüfung des Praktikum Informatik 1 anmelden, dies ist später wichtig für die Übermittlung der Credits an das ZPA. Diese Prüfung ist rein organisatorischer Natur, da nur für erbrachte Prüfungsleistungen Credits eingetragen werden können. Darüber hinaus haben Sie dadurch die Möglichkeit, sich eigenständig in RWTHonline von dieser Prüfung abzumelden, wenn Sie am Praktikum Informatik 1 doch nicht mehr teilnehmen wollen. Eine Abmeldung vom Praktikum selbst bei der Fachstudienberatung ist dann nicht mehr notwendig. Eine Abmeldung von der Prüfung ist bis drei Tage vor dem zweiten Testat möglich. Für alle Fristen bezüglich der Anmeldungen und Abmeldungen schauen Sie bitte in RWTHonline nach.

Abnahme und Bewertung eines Aufgabenblocks

Die Testate finden in zwei Wochen im Semester im CIP-Pool statt (15. Mai - 17. Mai und 3. Juli - 5. Juli). Testate sind Prüfungstermine, d.h. während dieser beiden Wochen besteht Anwesenheits- und

Erfolgreiche Praktikumsteilnahme

Abnahmepflicht. Suchen Sie sich zu Beginn des Praktikumstermins einen Sitzplatz und tragen Sie sich anschließend mit Name und Sitzplatznummer in der vorne am CIP-Pool Info-Desk ausliegenden Liste zum Testat ein. Sollten Sie einen Testattermin krankheitsbedingt verpassen, so ist eine Wiederholung des Testates nur gegen Vorlage eines Attests möglich (vorzulegen im CIP-Pool, nicht beim ZPA). Der Nachholtermin findet zum nächsten stattfindenden Praktikumstermin statt (Beispiel: Sind Sie bis einschließlich Dienstag krankgeschrieben, findet ihr Nachholtermin am Mittwoch um 12:15 Uhr statt). Eine vorzeitige Abnahme der Testate ist innerhalb der ersten 3 Wochen des jeweiligen Testat-Blocks möglich.

Bei der Abnahme müssen Sie die Programme des entsprechenden Aufgabenblocks vorführen sowie die Fragen des Betreuers beantworten können. Dabei gehen folgende Punkte in die Bewertung ein:

- Korrekte Umsetzung der Programme
- Aufbau und Kommentierung des Quellcodes
- Beantwortung von Verständnisfragen und Fragen zur Umsetzung der Aufgabenstellungen, die überprüfen sollen, ob Sie die Aufgaben selbstständig gelöst haben
- Fragen zu Inhalten aus dem Skript

Der Bewertung wird folgendes Schema zu Grunde gelegt:

- 3 Punkte: Inhalt des Skripts verstanden; korrekte Umsetzung der Aufgaben; gute Kommentierung; gute Erklärung des Quellcodes
- 2 Punkte: Inhalt des Skripts verstanden; kleine Fehler bei der Umsetzung; Erklärung des Quellcodes ist befriedigend
- 1 Punkt: Inhalt des Skripts nur teilweise verstanden; grobe Mängel bei der Implementierung; das Konzept zur Lösung der Aufgaben ist korrekt
- 0 Punkte: Ungenügende Kenntnisse zur Lösung des Aufgabenblocks; Lösung entspricht nicht der aktuellen Aufgabenstellung; Aufgabenblöcke bauen nicht aufeinander auf; Programme laufen nicht; Programme wurden nicht selbstständig erstellt. Das Testat gilt in diesem Fall als nicht bestanden (siehe Täuschungsversuche).

Erfolgreiche Praktikumsteilnahme

Zum Bestehen des Praktikums müssen folgende Kriterien erfüllt werden:

- Gesamtpunktzahl ≥ 4
- Bei Nichtbestehen (0 Punkte) eines Testats gilt das Praktikum als nicht bestanden.

Sind Sie Ihren Abnahmepflichten gerecht geworden und haben alle Aufgaben erfolgreich gelöst, gilt das Praktikum als bestanden. Eine Anwesenheitspflicht besteht nicht, es wird jedoch empfohlen, regelmäßig zu den Betreuungsterminen zu kommen. Bitte beachten Sie, dass während der Testate Anwesenheitspflicht besteht.

Zusatzaufgaben

An einigen Stellen im Skript werden Sie Zusatzaufgaben finden. Diese sind als freiwillig gekennzeichnet. Für das Bestehen der Testate sind die Zusatzaufgaben nicht notwendig. Auch für die volle Punktzahl (3 Punkte) sind die Zusatzaufgaben nicht notwendig. Die Zusatzaufgaben gehen etwas über die Anforderungen der normalen Aufgabenstellung hinaus, vermitteln jedoch interessante weitere Konzepte. Daher wird sich ein Blick auf diese lohnen.

Täuschungsversuche

Als Täuschungsversuch wird gewertet, wenn bei einer Abnahme eine Lösung vorgestellt wird, die offensichtlich nicht mit der aktuellen Aufgabenstellung übereinstimmt oder bei der die einzelnen Aufgabenblöcke nicht aufeinander aufbauen, sowie solche Lösungen, die offensichtlich nicht selbst erstellt wurden oder die Lösung nicht dem Kenntnisstand entspricht. In diesem Fall wird das Testat mit 0 Punkten bewertet. Eine Wiederholung ist nicht möglich.

Semesterübersicht

Um Ihnen einen Überblick über das Semester zu ermöglichen, sind in dem Kalender auf Seite 4 alle Termine dieses Praktikums eingetragen. Die Wochen, an denen Testate stattfinden, sind in der Übersicht mit "Testat 1" und "Testat 2" für das erste bzw. das zweite Testat gekennzeichnet. Die Angaben zu den Versuchen (V1 bis V9) hinter den Terminen geben eine grobe Orientierung zum Arbeitsaufwand der einzelnen Versuche.

Erfolgreiche Praktikumsteilnahme

	April		Mai		Juni		Juli	
1	Sa	Semesterbeginn	1	Mo	Maifeiertag	1	Do	Exkursionswoche
2	So		2	Di	Di2, Di3 (Dies ab 14:30)	2	Fr	Exkursionswoche
3	Mo	Vorlesungsbeginn	3	Mi	Mi1, Mi2, Mi3 (V3-V4)	3	Sa	
4	Di		4	Do		4	So	
5	Mi		5	Fr		5	Mo	Mo1, Mo2, Mo3 (V7-V8)
6	Do		6	Sa		6	Di	Di1, Di2, Di3 (V7-V8)
7	Fr	Kaffreitag	7	So		7	Mi	Mi1, Mi2, Mi3 (V7-V8)
8	Sa		8	Mo	Mo1, Mo2, Mo3 (V4-V5)	8	Do	Fronleichnam
9	So		9	Di	Di1, Di2, Di3 (V4-V5)	9	Fr	
10	Mo	Ostermontag	10	Mi	Mi1, Mi2, Mi3 (V4-V5)	10	Sa	
11	Di	Di1, Di2, Di3 (V1-V2)	11	Do		11	So	
12	Mi	Mi1, Mi2, Mi3 (V1-V2)	12	Fr		12	Mo	Mo1, Mo2, Mo3 (V8-V9)
13	Do		13	Sa		13	Di	Di1, Di2, Di3 (V8-V9)
14	Fr		14	So		14	Mi	Mi1 (V8-V9)
15	Sa		15	Mo	Testat 1 - Mo1, Mo2, Mo3 (V6)	15	Do	
16	So		16	Di	Testat 1 - Di1, Di2, Di3 (V6)	16	Fr	
17	Mo	Mo1, Mo2, Mo3 (V2-V3)	17	Mi	Testat 1 - Mi1, Mi2, Mi3 (V6)	17	Sa	
18	Di	Di1, Di2, Di3 (V2-V3)	18	Do	Christi Himmelfahrt	18	So	
19	Mi	Mi1, Mi2, Mi3 (V2-V3)	19	Fr		19	Mo	Mo1, Mo2, Mo3 (V9)
20	Do		20	Sa		20	Di	Di1, Di2, Di3 (V9)
21	Fr		21	So		21	Mi	Mi1, Mi2, Mi3 (V9)
22	Sa		22	Mo	Mo1, Mo2, Mo3 (V6-V7)	22	Do	
23	So		23	Di	Di1, Di2, Di3 (V6-V7)	23	Fr	
24	Mo	Mo1, Mo2, Mo3 (V3)	24	Mi	Mi1, Mi2, Mi3 (V6-V7)	24	Sa	
25	Di	Di1, Di2, Di3 (V3)	25	Do		25	So	
26	Mi	Mi1, Mi2, Mi3 (V3)	26	Fr		26	Mo	Mo1, Mo2, Mo3 (V9)
27	Do		27	Sa		27	Di	Di1, Di2, Di3 (V9)
28	Fr		28	So		28	Mi	Mi1, Mi2, Mi3 (V9)
29	Sa		29	Mo	Exkursionswoche	29	Do	
30	So		30	Di	Exkursionswoche	30	Fr	
			31	Mi	Exkursionswoche			31 Mo

Über kurzfristige Änderungen werden Sie über RWTTHonline und RWTThmoodle informiert.

1 Datenstrukturen und Operatoren

1.1 Motivation

Die Entwicklung eines Programms erfordert zwei grundlegend verschiedene Arbeitsphasen: die Modellierung und die Programmierung. Die Modellierung umfasst die Planung, Problemanalyse und den groben Entwurf der Softwarearchitektur (z.B. mit der Unified Modeling Language (UML)). Sie wird umso wichtiger, je komplexer das zu lösende Problem ist.

Die darauf folgende Programmierung beschreibt dann die Umsetzung der ausgearbeiteten Problemlösung in eine Programmiersprache. Einerseits ist natürlich der Modellierungsschritt auf die Art der Sprache (funktional, objektorientiert, usw.) abgestimmt, andererseits kommt es aber auch oft dazu, dass während der Programmierung Details im Modell überarbeitet werden müssen.

Beide Schritte könnte man prinzipiell mit wenigen Arbeitsmitteln bewältigen (Papier und Stift zum Modellieren, Editor und Compiler/Linker zum Programmieren). Der Einsatz spezieller Programme, welche den Programmierer während seiner Aufgabe so gut wie möglich unterstützen, erlauben aber einen weitaus effizienteren Entwicklungsprozess. Ein klassisches Beispiel ist die Benutzung einer *integrierten Entwicklungsumgebung* (Abkürzung *IDE*, von Integrated Development Environment). Eine IDE verfügt normalerweise über folgende Komponenten:

1. Einen Texteditor mit Syntax-Highlighting (farbliches Hervorheben besonderer Sprachelemente),
2. eine direkte Anbindung des Compilers und Linkers inklusive Anzeige der gefundenen Fehler direkt im Code,
3. einen Debugger und
4. eine Projektverwaltung, welche es ermöglicht, zusammengehörige Dateien, Verzeichnisse und Einstellungen kombiniert abzuspeichern.

In erster Linie sind IDEs hilfreiche Werkzeuge, die dem Software-Entwickler häufig wiederkehrende Aufgaben abnehmen und einen schnellen Zugriff auf wichtige Funktionen bieten. Der Entwickler kann sich dadurch ganz auf seine eigentliche Aufgabe, die Programmierung, konzentrieren.

In diesem Praktikum werden Sie eine der bekanntesten Entwicklungsumgebungen zum Programmieren (*Eclipse*) sowie das Plugin zur C/C++ Entwicklung (*CDT - C++ Development Tools*) kennenlernen.

1.2 Qualifikationsziele

In diesem Versuch werden alle Schritte aufgezeigt, um mittels *Eclipse*¹ C++ Programme zu entwickeln. Diese Schritte umfassen den vollen Programmierprozess:

- Anlegen eines neuen Projektes
- Programmierung, Verwaltung und Navigation im Programm
- Compilieren, Linken und Debuggen

Eclipse ist auf jedem Rechner des Rechner-Pools installiert.

¹ Andere IDE's sind erlaubt sofern sie den selben Funktionsumfang bieten. Bitte beachten Sie, dass für andere IDE's ggfs. keine Unterstützung der Betreuer bei der Installation/Konfiguration erfolgen kann.

1.3 Einführung in Eclipse

Neben einer Einführung in Eclipse zeigt dieser Versuch beispielhaft, welche Vorteile sich durch eine IDE ergeben. Da die Größe des Programms durch die kurze Versuchsdauer beschränkt ist, erscheinen viele Möglichkeiten der IDE als nett, aber nicht lebensnotwendig, z. B. die Integration des Versions-Management-Systems. Der Sinn dieser Hilfen wird erst bei großen Softwareprojekten, in denen mehrere Personen (an verschiedenen Orten) über mehrere Monate an einer Software arbeiten, ersichtlich.

Alle Projekte und die dazugehörigen Einstellungen und Dateien werden in einem Ordner namens *workspace* gespeichert; das dazugehörige Verzeichnis fragt Eclipse bei jedem Programmstart ab.

Wichtig: Wenn Sie an den Rechnern im CIP-Pool arbeiten, achten Sie unbedingt darauf, dass Ihr *workspace* auf dem Laufwerk U:\ liegt, siehe 1.1. Auf dem Laufwerk C:\ steht Ihnen nicht genügend Speicher zur Verfügung! Bestätigen Sie die Einstellung mit Ok.

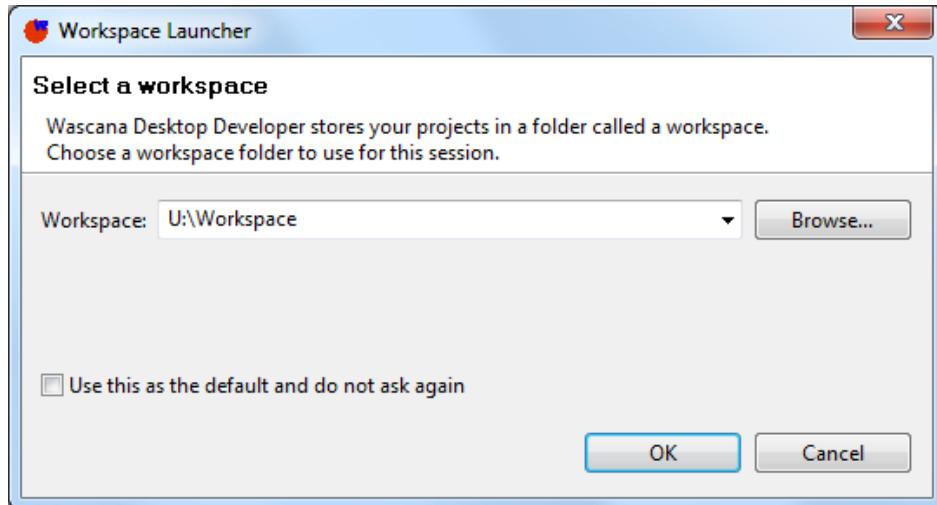


Abbildung 1.1: Einstellen des Verzeichnisses des Workspace

Im darauf folgenden Willkommen-Bildschirm wählen Sie direkt rechts oben den Pfeil aus: *Workbench*.

In Eclipse heißt die äußere Erscheinung der Benutzeroberfläche, die Anordnung und der Inhalt der einzelnen Fenster Perspektive. Diese wechselt, je nachdem, was Sie gerade tun. Sie sieht im Editiermodus anders aus als im Debuggermodus. In der sich öffnenden Perspektive sehen Sie unter anderem

- In der Mitte einen Editor, der speziell an C++ angepasst ist, d. h. er unterstützt Syntax-Hervorhebung, automatisches Ergänzen uvm.
- Links eine vollständige Integration der Projektverwaltung, also eine Zusammenfassung aller Dateien eines Projektes und automatisches Kompilieren und Linken des gesamten Projektes.
- Rechts eine einfache und schnelle Navigation durch das geschriebene Programm durch einen internen Index aller Deklarationen, Definitionen, Variablen, Instanzen und Funktionsaufrufe.
- Unten diverse Fenster, in denen Eclipse selbst Informationen zur Verfügung stellt, unter anderem eine Konsolenausgabe für Ihre Programme.

1.3.1 Einrichten der Arbeitsumgebung mit Eclipse

Die folgenden Einstellungen und Beschreibungen können Ihnen das Arbeiten mit Eclipse deutlich erleichtern und helfen, Fehler oder unnötigen Aufwand zu vermeiden. Nehmen Sie sich im eigenen Interesse die Zeit, ihre Arbeitsumgebung einzurichten.

Codevorlagen

Im CIP-Pool finden Sie die Codevorlagen auf dem Laufwerk *P*: unter

P:\UserGrp\PI1\Vorlagen

Ansonsten laden Sie die Zip-Datei aus dem RWTHmoodle-System herunter und entpacken Sie sie in ein eigenes Verzeichnis, nicht in den Workspace von Eclipse. Für jeden Versuch gibt es einen eigenen Unterordner mit den Vorlagen.

Vergessen Sie nicht, beim Importieren die Option *Copy files* auswählen, das ist wichtig.

Startverhalten (*Launching*) des Programms konfigurieren

Diese Einstellung kann Ihnen viel Kopfzerbrechen ersparen. Wenn Ihre Anwendung noch läuft, z.B weil sie auf eine Eingabe wartet, und Sie editieren in der Zwischenzeit Ihr Programm, weil Ihnen ein Fehler aufgefallen ist, kann Eclipse keine neue, ausführbare Version erstellen, da die alte Version noch ausgeführt wird.

Unter Windows ist es nicht möglich, geöffnete, ausführbare Dateien zu löschen. Der Compiler erstellt zwar eine neue Version Ihres Programms, aber der Linker, der die ausführbare Version erstellt, kann die alte nicht löschen, und damit keine neue Version erstellen. In der Konsole sehen Sie dann auch eine entsprechende Fehlermeldung und es öffnet sich ein Fenster, das auf den Fehler hinweist und fragt, ob Sie trotzdem starten (*Launch*) wollen. Wenn Sie jetzt auf *Launch* klicken, wird die vorhandene, also die alte Version, wieder gestartet, und Ihr Programm läuft dann zweimal. Und Sie wundern sich, warum Ihre Änderungen nicht angewendet wurden.

Das können Sie beliebig oft fortsetzen, und Ihre Änderungen erscheinen trotzdem nicht. Um dies zu vermeiden, nehmen Sie folgende Einstellung vor:

Window → Preferences → Run/Debug → Launching

Wählen Sie unten *Launch the associated project* und setzen Sie darunter das Häkchen bei *Tminate and Relaunch while launching(...)*. Stellen Sie sicher, dass etwas weiter oben unter *General Options* der Punkt *Build (if required) before launching* ausgewählt ist, dann *Apply*, wenn Sie noch mehr auf einmal konfigurieren wollen, oder *Apply and Close*.

Jetzt wird Eclipse zuerst Ihr Programm beenden, bevor dies neu kompiliert und dann ausgeführt wird.

Automatisches Speichern

Es ist sehr nützlich, Eclipse so einzustellen, dass alle Dateien automatisch gespeichert werden, bevor eine ausführbare Datei erstellt wird. Dies verhindert eine langwierige Fehlersuche, verursacht durch eine nicht gespeicherte Datei.

Gehen Sie hierzu in die Eclipse-Einstellungen:

Window → Preferences → General → Workspace → Build

Aktivieren Sie die automatische Speicherfunktion: *Save automatically before build*.

Eine Ebene höher direkt unter *Workspace* können Sie auch zusätzlich das Intervall festlegen, in dem Eclipse automatisch geöffnete Dateien im Hintergrund sichert. Geben Sie die Anzahl in Minuten unter *Workspace save interval* an, dann *Apply* oder *Apply and Close*.

MinGW als Default-Compiler

Stellen Sie den MinGW GCC als Default-Compiler ein.

Gehen Sie hierzu in die Eclipse-Einstellungen:

Window → Preferences → C/C++ → New C/C++ Projekt Wizard

Aktivieren Sie im Tab *Preferred Toolchains* die Punkte *Empty Project* und *MinGW GCC*, dann *Apply* oder *Apply and Close*.

Importieren von Codevorlagen

Im Laufe des Praktikums bekommen Sie oft Codevorlagen zur Verfügung gestellt, die Sie dann in ein leeres Projekt importieren und um eigenen Code ergänzen sollen.

So erstellen Sie ein leeres Projekt:

- *File → New → C++ Project → C++ Managed Build → Next*
- Tragen Sie den Projektnamen ein und wählen Sie *Empty Project* und *MinGW GCC*, dann *Finish*.

Nun können Sie die Vorlagen importieren.

- Im Windows-Explorer in das Vorlagenverzeichnis wechseln, zu importierende Dateien markieren und in Eclipse auf den Projektnamen ziehen.
- Im sich öffnenden Fenster die Option *Copy files* auswählen, dann *OK* (wichtig).

Die Codedateien werden dann automatisch in das Projektverzeichnis kopiert.

Code automatisch formatieren

Eclipse bietet dem Programmier die Möglichkeit, den Code nach verschiedenen Vorgaben (*Code-Convention*) automatisch zu formatieren. Dazu müssen Sie zuerst die richtige Vorlage auswählen. Die für das Praktikum geltende Code-Convention werden Sie noch im Kapitel 3 kennenlernen, Eclipse verfügt über die entsprechende Vorgabe. Gehen Sie wie folgt vor:

- *Window → Preferences → C/C++ → Code Style → Formatter*
- Wählen Sie unter *Active Profile* die Option *BSD/Allman [built-in]* aus. Im Fenster darunter sehen Sie ein Beispiel, wie Ihr Code formatiert wird.
- Dann wieder *Apply* oder *Apply and Close*.

Mit *Strg-Shift-F* im Texteditor öffnet sich ein Fenster, in dem Sie die ganze Datei oder den markierten Text auswählen können.

Perspektive wiederherstellen

Wenn Ihnen die Perspektive durcheinander geraten ist oder Fenster versehentlich verschwunden sind, können Sie die ursprüngliche Perspektive wie folgt wiederherstellen:

Window → Perspective → ResetPerspective

Aktive Zeile im Editor farbig unterlegen und Tabulatorgröße setzen

Damit Sie schneller sehen können, an welcher Stelle sich Ihr Cursor im Moment befindet, können Sie die aktive Zeile hervorheben.

- *Window → Preferences → General → Editors → Text Editors*
- Wählen Sie im unteren Fenster *Current line highlight* und rechts daneben die Farbe.
- Tragen Sie im Feld *Displayed tab width* "4" ein.
- Wählen Sie zusätzlich den Punkt *Insert spaces for tabs* aus.
- Dann wieder *Apply* oder *Apply and Close*.

Unterstützung für Doxygen (optional)

Eclipse kann Sie bei der Erstellung der Dokumentation mit *Doxygen* unterstützen. Das wird ab Kapitel 3 interessant. Doxygen können Sie ggfs. auch nachträglich über den Eclipse-Marketplace installieren. Wählen Sie bei

Window → Preferences → C++ → Editors

unten rechts für *Workspace default* die Einstellung *Doxygen*. Wie Sie diese Einstellung nutzen, erfahren Sie in Kapitel 3.

Verschiedenes

Wenn Sie einen ganzen Block auskommentieren wollen, um z.B. zu testen, ob ein Fehler in diesem Block ist, geht das recht einfach. Block markieren, dann *Strg-Shift-7*. Wenn Sie alles wieder einkommentieren wollen, genau so. Und genau so können Sie ganze Blöcke einrücken und wieder ausrücken, indem Sie *Tab* bzw. *Shift-Tab* drücken.

Eine Liste aller Tastaturbefehle bekommen Sie mit *Strg-Shift-L* angezeigt.

1.3.2 Erstellen des ersten Projektes

Um mit Hilfe von Eclipse eine ausführbare Datei zu erstellen, legen Sie zunächst ein neues *C++ Project* an, indem Sie im Menü *File → New → C++ Project* wählen. Wählen Sie im folgenden Fenster *C++ Managed Build*, dann *next*. Geben Sie im Wizard den Projektnamen („HelloWorld“) ein und wählen Sie als Projekttyp *Executable, Empty Project*. Da für dieses Praktikum der *MinGW* Compiler konfiguriert wurde, wählen Sie die Toolchain *MinGW GCC*. Beenden Sie den Wizard² mit *Finish* und in der Projektübersicht links sollte das leere *C++-Projekt HelloWorld* erscheinen, das nur die Includes der gewählten Toolchain enthält.

Mit *File → New → SourceFile* können Sie in diesem Projekt eine neue Quelldatei anlegen. Als Namen geben Sie bitte *main.cpp* an. Daraufhin öffnet sich ein leeres Editorfenster. Nun können Sie als Erstes das praktische Auto-Vervollständigen testen: Geben Sie einfach nur *main* ein und drücken Sie *Strg+Leerzeichen*. Daraufhin vervollständigt der Editor selbstständig zum Standard-Kopf eines *C++* Programmes!

Falls der Editor mehrere Möglichkeiten zur Vervollständigung kennt, erscheint ein kleines Fenster, mit dem Sie dann den passenden Code auswählen, bzw. mit Escape (Esc) abbrechen können. Geben Sie nun folgendes, kleines Programm ein. Probieren Sie währenddessen ruhig das automatische Vervollständigen aus, z. B. bei der Eingabe von “if” oder der “for”-Schleife.

```

1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!" << std::endl;
6
7     int max;
8     max = 10;
9
10    for (int var = 0; var < max; ++var)
11    {
12        std::cout << var << std::endl;
13    }
14
15    return 0;
16 }
```

²Eclipse bietet weitere Möglichkeiten zur Konfiguration des Projekts und der Toolchain an. Falls Sie schon einmal mit Make/g++ programmiert haben, sind die Einstellungen unter *Advanced Settings* sicherlich interessant; für den Anfang reichen jedoch die Standardeinstellungen aus.

1 Datenstrukturen und Operatoren

Die Zeile 1 wird später erklärt und kann hier ignoriert werden (siehe Kapitel 1.5.3). Das Programm zählt mit Hilfe einer for-Schleife von 0 bis 9 und gibt diese Zahlen aus.

Vergessen Sie nicht, am Schluss zu speichern, indem Sie auf das Diskettensymbol in der Schnellzugriffsleiste klicken, sofern Sie das Autospeichern nicht aktiviert haben. Am rechten Bildschirmrand unter *Outline* sehen Sie eine kleine Zusammenfassung des aktuellen Programmes, also die Includes und die geschriebenen Funktionen (hier z.B. *main*). Diese Zusammenfassung stellt eine der hilfreichen Navigationsfunktionen dar, welche natürlich erst in etwas größeren Projekten richtig zum Tragen kommt.

1.3.3 Compilieren

Die Entwicklungsumgebung Eclipse bringt zwar umfangreiche Quellcodeeditoren mit, bietet jedoch keine Toolchain, sodass Eclipse nicht in der Lage ist, Code eigenständig zu kompilieren. Hierzu wird stets ein externer Compiler benötigt, der von Eclipse aus genutzt werden kann. Der Grund dafür ist simpel. Für die Programmiersprache C++ existieren viele verschiedene Toolchains, und darüber hinaus existieren viele verschiedene Zielsysteme, für die C++ -Programme kompiliert werden können. Die Wahl des Compilers liegt demnach beim Entwickler. Im Rahmen des Praktikums wurde der Compiler MinGW gewählt.

Durch Anklicken des Hammers in der Schnellzugriffsleiste starten Sie das Compilieren und Linken des gesamten Projektes. Im unteren Fenster sollten Sie nun unter der Rubrik *Problems* die Statusmeldung "0 items" sehen, ansonsten ist Ihnen vermutlich ein Tippfehler unterlaufen. Unter der Rubrik *Console* finden Sie auch Informationen darüber, was Eclipse bzw. CDT im Hintergrund für Sie ausgeführt hat:

```
**** Build of configuration Debug for project HelloWorld ****
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "..\\main.cpp"
g++ -o HelloWorld.exe main.o
```

Build Finished

Hier können Sie gut die zwei verschiedenen Schritte zu einem ausführbaren Programm erkennen: Als Erstes wird jede einzelne Quelldatei des Projektes – also hier nur *main.cpp* – mit dem Befehl

```
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "..\\main.cpp"
```

kompliert, also in Maschinensprache übersetzt. Es wird die Objekt-Datei *main.o* erzeugt. Im zweiten Schritt werden mittels

```
g++ -o HelloWorld.exe main.o
```

sämtliche benötigten Objekt-Dateien des Projektes zur Datei *HelloWorld.exe* zusammengefügt, welche das ausführbare Programm darstellt.

Falls beim Compilieren oder Linken Fehler aufgetreten sind, können Sie die Fehlerbeschreibung und die dazugehörige Zeile im Quelltext unter *Problems* anzeigen. Fügen Sie einen absichtlichen Fehler in das Programm ein (z. B. durch das Löschen eines Semikolons) und beobachten Sie das Ergebnis!

1.3.4 Ausführen

Falls Ihr Programm fehlerfrei ist, können Sie es mit einem Klick auf den grünen Kreis mit dem weißen Dreieck ausführen. Im unteren Bereich sollten Sie jetzt unter der Konsole die richtige Ausgabe von *HelloWorld* sehen können:

```
Hello World!
0
1
2
3
4
5
6
7
8
9
```

1.4 Testen und Debuggen

Jeder Softwareentwickler ist bemüht, fehlerfreien Code zu erzeugen. Der eingesetzte Compiler kann zwar Syntax-Fehler erkennen, jedoch keine semantischen Fehler. Daher kann man in der Regel selbst in stabil laufender und getesteter Software noch etwa 1-2 Fehler pro 1000 Zeilen Code finden. Semantische Fehler äußern sich durch unerwartetes Verhalten der Applikation und sind im Programmcode schwer zu lokalisieren. Zur Erleichterung dieser Suche eignet sich der Einsatz eines so genannten Debuggers.

Das Wort “Debuggen” bezeichnet die Beseitigung von Fehlern (der englische Begriff *bug* entspricht im Kontext der Programmierung einem Fehler in der Software). Debugger erlauben es, die einzelnen Schritte eines Programms und ihre Auswirkungen auf den Datenspeicher im laufenden Betrieb näher zu analysieren. Dabei ist es z.B. möglich, Programme an bestimmten Stellen anzuhalten, um den Zustand verschiedener Variablen zu überprüfen.

1.4.1 Erzeugung von debugfähigem Code

Ein Debugger ermöglicht es, den Ablauf eines Programms an einer beliebigen Stelle zu unterbrechen. Dies wird durch das Setzen eines so genannten Haltepunktes ermöglicht, der mit Hilfe des Debuggers an gewünschten Codezeilen platziert werden kann. Der Entwickler kann das angehaltene Programm danach Befehl für Befehl in Einzelschritten fortführen und sein Verhalten analysieren. Die eigentliche Fehlerbeseitigung erfolgt jedoch nicht im Debugger selbst, denn dazu muss meist der Code verändert und das Programm neu compiliert werden.

Der Debugger benötigt für den Eingriff in den normalen Programmablauf Zusatzinformationen, die durch spezielle Compileroptionen automatisch generiert werden. Unter Eclipse sind diese Compileroptionen durch die aktive “Build Configuration” bestimmt, welche im Menü unter *Projekt → Build Configurations → Set Active* gesetzt werden kann. Per default gibt es hier zwei Möglichkeiten: “Debug” und “Release”.

Den Unterschied können Sie in der Konsolen-Ausgabe während des Compilierens erkennen:

- Debug: g++ -g3 ...
- Release: g++ -O3 ...

Das Flag **-g** bestimmt die Menge der erzeugten Debug-Zusatzinformationen, das Flag **-O** den Grad der Code-Optimierung. Die Zusatzinformationen erhöhen die Größe der ausführbaren Datei und verzögern den normalen Programmablauf. Daher sollte man bei der Erzeugung der finalen Version (*Release*-Version) eines Programms auf diese Informationen verzichten.

1.4.2 Debugger mit graphischer Menüführung

Der integrierte Debugger wird ähnlich wie das Ausführen des Programms mit einem Klick auf den kleinen Käfer links neben dem Startsymbol gestartet. Da der Debugger eine andere Perspektive als die zur C++ Entwicklung besitzt, sollte die folgende Frage, ob diese geöffnet werden soll, mit *Yes* beantwortet werden.

1.4.3 Funktionen eines Debuggers

Jeder Debugger besitzt einige typische Funktionen. Bei unterschiedlichen Debuggern können diese durchaus unterschiedlich bezeichnet sein, in der Arbeitsweise gleichen sie sich jedoch. Die im Folgenden dargestellten Bezeichnungen orientieren sich an der Benennung im *GDB*.

Haltepunkte

Möchte der Entwickler den Zustand eines Programms an einer bestimmten Stelle im Ablauf analysieren, dann setzt er vor der Ausführung an dieser Stelle im Code einen Haltepunkt (*Breakpoint*) (Abbildung 1.2). Erreicht das Programm während seiner Ausführung diese Stellen, dann unterbricht der Debugger den Ablauf. Nun lassen sich mit Hilfe des Debuggers die Werte der lokalen und globalen Variablen anzeigen und verändern.

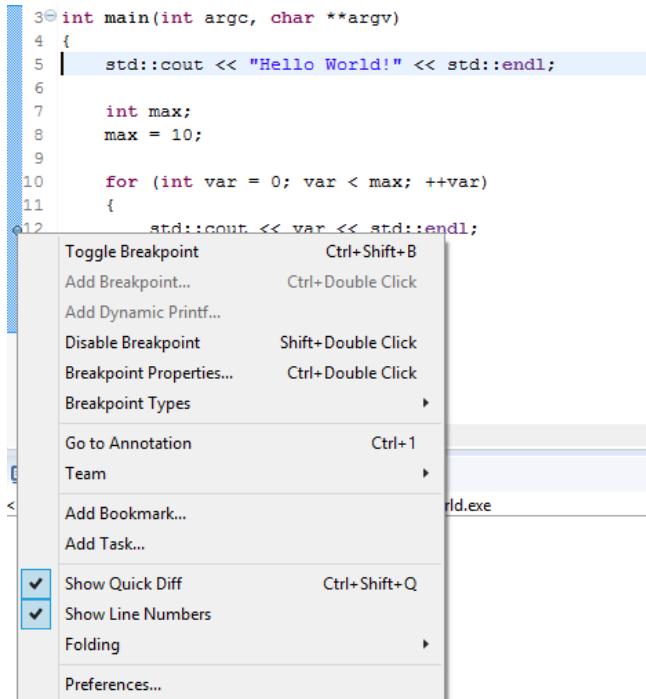


Abbildung 1.2: Hinzufügen eines Haltepunktes mit der rechten Maustaste oder Doppelklick auf die angezeigte Stelle.

Einzelschritt und Prozedurschritt

Nach der Unterbrechung des Programmablaufs durch einen Haltepunkt lässt sich das Programm schrittweise fortführen. Dabei gibt es zwei Optionen. Beim Einzelschritt (*Step Into*) wird jeder Befehl einzeln abgearbeitet und in Unterprogramme gesprungen. Der Prozedurschritt (*Step Over*) führt dagegen komplett Befehlszeilen aus und behandelt auch komplexe Unterprogramme wie einen einzelnen Programmschritt. Bereits getestete Unterprogramme müssen auf diese Weise nicht in Einzelschritten durchlaufen werden. Das graphische Frontend zeigt bei jedem Schritt die aktuelle Befehlszeile an. Dieser Unterschied wird später in Kapitel 2.1.8 detailliert dargestellt.

1.4.4 Bedienung des Debuggers in Eclipse

Anhand des bisher implementierten Beispielprogramms soll die Bedienung der grundlegenden Debug-Funktionen in der Eclipse-Umgebung erläutert werden. Weitere Funktionen des Debuggers werden in 2.1.8 gezeigt, da dafür die Theorie des 2. Versuches benötigt wird.

Die Debug-Perspektive bietet separate Fenster für den Quellcode, Variablen und weitere mächtige Werkzeuge, welche später genauer behandelt werden, um Fehler analysieren zu können. Um die Programmausführung im Debug-Modus zu starten, klicken Sie auf das Käfer-Symbol (Markierung 1 in der Abbildung 1.3). Der Debugger wird nun die Ausführung allerdings nicht wie erwartet in Zeile 12 anhalten, sondern in der ersten auszuführenden Code-Zeile. Dies ist eine Standardeinstellung von Eclipse, welche unter *Run → Debug Configurations* unter dem Tab *Debugger* mit *Stop on startup at :* geändert werden kann.

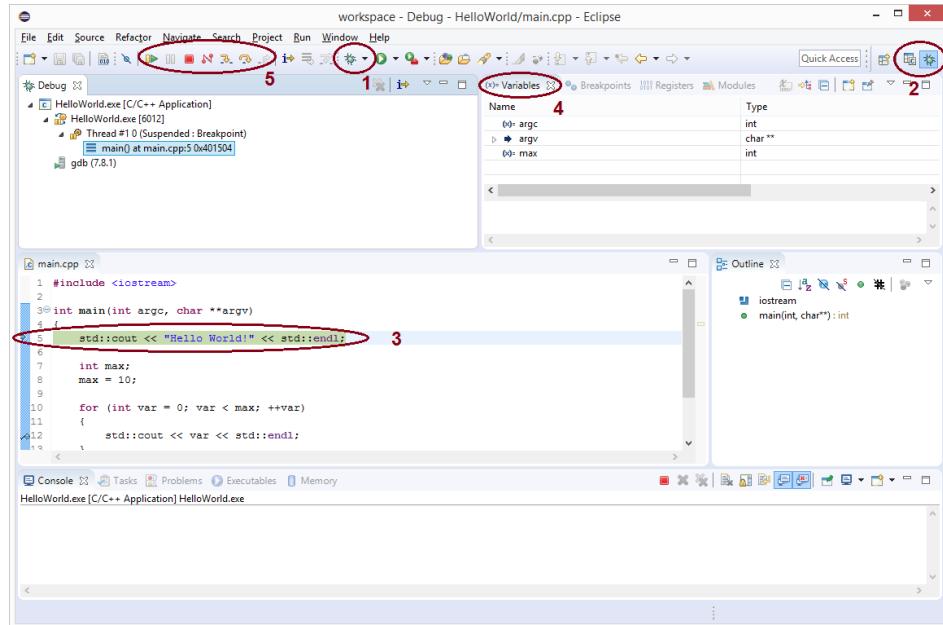


Abbildung 1.3: Das Fenster der Debug-Perspektive.

Die Abbildung 1.3 zeigt nun den aktuellen Zustand der Debug-Perspektive:

- Markierung 1 zeigt den *Run with Debugger* Button.
- Markierung 2 zeigt die Perspektivenauswahl. Es ist zu sehen, dass sich Eclipse momentan in der Debug-Perspektive befindet. Durch Anklicken des C/C++-Symbols gelangen Sie zurück in die Editorsperspektive.
- Markierung 3 zeigt, dass der Programmablauf in Zeile 5 angehalten wurde.
- Markierung 4 zeigt das Fenster, in dem die Werte lokaler und globaler Variablen überprüft werden können. Der Reiter *Breakpoints* dient zur Aktivierung bzw. Deaktivierung vorhandener Haltepunkte.
- Markierung 5 zeigt die Buttons, mit denen die Einzelschritte (*Step Into*, *Step Over*) ausgeführt werden können.

Es gibt nun mehrere Möglichkeiten, das Programm weiter laufen zu lassen. Bedienelement hierfür ist die Navigationsleiste (siehe Abbildung 1.4).



Abbildung 1.4: Die Navigationsleiste im Debug-Modus.

1 Datenstrukturen und Operatoren

- Markierung 1 zeigt den *Resume*-Button, welcher die Ausführung bis zum nächsten Haltepunkt oder zum Programmende fortsetzt.
- Markierung 2 zeigt den *Stop*-Button, welcher die Ausführung sofort beendet, ohne weitere Teile des Programms auszuführen.
- Markierung 3,4,5 zeigen *Step Into*, *Step Over* und *Step return*, in dieser Reihenfolge. Auf die Besonderheiten von *Step Into* und *Step return* wird später noch genauer eingegangen. *Step over* führt die aktive Zeile und alle sich dahinter verborgenden Instruktionen aus und pausiert dann den Ablauf wieder.

Button 3,4,5 werden in 2.1.8 genauer erklärt.

Mit dem *Resume*-Button (Markierung 1, Tastenkombination F8) läuft das Programm bis zum nächsten Haltepunkt oder bis zum Ende des Programms, falls es keinen weiteren Haltepunkt gibt. Hier wird die Ausführung bis in die *for*-Schleife fortgesetzt, bevor der Debugger das Programm wieder in Zeile 12 anhält. Die Variablen *max* und *var* haben nun die erwarteten Werte, siehe Abbildung 1.5.

Name	Type	Value
0x0+ argc	int	1
0x1+ argv	char **	0x3e4300
0x2+ var	int	0
0x3+ max	int	10

Abbildung 1.5: Anzeige der im Kontext vorhandenen lokalen und globalen Variablen

Durch Klick auf den *Step-Over*-Button (Markierung 4, Tastenkombination F6) wird Zeile 12 ausgeführt und *0* in die Ausgabe geschrieben. Nach nochmaligem Betätigen von *Step-Over* wird *var* erhöht und die Bedingung der Schleife überprüft. Da *var < max* immer noch gilt, hält das Programm nun wieder in Zeile 12.

Eine weitere Möglichkeit, die der Debugger bietet, ist es, Variablen während der Ausführung zu ändern, um den Ablauf des Programms manuell zu beeinflussen oder manuell Fehlerkorrekturen testen zu können, ohne den Debugmodus verlassen zu müssen. Ändert man beispielsweise *max* auf 5, sollte die *for*-Schleife nur noch bis 4 Zählen. Dies ist möglich, indem auf den Wert der Variable *max* in der Variablenübersicht geklickt wird.

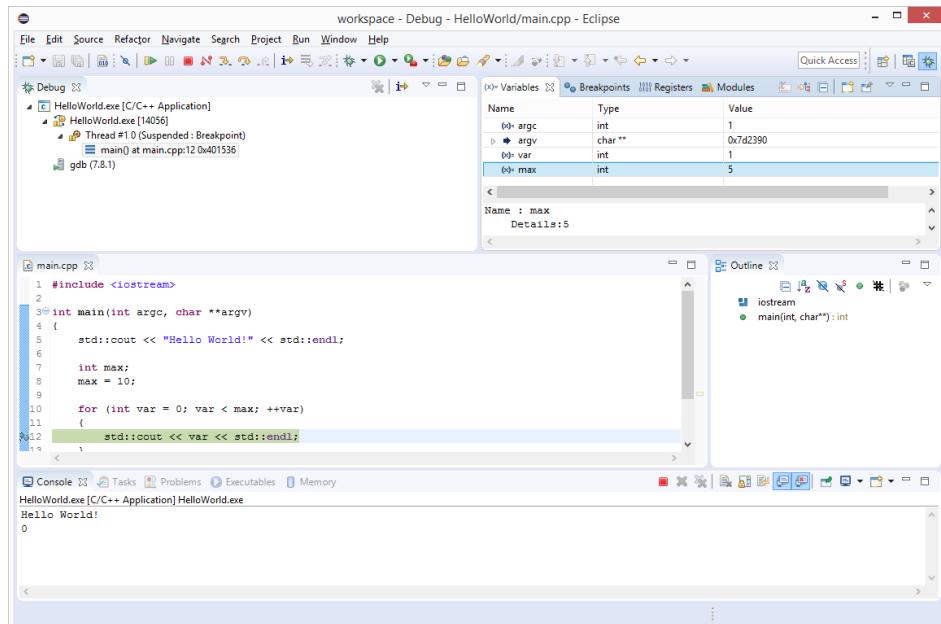


Abbildung 1.6: Debug-Sicht nach ändern von *max*.

Es ist möglich, während des Debuggens weitere Haltepunkte hinzuzufügen und zu entfernen. Haltepunkte lassen sich mit der rechten Maustaste oder Doppelklick auf den Haltepunkt entfernen (siehe Abbildung 1.7). Entfernen Sie nun den Haltepunkt in Zeile 12.

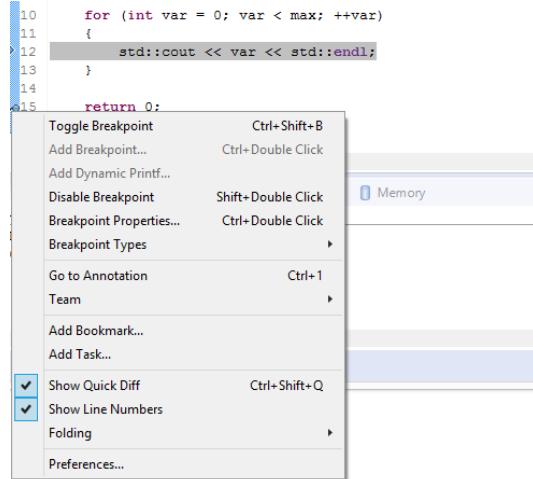


Abbildung 1.7: Entfernen eines Haltepunktes mit Hilfe der rechten Maustaste.

Nun wird ein neuer Haltepunkt in Zeile 15 hinzugefügt, um das Programm kurz vor dessen Beendigung noch einmal anzuhalten. Nach Klick auf den *Resume*-Button (Markierung 1, Tastenkombination F8) stoppt das Programm in Zeile 15. Die Ausgabe zeigt nun, dass die Schleife wie erwartet nach dem Ändern von *max* nur 5 mal durchlaufen wurde.

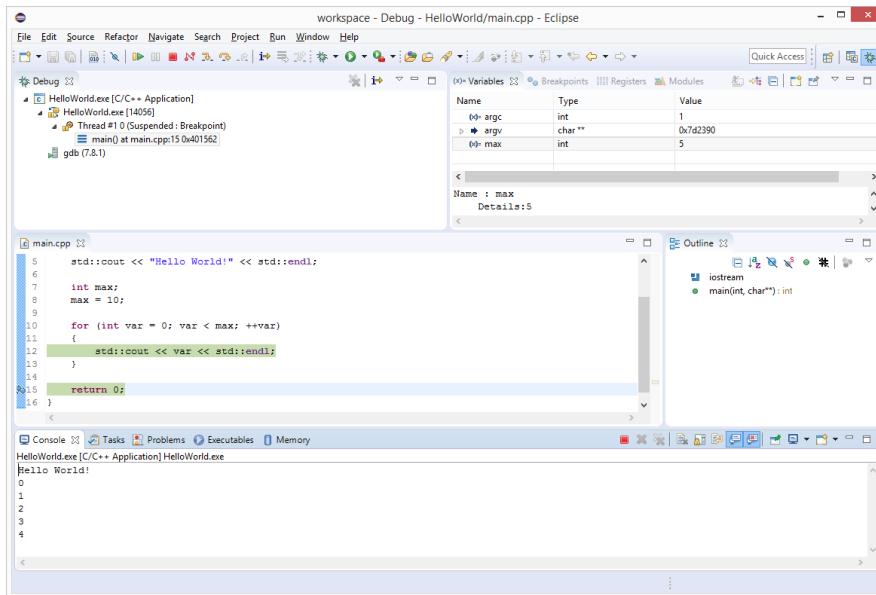


Abbildung 1.8: Neuer Haltepunkt am Ende des Programms.

Die bisherigen Schritte geben eine kurze Einführung in Eclipse, ohne jedoch alle Möglichkeiten genau zu erklären. Als beste Methode, sich mit der IDE auseinanderzusetzen, wird das regelmäßige Benutzen und Ausprobieren der verschiedenen Optionen empfohlen. Nehmen Sie sich in den kommenden Versuchen die Zeit, es gibt noch einiges zu Entdecken!

Natürlich ist das Internet voll mit meist englischen Tutorials über Eclipse und C++.

Im RWTHmoodle finden Sie unter der Rubrick *Hyperlinks* eine Liste mit Links und Beschreibungen.

1.5 Theorie

1.5.1 Zielsetzung und Einordnung

Sie haben nun die Entwicklungsumgebung *Eclipse* kennengelernt. In diesem Versuch werden Sie einen genaueren Blick auf Variablen, ihre Datentypen und grundlegende Konzepte zur Handhabung dieser Variablen werfen. Sie werden lernen, wie man Variablen über Operatoren miteinander verknüpft und welche Möglichkeiten es gibt, um zusammengehörende Daten geeignet abzuspeichern.

Ziel dieses Versuches ist es, Ihnen die grundsätzlichen Werkzeuge der Programmiersprache *C++* mit auf den Weg zu geben. Am Ende dieses Versuches sollten Sie wissen, was

- Datentypen
- Operatoren
- Felder und Strukturen

sind und wie man diese einsetzen kann.

1.5.2 Datenobjekte und Datentypen

Grundsätzlich besteht ein *C++*-Programm aus *Funktionen* und *Variablen*. Die Funktionen bestehen aus einer Folge von *Anweisungen*, die die Aktionen des Programms bestimmen. Die *Datenobjekte* (Variablen und Konstanten) enthalten die Werte, mit denen das Programm arbeitet.

Variables

Jede Variable in *C++* besitzt einen bestimmten Datentyp. Genau wie sich in der Mathematik eine Zahl als *natürliche*, *ganze*, *rationale* oder *reelle* Zahl charakterisieren lässt, lassen sich auch die Variablen eines *C++*-Programms durch ihre Datentypen charakterisieren. Es gibt Datentypen zum Speichern ganzer oder natürlicher Zahlen, sowie Variablen, die Zahlen mit Nachkommastellen aufnehmen können (sog. *Fließkommazahlen*). Da der Speicherplatz im Rechner nicht beliebig groß ist, können die Variablen auch nicht beliebig große Werte annehmen. Die folgende Tabelle zeigt die gebräuchlichsten, elementaren Datentypen von *C++* sowie ihre Wertebereiche. Beachten Sie, dass der tatsächliche Wertebereich system- und prozessorabhängig ist. Die Standard-Headerdatei "*limits*" definiert die tatsächlichen Wertebereiche.

Ein *C++*-Compiler muss immer wissen, welchen Datentyp eine Variable hat, damit er genug Speicherplatz reservieren kann und weiß, wie er die Daten im Speicher interpretieren muss. Man bezeichnet dies als *Vereinbarung* oder *Definition*. Die einfachste Vereinbarung beginnt mit dem Schlüsselwort des Datentyps, gefolgt von dem Namen der Variable:

```
1 || int value;           // Definition einer Integer-Variablen
```

Bei der Vereinbarung kann der Variablen auch direkt ein Wert zugewiesen werden. Die Variable wird *initialisiert*. Dies sollte sogar immer mit sinnvollen Werten geschehen, um Fehler durch beliebig initialisierte Variablen zu vermeiden. In diesem Praktikum sind Sie daher dazu angehalten, allen Variablen, die Sie einführen, auch einen Wert zuzuweisen.

```
1 || int value = 2017; // Definition mit Initialisierung
```

Auch die gleichzeitige Vereinbarung mehrerer Variablen des gleichen Datentyps ist zulässig, wenn auch von einigen Autoren empfohlen wird, pro Zeile nur eine Variable zu definieren, um potentielle Fehlerquellen zu vermeiden und leichter Kommentare einzufügen zu können. Die Variablen werden durch Kommata getrennt:

```
1 || // mehrere Definitionen vermeiden!!!
2 || double dValue1 = 3.2, dValue2, dValue3 = -0.5;
```

C++ erlaubt die Definition von Variablen praktisch überall im Programm, auch mitten in einer Funktion oder in einem Block. Die meisten Programmierer finden es übersichtlicher, eine Variablen erst unmittelbar vor (oder sogar praktisch mit!) ihrer ersten Benutzung zu definieren, so dass man die Variable erst dann sieht, wenn man sie wirklich braucht. Dies ist bei der Programmierung in C++ üblich.

DatenTyp	Erläuterung	Wertebereich	Länge (in Byte)
ganzzahlige Typen (vorzeichenbehaftet)			
char	ein Zeichen	-128 ... 127	1
short	kleiner Wertebereich	-32768 ... 32767	2
int	einfacher Wertebereich	$-2^{31} \dots 2^{31} - 1$	4
long	doppelter Wertebereich	$-2^{31} \dots 2^{31} - 1$	4
long long	verdoppelter Wertebereich	$-2^{63} \dots 2^{63} - 1$	8
ganzzahlige Typen (nicht vorzeichenbehaftet)			
unsigned char	ein Zeichen	0 ... 255	1
unsigned short	kleiner Wertebereich	0 ... 65535	2
unsigned int	einfacher Wertebereich	0 ... 4294967295	4
unsigned long	doppelter Wertebereich	0 ... 4294967295	4
unsigned long long	doppelter Wertebereich	0 ... 18446744073709551615	8
Fließkommazahlen			
float	einfache Genauigkeit	$\approx 1.4 \cdot 10^{-45} \dots 3.4 \cdot 10^{+38}$	4
double	doppelte Genauigkeit	$\approx 4.94 \cdot 10^{-324} \dots 1.8 \cdot 10^{+308}$	8
Wahrheitswerte			
bool	nur zwei Werte	true, false	1

Konstanten

Manche Dinge ändern sich nie, z.B die Zahl π . Stellt man einer Variablendefinition das Schlüsselwort *const* voran, so wird die Variable als *Konstante* vereinbart. Ihr Wert kann später nicht mehr verändert werden. Dies bedeutet auch, dass eine Konstante bei ihrer Definition immer initialisiert werden muss:

```
1 || const double PI = 3.1415; // Definition der Konstanten PI
2 || const double quote;      // Fehler: uninitialized Konstante
3 || PI = 3.14159265;        // Fehler: PI ist nicht veränderbar
```

Grundsätzlich gehört zu einem guten Programmierstil, dass Sie alle Variablen, deren Wert sich nicht ändern kann (oder soll), auch als Konstante definieren!

Boolsche Variablen

Boolsche Variablen stellen sogenannte Wahrheitswerte dar und können nur zwei Werte annehmen: *true* (wahr) oder *false* (falsch).

Sie stellen in C++ einen eigenen Datentypen dar und werden mit dem Schlüsselwort *bool* angelegt.

```
1 || bool ok1;
2 || bool ok2 = true;
```

Die interne Darstellung erfolgt mittels Zahlenwerten, wobei 0 für *false* steht, und alle anderen Werte für *true*. Sie sollten aber immer *true* und *false* benutzen, damit Ihr Code besser zu verstehen ist.

Character Variablen

Variablen vom Typ *char* dienen der Aufnahme eines einzelnen Zeichens (engl. *character*) und sind genau ein Byte groß. Da Rechner aber nur Zahlen kennen, werden alle Zeichen als Zahl codiert. Erst durch die Interpretation wird aus dieser Zahl ein Zeichen. Daher ist es wichtig, diese Variable als *char*

1 Datenstrukturen und Operatoren

zu definieren.

Die sogenannte *ASCII-Tabelle* legt dabei die Codierung der ersten 128 Zeichen fest, die anderen 128 sind nicht fest definiert und variieren je nach Rechner und z.B. Landessprache. Aber im Moment reichen die ersten 128 aus. Suchen Sie im Internet nach der *ASCII-Tabelle* und schauen Sie sich ihre Belegung an.

```
1 || char zeichen1;           // Definition eines char Wertes
```

Die Initialisierung erfolgt durch einfache Anführungszeichen.

```
1 || char zeichen1 = 'A'; // Definition eines char Wertes mit Initialisierung
```

Da *char*-Variablen wie vorzeichenbehaftete Integervariablen gehandhabt werden, wäre eine Initialisierung durch

```
1 || char zeichen1 = 65;
```

gleichwertig zu

```
1 || char zeichen1 = 'A';
```

Es lassen sich alle Rechenoperationen, die mit Integerwerten möglich sind, auch auf den Datentyp *char* anwenden, solange man den recht kleinen Wertebereich von *char* im Auge hat.

Möglich sind dadurch auch folgende Ausdrücke

```
1 || char zeichen1 = 'A' + 1;    // zeichen1 ist 'B'  
2 || int i1 = zeichen1 % 32;    // i1 ist 2, z.B. 2. Stelle im deutschen Alphabet  
3 || char zeichen2 = 'a';       // zeichen2 ist 'a'  
4 || int i2 = zeichen2 % 32;    // i2 ist 1, z.B. 1. Stelle im deutschen Alphabet
```

1.5.3 Operatoren

Variablen an sich sind relativ langweilige Gefährten und werden erst dadurch interessant, dass man sie miteinander verknüpfen kann. Hierzu dienen *Operatoren*, aus denen zusammen mit den *Operanden* (Variablen, Konstanten, Zahlen, Funktionsaufrufe, etc.) *Ausdrücke* gebildet werden. Zum Beispiel ist $a + b - 2 * \sqrt{4}$ ein Ausdruck. Ein Ausdruck hat einen Wert und kann immer dort stehen, wo ein Wert verlangt wird. C++ kennt eine Vielzahl von Operatoren, die mächtige Ausdrücke in einer einzelnen Zeile liefern können. Gleichzeitig werden diese Ausdrücke auch beliebig kompliziert, fehleranfällig und unleserlich. Sofern Ihre Anweisung nicht laufzeitkritisch ist, vermeiden Sie es, zu viele Operationen in einen Ausdruck zu pressen. Jemand, der das Programm später warten muss, wird es Ihnen danken.

Eine weitere Quelle für fehlerhafte Programme ist die Reihenfolge, in der die Operatoren ausgewertet werden. Wenn Sie sich nicht sicher sind, welche Priorität die Operatoren haben, verwenden Sie in Ihrem Ausdruck die runden Klammern.

Arithmetische Operatoren

Die in C++ bekannten arithmetischen Operatoren sind $+$, $-$, $*$, $/$ und $\%$. Letzterer liefert den Divisionsrest (Modulo-Operator), also zum Beispiel $5 \% 3 = 2$. Er kann daher nur auf ganzzahlige Datentypen angewendet werden. Wie üblich gilt, dass Multiplikation und Division Vorrang haben vor Addition und Subtraktion.

Vergleiche

Die in C++ bekannten *Vergleichsoperatoren* sind $>$ (*größer*), \geq (*größer oder gleich*), $<$ (*kleiner*) und \leq (*kleiner oder gleich*). Die Operatoren $==$ (*gleich*) und \neq (*ungleich*) werden manchmal auch als *Äquivalenzoperatoren* bezeichnet. Sie haben geringeren Vorrang als die Vergleichsoperatoren. Diese wiederum haben geringeren Vorrang als die zuvor vorgestellten arithmetischen Operatoren. Daher wird ein Ausdruck wie $i < a + b$ als $i < (a + b)$ ausgewertet, was man auch intuitiv erwarten würde. Beachten Sie auch den Unterschied zwischen dem Zuweisungsoperator $=$ und dem Vergleichsoperator $==$. Die Verwechslung dieser beiden Operatoren stellt eine häufige Fehlerquelle in C++-Programmen dar.

Logische Verknüpfungen

C++ kennt die logischen Verknüpfungen ! (NOT), && (AND) und || (OR) mit Priorität in der genannten Reihenfolge. Logische Verknüpfungen haben geringere Priorität als Vergleichs- und Äquivalenzoperatoren.

Zu bemerken ist noch, dass Ausdrücke, die mit && oder || verknüpft sind, von links nach rechts ausgewertet werden, jedoch nur solange, bis das Ergebnis der logischen Verknüpfung feststeht.

Der Ausdruck $(i < 0 \parallel i \geq 7)$ ergibt *true*, wenn i kleiner 0 oder i größer 7 ist. Die Auswertung endet bereits, wenn i kleiner als 0 ist. Sie wissen also nicht, welche der Bedingungen erfüllt ist, sondern nur, dass mindestens eine erfüllt ist.

Inkrement- und Dekrement-Operator

Zum Erhöhen und Erniedrigen ganzzahliger Variablen um 1 stellt C++ die Operatoren ++ und -- zur Verfügung:

```
1 || int value = 0;
2 || value++;                                // value ist jetzt 1
3 || value--;                                // value ist wieder 0
```

Beachten Sie, dass der Operator vor (Präfix-Version) oder hinter (Postfix-Version) der Variablen stehen kann. Diese Unterscheidung kann wesentlich sein. Durch die Anweisung

```
1 || value1 = value2++;
```

erhält *value1* zuerst den Wert von *value2* und erst anschließend wird *value2* um 1 erhöht. Die Anweisung

```
1 || value1 = ++value2;
```

hingegen erhöht zunächst *value2*, und anschließend erhält *value1* diesen neuen Wert.

Eingabe- und Ausgabe-Operator

Um Daten auf die Konsole auszugeben oder Daten über die Tastatur einzulesen, werden der Ausgabe-Operator << bzw. der Eingabe-Operator >> aus der C++-Standardbibliothek benutzt. Daten werden mit Hilfe dieser zwei Operatoren an *cout*³ oder von *cin* in den Speicher einer Variablen geschickt. Dazu muss zunächst die benötigte Funktionalität geladen werden:

```
1 || #include <iostream>
```

Danach kann mit dem Operator << z.B. ein Text nach *std::cout* ausgegeben werden:

```
1 || std::cout << "Ausgabe" << std::endl;
```

oder von *std::cin* mit dem Operator >> z.B. ein Integerwert eingelesen werden.

```
1 || int nummer = 0;
2 || std::cin >> nummer;
```

³Wie Sie wahrscheinlich bereits bemerkt haben, wird der Befehl immer mit *std::* begonnen. Dies liegt daran, dass *cout* und *cin* in einem sogenannten *Namespace* liegen. Ein Namespace dient dazu, dass man nicht, wenn man zum Beispiel eine Bibliothek inkludiert, alle möglichen Funktionen in seinem Programm hat, die eventuell gleich heißen. Mit *std::cout* teilt man also mit, welches *cout* gemeint ist. Mit dem Schlüsselwort *using* kann man entweder den Inhalt eines ganzen Namespace (z.B. *using namespace std;*) oder spezielle Member (z.B. *using std::cout;*) bekanntmachen, sodass man nicht bei jedem Aufruf spezifizieren muss, was gemeint ist. Einen ganzen Namespace wie *std* bekanntzumachen birgt jedoch auch Risiken, da es zu ungewollten Konflikten kommen kann. Daher empfiehlt es sich im Allgemeinen, nur Member bekanntzumachen, die auch wirklich genutzt werden, um sowohl den Überblick zu bewahren als auch die Lesbarkeit des Codes durch Andere nicht zu gefährden.

Hinweis: Beim Debuggen von Programmen mit `std::cin`-Inputs kann es dazu kommen, dass Eingaben nicht wie erwünscht verarbeitet werden. Um diesem unerwünschten Verhalten vorzubeugen, empfiehlt es sich, derartige Programme in einer separaten Konsole auszuführen. Nehmen Sie dazu folgende Einstellung vor:

Window → Preferences → C/C++ → Debug → GDB

Setzen Sie das Häkchen bei *use external console for inferior (open a new console window for input/output)*

1.5.4 Typumwandlungen

Die Variablen in *C++* können sich für etwas ausgeben, was sie gar nicht sind, um mit anderen Variablen operieren zu können. Sie können sie zu dieser Mogelei (*explizit*) zwingen, meist jedoch erfolgt dies automatisch (*implizit*) ohne Ihr Zutun, wenn Sie Variablen verschiedener Datentypen über einen Operator miteinander verknüpfen. Dabei wird einer der beiden Datentypen in den anderen Datentyp überführt, sofern dabei keine Information verloren geht. Diesen Vorgang bezeichnet man auch als *Typecasting*, kurz *Casting*. Im folgenden Beispiel wird bei der Addition der *int*-Wert in einen *double* umgewandelt.

```
1 || int value1 = 42;
2 || double value2 = 42.42;
3 || // Zwischenwert von value1 wird zu double
4 || double value3 = value1 + value2;
```

Beachten Sie, dass „Umwandlung“ **nicht** bedeutet, dass sich der Datentyp der Variablen ändert. Es wird lediglich der *Wert* der Variablen in dem gewünschten Datentyp geliefert, d.h. im obigen Beispiel bekommt der Plus-Operator den Wert von *value1* als *double* geliefert, wohingegen *value1* selbst ein *int* bleibt.

Allgemein werden implizit „schmalere“ Typen in „breitere“ Typen umgewandelt, also z.B. ein *int* in einen *float*, nicht jedoch umgekehrt, da dabei die Nachkommastellen verloren gehen würden. Solche Anweisungen sind zwar nicht verboten, ein vorsichtiger Compiler generiert jedoch eine Warnung:

```
1 || int value1;
2 || double value2 = 42.42;
3 ||
4 || value1 = value2;           // Compilerwarnung
```

Ein Typecast kann unter Zuhilfenahme eines Umwandlungsoperators auch explizit erzwungen werden. Der Umwandlungsoperator besteht aus dem Namen des gewünschten Datentyps, eingeschlossen in Klammern:

```
1 || int value1;
2 || double value2 = 42.42;
3 ||
4 || value1 = (int) value2;    // expliziter Typecast von value2
```

Beim expliziten (also vom Programmierer gewollten) Typecast generiert der Compiler auch dann keine Warnung, wenn beim Casten Information verloren geht (wie in obigem Beispiel der Nachkommateil von *value2*). Durch einen expliziten Typecast teilt man dem Compiler sozusagen mit, dass es so beabsichtigt war. Neben dem sogenannten *C-Cast*, der gelegentlich auch als unsicher bezeichnet wird, bietet die Sprache *C++* in Verbindung mit komplexen Datentypen, die im Folgenden behandelt werden, und Polymorphie weitere, sichere Möglichkeiten Typumwandlungen durchzuführen. An dieser Stelle werden diese jedoch nicht behandelt.

1.5.5 Datenstrukturen

Ein fundamentales Problem bei der Erstellung von Programmen ist die Frage, in welcher Form die zu bearbeitenden Daten im Speicher abgelegt werden sollen. Die gewählte *Datenstruktur* beeinflusst ganz wesentlich die Übersichtlichkeit, Leistungsfähigkeit, Komplexität und Wartbarkeit Ihres Programms. Die Wahl der Datenstruktur hängt normalerweise von den Aktionen ab, die mit den Daten durchgeführt werden sollen:

- Ihr Telefonbuch enthält eine Reihe von Namen. Damit Sie einen Namen schnell wiederfinden, sind die Einträge alphabetisch sortiert. Ihr Telefonbuch ist somit als Datenstruktur *Liste* implementiert. Jeder Eintrag in Ihrem Buch, bestehend aus den *Datenelementen* Name, Adresse und Telefonnummer, stellt einen *Datensatz* dar.
- Ihre Deutschland-Straßenkarte zeigt Ihnen viele Städte und Straßen, welche die Städte miteinander verbinden. Jede Stadt als *Knoten* und jede Straße als *Kante* stellt somit ein Datenelement in der Datenstruktur *Graph* dar. Würden Sie jemals den Weg von Aachen nach München finden, wenn die Karte als Liste von Städten und Straßen implementiert wäre?
- Ihr Familienstammbuch zeigt Ihnen Ihre Vorfahren, ausgehend z.B. von Ihren Urgroßeltern bis zu Ihren Eltern. Ähnlich wie in einem Graph gibt es Knoten, welche Personen darstellen, und Kanten, die Beziehungen zwischen Personen (ist Kind von, Heirat) symbolisieren. Es sollte jedoch in der Durchschnittsfamilie keine Kanten über mehrere Generationen hinweg geben, also keine *Zykel*. Ihr Stammbuch hat somit die Datenstruktur *Baum*.
- Sofern Sie Ihre leeren Milchtüten nicht zurück in den Kühlschrank stellen, werfen Sie sie (wie auch jeglichen sonstigen Abfall) in den Mülleimer. Dabei ist Ihnen die Position, an der die Tüte im Eimer zu liegen kommt, i.A. egal. Die Datenelemente in Ihrem Mülleimer sind daher ungeordnet. Mit ein wenig Suchen schaffen Sie es trotzdem noch, eine versehentlich weggeworfene Adresse wiederzufinden. Sie dürfen Ihren Müllhaufen daher guten Gewissens als Datenstruktur *Heap* betrachten.

Im Folgenden werden Sie drei einfache Datenstrukturen kennenlernen, mit denen Sie in dieser und den folgenden Übungen arbeiten werden: das *Feld*, die *Zeichenkette* und die *Struktur*.

Felder

Ein *Feld* oder *Array* ist sicherlich die am häufigsten verwendete Datenstruktur. Es ist eine Ansammlung von Datenelementen gleichen Typs, auf die mit einem gemeinsamen Namen zugegriffen werden kann. Die Daten werden mit Hilfe eines *Index* durchnummeriert, über den jedes Feldelement direkt angesprochen werden kann. Die Größe des Feldes muss von Anfang an bekannt sein, sie kann sich zur Laufzeit des Programms nicht ändern. Man sagt deshalb auch, es handelt sich um eine *statische* Datenstruktur.

Eindimensionale Felder Die Eigenheime auf der Schlossallee sind ein typisches Beispiel für ein (eindimensionales) Feld. Alle Eigenheime sind irgendwie ähnlich und haben den Datentyp *Haus*. Die Anzahl der Häuser ist bekannt und nicht veränderlich (Baustellen und Gasexplosionen mal ausgenommen). Jedes Haus ist über seine Hausnummer als Index eindeutig identifizierbar, und man kann auf jedes Haus direkt zugreifen (also hineingehen), ohne vorher irgendwelche anderen Häuser besucht zu haben.

Um ein Feld zu deklarieren, fügen Sie dem Variablennamen einfach die gewünschte Größe des Feldes, eingeschlossen in eckige Klammern, hinzu. Auf die gleiche Weise wird auch auf ein bestimmtes Element des Feldes zugegriffen:

```
1 || int feld[10]; // Definition eines Feldes mit 10 int-Werten
2 || feld[3] = 5; // Dem Element mit Index 3 den Wert 5 zuweisen
4 || std::cout << feld[1]; // Ausgeben des Elements mit Index 1
```

Die Indexnummierung beginnt bei 0 und endet bei $n - 1$, wenn n die Größe des Feldes ist. Die Nummerierung der Feldelemente aus obigem Beispiel geht also von 0 bis 9.

Mehrdimensionale Felder Oft ist die Verwendung eindimensionaler Felder nicht zweckmäßig. Viele Problemstellungen sind durch Tabellen mit n Zeilen und m Spalten abbildbar. Eine Liste der Bewohner Ihres Studentenwohnturms könnte z.B. nach Etage (Zeile) und Zimmernummer (Spalte) geordnet sein. Das Walter-Eilender-Haus ließe sich somit durch folgende Vereinbarung erzeugen:

```
1 || // Definition eines Feldes mit 17 Zeilen und 16 Spalten
2 || int feld[17][16];
```

1 Datenstrukturen und Operatoren

Beachten Sie, dass die Indizierung mit der Notation `feld[j][i]` und nicht mittels `feld[j,i]` erfolgt. Dies liegt daran, dass ein zweidimensionales Feld eigentlich ein eindimensionales Feld ist, wobei jedes Feldelement wiederum ein eindimensionales Feld ist. Dieses Spiel kann man natürlich auch noch in höhere Dimensionen treiben, bis man ein genügend dimensionales Feld erzeugt hat.

Initialisierung von Feldern Genau wie normale Variablen können auch Felder direkt bei Ihrer Definition initialisiert werden. Die Initialisierungsliste ist in geschweifte Klammern eingeschlossen, die Werte für die einzelnen Feldelemente werden durch Kommata voneinander getrennt:

```
1 // Initialisierung eines Feldes
2 int feld[10] = { 2, 3, 8, 2, 5, 6, 2, 4, 9, 3 };
3 char textfeld[] = {'T','E','X','T',' ','F','E','L','D'}; // Feld mit 9 char-Werten
```

Auch die Initialisierung mehrdimensionaler Felder ist möglich, wobei wir uns hier auf zweidimensionale Felder beschränken.

```
1 // Initialisierung eines zweidimensionalen Feldes
2 int feld[2][5] = { { 2, 3, 8, 2, 5 },           // 1. Zeile
3                   { 6, 2, 4, 9, 3 } };      // 2. Zeile
```

Zeichenketten

Eine *Zeichenkette* (*engl. string*) ist die Darstellung einer Folge von Zeichen. C++ stellt Ihnen dazu den Datentyp `string` aus der C++-Standardbibliothek zur Verfügung. Ein String mit Initialisierung wird dann folgendermaßen definiert:

```
1 #include <string>
2 std::string text = "Dies ist ein Text!";
```

Beachten Sie die `#include`-Anweisung in der ersten Zeile. Diese ist notwendig, um den Datentyp `string` nutzen zu können. Damit wird auch die gesamte Funktionalität zur Verfügung gestellt.

Das Besondere am Datentyp `string` ist, dass sich seine Größe auch zur Laufzeit ändern kann.

```
1 #include <string>
2 std::string text = "Dies ist ein Text!";
3 text = "Und das ist jetzt ein ganz anderer Text, viel länger!";
```

Auch Zahlen lassen sich in einem String speichern, dabei werden die einzelnen Ziffern gemäß der *ASCII*-Tabelle gespeichert.

```
1 #include <string>
2 std::string matrikel_nummer = "123456"; // "123456" ist nicht Integer 123456
```

Arbeiten mit Strings

Es gibt viele Möglichkeiten, Strings einzusetzen und mit ihnen zu arbeiten. Hier sollen die wichtigsten Konzepte und Funktionen vorgestellt werden.

Bei dem Datentyp `string` handelt es sich nicht um einen einfachen Datentyp wie `int` oder `char`, sondern um einen Datentyp, der durch eine *Klasse* repräsentiert wird. Klassen werden Sie in Kapitel 4 näher kennenlernen und selbst erstellen. Daher hier nur das Notwendigste.

Klassen speichern normalerweise nicht nur Daten, sondern stellen auch Funktionen zu Verfügung, mit denen diese Daten bearbeitet werden können. Die Klasse `string` besitzt eine ganze Reihe Funktionen, die speziell auf Strings abgestimmt sind.

Länge von Strings bestimmen Strings sind dynamisch, d.h. sie können sich zur Laufzeit des Programms ändern und in ihrer Länge variieren.

Um die Länge eines Strings zu bestimmen, stellt der Datentyp `string` eine Funktion `length()` bereit. Um Funktionen von Klassen aufrufen zu können, benötigen Sie (fast) immer ein konkretes Objekt dieser Klasse, im folgenden Beispiel ist dies `str`.

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str = "Dies ist ein Satz.";
7     int strLaenge = str.length();
8     std::cout << str << " Er ist " << strLaenge << " Zeichen lang." << std::endl;
9     return 0;
10}

```

Dies ist ein Satz. Er ist 18 Zeichen lang.

Verkettung von Strings Um mehrere Strings zu verketten (aneinander zu hängen), kann der + Operator genutzt werden.

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str1 = "Dies ist";
7     std::string str2 = " ein Satz.";
8     std::string str12 = str1 + str2;
9     std::cout << str12 << std::endl;
10
11}

```

Dies ist ein Satz.

Mit dem Operator += können Sie sowohl einen String als auch ein einzelnes Zeichen an einen String am Ende hinzufügen.

Mit der Funktion *push_back()* wird ein einzelnes Zeichen an den String angehängt, keine Strings. Bei beiden Möglichkeiten wird die interne Variable, in der die Länge gespeichert wird, angepasst.

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str1 = "Dies ist";
7     std::string str2 = " ein Sat";
8     str1 += str2;
9     std::cout << str1 << std::endl;
10    str1 += 'z';
11    std::cout << str1 << std::endl;
12    str1.push_back('!');
13    std::cout << str1 << std::endl;
14
15}

```

Dies ist ein Sat
Dies ist ein Satz
Dies ist ein Satz!

Zugriff auf einzelne Zeichen Sie können auch auf einzelne Zeichen innerhalb des Strings zugreifen.

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str = "Dies ist ein Satz";

```

1 Datenstrukturen und Operatoren

```
7 ||     char zeichen = str[2];
8 ||     std::cout << "zeichen: " << zeichen << std::endl;
9 ||     return 0;
10|| }
```

```
zeichen: e
```

Sie können auf die einzelnen Zeichen wie bei Feldern mit dem Zugriffsoperator `[]` zugreifen. Dabei gilt es zu beachten, dass, wie bei Feldern, der Index bei 0 beginnt, und dass keine Bereichsüberprüfung stattfindet. Sie sind selbst für den Index verantwortlich.

Der Zugriffsoperator ist, wie der Name schon sagt, für Zugriffe vorgesehen. Er kann jedoch auch dazu genutzt werden, einzelne Zeichen innerhalb des Strings zu ändern. Werden allerdings Zeichen am Ende hinzugefügt, wird die Länge *nicht* angepasst, und daher auch nicht ausgegeben. Ein leerer String bleibt leer. Benutzen Sie dafür die im vorherigen Abschnitt beschriebenen Funktionen.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str = "Dies ist ein Satz";
7     std::cout << str.length();
8     str[17] = '!';
9     std::cout << str.length();
10    std::cout << str << std::endl;
11    return 0;
12 }
```

```
17
17
Dies ist ein Satz
```

Die Länge ändert sich nicht und das Ausrufezeichen wird nicht ausgegeben.

Maskierung (Escaping) Wie bereits gesagt, werden String-Literale mit `""` dargestellt. Soll nun ein solches Zeichen im Text selber stehen, muss dies dem Compiler mitgeteilt werden, da der Compiler nicht wissen kann, ob der Programmierer nun den String beenden will oder das Zeichen im String enthalten sein soll. Dies ist möglich, indem vorher das Zeichen mit \ *maskiert (escaped)* wird. Dies bewirkt, dass das nachfolgende Zeichen nicht mehr als Funktionszeichen interpretiert wird. Das Beispiel wird nun so erweitert, dass in der Ausgabe `str12` in `""` steht:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string str1 = "\"Dies ist";
7     std::string str2 = " ein Satz.\"";
8     std::string str12 = str1 + str2;
9     int str12Laenge = str12.length();
10    std::cout << str12 << "Er ist " << str12Laenge << " Zeichen lang." << std::endl;
11 }
```

```
"Dies ist ein Satz." Er ist 20 Zeichen lang.
```

Strukturen

Stellen Sie sich vor, Sie möchten die Mitglieder in dem von Ihnen ins Leben gerufenen, örtlichen Verein elektronisch speichern. Welche Datenstruktur verwenden Sie? Sie könnten die einzelnen Charakteristika jeder Person (Name, Mitgliedsnummer, Straße, Tel.-Nr.) je in einem eigenen Feld ablegen. Dann müssten Sie auf vier Felder zugreifen, um eine einzelne Person auszugeben. Dies ist weder elegant noch besonders gut wartbar. Schöner wäre es, Sie könnten alle Charakteristika zu einer Einheit (in diesem Falle zu einer Person) zusammenfassen, um dann die Einheit als Ganzes abzuspeichern. Es wird Sie nicht überraschen: Sie können! In anderen Programmiersprachen heißt die hierzu verwendete Datenstruktur *Record*, während man sie in *C++ Struktur* nennt.

Deklaration einer Struktur Eine Struktur ist also eine Ansammlung von Variablen mit möglicherweise verschiedenen Typen. Die Deklaration eines Mitglieds sieht so aus:

```
1 || struct Mitglied
2 || {
3 ||     std::string name;
4 ||     int nr;
5 ||     std::string strasse;
6 ||     int telefon;
7 || };
```

Eine so deklarierte Struktur stellt einen neuen Datentyp dar. Dem Schlüsselwort *struct* kann ein sogenanntes *Etikett* folgen, welches anschließend als Abkürzung für den in den geschweiften Klammern stehenden Deklarationsteil dienen kann. Hier ist das Etikett das Wort *Mitglied*. Die einzelnen Variablen, aus denen eine Struktur besteht, werden als *Komponenten* bezeichnet. Noch einmal: Sie haben jetzt einen neuen Datentyp mit dem Namen *Mitglied* definiert, aber noch keine Variable dieses Typs. Wie sähe jetzt die Variablendeklaration aus? Wie vorher auch:

```
1 || Mitglied hans;      // Definition einer Strukturvariablen
```

Sie haben soeben das Mitglied „hans“ Ihres Vereins ins Leben gerufen. Auf die einzelnen Komponenten von *hans* greifen Sie mit Hilfe des *Punktoperators* (engl.: *member dot operator*) zu:

```
1 || hans.name = "Hans Wurst";
2 || hans.nr = 42;
3 || hans.strasse = "Am Schlachthof 1";
4 || hans.telefon = 420815;
```

Ein kurzes Wort zu den Begriffen *Deklaration* und *Definition*. Wenn Sie eine Variable anlegen, dann reserviert der Compiler Speicher, daher spricht man an dieser Stelle von einer *Definition*. Bei der Erstellung der Struktur *Mitglied* wird aber noch kein Speicher reserviert, sondern Sie teilen dem Compiler erst mal nur mit, wie Ihr benutzerdefinierter Datentyp *Mitglied* aufgebaut ist. Daher nennt man das eine *Deklaration*. Erst wenn Sie schreiben *Mitglied Hans;*, definieren Sie eine Variable *hans* vom Typ *Mitglied* und es wird Speicher reserviert.

Ähnliches gilt für Funktionen, die Sie noch kennenlernen werden.

Bei den normalen Variablen ist die *Definition* genau genommen gleichzeitig auch die *Deklaration* der Variablen. Das lässt sich bei *C++* nicht trennen.

Kopieren von Strukturen Sie können auch eine Strukturvariable einer anderen zuweisen. Die Struktur wird dann automatisch komponentenweise kopiert. Der Code

```
1 || Mitglied peter;
2 || peter = hans;
```

ist äquivalent zu

```
1 || Mitglied peter;
2 ||
3 || peter.name = hans.name;
4 || peter.nr = hans.nr;
5 || peter.strasse = hans.strasse;
6 || peter.telefon = hans.telefon;
```

1 Datenstrukturen und Operatoren

Strukturen in Feldern Ihr Verein besteht nicht nur aus dem Mitglied Hans Wurst, sondern aus vielen, motivierten Mitgliedern, die Sie alle archivieren wollen. Zu diesem Zweck kombinieren Sie jetzt die Datenstruktur *Struktur* mit der Datenstruktur *Feld* und legen einen ganzen Verein an:

```
1 || Mitglied verein[100];           // Definition eines Vereins
```

Die Bildschirmausgabe des 25. Mitglieds stellt nun kein großes Geheimnis mehr dar, wenn Sie noch daran denken, dass die Nummerierung der Feldindizes bei 0 beginnt:

```
1 || std::cout << "Name: " << verein[24].name << std::endl;
2 || std::cout << "Mitgliedsnr.: " << verein[24].nr; << std::endl;
3 || std::cout << "Straße: " << verein[24].strasse << std::endl;
4 || std::cout << "Telefon: " << verein[24].telefon << std::endl;
```

1.5.6 Blöcke

Definitionen und Anweisungen können in einem *Block* zusammengefasst werden. Dazu dienen die geschweiften Klammern:

```
1 || {
2 ||     Anweisung;
3 ||     Anweisung;
4 ||     ...
5 || }
```

Ein Block ist syntaktisch äquivalent zu einer einzelnen Anweisung, d.h. überall dort, wo eine Anweisung stehen kann, kann auch ein Block stehen. Das offensichtlichste Beispiel ist der Block, der die Anweisungen in einer Funktion, z.B. der *main*-Funktion, zusammenfasst. Weiterhin dienen Blöcke dazu, die Anweisungen hinter einer Kontrollstruktur wie *while* oder *if*⁴ zusammenzufassen.

1.5.7 Gültigkeitsbereiche

Wenn Sie in einen Vergnügungspark gehen und dort eine Eintrittskarte kaufen, so gilt diese Eintrittskarte nur so lange, bis Sie den Park verlassen. Dann wird die Karte ungültig und kann nicht mehr benutzt werden. Beim erneuten Eintreten in den Park müssen Sie eine neue Karte kaufen. So ähnlich funktioniert es auch mit Variablennamen. Der *Gültigkeitsbereich* eines Namens ist der Teil des Programms, in dem der Name benutzt werden kann. Für den Anfang reicht es, wenn Sie sich eine Regel merken: **Ein Name ist nur in dem Block gültig, in dem er definiert wurde.** Mit „Block“ ist hierbei alles gemeint, was in geschweiften Klammern steht, insbesondere auch Funktionen.

In einem Block können Sie beliebige Variablen definieren, die dann nur lokal in diesem Block bekannt sind. Nach dem Verlassen des Blocks wird die Variable ungültig:

```
1 || #include <iostream>
2
3 || int main()
4 || {
5 ||     const double PI = 3.1415;    // PI ist in main() bekannt
6
7 ||     // Beginn des Blockes
8 ||     {
9 ||         double radius;
10 ||         std::cout << "Radius: ";
11 ||         std::cin >> radius;
12 ||         double umfang = 2 * PI * radius;
13 ||         std::cout << "Der Kreisumfang beträgt: " << umfang << std::endl;
14 ||     }
15 ||     // Ende des Blockes
16
17 ||     // Jetzt kommt ein Fehler
18 ||     std::cout << "Nochmal : Der Kreisumfang beträgt: " << umfang << std::endl;
19 ||     return 0;
20 || }
```

⁴Kontrollstrukturen lernen Sie im nächsten Versuch kennen

Beim Versuch, dieses Programm zu kompilieren, meldet der Compiler '*umfang*' was not declared in this scope. Die Variable *umfang* ist an dieser Stelle nicht mehr gültig, da sie innerhalb des vorherigen Blocks deklariert wurde.

Beachten Sie auch, dass Variablen mit gleichem Namen in verschiedenen Blöcken nichts miteinander zu tun haben:

```

1 #include <iostream>
2
3 int main()
4 {
5     // Beginn des 1. Blockes
6     {
7         // Definition einer Variablen wert
8         int wert = 4;
9         std::cout << "Der Wert beträgt: " << wert << std::endl;
10    }
11
12    // Beginn des 2. Blockes
13    {
14        // Definition einer anderen Variablen wert
15        int wert = 2;
16        std::cout << "Der Wert beträgt: " << wert << std::endl;
17    }
18    return 0;
19 }
```

Dies gilt auch für gleiche Namen in verschachtelten Blöcken. Es gilt jeweils die innerste Definition, die weiter außen definierten Variablen behalten zwar ihren Wert, sind jedoch im inneren Block nicht sichtbar:

```

1 #include <iostream>
2
3 int main()
4 {
5     // Definition der Variablen wert im äußeren Block
6     int wert = 4;
7
8     // Beginn des inneren Blockes
9     {
10        // Definition der Variablen wert im inneren Block
11        int wert = 2;
12        std::cout << "Der Wert innen beträgt: " << wert << std::endl; // wert: 2
13    }
14
15    std::cout << "Der Wert außen beträgt immer noch: " << wert << std::endl;
16    return 0;
17 }
```

1.6 Aufgaben

Für die Bearbeitung der Aufgaben werden Ihnen vorgefertigte Codefragmente zur Verfügung gestellt. Diese sind den Aufgabestellungen entsprechend zu ergänzen.

Um die vorgefertigten Codefragmente in Eclipse zu bearbeiten, importieren Sie diese wie auf Seite 7 und auf Seite 8 beschrieben.

1.6.1 Datentypen und Typumwandlung

Erstellen Sie ein neues Projekt *Versuch01Teil1* und importieren Sie die Vorlage *Variablen.cpp*. Implementieren Sie in dieser Datei nun nacheinander folgende Funktionen und testen Sie diese:

1. Fordern Sie den Benutzer auf, zwei ganze Zahlen (*iErste* und *iZweite*) einzugeben, speichern Sie die Summe in *iSumme* und das Ergebnis der Division von *iErste* durch *iZweite* in *iQuotient*.

1 Datenstrukturen und Operatoren

Geben Sie die Ergebnisse mit einer entsprechenden Meldung auf dem Bildschirm aus. Hier und im Folgenden sollten Sie bei der Ausgabe immer einen entsprechenden Hinweis für den Benutzer ausgeben, damit dieser weiß, was sich hinter der Ausgabe von welchen Werten verbirgt.

2. Berechnen Sie die Werte erneut, aber speichern Sie nun in den Variablen *dSumme* und *dQuotient* vom Typ *double* und geben Sie auch diese Ergebnisse aus.
3. Berechnen Sie nun die Summe und den Quotienten abermals. Diesmal allerdings, indem Sie jeden der ganzzahligen Operanden einem Typecasting zum Typ *double* unterziehen. Speichern Sie die Ergebnisse in den Variablen *dSummeCast* und *dQuotientCast* und geben Sie diese aus. Warum unterscheiden sich die Ergebnisse bei entsprechender Wahl der Eingabedaten von denen ohne Typecasting?
4. Lassen Sie den Benutzer seinen Vornamen und seinen Nachnamen (Variablen: *sVorname* und *sNachname* vom Typ *string*) eingeben und speichern Sie den kompletten Namen in der Form „Vorname Nachname“ in *sVornameName* und in der Form „Name, Vorname“ in *sNameVorname*. Nutzen Sie den + -Operator für Strings. Geben Sie beide Formen des Namens aus.
5. Legen Sie für die folgenden Unterpunkte einen eigenen Block an. Ein Block besteht aus einer öffnenden und einer schließenden geschweiften Klammer({ }).
 - a) Legen Sie ein Feld *iFeld* aus Ganzzahlen mit 2 Elementen an und geben Sie den Elementen die Werte 1 und 2. Geben Sie beide Werte aus.
 - b) Erzeugen Sie ein Feld aus 2 mal 3 Elementen mit den Werten 1 bis 6 mit dem Namen *spielfeld*. Geben Sie die Werte aus, 3 pro Zeile mit Leerzeichen dazwischen, 2 Zeilen.
 - c) Definieren Sie eine Konstante *iZweite* mit dem Wert 1 und geben Sie diese aus.

Geben Sie *iZweite* nach Ende des Blockes erneut aus.
6. Wandeln Sie den ersten und den zweiten Buchstaben des Vornamens des Benutzers anhand der ASCII-Tabelle in eine Zahl um (Variablen: *iName1* und *iName2*). Geben Sie diese Zahlen aus.
7. Berechnen Sie anhand dieser Zahlen die Position der Buchstaben im deutschen Alphabet unabhängig von Groß- und Kleinschreibung, z.B. hat *A* den ASCII-Wert 65 und *A* ist der erste Buchstabe im deutschen Alphabet. Der modulo Operator(%) kann dabei sehr hilfreich sein.

1.6.2 Strukturen

Erstellen Sie nun ein weiteres Projekt *Versuch01Teil2* und importieren Sie die Vorlage *Strukturen.cpp*. Deklarieren Sie in dieser Datei eine Struktur mit dem Namen *Person*, die aus den Zeichenketten *sNachname*, *sVorname* und aus den ganzen Zahlen *iGeburtsjahr* und *iAlter* besteht. Definieren Sie eine Variable vom Typ *Person* mit dem Namen *nBenutzer*. Lassen Sie den Benutzer seinen Namen und Vornamen sowie sein Geburtsjahr und sein Alter eingeben. Geben Sie den Inhalt der gesamten Struktur sinnvoll aus.

Kopieren Sie den Inhalt von *nBenutzer* in eine neue Variable.

Kopieren Sie

1. jedes Element einzeln in *nKopieEinzeln* und
2. die gesamte Struktur in *nKopieGesamt*.

Geben Sie beide Ergebnisse aus.

2 Kontrollstrukturen, Referenzen, Zeiger und Funktionen

2.1 Theorie

2.1.1 Zielsetzung und Einordnung

Durch diesen Versuch sollen Sie lernen, mit Hilfe von Funktionen und Kontrollstrukturen Ihr Programm so zu verzweigen, dass Sie auch komplexere Aufgabenstellungen lösen können. Darüber hinaus gibt Ihnen der Einsatz von Schleifen die Möglichkeit, wiederkehrende Programmteile mehrfach ausführen zu lassen, bis ein bestimmtes Ziel erreicht ist. Außerdem sollen fortgeschrittenere Funktionen des Debuggers gezeigt werden, um die nun komplexer werdenden Programme trotzdem effizient debuggen zu können. All dies zusammen versetzt Sie in die Lage, auch größere Aufgaben strukturiert und übersichtlich zu lösen. Setzen Sie diese Möglichkeit ein und gestalten Sie Ihr Programm so, dass andere Studenten sich schnell zurechtfinden. Dies bedeutet auch, Funktionen sinnvoll zu benennen und gegebenenfalls den Quellcode zusätzlich zu *kommentieren*.

Ziel dieses Versuches ist es, Ihnen die grundsätzlichen Prinzipien von Gültigkeitsbereichen, Funktionen und Kontrollstrukturen zu erklären, damit Sie Ihren Programmablauf steuern und die komplexer werdenden Programme übersichtlich halten können. Am Ende dieses Versuches sollten Sie wissen, was

- Kontrollstrukturen
- Referenzen und Zeiger
- Funktionen

sind und wie man diese einsetzen kann.

2.1.2 Einfache Kontrollstrukturen

Kontrollstrukturen dienen dazu, die sequentielle Ausführungsreihenfolge der Anweisungen aufzubrechen. Die häufigsten Kontrollstrukturen sind Verzweigungen, um unter einer von mehreren Alternativen zu wählen, und Schleifen, um einen bestimmten Teil des Programmcodes mehrfach auszuführen. In diesem Abschnitt werden die *if-else*-Verzweigung und ihre Abwandlungen beschrieben, und im nächsten Abschnitt werden die *for*- und *while*-Schleifen thematisiert.

***if-else* Anweisung**

Normalerweise arbeitet ein Programm, beginnend bei der ersten Anweisung, alle Anweisungen der Reihe nach ab, bis die letzte Anweisung ausgeführt wurde. Dann stoppt es. Eine Verzweigung des Programmes aufgrund von bestimmten Bedingungen, z.B. Eingaben, wird erst durch die *if-else*-Anweisung ermöglicht. Formal gilt folgende Syntax, wobei der *else*-Teil entfallen kann.

```
1 | if (expression)
2 | {
3 |     statement1;
4 | }
5 | else
6 | {
7 |     statement2;
8 | }
```

Der Ausdruck *expression* kann jeder Ausdruck sein, der als Ergebnis die Wahrheitswerte *true* oder *false* liefert. Trifft die Bedingung zu (hat *expression* den Wert *true*, so wird die Anweisung *statement1* ausgeführt. Ist die Bedingung nicht erfüllt (hat *expression* den Wert *false*), so wird *statement2* ausgeführt, sofern der *else*-Teil vorhanden ist. Beachten Sie, dass es sich bei einem *statement* um eine einzelne Anweisung oder um einen Block, also um eine in geschweifte Klammern eingeschlossene Kette von Anweisungen, handeln kann. Ein abschließendes Semikolon ist immer fester Bestandteil einer einzelnen Anweisung.

else-if-Anweisung

Manchmal stehen nicht nur zwei, sondern gleich mehrere Alternativen zur Verfügung. Um unter diesen einer Entscheidung zu treffen, wird das *else-if*-Statement verwendet:

```

1  if (expression1)
2  {
3      statement1;
4  }
5  else if (expression2)
6  {
7      statement2;
8  }
9  else if (expression3)
10 {
11     statement3;
12 }
13 else
14 {
15     statement4;
16 }
```

Hierbei kann *expression{1, 2, 3}* wiederum ein beliebiger Ausdruck sein, der als Ergebnis *true* oder *false* liefert. Es wird eine *expression* nach der anderen bewertet. Sobald eine dieser Bedingungen zutrifft, wird das zugehörige *statement* ausgeführt und die Kette beendet. Trifft keine der Bedingungen zu, so wird der *else*-Teil ausgeführt, sofern er vorhanden ist. Wie gehabt, kann es sich bei *statement* um eine einzelne Anweisung oder um einen Block handeln. Im Folgenden wird dies nicht weiter explizit erwähnet.

Beispiel

```

1 #include <iostream>
2
3 int main()
4 {
5     double entfernung = 0;
6
7     std::cout << "Geben Sie die zu überbrückende Entfernung in km ein : ";
8     std::cin >> entfernung;
9
10 if (entfernung <= 1)
11 {
12     std::cout << "Benutzen Sie doch mal wieder Ihre Beine." << std::endl;
13 }
14 else if (entfernung <= 3)
15 {
16     std::cout << "Nehmen Sie Ihr Rad!" << std::endl;
17 }
18 else if (entfernung <= 42)
19 {
20     std::cout << "Entweder per Auto oder per Anhalter..." << std::endl;
21 }
22 else
23 {
24     std::cout << "Mondsüchtig?" << std::endl;
25 }
```

```
26 }     return 0;
27 }
```

Das Beispielprogramm bildet ein (zugegebenermaßen noch sehr rudimentäres) Expertensystem zur Wahl eines geeigneten Verkehrsmittels bei gegebener Entfernung.

Geben Sie die zu überbrückende Entfernung in km ein : 1
Benutzen Sie doch mal wieder Ihre Beine.

Geben Sie die zu überbrückende Entfernung in km ein : 90000
Mondsüchtig?

Geben Sie die zu überbrückende Entfernung in km ein : 42
Entweder per Auto oder per Anhalter...

switch-Anweisung

Besitzt ein Ausdruck einen *ganzzahligen* Wert (int, long, char,...), so kann auch die *switch*-Anweisung verwendet werden, um in Abhängigkeit des Wertes im Programmcode zu verzweigen:

```
1 switch (expression)
2 {
3     case constValue1 : statement1;
4     case constValue2 : statement2;
5     default: statement3;
6 }
```

Jede *case*-Marke leitet eine Alternative ein. Hat die *expression* bei *switch* den Wert einer der *constValues* bei *case*, so wird die Ausführung bei dieser *case*-Marke fortgesetzt. Passt keiner der konstanten Ausdrücke, wird die Programmausführung bei der *default*-Marke fortgesetzt, welche optional ist.

```
1 #include <iostream>
2
3 int main()
4 {
5     char zeichen;
6     std::cout << "Geben Sie ein Zeichen ein : ";
7     std::cin >> zeichen;
8     switch (zeichen)
9     {
10         // x wird durch * ersetzt
11         case 'x':
12             zeichen = '*';
13             break;
14         // Vokale werden durch ? ersetzt
15         case 'a':
16         case 'e':
17         case 'i':
18         case 'o':
19         case 'u':
20             zeichen = '?';
21             break;
22         // Alle anderen werden durch Leerzeichen ersetzt
23         default:
24             zeichen = ' ';
25             break;
26     }
27
28     std::cout << "Ihre Eingabe wurde umgewandelt zu '" << zeichen << "'" << std::endl;
29     return 0;
30 }
```

Das Programm erwartet die Eingabe eines Buchstabens und wandelt diesen entweder in ein '*', '?' oder Leerzeichen um:

```
Geben Sie ein Zeichen ein : q
Ihre Eingabe wurde umgewandelt zu ''
```

```
Geben Sie ein Zeichen ein : e
Ihre Eingabe wurde umgewandelt zu '??'
```

```
Geben Sie ein Zeichen ein : x
Ihre Eingabe wurde umgewandelt zu '*'
```

Beachten Sie, dass es sich bei *case* um eine Einsprungsmarke handelt, von der aus der Programmablauf fortgesetzt wird. Insbesondere werden auch die Anweisungen der nachfolgenden *case*-Labels ausgeführt, solange bis auf ein *break*-Statement getroffen wird (sog. *fall-through*, in diesem Beispiel bei den Vokalen zu sehen). Im Regelfall wird daher jeder Block hinter einem *case*-Label durch ein *break*-Statement abgeschlossen. Ein vergessenes *break* in einem *case*-Label ist einer der beliebtesten Programmierfehler in C++. Achten Sie also beim Verwenden einer *switch*-Anweisung besonders darauf!

2.1.3 Weitere Kontrollstrukturen: Schleifen

Sie haben nun die Möglichkeiten kennengelernt, im Programmcode zu verzweigen. Weitere, wichtige Kontrollstrukturen sind Schleifen, die es ermöglichen, eine Folge von Anweisungen wiederholt zu durchlaufen, solange eine bestimmte Bedingung erfüllt ist. Eine Form einer solchen Schleife stellt die *for*-Anweisung dar.

for-Anweisung

Formal dargestellt sieht die *for*-Anweisung wie folgt aus:

```
1 || for (expr1; expr2; expr3)
2 | {
3 |   statement;
4 | }
```

Zunächst wird *expr1* ausgewertet (Initialisierung), bei der es sich normalerweise um eine Zuweisung oder einen Funktionsaufruf handelt. Anschließend erst erfolgt die eigentliche Iteration der Schleife. Dazu wird zunächst *expr2* ausgewertet, bei der es sich meist (jedoch nicht notwendigerweise) um einen Vergleich handelt. Wird der Ausdruck zu *true* ausgewertet, wird der Schleifenrumpf betreten, d.h. *statement* wird ausgeführt. Nach Durchlauf des Schleifenrumpfes (und erst dann!), wird *expr3* ausgewertet, bei dem es sich wiederum meist um eine Zuweisung oder einen Funktionsaufruf handelt. Nun ist die Schleife einmal durchlaufen worden und das Spiel beginnt von neuem, d.h. *expr2* wird ausgewertet. Der Vorgang wird so lange wiederholt, bis *expr2 false* ergibt. In diesem Fall wird die Programmausführung mit der ersten Anweisung hinter der Schleife fortgesetzt. Anschaulich lässt sich die *for*-Anweisung in Pseudocode so schreiben:

```
1 || expr1;
2 || solange expr2 erfüllt ist
3 | {
4 |   statement;
5 |   expr3;
6 | }
```

Sie haben beispielsweise ein Feld mit 10 Einträgen, die sie ausgeben wollen:

```
1 || int feld = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 || for (int i = 0; i < 10; i++)
3 | {
4 |   std::cout << feld[i] << std::endl;
5 | }
```

Es ist möglich, *for*-Schleifen ineinander zu verschachteln. Dies ist nützlich, um beispielsweise auf einem mehrdimensionalen Feld zu arbeiten. Zwei Schleifen, die jede Zeile eines zweidimensionalen Felds mit den Ziffern 0-9 füllen, könnten zum Beispiel so aussehen:

```

1 || int feld[10][10];
2 || for (int y = 0; y < 10; y++)
3 || {
4 ||     for (int x = 0; x < 10; x++)
5 ||     {
6 ||         feld[y][x] = x;
7 ||     };
8 || };

```

In diesem Fall wird also in jedem Schleifendurchlauf der ersten Schleife die zweite einmal vollständig durchlaufen. Insgesamt wird jedes der 100 Feldelemente des Felds angesprochen.

while-Anweisung

Normalerweise verwendet man eine *for*-Schleife dann, wenn man schon von vornherein die Anzahl der Schleifendurchläufe kennt (vgl. das Laufbeispiel aus dem letzten Abschnitt). Dies muss nicht immer der Fall sein. Dafür gibt es in C++ die *while*-Schleife, bei der vor jedem Durchlauf überprüft wird, ob eine weitere Wiederholung erwünscht wird:

```

1 || while (expression)
2 || {
3 ||     statement;
4 || }

```

Zunächst wird die Bedingung *expression* bewertet. Ist *expression* gleich *true*, wird *statement* ausgeführt. Anschließend wird *expression* erneut bewertet. Dies wiederholt sich, bis *expression* den Wert *false* hat. Die Ausführung des Programms wird dann hinter *statement* fortgesetzt. Beachten Sie, dass die Anzahl der Schleifendurchläufe also auch Null betragen kann, falls die Bedingung von vorneherein nicht erfüllt ist.

Das folgende Programm gibt beispielsweise die Fakultät von 7 aus:

```

1 || #include <iostream>
2
3 || int main()
4 || {
5 ||     int n = 7;
6 ||     int fakultaet = n;
7
8 ||     while(n > 2)
9 ||     {
10 ||         n = n - 1;
11 ||         fakultaet = fakultaet * n;
12 ||     }
13
14 ||     std::cout << fakultaet << std::endl;
15 || }

```

Interessant ist noch, dass jede *for*-Schleife in eine äquivalente *while*-Schleife überführt werden kann und umgekehrt. Das Konstrukt

```

1 || for (expr1; expr2; expr3)
2 || {
3 ||     statement;
4 || }

```

ist identisch zu:

```

1 || expr1;
2 || while (expr2)
3 || {
4 ||     statement;
5 ||     expr3;
6 || }

```

do-while-Anweisung

Charakteristisch für eine *while*-Schleife ist, dass die Anzahl der Schleifendurchläufe auch Null betragen kann. Soll eine Schleife mindestens einmal durchlaufen werden, kann stattdessen auch die *do-while*-Anweisung zum Einsatz kommen:

```

1 do
2 {
3     statement;
4 }
5 while (expression);

```

Zunächst wird das *statement* ausgeführt. Anschließend wird die Bedingung *expression* bewertet. Ist ihr Wert *true*, wird *statement* erneut ausgeführt. Dies wiederholt sich so lange, bis *expression* *false* ist und die Schleife damit beendet wird.

Als Beispiel dient hier ein kleines Spiel, bei dem man eine Zahl zwischen 1 und 10 raten soll:

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main()
6 {
7     int geheimnis;
8     int versuch;
9
10    srand(time(nullptr)); // Initialisierung Zufallsgenerator
11    secret = rand() % 10 + 1; // Zufallszahl zwischen 1 und 10
12
13    do
14    {
15        std::cout << "Geben Sie eine Zahl zwischen 1 und 10 ein: " << std::endl;
16        std::cin >> versuch;
17    }while(geheimnis != versuch);
18    std::cout << "Herzlichen Glückwunsch!" << std::endl;
19 }

```

break-Anweisung

Mit der *break*-Anweisung können alle Schleifen (*for*-, *while*- und *do*) vorzeitig verlassen werden, ohne dass die Abbruchbedingung geprüft wird. Z.B. kann man hierdurch auch „unendliche“ Schleifen programmieren, die dann bei Eintreten einer bestimmten Bedingung über die *break*-Anweisung verlassen werden:

```

1 #include <iostream>
2
3 int main()
4 {
5     int zahl = 0;
6     // unendliche Schleife starten
7     while (true)
8     {
9         // Eingeben einer Zahl
10        std::cout << "Geben Sie eine Zahl zwischen 1 und 10 ein: " ;
11        std::cin >> zahl;
12        // Zahl innerhalb des Wertebereichs?
13        if (zahl >= 1 && zahl <= 10)
14        {
15            break; // verlassen von while
16        }
17        // Fehlermeldung
18        std::cout << "Das war wohl nichts, versuchen Sie es nochmal!" << std::endl;
19    }
20    std::cout << "Ihre Zahl war " << zahl << std::endl;
21    return 0;
22 }

```

Ein Beispiellauf sieht wie folgt aus:

```
Geben Sie eine Zahl zwischen 1 und 10 ein : -1
Das war wohl nichts, versuchen Sie es nochmal!
Geben Sie eine Zahl zwischen 1 und 10 ein : 19
Das war wohl nichts, versuchen Sie es nochmal!
Geben Sie eine Zahl zwischen 1 und 10 ein : 3
Ihre Zahl war 3
```

Zu beachten ist noch, dass bei verschachtelten Schleifen durch eine *break*-Anweisung immer die *innerste* Schleife verlassen wird.

continue-Anweisung

Mit der *continue*-Anweisung kann innerhalb von Schleifen (*for*-, *while*- und *do*) unmittelbar der nächste Schleifendurchlauf begonnen werden. Bei einer *for*-Schleife wird hierbei zunächst noch die iterierte Anweisung (*expr3* in der Beschreibung der *for*-Schleife) ausgeführt.

Beispielsweise lassen sich auf diese Weise nur die nicht-negativen Elemente eines Feldes bearbeiten.

```
1 #include <iostream>
2
3 const int MAX = 10;
4
5 void ausgabe(int feld[MAX])
6 {
7     for (int i = 0; i < MAX; i++)
8     {
9         std::cout << feld[i] << " ";
10    }
11    std::cout << std::endl;
12}
13
14 int main()
15 {
16     // Feld initialisieren
17     int feld[10] = { -1, 3, 5, -2, -7, -9, 9, 8, 0, -7 };
18     ausgabe(feld);
19
20     // Positive Elemente Nullsetzen
21     for (int i = 0; i < MAX; i++)
22     {
23         // negative Elemente überspringen
24         if (feld[i] < 0)
25         {
26             continue;
27         }
28
29         feld[i] = 0;
30     }
31     ausgabe(feld);
32     return 0;
33 }
```

Innerhalb der *for*-Schleife wird zunächst überprüft, ob das aktuelle Feldelement kleiner Null ist. Falls ja, wird die *continue*-Anweisung ausgeführt. Diese bewirkt automatisch die Ausführung der iterierten Anweisung (*i++*), d.h. die Laufvariable wird erhöht und unmittelbar der nächste Schleifendurchlauf gestartet. War das Feldelement größer oder gleich Null, wird ganz normal die Schleife weiter abgearbeitet, d.h. das Feldelement wird zu Null gesetzt und erst dann der nächste Schleifendurchlauf begonnen. Dieses Beispielprogramm produziert folgende Ausgabe:

```
-1 3 5 -2 -7 -9 9 8 0 -7
-1 0 0 -2 -7 -9 0 0 0 -7
```

Eine *continue*-Anweisung wird gerne dazu verwendet, eine große Verschachtelungstiefe (die bei mehrfachem Verwenden von *if-else* unweigerlich auftritt) zu vermeiden, da ansonsten der Programmcode schlecht lesbar wird.

2.1.4 Referenzen und Zeiger

Referenzen sind Aliasnamen für bereits existierende Variablen. Eine Referenz existiert also nie allein, sondern ist immer an eine Variable gebunden. Ändert man eine Referenz, so ändert man dadurch auch die Variable, die durch die Referenz referenziert wird. Referenzen müssen bei ihrer Erzeugung immer an eine Variable gebunden werden, sonst liefert der Compiler einen Fehler. Nach der Deklaration lässt sich eine Referenz verwenden, als handele es sich um die entsprechende Variable. Über den Typ der Referenz (z.B. `double&`) wird festgehalten, welcher Datentyp sich an der Speicheradresse befindet.

Zeiger sind Variablen, die eine Speicheradresse enthalten. Über den Typ des Zeigers (z.B. `int*`) wird festgehalten, welcher Datentyp sich an der Speicheradresse befindet. Im Gegensatz zum Zeiger kann die Adresse einer Referenz nicht wieder verändert werden.

```
1 || int zahl;
2 || int* zeiger = nullptr;
3 || int& referenz = zahl;
```

Ohne Initialisierung der Referenz meldet der Compiler einen Fehler. Zeiger müssen dagegen nicht zwingend initialisiert werden, verweisen in diesem Fall jedoch abhängig vom verwendeten Compiler unter Umständen auf einen zufallsbestimmten Speicherbereich. Daher sollten sie zunächst mit einem `nullptr`-Zeiger initialisiert werden. Eine alternative Zeigerinitialisierung wäre:

```
1 || int* zeiger = &zahl;
```

Der verwendete Adresse-Operator `&` gibt die Adresse der jeweiligen Variable zurück (hier: Adresse der Variable `zahl`) und hat dabei eine andere Bedeutung als bei der Verwendung des `&` bei der Deklaration einer Referenz.

Möchte man eine lesende Operation auf der Zeiger-Variable durchführen, so muss der Zeiger zunächst mit dem Operator `*` dereferenziert werden:

```
1 || int zahlKopie = *zeiger;
```

Im Gegensatz zu Referenzen kann die im Zeiger gespeicherte Adresse zur Laufzeit geändert werden. Bei unsachgemäßem Einsatz kann der Zeiger auf andere Datentypen oder gar auf nicht verwendete Speicherbereiche verweisen. Daher sollte man vor jedem Einsatz eines Zeigers prüfen, ob nicht auch eine Referenz für das zu lösende Problem ausreicht.

Das folgende Beispiel soll anhand der so genannten Zeigerarithmetik verdeutlichen, welche Einsatzmöglichkeiten und Gefahren sich bei der Verwendung von Zeigern ergeben: Bei der Definition eines Feldes sorgt der Compiler dafür, dass der benötigte Speicher zur Laufzeit reserviert wird. Gängige C++-Compiler für PC-Plattformen reservieren für eine Integer-Variable 4 Byte, so dass für ein Feld mit 5 Integerwerten insgesamt 20 Byte benötigt werden. Ab dem 21. Byte folgen andere Variablen, die im weiteren Verlauf des Programmes benötigt werden können. Da ein Feld C++-intern über Zeiger realisiert ist, lässt es sich mit Hilfe eines zusätzlichen Zeigers manipulieren.

```
1 || int feld[5] = { 3, 6, 9, 12, 15 };
2 || int* feldZeiger = feld;
```

Die Variable `feldZeiger` verweist hier zunächst auf das erste Element des Feldes (eine gleichwertige Initialisierung wäre `int* feldZeiger = &feld[0];`). Inkrementiert man nun den Zeiger mittels `++feldZeiger`, so wird dessen Adresse automatisch um 4 Byte erhöht, so dass er auf das zweite Element verweist. Bei dieser Operation wird jedoch nicht geprüft, ob die neue Adresse weiterhin auf einen gültigen Speicherbereich verweist, der eine 4-Byte-Integer-Variable enthält. Nach fünfmaligem Inkrementieren verlässt der Zeiger damit ungeprüft den gültigen Speicherbereich. Ein Schreibvorgang außerhalb des gültigen Bereiches ändert die dort gespeicherten Variablen und kann somit während des weiteren Verlaufs des Programms zu unerwünschten und nicht nachvollziehbaren Fehlern führen.

2.1.5 Speicherbereich und Sichtbarkeit

Der Gültigkeitsbereich einer Variablen bezieht sich auf den Bereich eines Programms, in dem für diesen eine Adresse im Arbeitsspeicher reserviert ist. Dies ist in der Regel innerhalb des durch { und } gebildeten Blockes der Fall, in dem sie definiert wurde. Der Sichtbarkeitsbereich einer Variablen beschreibt den Bereich eines Programms, in dem der Programmentwickler explizit lesend und schreibend auf ihre Adresse im Arbeitsspeicher zugreifen kann.

Die Programmiersprache *C++* unterscheidet drei elementare Speicherklassen:

- **Automatischer Speicher:** Hier werden lokale Variablen und Funktionsargumente abgelegt. Sie werden automatisch bei der Definition angelegt und am Ende ihres Gültigkeitsbereichs wieder gelöscht. Die Sichtbarkeit innerhalb eines Gültigkeitsbereichs kann verdeckt werden, wenn mit { und } ein neuer Bereich gebildet wird, in dem sich eine Variable gleichen Namens befindet. Aus Gründen der Übersichtlichkeit und vereinfachten Fehlersuche sollte dies jedoch möglichst vermieden werden.
- **Statischer Speicher:** Im statischen Speicher werden globale Variablen, Variablen von Namensbereichen, statische Klassenelemente und statische Variablen in Funktionen abgelegt. Derartige Objekte existieren und behalten ihren Wert während der gesamten Ausführung des Programms. Allerdings ist ihre Sichtbarkeit nur auf den Block beschränkt, in dem die Variable deklariert wurde.
- **Dynamischer Speicher:** Diese Speicherklasse wird benutzt, wenn Speicherplatz zur Laufzeit explizit angefordert wird (Allokation). Diese Art Speicher wird auch *Heap* genannt und muss vom Programm selbst wieder explizit freigegeben werden.

2.1.6 Dynamische Speichernutzung

Gewöhnlich haben Objekte eine Lebensdauer, die durch ihren Gültigkeitsbereich bestimmt wird. Manchmal ist es jedoch sinnvoll, Objekte zu erzeugen, deren Lebensdauer vom aktuellen Gültigkeitsbereich unabhängig ist bzw. deren Art und Anzahl erst zur Ausführung des Programms bekannt ist. Die Operatoren *new* und *delete* bieten die Möglichkeit, Speicherplatz zur Laufzeit zu allozieren, um die gewünschten Datenelemente anzulegen.

Bei dieser so genannten dynamischen Programmierung ist der *C++*-Entwickler selbst für die Speicherverwaltung verantwortlich. Er muss daher darauf achten, dass alle Datenelemente, die mit *new* explizit angelegt wurden, auch wieder explizit mit *delete* gelöscht werden. Vergisst der Programmierer, nicht mehr benötigte Datenelemente zu löschen, dann bleibt der Speicher bis zum Ende des Programms reserviert. Diese Art der Speicherverschwendungen nennt man Speicherleck. Gibt es weiterhin Zeiger auf ein Objekt, obwohl dieses per *delete* gelöscht wurde, so kann es zu Speicherschutzverletzungen (auch *Segmentation Fault* genannt) kommen. Solche Verweise werden auch „wilde“ Zeiger genannt.

```
1 || int* intZeiger = new int;
2 || delete intZeiger;
```

Die Variable *intZeiger* enthält die Adresse des neu erzeugten Speicherplatzes, der sich auf dem so genannten Heap befindet. Nachdem dieser dynamisch zur Laufzeit allozierbare Speicherplatz durch *delete* wieder freigegeben wurde, zeigt *intZeiger* auf einen ungültigen Speicherbereich, in dem im weiteren Verlauf des Programmes beliebige andere Variablen abgelegt werden können. Daher sollte man nach der Speicherplatzfreigabe grundsätzlich dem Zeiger *nullptr* zuweisen (*intZeiger = nullptr*).

2.1.7 Funktionen

Funktionen sind das Salz in der Programmiersuppe. Eine Funktion ist eine Zusammenfassung von Anweisungen in einer Einheit. Sie erhält i.A. Argumente, mit denen sie rechnet, und liefert ein bestimmtes Ergebnis zurück. Es kann aber auch Funktionen geben, die keine Argumente bekommen und/oder auch kein Ergebnis zurückliefern.

Eine Funktion in C++ können Sie problemlos mit einer mathematischen Funktion vergleichen. Betrachten Sie hierzu die mathematische Funktion $z = f(x, y) = \sqrt{x^2 + y^2}$. Diese Funktion erhält als Argumente die reellen Zahlen x und y und liefert als Ergebnis die reelle Zahl z zurück.

Eine Funktionsdefinition in C++ spezifiziert zunächst den Datentyp des zurückgelieferten Ergebnisses, dann den Namen der Funktion und schließlich die Typen und Namen der Parameter:¹

```
1 || double f(double x, double y);
```

Die obige Zeile zeigt den Kopf (die sog. Signatur) einer Funktion, die einen Gleitkommawert zurückliefert, den Namen f hat und als Parameter zwei Gleitkommawerte erhält, auf die sie über die Namen x und y zugreift.

Das Zurückliefern eines Wertes aus einer Funktion geschieht mit Hilfe der *return*-Anweisung. Diese bewirkt ein sofortiges Hinausspringen aus der Funktion. Der Programmablauf wird an der Stelle fortgesetzt, an der die Funktion aufgerufen wurde. Die Implementierung obiger, mathematischer Funktion sieht also so aus:

```
1 || double f(double x, double y)
2 {
3     double z = sqrt(x * x + y * y);
4     return z;
5 }
```

Hierbei bezeichnet *sqrt* die Wurzelfunktion (*square root*), zu deren Benutzung die Headerdatei *<cmath>* eingebunden werden muss. Beachten Sie also, dass Sie zur Implementierung Ihrer Funktion bereits eine weitere Funktion verwendet haben!

Sie sehen, dass man von einer einmal entworfenen Funktion nicht wissen muss, *wie* sie eine Aufgabe löst, sondern lediglich, *was* die Funktion tut und wie die Funktion aufgerufen wird. Man spricht davon, dass zur Benutzung einer Funktion lediglich ihre Schnittstelle, nicht jedoch ihre Implementierung bekannt sein muss. Im obigen Beispiel wissen Sie nicht, wie die Wurzelfunktion implementiert wurde. Es reicht, dass Sie wissen, dass die Quadratwurzel der übergebenen Zahlen geliefert wird.

Das folgende Beispiel verdeutlicht das Einbinden einer Funktion in ein Gesamtprogramm. Das Programm besteht aus der soeben definierten Funktion $f(x, y)$ und der Funktion *main()*. Letztere Funktion muss in jedem C++-Programm vorhanden sein, da sie automatisch beim Start des Programms aufgerufen wird.

```
1 #include <iostream>
2 #include <cmath>
3
4 double f(double x, double y)
5 {
6     // Kürzere Implementierung als oben:
7     return sqrt(x * x + y * y);
8 }
9
10 int main()
11 {
12     double x;
13     std::cout << "x = ";
14     std::cin >> x;
15     double y;
16     std::cout << "y = ";
17     std::cin >> y;
18
19     double z = f(x, y);
20     std::cout << "sqrt(" << x << "^2 + " << y << "^2) = "
21             << z << std::endl;
22
23     return 0;
24 }
```

¹In der Literatur wird manchmal zwischen den Begriffen Argument und Parameter unterschieden. Hier werden beide Begriffe synonym verwendet.

Das Programm bindet zunächst ein paar Header ein und implementiert die Funktion $f(x, y)$. Die Funktion `main()`, die im Folgenden auch synonym mit dem Wort Hauptprogramm bezeichnet wird, erwartet die Eingabe der Werte für x und y . Anschließend wird die Funktion $f(x, y)$ aufgerufen. Der Ausdruck $f(x, y)$ stellt nach Auswertung der Funktion genau das Funktionsergebnis dar und wird in der Variablen z abgespeichert. Somit kann ein Funktionsaufruf überall dort stehen, wo ein Wert erwartet wird. Ein Beispielauf des Programms auf der Konsole sieht wie folgt aus:

```
x = 3
y = 4
sqrt(3^2 + 4^2) = 5
```

Gültigkeitsbereich von Funktionsparametern

Die Argumentnamen einer Funktionsdefinition (bisher x und y) sind nur lokal in der Funktion gültig. Sie haben nichts zu tun mit den Namen der Variablen, die beim Aufruf an die Funktion übergeben werden. Das (etwas abgeänderte) Beispiel von oben ließe sich also auch so schreiben:

```
1 #include <cmath>
2
3 double f(double a, double b)
4 {
5     return sqrt(a * a + b * b);
6 }
7
8 int main()
9 {
10    double x = 3, y = 4;
11    double z = f(x, y);      // Aufruf der Funktion
12    return 0;
13 }
```

Konstante Funktionsparameter

Es ist auch möglich, Funktionsparameter als `const` zu deklarieren. Das ist immer dann sinnvoll, wenn ein Parameter innerhalb einer Funktion nicht verändert, sondern zum Beispiel nur ausgegeben wird. Wenn man einheitlich alle Parameter, die nicht verändert werden, konstant setzt, erhöht das deutlich die Übersichtlichkeit und verringert die Fehleranfälligkeit.

Der Datentyp `void`

Im Zusammenhang mit Funktionen gibt es den speziellen Datentyp `void`. Gibt eine Funktion keinen Wert zurück, so wird ihr Rückgabetyp als `void` spezifiziert:

```
1 // Funktion liefert kein Ergebnis
2 void printValue(const int value)
3 {
4     std::cout << value << std::endl;
5 }
```

Werden der Funktion keine Argumente übergeben, so kann die Argumentliste in der Funktionsdefinition den Datentyp `void` enthalten. Eine leere Argumentliste ist hingegen gebräuchlicher. Der Funktionsaufruf erfolgt mit leerer Argumentliste:

```
1 // Funktionsdefinition ohne Argumente
2 double pi(void)
3 {
4     return 3.14159265;
5 }
6 int main()
7 {
8     double value;
```

```

9 // Funktionsaufruf mit leerer Argumentliste
10 value = 2 * pi();
11 return 0;
12 }
```

Call by value

Funktionsparameter werden in C++ normalerweise als Wert übergeben (*Call by value*). Für das Beispiel aus Abschnitt 2.1.7 heißt das, dass der Wert von *x* beim Aufruf der Funktion in *a* kopiert wird. Analoges gilt für *y* und *b*. Im Speicher sähe das etwa wie folgt aus - siehe Abbildung 2.1.

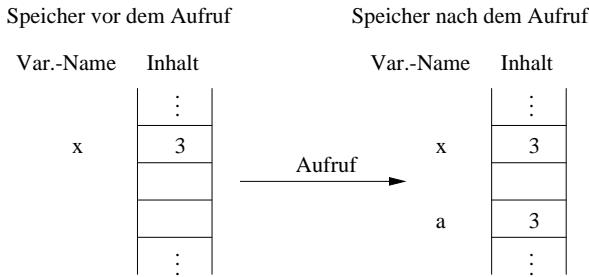


Abbildung 2.1: Call by value

Eine Veränderung des Wertes von *a* hat also keinen Einfluss auf den Wert von *x*.

Call by reference

Statt *Call by value* lässt sich in C++ auch *Call by reference* realisieren. Dabei stehen zwei Möglichkeiten zur Verfügung: *Zeiger* und *Referenzen*.

Zeiger sind recht umständlich in der Handhabung. Sie müssen jedesmal dereferenziert werden, wenn der dahinterstehende Wert benötigt wird.

C++ bietet daher die Möglichkeit, ein Funktionsargument als Referenz zu übergeben. Dies bedeutet für das obige Beispiel, dass der Wert von *x* nun nicht mehr in *a* kopiert wird, sondern *a* stellt lediglich einen synonymen Namen (eine Referenz) für dieselbe Speicherzelle (also für *x*) dar. Jede Veränderung von *a* bedeutet auch eine Veränderung von *x* - siehe Abbildung 2.2.

Der Vorteil einer Referenz gegenüber einem Zeiger besteht darin, dass sie sich fast wie ein Zeiger verhält, man aber nicht immer daran denken muss, diesen auch zu dereferenzieren. Man benutzt einfach den Namen.

Ein Funktionsargument kennzeichnen Sie als Referenz, indem Sie in der Funktionsdefinition dem Namen des Arguments ein *Ampersand* (&) voranstellen. Ansonsten ändert sich nichts, weder die Benutzung des Arguments im Funktionsrumpf noch der Aufruf im Hauptprogramm.

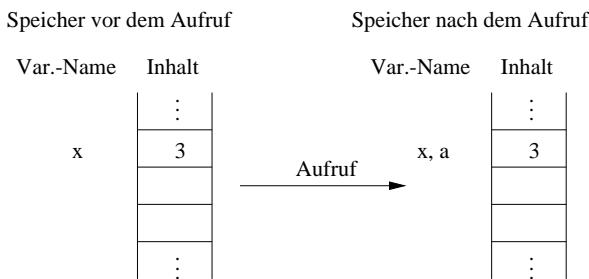


Abbildung 2.2: Call by reference

Der Unterschied zwischen *Call by reference* und *Call by value* wird bei Funktionen deutlich, welche die Werte ihrer Argumente verändern. Betrachten Sie das folgende kleine Beispielprogramm:

```

1 #include <iostream>
2
3 void inc(int value)
4 {
5     value++;
6     std::cout << "value ist " << value << std::endl;
7 }
8
9 int main() // Hauptprogramm
10 {
11     int x = 0;
12     std::cout << "Zunächst ist x = " << x << std::endl;
13     inc(x);
14     std::cout << "Nach dem Inkrementieren ist x = " << x << std::endl;
15 }
```

Der Wert x wird im Hauptprogramm per Call by value an die Funktion `inc(int value)` übergeben, welche den Wert um eins erhöht. Trotzdem bleibt die Variable x des Hauptprogramms davon unberührt:

```

Zunächst ist x = 0
value ist 1
Nach dem Inkrementieren ist x = 0
```

Übergibt man die Variable jedoch per *Call by reference*, indem man Zeile 3 zu

```
1 void inc(int& value)
```

ändert, wird auch x inkrementiert:

```

Zunächst ist x = 0
value ist 1
Nach dem Inkrementieren ist x = 1
```

Felder als Funktionsargumente

Sie haben soeben gelernt, dass in C++ Argumente normalerweise per *Call by value* an Funktionen übergeben werden. Keine Regel ohne Ausnahme: Felder werden *immer* als Referenz übergeben, ohne dass Sie etwas dafür tun müssten (oder dagegen tun könnten). In dem Beispiel

```

1 #include <iostream>
2
3 // Funktion erwartet Feld als Argument:
4 void aendern(int feld[2][2])
5 {
6     feld[0][0] = 2;
7 }
8
9 void feldAusgeben(const int feld[2][2])
10 {
11     std::cout << feld[0][0] << " " << feld[0][1] << std::endl;
12     std::cout << feld[1][0] << " " << feld[1][1] << std::endl;
13 }
14
15 int main()
16 {
17     // Initialisierung:
18     int werte[2][2] = {{1, 2}, {3, 4}};
19     // Ausgabe:
20     std::cout << "Das Feld lautet " << std::endl;
21     feldAusgeben(werte);
22     // Aufruf der Funktion mit einem Feld
23     aendern(werte);
24     // Ausgabe:
25     std::cout << "Das Feld ist nun " << std::endl;
26     feldAusgeben(werte);
27     return 0;
28 }
```

ist der Name *feld* in der Funktion *aendern()* lediglich ein Synonym für den Namen des übergebenen Feldes (in diesem Fall *werte*), daher wirken sich Änderungen an *feld* auch auf *werte* aus. Das Beispielprogramm produziert folgende Ausgabe:

```
Das Feld lautet
1 2
3 4
Das Feld ist nun
2 2
3 4
```

Wie aus dem Beispiel ersichtlich ist, werden Felder ohne Größenangaben, also nur mit dem Namen, an eine Funktion übergeben. Die Parameter in der Funktion, hier `int feld[2][2]` bzw. `const int feld[2][2]`, werden mit der Größe des Felds angegeben, damit der Compiler weiß, dass hier ein Feld übergeben werden soll. Rufen Sie die Funktion *aendern()* mit `aendern(feld[2][2])`; auf, dann interpretiert der Compiler dies als das Element `[2][2]` aus *feld* und versucht, den entsprechenden Integerwert, den es dazu auch noch nicht mal gibt, zu übergeben. Das führt natürlich zu einem Fehler, weil er in der Funktion einen Zeiger auf ein Feld erwartet und keinen Integerwert (*invalid conversion from 'int' to 'int**).

Funktionsdeklarationen

In den bisherigen Beispielen wurden die Funktionen immer vor der *main()*-Funktion, in der sie dann aufgerufen wurden, definiert. Dadurch war dem Compiler bekannt, worum es sich bei dem Funktionsaufruf handelt. Dies muss aber nicht immer sinnvoll oder möglich sein.

In C++ ist es möglich, und auch gebräuchlich, Funktionen vor ihrer ersten Verwendung nur zu deklarieren. Dadurch kennt der Compiler die Funktion samt Übergabeparameter und Rückgabewert und kann dies beim Funktionsaufruf berücksichtigen. Obiges Beispiel lässt sich auch folgendermaßen implementieren.

```
1 #include <iostream>
2
3 void aendern(int feld[2][2]);
4 void feldAusgeben(const int feld[2][2]);
5
6 int main()
7 {
8 // Initialisierung:
9 int werte[2][2] = {{ 1, 2 }, { 3, 4 }};
10 // Ausgabe:
11 std::cout << "Das Feld lautet " << std::endl;
12 feldAusgeben(werte);
13 // Aufruf der Funktion mit einem Feld
14 aendern(werte);
15 // Ausgabe:
16 std::cout << "Das Feld ist nun " << std::endl;
17 feldAusgeben(werte);
18 return 0;
19 }
20
21 // Funktion erwartet Feld als Argument:
22 void aendern(int feld[2][2])
23 {
24 feld[0][0] = 2;
25 }
26
27 void feldAusgeben(const int feld[2][2])
28 {
29 std::cout << feld[0][0] << " " << feld[0][1] << std::endl;
30 std::cout << feld[1][0] << " " << feld[1][1] << std::endl;
31 }
```

Die Funktionen werden dem Compiler nur bekannt gemacht, die Implementierung kann woanders erfolgen, auch in einer anderen Datei. Wichtig ist dabei, dass die Deklarationen im globalen Bereich erfolgen, also ausserhalb anderer Funktionen. Indirekt haben Sie diese Möglichkeit schon verwendet. Durch die `#include`-Anweisung wird eine Datei textlich geladen und an dieser Stelle eingefügt, im obigen Beispiel `iostream` aus der Standardbibliothek. Diese enthält unter anderem alle Funktionsdeklarationen, die diese Bibliothek zur Ein- und Ausgabe zur Verfügung stellt.

2.1.8 Fortgeschrittenes Debuggen

Wie bereits in Versuch 1 erwähnt, besitzt der Debugger neben der bereits vorgestellten Funktionalität noch viele weitere, nützliche Funktionen, um sogar die komplexesten Programme debuggen zu können. Hier sollen nun einige dieser Funktionen vorgestellt werden, welche, dem Umfang der Versuche dieses Praktikums entsprechend, sehr nützlich werden können. Um die Funktionalitäten anschaulich darstellen zu können, wird das *Hello-World* Beispiel aus Versuch 1 erweitert:

```

1 #include <iostream>
2
3 void zaehlen(int);
4
5 int main()
6 {
7     std::cout << "Hello World!" << std::endl;
8
9     zaehlen(10);
10
11    return 0;
12 }
13
14 void zaehlen(int max)
15 {
16     for (int var = 0; var < max; ++var)
17     {
18         std::cout << var << std::endl;
19
20         if(var == (int)(max/2))
21         {
22             std::cout << "Haelfte ist geschafft!" << std::endl;
23         }
24     }
25 }
```

Call-Stack

In C++ werden häufig Unterprogramme aufgerufen. Der so genannte *Call Stack* speichert diese Aufrufkette. Damit kann nach Beendigung eines Unterprogramms nachvollzogen werden, wo das Rücksprungziel des Programms liegt. Der *Call-Stack* kann in Kombination mit einem Debugger ein nützliches Werkzeug zur schnellen Lokalisierung von Programmfehlern sein. Startet man ein fehlerhaftes Programm ohne gesetzte Haltepunkte im Debugger, dann lässt sich nach einem Absturz der zuletzt aufgerufene Befehl über den Call-Stack ermitteln. Leider führen insbesondere Bereichsüberschreitungen von Zeigern oft dazu, dass das Programm erst im weiteren Verlauf abstürzt, wenn auf den unzulässigerweise überschriebenen Speicher zugegriffen wird.

Der Call-Stack ist in der Debug-Ansicht als Standardeinstellung oben links platziert. Wird beispielsweise in `zaehlen` in Zeile 16 ein Haltepunkt gesetzt, zeigt der Call-Stack, dass sich die Ausführung nun in `main → zaehlen` befindet:

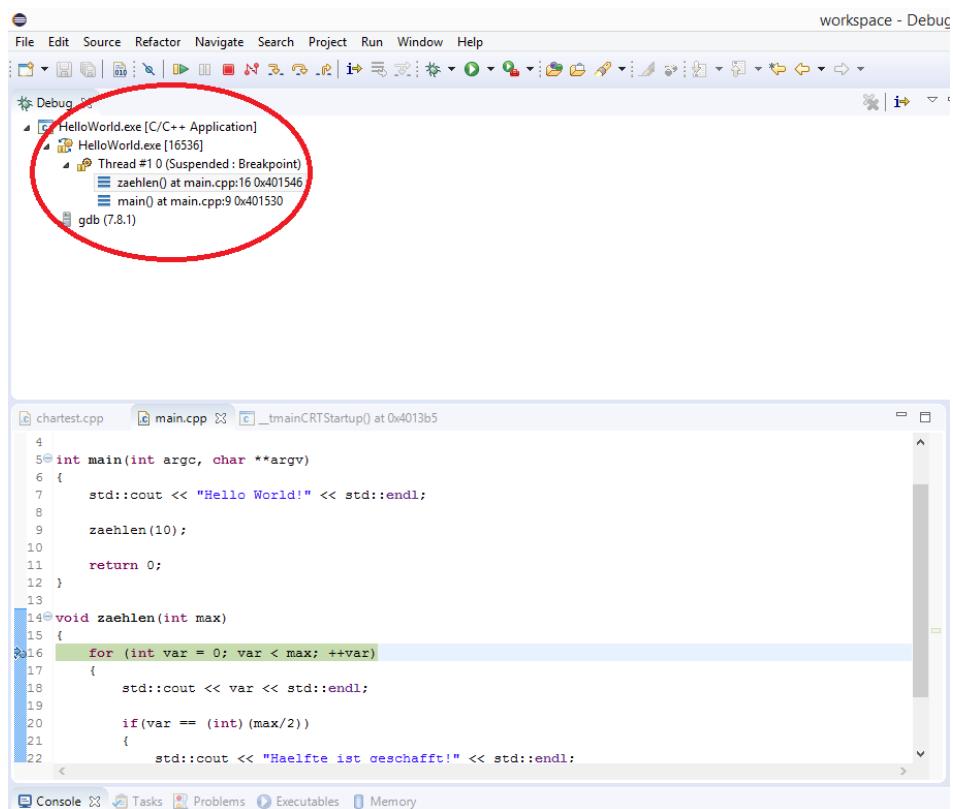


Abbildung 2.3: Call Stack nach Aufrufen von *zaehlen*.

Schrittweise Ausführung



Abbildung 2.4: Die Navigationsleiste im Debug-Modus.

- Markierung 1 zeigt den *Resume*-Button, welcher die Ausführung bis zum nächsten Haltepunkt oder Programmende fortsetzt.
- Markierung 2 zeigt den *Stop*-Button, welcher die Ausführung sofort beendet.
- Markierung 3 zeigt den *Step – Into*-Button, welcher, wenn möglich, in die aufzurufende Funktion springt.
- Markierung 4 zeigt den *Step – Over*-Button, welcher die aufzurufende Funktion in einem Schritt ausführt.
- Markierung 5 zeigt den *Step – Return*-Button, welcher den Programmablauf soweit ausführt, bis aus der aktuellen Funktion herausgesprungen wird.

Befindet sich die Ausführung des Beispielprogrammes nun z.B. in Zeile 9, wird der Debugger durch *Step – Into* in *zaehlen(10)* springen. Durch *Step – Over* wird der Debugger *zaehlen(10)* komplett ausführen und wieder in Zeile 11 anhalten. *Step – Return* sollte in der beschriebenen Situation nicht aktiv sein, da sich der Programmablauf in der obersten Funktion befindet. Wird das Programm allerdings wieder in *zaehlen* pausiert, wird *Step – Return* den Programmablauf solange fortsetzen, bis *zaehlen* komplett ausgeführt wurde, und dann in *main* anhalten.

Haltepunkte mit Bedingung

Oft passiert es, dass Fehler im Programm nur in bestimmten Spezialfällen auftauchen, da genau diese nicht in der Implementierung bedacht wurden. In dem Fall ist es sehr hilfreich, wenn der Debugger nur genau in diesen Spezialfällen an Haltepunkten anhält. Um dies zu realisieren, können Bedingungen an Haltepunkte geknüpft werden. Dafür kann über einen Rechtsklick auf die Zeile, in der der Haltepunkt hinzugefügt werden soll, die Option *Add Breakpoint* ein Haltepunkt mit Bedingung erstellt werden. Alternativ ist dies auch mit *Strg+Doppelklick* möglich. Im Beispiel soll der Spezialfall für die Ausgabe nach Ablauf der Hälfte der Schleife betrachtet werden:

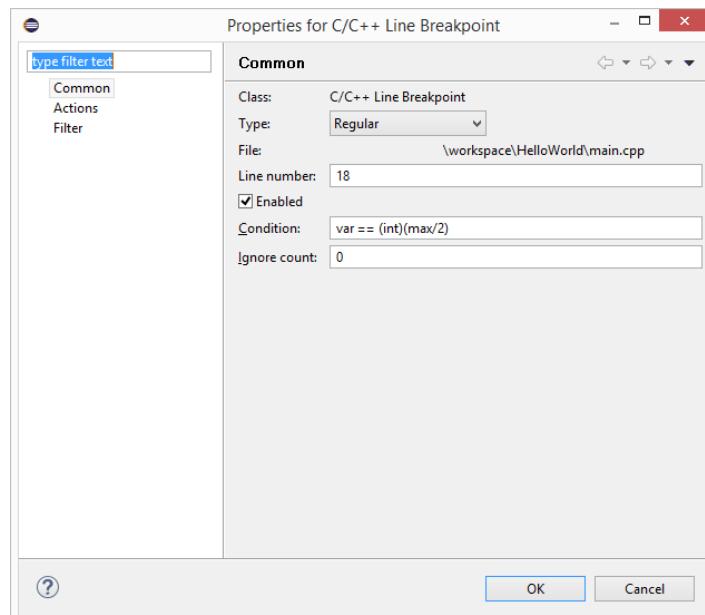


Abbildung 2.5: Erstellen eines Haltepunktes mit Bedingung.

Die Syntax, die zur Formulierung der Bedingungen genutzt wird, ist die der Projektssprache. Nach Hinzufügen des in 2.5 gezeigten Haltepunktes unterbricht der Programmablauf nur noch dann, wenn die Bedingung der If-Abfrage auch zutrifft. Im Beispielfall also vor der Ausgabe von 5. Es ist sogar möglich, die Bedingungen der Haltepunkte während des Debuggens anzupassen. Dazu lassen sich die Eigenschaften des Haltepunktes über das *Breakpoint*-Menü in der oberen, rechten Ecke anpassen:

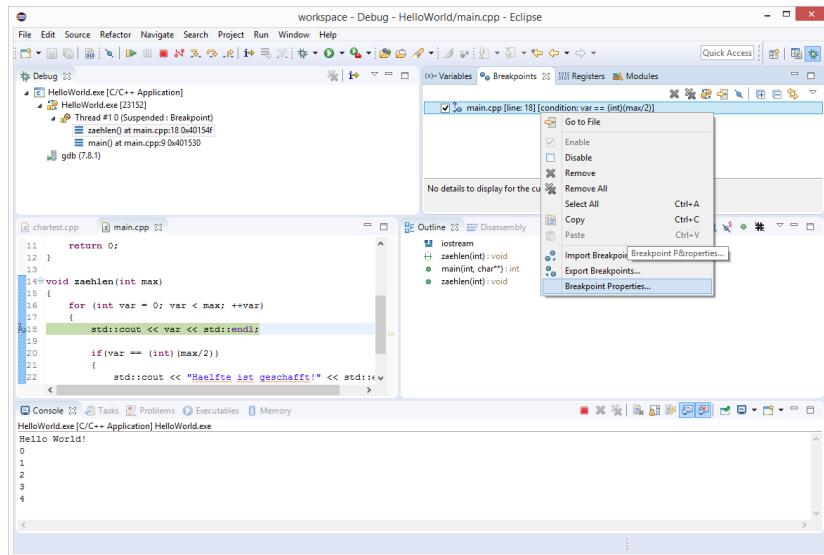


Abbildung 2.6: Editieren eines Haltepunktes mit Bedingung.

2.1.9 Rekursion

Das allgemeine Konzept der Rekursion sollte bereits aus der Mathematik bekannt sein. Soll zum Beispiel eine Fakultät berechnet werden, so lässt diese rekursiv wie folgt beschreiben:

$$\forall n \in \mathbb{N}_0^+ : f_n := \begin{cases} 1 & n = 0 \\ n \cdot f_{n-1} & n > 0 \end{cases}$$

In dieser noch komplett mathematischen Definition stecken nun auch alle Komponenten, die in der Informatik für einen rekursiven Algorithmus benötigt werden. So lässt sich der Wertebereich direkt ablesen: \mathbb{N}_0^+ . Der Werte- und Lösungsbereich kann also mit einem *unsigned int* realisiert werden. Welche Größe hier zu nehmen ist, muss vom Programmierer abgeschätzt werden. Im Beispiel soll der einfache Wertebereich genügen.

```
1 || unsigned int fak(unsigned int n);
```

Damit ist die Deklaration der rekursiven Funktion bekannt.

Die Definition lässt sich nun anhand zweier weiterer Kriterien bestimmen: Rekursionsanfang und Rekursionsschritt. Der Rekursionsanfang benennt die trivialen Lösungen der Funktion, im Beispiel also: $f_0 = 1$

```
1 || unsigned int fak(unsigned int n)
2 {
3     if(n == 0)
4     {
5         return 1;
6     }
7 }
```

Der Rekursionsschritt beinhaltet nun den rekursiven Aufruf (Selbstaufruf) der Funktion und die Regel, nach der gerechnet werden soll: $f_n = n \cdot f_{n-1}$

```
1 || unsigned int fak(unsigned int n)
2 {
3     if(n == 0)
4     {
5         return 1;
6     }
7     else if(n > 0)
8     {
```

```

9 ||     return n * fak(n-1);
10 || }
11 ||

```

Somit ist die Rekursion vollständig. Angemerkt sei, dass rekursive Implementierungen oft verständlicher und einfacher sind. Jedoch ist ihre äquivalente, iterative Implementierung in fast jedem Fall signifikant schneller. Das liegt daran, dass bei jedem Selbstaufruf nicht nur das Ergebnis berechnet wird muss, sondern auch auf dem Call-Stack gearbeitet wird. Dies kostet im Gegensatz zu den oft sehr einfachen Berechnungsregeln viel Zeit. Exemplarisch ist dies für die Berechnung der Fakultät dargestellt, siehe Abbildung 2.7. Um die Fakultät von 4 zu berechnen, wird entsprechend der Implementierung zunächst die Fakultät von 3 berechnet. Hierfür ist wiederum die Berechnung der Fakultät von 2 notwendig. Dies wird solange forgeföhrt, bis die Fakultät von 0 berechnet wird. Dies ist die Abbruchbedingung der Rekursion, da $0! = 1$. Nun wird der aufgebaute Call-Stack von unten wieder abgebaut und alle Berechnungen werden durchgeführt. Ist der komplette Call-Stack abgebaut, ist die Fakultät von 4 berechnet.

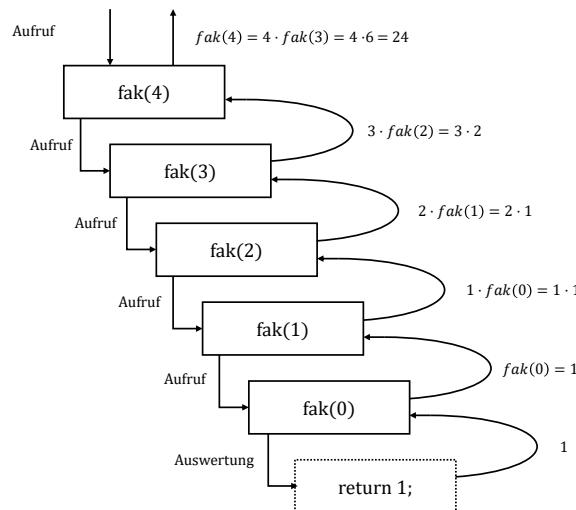


Abbildung 2.7: Call-Stack während der Rekursion zur Berechnung einer Fakultät

Der oben beschriebene Call-Stack lässt sich auch in Eclipse und jeder anderen IDE betrachten. Beispielhaft ist er in Abbildung 2.8 dargestellt.

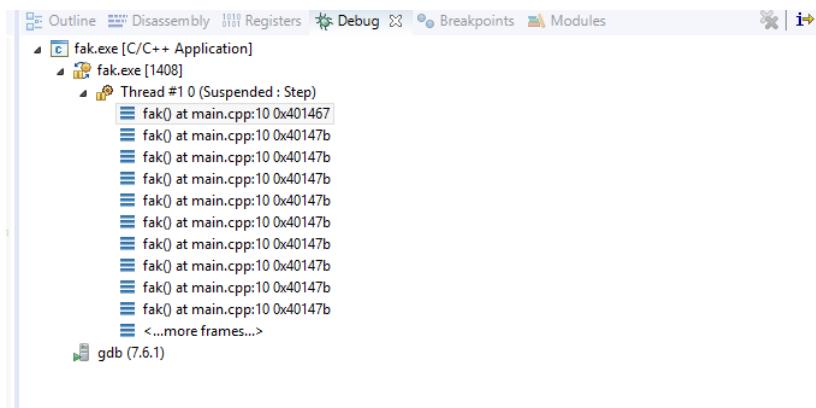


Abbildung 2.8: Call-Stack während der Rekursion zur Berechnung einer Fakultät

2.2 Aufgaben

Die Fibonacci-Folge ist eine unendliche Folge von Zahlen (den Fibonacci-Zahlen), bei der sich die jeweils folgende Zahl durch Addition ihrer beiden vorherigen Zahlen ergibt. Benannt ist sie nach Leonardo Fibonacci, der damit 1202 das Wachstum einer Kaninchenpopulation beschrieb. Die ersten Elemente der Folge sind: 0, 1, 1, 2, 3, 5, 8, 13, ...

2.2.1 Rekursive Berechnung der Fibonacci-Zahlen

1. Stellen Sie eine rekursive Formel $f(n)$ zur Berechnung der Fibonacci-Zahlen auf. Für die Anfangswerte soll gelten $f(0) = 0$ und $f(1) = 1$.
2. Legen Sie ein neues Eclipse-Projekt mit Namen *Fibonacci* an.
3. Schreiben Sie ein Programm, welches die Fibonacci-Zahlen für $n = 0$ bis 25 mithilfe der rekursiven Formel berechnet und der Reihe nach ausgibt.

2.2.2 Iterative Berechnung der Fibonacci-Zahlen

1. Erstellen Sie ein neues Projekt *Versuch02Teil2* und fügen Sie die Vorlagedatei hinzu (siehe Seite 8).
2. Das Programm berechnet angeblich die Fibonacci-Zahlen, wie sie in Aufgabe 1 definiert wurden. Leider ist das Ergebnis nicht korrekt. Benutzen Sie den Debugger von Eclipse, um sich die Funktion des Programmes klar zu machen. Korrigieren Sie die Fehler.
3. Vergleichen Sie für $n = 42$ die Geschwindigkeit Ihres Programms mit dem aus dieser Aufgabe. Gibt es einen Unterschied? Wenn ja, warum?
4. Zusatzaufgabe: Ab der 47-ten Fibonacci-Zahl geben beide Programme ein falsches Ergebnis aus. Warum? Wie kann das beheben? Was ist die größte mit diesem Programm berechenbare Fibonacci-Zahl?

2.2.3 Felder

Es gibt zahlreiche Möglichkeiten, Texte zu verschlüsseln. Eines der einfachsten Verfahren besteht darin, jeden Buchstaben des Quelltextes nach einer festen Tabelle in einen anderen umzuwandeln. Aufgrund seiner Einfachheit eignet es sich zwar gut für anschauliche Zwecke, wird in der Praxis jedoch kaum eingesetzt, da es relativ einfach, z.B über Häufigkeitsanalysen, zu entziffern ist.

Bei dem Verfahren handelt es sich um eine monographische, monopartite Substitution, das heißt, jedes Zeichen im zu verschlüsselnden Text wird in ein ihm zugeordnetes Geheimzeichen umgewandelt. Dabei muss jedoch gewährleistet sein, dass die Eindeutigkeit der Verschlüsselung erhalten bleibt, also die Verschlüsselungsfunktion injektiv ist. Ein Geheimzeichen darf nicht zwei verschiedenen Klarzeichen entsprechen.

In diesem Versuch werden Sie eine einfache Verschlüsselung entwerfen, sodass Sie kurze Worte verschlüsseln und wieder entschlüsseln können. Sie werden die Darstellung der Zeichen im Speicher ausnutzen, damit die Zugriffe auf die Lookup-Tabellen berechnen und so die Zeichenersetzung durchführen.

1. Erstellen Sie ein neues Projekt mit dem Namen *Versuch02Teil3*.
2. Legen Sie in Ihrer *main()*-Funktion die Lookup-Tabelle für die Verschlüsselung an. In dieser wird jeder Großbuchstabe des Alphabets mit einer entsprechenden Substitution im Geheimtext verknüpft. Der Verschlüsselungsalgorithmus muss nun nur von Zeile 1 der Tabelle (Klartext) in Zeile 2 der Tabelle (Geheimtext) wechseln, um die richtige Entsprechung zu finden. Verwenden Sie ein Feld der Größe *char [2]/[26]* und füllen Sie die erste Zeile mit den Großbuchstaben von A-Z. Die zweite Zeile füllen Sie ebenfalls mit den Großbuchstaben A-Z, jedoch in beliebiger Reihenfolge. Stellen Sie sicher, dass jeder Buchstabe nur einmal vorhanden ist.

3. Entwerfen Sie eine Funktion zu Verschlüsselung eines Wortes. Die Funktion soll zwei Parameter besitzen:
 - Das zu verschlüsselnde Wort in Großbuchstaben vom Datentyp *std::string*.
 - Die Lookup-Tabelle, die zur Verschlüsselung benutzt werden soll.

Die Funktion soll das verschlüsselte Wort als *std::string* zurückgeben. Nutzen Sie die Funktionen, die der Datentyp *String* Ihnen zur Verfügung stellt, z.B. *length()* in Verbindung mit einer geeigneten Schleife.

4. Testen Sie ihre Funktion, in dem Sie ihr Programm derart erweitern, dass der Benutzer ein Wort eingeben kann. Dieses Wort soll in einer Variable des Datentyps *std::string* gespeichert werden. Geben Sie das Wort im Klartext und anschließend verschlüsselt in der Konsole aus.
5. Schreiben Sie analog zu ihrer Verschlüsselungsfunktion eine Entschlüsselungsfunktion. Dazu können Sie eine weitere Lookup-Tabelle für die Entschlüsselung benutzen, die der Funktion anstelle der Verschlüsselungs-Lookup-Tabelle übergeben wird.
6. Entschlüsseln Sie nun die verschlüsselte Benutzereingabe und geben Sie sie in der Konsole aus.

3 Komplexere Projekte und Dokumentation

3.1 Theorie

3.1.1 Zielsetzung und Einordnung

Mit den Programmen, die Sie im den vorherigen Versuchen geschrieben haben, können Sie nun auch komplexere Probleme lösen. Doch da größere Programme tendenziell immer unübersichtlicher werden, soll in diesem Versuch noch gezeigt werden, wie Projekte dokumentiert werden können, um sich im Code besser zurecht zu finden. Weiterhin soll das Konzept der testgetriebenen Entwicklung vorgestellt und in den Aufgaben angewandt werden. Dies ist eine gute Methode, sich während der Implementierung vor Bugs zu schützen und hilft auch beim späteren Debuggen. Im Aufgabenteil werden Sie Ihr neues Wissen anwenden und das Spiel *Reversi* programmieren.

Grundlegend soll dieser Versuch folgende Kenntnisse vermitteln:

- Dokumentation mit Doxygen (Optional)
- Schreiben von sauberem und übersichtlichem Code
- Überprüfung des eigenen Programms durch Tests

Die gestellte Aufgabe ermöglicht es Ihnen, das zu erstellende Programm zu erweitern und eigene Ideen einzubringen.

Außerdem sollen Sie in diesem Versuch das erste Mal selbstständig eine Dokumentation anfertigen. Beachten Sie, dass eine ausführliche Dokumentation einen Teil der Bewertung bei den Testatoren darstellt und ab jetzt bei *jedem* Versuch erwartet wird.

3.1.2 Testgetriebene Entwicklung

Bei der testgetriebenen Entwicklung wird die Softwareentwicklung durch Tests gesteuert. Das bedeutet, dass die Tests bestimmen, was und wie programmiert wird. Diese Entwicklung läuft zyklisch ab. Zuerst schreibt man einen Test, der noch nicht erfüllt wird. Dann schreibt man das eigentliche Programm, das schließlich den Test besteht. Nun kann man den Test weiter verbessern und das Programm entsprechend anpassen. Wichtig ist, dass dieser Test möglichst weit automatisiert ist, damit man schnell überprüfen kann, ob das Programm den Anforderungen entspricht.

Dieses Verfahren hat einige Vorteile. So weiß man jederzeit, welche Teile eines umfangreicheren Softwareprojekts funktionstüchtig sind, und man kann sich so beim Wiederverwenden alter Programmteile auf den neuen Code konzentrieren, weil man sichergehen kann, dass diese alten Programmteile nicht zu neuen Fehlern führen. Dadurch, dass man die Tests zuerst schreibt, wird erreicht, dass zuerst klar definiert werden muss, was ein Programm zu leisten hat, ohne vorher zu wissen, welche Probleme beim Programmieren auftreten. Auf diese Weise kommt es seltener vor, dass Fehler, die beim Implementieren der Funktion gemacht werden, auch beim Schreiben der Tests gemacht werden.

Im vorliegenden Versuch soll dieses Konzept angewendet werden, um bei einem umfangreichen Projekt die Übersicht zu behalten, und Fehler frühzeitig zu bemerken und zu beheben.

3.1.3 Übersichtlicher und verständlicher Code

Da in der Softwareentwicklung Zeit und damit Kosten eine wichtige Rolle spielen, ist es wichtig, Programmfehler schnell zu finden. Programmfehler müssen nicht unbedingt in einer Fehlermeldung resultieren. Wenn das Ergebnis dessen, was das Programm ausgibt, vom erwarteten Verhalten abweicht,

muss analysiert werden, worin dies begründet ist. Dabei ist es hilfreich, wenn der Code gut strukturiert und dokumentiert ist.

Größere Anwendungen werden von ganzen Programmiererteams erstellt. Daher ist es sinnvoll, wenn alle Programmierer einen ähnlichen Programmierstil haben. Dies geschieht durch Regeln (Code Convention). Diese erleichtern die Fehlerkorrektur in fremdem Code.

Während dieses Praktikums erstellen auch Sie Code, den andere, z.B. Betreuer bei einem Testat, schnell erfassen können sollen. Daher sollen die folgenden Regeln für dieses Praktikum bei der Erstellung von neuem Code gelten. Bitte lesen Sie diese Regeln sorgfältig, und programmieren Sie von nun an nach diesen Vorgaben. Die Einhaltung der Code Convention geht mit in die Beurteilung Ihrer Versuche bei den einzelnen Testaten ein.

Code Convention

- Funktionen, Variablen, Zeiger beginnen mit einem Kleinbuchstaben.
- Die Namen folgen dem *lowerCamelCase*-Prinzip, z.B. aktuellerSpieler, zugAusfuehren() oder spielfeld. Besteht der Name aus mehr als einem Wort, wird das erste Wort klein geschrieben und der Beginn des folgenden Worts groß.
- Klassen und Structs beginnen mit einem Großbuchstaben.
- Variablen und Zeiger werden beim Anlegen initialisiert.
- Aus dem Namen einer Funktion, Variablen usw. muss ihre Aufgabe ersichtlich sein.
- Geschweifte Klammern ({, }) stehen immer alleine in einer neuen Zeile.
- Der Code ist eingerückt (Erleichtert die richtige Anzahl von { und } zu verwenden).
- Ausreichend kommentieren (vor allem Funktionen).
- Kommentare über mehrere Zeilen beginnen mit /* und enden mit */ (jeweils in einer eigenen Zeile)
- Mehrzeilige Kommentare innerhalb von Funktionen vermeiden.
- Mit Kommentaren innerhalb von Funktionen nicht übertreiben.
- Zeiger und Referenzen stehen direkt hinter dem Datentyp (double* var, double& var).

Beispiele

Welche der beiden nachfolgenden Funktionen ist besser zu verstehen und was tun diese?

```
1 || int p(int f, int g){ int h = f*f+g*g;
2 || return h;}
3
4 || /*
5 || Die Funktion Pythagoras realisiert den Satz des Pythagoras.
6 || Sie bekommt a und b übergeben und liefert c^2 zurück.
7 || */
8 || int pythagoras(int a, int b)
9 || {
  ||   int c2 = a*a+b*b; //c2 = c^2
  ||   return c2;
  || }
```

Die beiden Funktionen realisieren dasselbe. Durch die vereinbarte Code Convention ist der untere Code deutlich übersichtlicher und schneller zu verstehen.

3.1.4 Dokumentation mit Doxygen

Dokumentation einer Software

Kommentare unterstützen die Entwickler insbesondere bei der Wartung der Software (Fehlersuche und -korrektur). Dabei kann es vorkommen, dass nach Jahren ein Code überarbeitet werden muss. Da dies dann in der Regel nicht mehr vom eigentlichen Urheber des Sourcecodes durchgeführt wird, sondern von weiteren, muss darauf geachtet werden, dass der Code leicht zu verstehen ist.

Die Dokumentation umfasst aber nicht nur die Kommentierung des Quelltextes, sondern auch eine ausführliche Beschreibung jedes einzelnen Bausteins des Programms, vor allem der Klassen. Dabei sollte die Dokumentation der Funktionsweise von Schnittstellen einer Klasse im Vordergrund stehen. Wurde eine Klasse ordentlich dokumentiert, kann sie problemlos von anderen Programmierern genutzt werden, ohne dass diese den Code dahinter verstehen müssen. Insbesondere bei der *Black Box Reuse* ist eine saubere Kommentierung nötig, da die Implementierung selbst nicht eingesehen werden kann. Ein weit verbreitetes Hilfsmittel zur Dokumentation ist das unter der GNU Public License stehende *Doxygen*. Für Eclipse gibt es ein entsprechendes Frontend namens *Eclox*, das Doxygen nutzt.

Bitte beachten Sie, dass die Doxygen-Kommentierung Teil der Bewertung ist und von nun an bei jedem Versuch vorgenommen werden muss. Das Anlegen einer Doxygen Datei wiederum ist eine optionale Aufgabe. Dazu ist die Installation von Doxygen notwendig.

Einsatz von Doxygen

Doxygen generiert aus C++-Kommentaren, die mit speziellen "Markern" versehen sind, eine Dokumentation in HTML, LaTeX oder anderen Formaten. Diese Kommentare können auf unterschiedliche Art und Weise gesetzt werden (siehe nachfolgende Codebeispiele). Damit Doxygen die Kommentare richtig zuordnet, müssen diese immer in der Zeile vor der betreffenden Funktion stehen. Dabei sind Doxygen-Kommentare normale C++-Kommentare mit einem Hinweis für Doxygen, diesen Kommentar in die Dokumentation aufzunehmen. Generell gibt es 4 verschiedene Möglichkeiten dies zu tun.

- Blockkommentare, die mit zwei “**” eingeläutet werden:

```
1 || /**
2 | * ... text...
3 | */
```

- Blockkommentare, die mit einem “!” gekennzeichnet sind:

```
1 || /*!
2 | * ... text...
3 | */
```

- C++-Zeilenkommentare, die mit “//” anfangen:

```
1 || //... Erste Zeile
2 || //... Zweite Zeile
```

- C++-Zeilenkommentare, die mit “//!” anfangen:

```
1 || //!... Erste Zeile
2 || //!... Zweite Zeile
3 || //!... Dritte Zeile
```

Hinweis: Beachten Sie bei diesen letzten beiden Methoden, dass eine leere Zeile zwischen den Kommentarzeilen den Doxygen-Kommentarblock beendet.

Mit den oben kennengelernten Kommentaren können Sie nun schon sehr detaillierte Beschreibungen für Ihre Funktionen erstellen. Doxygen bietet die Möglichkeit, die Dokumentation noch weiter zu verfeinern. Dazu können über \ oder @ und ein nachfolgendes Schlüsselwort z.B Rückgabewerte oder Parameter kommentiert werden. Einige mögliche Schlüsselworte sind:

- brief - Kurze Zusammenfassung des folgenden Codes

3.2 Aufgaben

- class xy - Beschreibung der Klasse xy
- file xy.h - Beschreibung der Datei xy.h
- return - Beschreibung des Rückgabewerts einer Funktion
- param xy - Beschreibung des Parameters xy einer Funktion

Es bietet sich immer an, die Funktion in einem Satz zusammenzufassen, um einen groben Überblick über die Funktionalität zu bekommen. Hierfür nutzt man den Befehl \brief . Ein Beispiel:

```
1 /*! \brief Brief description.
2 *      Brief description continued.
3 *
4 * Detailed description starts here.
5 */
```

Damit andere Ihre Funktionen wiederverwenden können, ist es besonders wichtig, die Schnittstelle sauber zu kommentieren. Dazu wird das vorherige Beispiel um die Befehle \param und \return erweitert:

```
1 /*! \brief Brief description.
2 *      Brief description continued.
3 *
4 * Detailed description starts here.
5 * Detailed description continued.
6 *
7 * \return Beschreibung das Rückgabewerts der Funktion.
8 * \param x Beschreibung des Parameters x.
9 * \param y Beschreibung des Parameters y.
10 */
```

Ein Programmierer, der Ihre Funktion benutzen möchte, kann nun einfach in der Dokumentation nachlesen, welchen Zweck die Funktion erfüllt, und welche Parameterübergaben sie dazu braucht.

Bitte beachten Sie, dass bei jeder .cpp sowie .h Datei „\file: Dateiname.endung“ in der erste Zeile stehen muss. Eine ausführliche Übersicht aller Befehle finden Sie unter
<https://www.doxygen.nl/manual/commands.html>

Zur Trennung der Kurz- und der Langbeschreibung wird eine Leerzeile benutzt. Beachten Sie außerdem, dass Zeilenkommentare, die eine detaillierte Beschreibung enthalten, mindestens 3 Zeilen lang sein sollten. Hier können Sie mit
 eine neue Zeile im resultierenden HTML Dokument beginnen. Wie gezeigt kann die Dokumentation sowohl in der Header-Datei als auch in der Implementierungs-Datei stehen.

Tipp: Wenn Sie in der Zeile unmittelbar über Ihrer Funktion die Kommentarzeichen /** eingeben und dann Enter drücken, erzeugt Ihnen Eclipse eine Vorlage für Ihren Kommentar. Dazu muss die Unterstützung für Doxygen eingeschaltet sein (siehe Seite 9)

Anlegen von Doxy-Files (optional)

Um zu einem Eclipseprojekt eine Doxygen-Doku zu erstellen, brauchen Sie nur ein neues Doxyfile hinzuzufügen (mittels *Ctrl+N* → *Others* → *Doxyfile*). Durch Doppelklick auf den Doxyfile konfigurieren Sie die Dokumentation. Aktivieren Sie *Scan recursively* und als Mode *all entities*. Zur besseren Navigation wählen Sie als *Output Format* die Option *with frames and navigation tree*. Löschen Sie das Häkchen bei *Latex*. Speichern Sie das Doxyfile. Wenn Sie in der Schnellzugriffsleiste auf das blaue @-Zeichen klicken, wird die Dokumentation erstellt. Sie wird im Projektordner unter *html* abgelegt. Die Startseite ist *index.html*

3.2 Aufgaben

Die Aufgabe ist die Implementierung des Spiels *Reversi* für zwei Spieler. Dieses Problem soll hier mit einem *bottom-up* Ansatz gelöst werden, wobei der Fortschritt regelmäßig mit den Mitteln der testgetriebenen Entwicklung überprüft werden soll. Beginnen Sie daher zunächst mit der Implementierung

kleinerer Funktionen und deren Tests, die dann am Ende zu einem kompletten Programm zusammengefügt werden.

Spielregeln

Die Spielregeln von *Reversi* sind recht einfach zu verstehen, nehmen dem Spiel aber nicht seine strategische Komplexität. *Reversi* wird auf einem 2-dimensionalen Spielfeld gespielt, wie es in Abb. 3.1 dargestellt ist. Jeder Spieler hat seine eigenen Spielsteine (schwarz oder weiß), und jedes Spiel wird in der dargestellten Ausgangsposition (linkes Feld) begonnen.

Ziel ist es, am Ende des Spiels die meisten Felder mit Steinen der eigenen Farbe besetzt zu haben. Dies geschieht, indem man gegnerische Steine zwischen zwei Steinen der eigenen Farbe einschließt. Alle Steine, die auf diesem Wege eingeschlossen wurden, wechseln ihre Farbe und gehören nun zu den eigenen Steinen. Es sind nur Züge möglich, die gegnerische Steine einschließen. Bild 3.1 zeigt auch ein Beispiel. Ausgehend von der Aufstellung auf dem zweiten Spielfeld ist Spieler *Weiß* am Zuge. *Weiß* platziert seinen Stein auf **C5**. Zwei schwarze Steine wechseln ihre Farbe und gehören nun *Weiß*.

Das Spiel ist beendet, wenn kein Spieler mehr Züge ausführen kann. Der Gewinner ist derjenige mit den meisten Steinen auf dem Spielbrett. Eine ausführliche Erläuterung der Spielregeln finden Sie zudem in der freien Enzyklopädie Wikipedia¹.

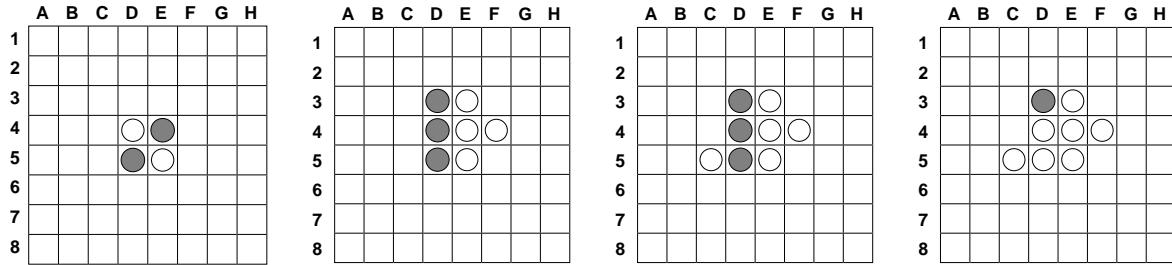


Abbildung 3.1: Startaufstellung und Beispiel

Obwohl die Regeln einfach sind, ist es nicht einfach, das Spiel zu gewinnen, da es bis zum Ende möglich ist, sehr viele Steine zu erobern, selbst mit dem letzten Stein.

3.2.1 Vorbereitung

Importieren der Vorlage

Zu diesem Versuch gibt es Code-Vorlagen.

1. Erstellen Sie ein neues Projekt mit dem Namen *Versuch03*.
2. Importieren Sie die Codevorlagen (siehe Seite 8).
3. Die Vorlagen sind so gestaltet, dass sie sofort compiliert werden können.

Die Vorlagen beinhalten alle benötigten Dateien. Dabei stellen *reversiKI.h* und *reversiKI.cpp* eine komplett vorimplementierte KI zur Verfügung, welche in der Lage ist, Reversi zu spielen. In der Datei *config.h* sind globale Konstanten definiert, welche in mehreren oder allen Dateien benötigt werden. Damit diese nicht in jeder Datei geändert werden müssen, sind diese in einer Header-Datei zusammengefasst, welche dann in den übrigen Dateien importiert wird. In diesen Dateien müssen Sie keine Änderungen mehr vornehmen.

Die Dateien *test.cpp* und *test.h* beinhalten die Test-Funktionen für das Programm, welches in der Datei *reversi.cpp* implementiert wird. In diesen Dateien ist die Programmstruktur bereits vorgegeben,

¹<http://de.wikipedia.org/wiki/Reversi>

3.2 Aufgaben

sodass Sie nur noch die Funktionsrümpfe implementieren müssen. Natürlich können Sie zusätzliche Funktionen dort einfügen, wo es Ihnen sinnvoll erscheint. Zur Implementierung einer einzelnen Funktion benötigen Sie am Anfang kein Wissen über die anderen Funktionen. Konzentrieren Sie sich auf das Ein-Ausgabe-Verhalten und testen Sie Ihre Funktionen. Nutzen Sie hierfür einfach die Funktionen aus der Datei *test.cpp*. Die Funktion

```
bool ganzenTestAusfuehren()
```

ist dazu gedacht, alle implementierten Funktionen nacheinander zu testen und wird in der *main*-Funktion aufgerufen. Später werden Sie die einzelnen, getesteten Funktionen zum Spiel Reversi zusammenfügen.

Ein Reversi-Spielfeld wird hier, wie die meisten Brettspiele, immer in einem zweidimensionalen Feld der Form *int spielfeld[GROESSE_Y][GROESSE_X]* gespeichert. Die Bezeichner *GROESSE_Y* und *GROESSE_X* stehen für die schon vorgegebene Größe des Feldes. Die einzelnen Positionen auf dem Feld werden mithilfe eines x-Index und eines y-Index adressiert. Hierbei steht der x-Index immer für die Spalte des Feldes und der y-Index für die Zeile. Beachten Sie, dass der Index für Zeilen und Spalten immer bei 0 beginnt.

3.2.2 Basisfunktionen

Das *bottom-up* Design schreibt vor, die Entwicklung mit den Basisfunktionen zu beginnen. In diesem Aufgabenblock werden Sie einzelne Funktionen, deren Tests und deren Dokumentation schreiben. Im nächsten Aufgabenblock werden Sie diese gemeinsam nutzen, um die übergeordnete Spiellogik zu implementieren.

Die Funktionen *initialisiereSpielfeld()* und *zeigeSpielfeld()* ist in den Vorlagen bereits vorhanden. Nutzen Sie diese als Orientierung bei der Implementierung der weiteren Funktionen.

Wer ist der Gewinner?

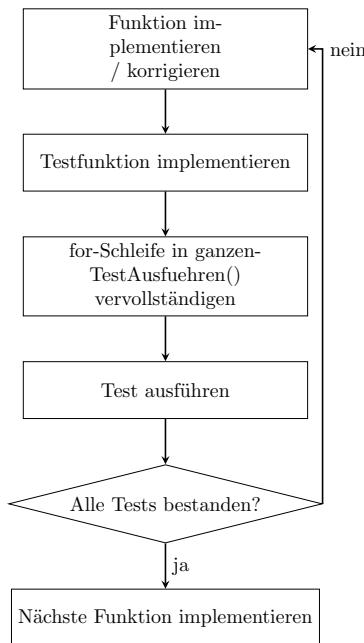
Am Ende des Spiels wird es notwendig sein, den Gewinner zu ermitteln. Auch diese Funktion ist bereits implementiert und mit Kommentaren versehen.

1. Die Funktion *int gewinner(const int spielfeld[GROESSE_Y][GROESSE_X])* zählt alle Felder, die Spieler 1, sowie alle Felder, die Spieler 2 besetzt hat, und liefert anschließend den Gewinner:
 - 0: falls das Spiel unentschieden ist,
 - 1: falls Spieler 1 gewinnt,
 - 2: falls Spieler 2 gewinnt.

Diese Funktion soll den Ablauf der einzelnen Funktionstests veranschaulichen. Anschließend werden Sie die restlichen Basisfunktionen in der gleichen Weise bearbeiten.

Funktionstest

Zu jeder Funktion, die sie implementieren werden, werden Sie auch eine dazugehörige Testfunktion implementieren. Der Ablauf dieser Tests ist immer gleich. Implementieren Sie zuerst die Funktion, vervollständigen Sie anschließend die Testfunktion in der Datei *test.cpp*. Vervollständigen noch die for-Schleife in der Funktion *ganzenTestAusfuehren()*. Die Signatur der Testfunktion gibt Ihnen bereits einen Anhaltspunkt, wie der Test abläuft. Die Testfunktion für *gewinner()* ist bereits implementiert und soll das Prinzip veranschaulichen.

**Abbildung 3.2:** Grundsätzlicher Ablauf aller Tests.

In der Funktion *ganzenTestAusfuehren()* sind bereits drei Felder (*eingabeFeld*) sowie ein Feld mit dem dazugehörigen richtigen Ergebnis (*korrektesErgebnis*) enthalten. In einer Schleife wird die Testfunktion *gewinnerTest()* drei mal aufgerufen, und das zu testende Feld, das korrekte Ergebnis sowie die Nummer des Tests übergeben. Die Testfunktion *gewinnerTest()* ruft die Spielfunktion *gewinner()* mit diesem Feld auf. Die Funktion *gewinner()* berechnet nun den Gewinner und gibt diesen Wert zurück. Die Funktion *gewinnerTest()* vergleicht, ob das von der Funktion *gewinner()* gelieferte Ergebnis mit dem korrekten Ergebnis übereinstimmt. Wenn das nicht der Fall ist, enthält die Implementierung der Spielfunktion *gewinner()* Fehler. In diesem Fall ist es sinnvoll, die Daten und die Nummer des Tests auszugeben, um den Fehler zu suchen. Die Testfunktion signalisiert über ihren Rückgabewert (*bool*), ob der Test erfolgreich war.

In der Funktion *ganzenTestAusfuehren()* wird im Fehlerfall der Wert von *gesamtErgebnis* auf *false* gesetzt und bleibt das danach auch. Daran kann man erkennen, das ein Test nicht bestanden wurde.

Sie können das Verhalten der Testfunktion studieren, indem Sie in der Funktion *ganzenTestAusfuehren()* einen Wert in *korrektesErgebnis* ändern, um absichtlich einen Fehler herbeizuführen.

Projektdokumentation

In diesem Versuch werden Sie die Doxygen-Kommentierung zur Dokumentation des Projekts nutzen. Doxygen wurde auf Seite 53 eingeführt².

1. Dokumentieren Sie alle weiteren Funktionen, die Sie implementieren werden, so wie die bereits mit Dokumentation vorhandenen Funktionen.
2. Erstellen Sie ein neues Doxyfile (optional).
3. Erstellen Sie die Dokumentation durch Klick auf das Doxygen Symbol (blaues @). Sie finden die erstellte Dokumentation in dem Unterordner *html* (optional).

Sie haben nun die Werkzeuge für die Dokumentation vorbereitet. Wenn Sie im Verlauf des Versuches Ihre Testfunktionen und die Dokumentation aktuell halten, werden Sie am Ende ein vollständig getestetes und dokumentiertes Projekt implementiert haben.

²Nutzen Sie als Referenz auch die Dokumentation von Doxygen unter: <http://www.doxygen.org>

3.2 Aufgaben

Weitere Basisfunktionen

1. Implementieren Sie zuerst die Funktion *aufSpielfeld()*, welche überprüfen soll, ob zwei übergebene Indizes (*posX* und *posY*) eine gültige Position auf dem Feld sind. Bedenken Sie, dass die Größe des Spielfeldes durch die konstanten Werte *GROESSE_X* und *GROESSE_Y* festgelegt ist. Die Funktion soll für erlaubte Indizes den Wert *true* zurückgeben, für Indizes außerhalb des Feldes dagegen *false*.
2. Vervollständigen Sie die entsprechende Testfunktion *aufSpielfeldTest()*. Orientieren Sie sich an den schon implementierten Testfunktionen.
3. Überprüfen Sie, ob der Spieler *aktuellerSpieler* an der Position (*posX*, *posY*) einen Zug durchführen darf in der Funktion:

```
bool zugGueltig(const int spielfeld[GROESSE_Y] [GROESSE_X] ,  
                 const int aktuellerSpieler,  
                 const int posX,  
                 const int posY)
```

4. Vervollständigen Sie nun die entsprechende Testfunktion *zugGueltigTest()*. Geben Sie das Feld nur aus, wenn ein Fehler aufgetreten ist. Sonst geben Sie nur aus, welcher Test bestanden wurde, wie bei *gewinnerTest()*.
5. Überprüfen Sie die Funktion *zugGueltig()*. Lassen Sie hierzu die Testfunktion wieder über die vorgegebenen Testfälle innerhalb der Funktion *ganzenTestAusfuehren()* laufen. Stellen Sie sicher, dass alle Fälle wie erwartet funktionieren.
6. Führen Sie den Zug eines Spielers innerhalb der Funktion *zugAusfuehren()* aus. Diese Funktion hat eine Ähnlichkeit mit der Funktion *zugGueltig()*. Nutzen Sie Teile ihres bestehenden Quellcodes und erweitern Sie diese hier geschickt.
Das Ablaufdiagramm auf der rechten Seite verdeutlicht die Funktionsweise dieser Funktion.
7. Stellen Sie auch hier sicher, dass die Funktion *zugAusfuehren()* korrekt funktioniert, indem Sie die Testfunktion *zugAusfuehrenTest()* vervollständigen und diese in der Funktion *ganzenTestAusfuehren()* über alle vorgegebenen Testfälle laufen lassen.
8. Implementieren Sie die Funktion *moeglicheZuege()*. Diese Funktion berechnet die Anzahl möglicher Züge eines Spielers und gibt diese zurück. Testen Sie auch diese Funktion.

Mit diesen Basisfunktionen ist die erste Ebene des *bottom-up* Designs abgeschlossen. Sie haben die benötigten Funktionen implementiert, ohne jedoch deren realen Einsatzort zu haben. Dennoch haben Sie sie unabhängigen Tests unterzogen und können sich sicher sein, dass sie in den Fällen, die durch die Tests abgedeckt werden, fehlerfrei funktionieren. Nun werden Sie sich mit der nächsten Designebene beschäftigen.

3.2.3 Die Ablaufsteuerung

Um Reversi spielen zu können, werden Sie Ihr Programm in diesem Teil des Versuches mit einer Ablaufsteuerung ausstatten. Sie werden erst das Spiel für zwei menschliche Spieler implementieren. Anschließend werden Sie das Spiel mit einer künstlichen Intelligenz ausstatten.

Der Mensch ist am Zug

Die Funktion *menschlicherZug()* führt einen menschlichen Zug aus. Hierbei wird der Benutzer erst nach seinem Zug gefragt und kann das gewünschte Feld in der Form A1 oder a1 für die obere, linke Ecke und H8 oder h8 für die untere, rechte Ecke auf der Konsole eingeben. Aus den Benutzereingaben werden die zugehörigen Feldindizes bestimmt. Dann wird mithilfe der Funktion *zugGueltig()* geprüft, ob der Zug gültig ist und, falls ja, die Funktion *zugAusfuehren()* aufgerufen.

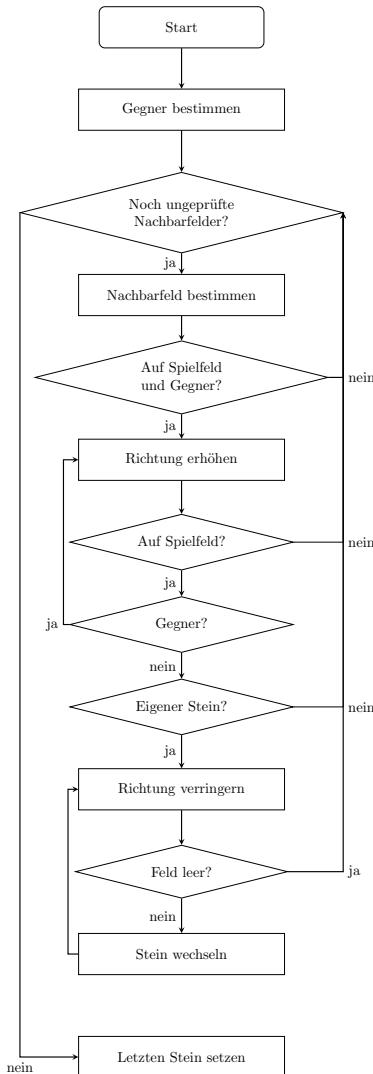


Abbildung 3.3: Ablaufdiagramm zur Funktion *zugAusfuehren()*.

Diese Funktion ist in der Vorlage bereits fertig implementiert. Sie steht Ihnen zur Verfügung, um die Ablaufsteuerung in der Funktion *spielen()* zu realisieren. Ein Gerüst hierfür wurde Ihnen ebenfalls bereits vorgegeben. Ergänzen Sie dieses schrittweise.

1. Implementieren Sie eine Ablaufsteuerung für zwei menschliche Spieler.
 - Lassen Sie abwechselnd Spieler 1 und Spieler 2 einen Zug machen.
 - Testen Sie, ob für die Spieler noch Züge möglich sind.
 - Lassen Sie einen Spieler einmal aussetzen, wenn für ihn kein Zug möglich ist.
 - Beenden Sie das Spiel, wenn beide Spieler keine weiteren Züge mehr haben.
 - Geben Sie den Gewinner bekannt.

Testen Sie Ihre Funktion hierbei mit folgendem Hauptprogramm:

```

1 || int main()
2 | {
3 |   int spielerTyp[2] = { MENSCH, MENSCH };
4 |   spielen(spielerTyp);
5 |   return 0;
6 | }
  
```

3.2 Aufgaben

Der Computer spielt mit

Die Funktion `computerZug()` in der Datei `reversiKI` führt, analog zu `menschlicherZug()`, einen Zug aus, den eine hoch entwickelte künstliche Intelligenz vorschlägt. Diese Funktion ist in der Vorlage bereits implementiert und steht Ihnen zur Verfügung.

1. Erweitern Sie die Funktion `spielen()` so, dass auch der Computer einen Zug machen kann. Der Spielertyp wird jeweils mit dem zweistelligen Feld `spielerTyp` per Parameter übergeben. Nutzen Sie diesen, um zu entscheiden, ob der jeweilige Spieler menschlich oder künstlich ist.
2. Testen Sie ihre Implementierung, indem Sie die Spielertypen im Hauptprogramm so variieren, dass zwei Computer gegeneinander spielen. Die korrekte Lösung dieser Partie finden Sie zur Kontrolle in der Vorlage des Versuchs.

Freiwillige Zusatzaufgaben

- Fragen Sie vor Beginn einer Partie, ob ein Computer oder ein Mensch spielen soll.
- Ermöglichen Sie es, dass mehrere Spiele gespielt werden können. Fragen Sie hierzu nach Spielende, ob eine weitere Partie gewünscht wird (z.B. mittels einer do-while Schleife).

Sie haben nun das Software-Projekt *Reversi* fertig gestellt. Sie haben gelernt, sich entlang einer *bottom-up* Lösung nach oben zu hängeln und einzelne Teilprobleme nach Spezifikation zu lösen. Hierbei haben Sie die Basisfunktionen so implementiert, dass sie unabhängig voneinander funktionieren und somit auch umfassend getestet werden können. Sie haben diesen Funktions-Pool anschließend genutzt und schrittweise ein Programm mit einer Ablaufsteuerung, einer Benutzerschnittstelle und einer künstlichen Intelligenz implementiert.

4 Einführung in Klassen

Bisher haben Sie in diesem Praktikum nach dem Prinzip der sogenannten *Strukturierten Programmierung* gearbeitet, in dem das zentrale Element die Funktionen sind. In diesem und den folgenden Versuchen werden Sie nun ein weiteres Paradigma der Programmierung kennenlernen, in dem die Daten und deren Eigenschaften das zentrale Element darstellen. Dieser Ansatz nennt sich *Objektorientierte Programmierung (OOP)*.

Nach einer kurzen Einführung in die *objektorientierte Programmierung* wird der Fokus auf die Hilfsmittel gelegt, die C++ in diesem Bereich bietet. Die Zielsetzung des Praktikums ist es, Ihnen die Syntax und einige Beispiele zu zeigen, wie man Programme nach dem Prinzip der *objektorientierten Programmierung* aufbaut. Literatur und Quellen zu weiterführenden Techniken und Konzepten im Bereich der *objektorientierten Programmierung* finden Sie im Literaturverzeichnis: [1, 2, 3].

4.1 Objektorientierte Programmierung

Die Objektorientierte Programmierung stellt ein höheres Abstraktionsniveau als die Strukturierte Programmierung dar. Daten und Operationen sowie Nachrichtenaustausch stehen bei diesem Ansatz im Vordergrund. Die zentralen Elemente sind hier Objekte und Klassen von Objekten. Was diese Begriffe für die Objektorientierte Programmierung bedeuten, wird in den nachfolgenden Abschnitten kurz erläutert.

4.1.1 Objekte

Wenn man Parallelen zwischen der *objektorientierten Programmierung* und der Linguistik zieht, dann entsprechen die Objekte allen Substantiven der deutschen Sprache.

Man könnte auch sagen, dass Objekte in der *objektorientierten Programmierung* nichts anderes sind als Objekte in unserer Umwelt. Allerdings muss an dieser Stelle erwähnt werden, dass auch Dinge, die wir nicht direkt wahrnehmen, Objekte sein können. Zum Beispiel stellt ein Bank- oder Fahrscheinautomat ein Objekt dar, aber auch das Girokonto, auf welches man über den Bankautomaten Zugriff erhält, ist ein Objekt. Konkrete Objekte werden in der *objektorientierten Programmierung* auch Instanz genannt, die Begriffe werden oft synonym verwendet.

Am Beispiel des Fahrkartenautomaten kann man die grundlegenden Gedanken hinter der *OOP* gut nachvollziehen:

- Es existieren viele verschiedene Geräte mit demselben Bauplan.
- Wichtig ist, was das Gerät „kann“, und nicht, wie es dies tut.
- Das Innenleben der Automaten ist den wenigsten bekannt, und es interessiert sie auch nicht.
- Der Automat stellt eine Schnittstelle bereit, die das innenliegende System für den Benutzer abstrahiert.
- Der Automat kann wiederum Teilsysteme (z.B. Drucker, Geldtresor) beinhalten, die der Benutzer nicht sieht.
- Die Benutzung ist (hoffentlich auf dem Automaten selbst) dokumentiert.

4.1.2 Klassen

In der realen Welt werden Objekte, die ähnliche Eigenschaften oder Funktionalitäten haben, in spezielle Gruppen eingeteilt und somit klassifiziert. Diese Klassifizierung hilft dabei, die komplexe Umwelt zu abstrahieren, damit sie leichter verständlich ist.

Ähnlich verhält es sich in der *objektorientierten Programmierung*. Den Begriff Klasse könnte man beispielsweise folgendermaßen definieren:

„Klasse ist in der objektorientierten Programmierung ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von Objekten (Klassifizierung). Sie dient dazu, Objekte zu abstrahieren. Im Zusammenspiel mit anderen Klassen ermöglichen sie die Modellierung eines abgegrenzten Systems.“ [1]

Der Begriff *System* kann in diesem Zusammenhang als die Abgrenzung zwischen einer Innen- und einer Außenwelt bezeichnet werden. Systeme selbst enthalten wiederum Teilsysteme, die ebenfalls eine solche Grenzziehung darstellen¹. In der *objektorientierten Programmierung* stellen Klassen also Systeme dar und mehrere Klassen bilden ebenfalls wieder ein System.

4.2 Klassen in C++

Obwohl C++ bereits eine ganze Reihe fundamentaler Datentypen zur Verfügung stellt (z.B. int, float, double, char, bool, ...), die durchaus reichen, um einfache Aufgaben zu lösen, kann es bei komplexeren Aufgaben schwierig werden.

Hier bietet C++ die Möglichkeit, eigene, benutzerdefinierte Datentypen zu erstellen, die an das zu lösende Problem angepaßt sind, ähnlich den Strukturen in C. Das besondere an Klassen ist, dass hier nicht nur Daten gespeichert werden, sondern auch die Funktionen, die mit diesen Daten arbeiten.

Klassen haben ein eigenes Schlüsselwort *class* und einen Namen. Variablen innerhalb von Klassen nennt man *Attribute* und Funktionen innerhalb von Klassen *Methoden*. Beides zusammen wird unter dem Begriff *Member* zusammengefasst.



Abbildung 4.1: Die Klasse *Lampe* und ihr UML-Diagramm.

Ein einfaches Beispiel ist in Abbildung 4.1 zu sehen. Zur Veranschaulichung werden in diesem Skript UML-Diagramme² verwendet. Das Diagramm zur Klasse *Lampe* sehen Sie ebenfalls in der Abbildung. Es zeigt, dass die Klasse *Lampe* sechs Member besitzt, von denen jeweils drei Methoden bzw. Attribute sind.

Mit Hilfe des Klassennamens lassen sich Variablen vom Typ dieser Klasse anlegen. Mit

¹Dieser Systembegriff wurde wesentlich von Niklas Luhmann geprägt.

²Sollten Sie mehr über UML-Diagramme erfahren wollen, können sie zum Beispiel hier weiterlesen: https://de.wikipedia.org/wiki/Unified_Modeling_Language

```
1 || Lampe Beispiellampe;
```

wird eine Variable vom Typ Lampe angelegt.

Zusätzlich unterstützt das C++ Klassenkonzept Zugriffsbeschränkungen, die hier mit den Schlüsselwörtern *private* und *public* gekennzeichnet sind. Dies wird in Abschnitt 4.2.5 genauer erläutert.

4.2.1 Methoden und Attribute

Funktionen, die in einem direkten Zusammenhang mit den Daten einer Klasse stehen, werden als deren Methoden in die Klasse eingebettet. Die Variablen einer C++ Klasse — also ihre *Attribute* — sind standardmäßig nach außen hin nicht sichtbar. Dies entspricht einem Grundprinzip der objektorientierten Programmierung, dem *Delegationsprinzip*: Attribute werden nach außen hin unsichtbar gemacht und der Zugriff auf sie erfolgt über die Methoden der Klasse. Dieses Prinzip hat für den Programmierer wesentliche Vorteile (z.B. Konsistenz der Daten) und wird entsprechend häufig verwendet.[1]

Der folgende Codeausschnitt zeigt beispielhaft die Definition einer Klasse *Student*, die eine Schablone für Studenten enthält. Eine Übersicht über alle Methoden und Attribute sehen Sie auch noch einmal im zugehörigen UML-Diagramm. Mit dieser Klasse ist es zum Beispiel möglich, eine Datenbank mit allen Studenten, die am Praktikum teilnehmen, zu erstellen. Wie die Beispielklasse am Anfang des Kapitels besteht auch diese Klasse aus Attributen und Methoden³. Sie enthält Variablen für die Matrikelnummer, den Namen des Studenten, dessen erreichte Punktzahl und ob das Praktikum bestanden wurde.

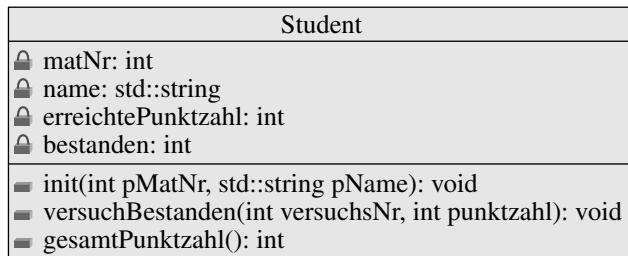


Abbildung 4.2: UML-Diagramm zur Klasse *Student*

```
1 || #ifndef STUDENT_H
2 || #define STUDENT_H
3 |
4 class Student
5 {
6 private:
7     int matNr;
8     std::string name;
9     int erreichtePunktzahl;
10    int bestanden;
11
12 public:
13     /* Initialisiert das Objekt,
14      * auf jeden Fall nach der Erzeugung eines Objekts aufrufen!
15      */
16     void init(int pMatNr, std::string pName);
17     void versuchBestanden(int versuchsNr, int punktzahl);
18     int gesamtPunktzahl();
19 };
20
21 #endif
```

Listing 4.1: Die Klasse Student (Datei *Student.h*)

³Die Befehle in den Zeilen 1, 2 und 21 werden in Abschnitt 4.5 erläutert und sind hier nur der Vollständigkeit halber erwähnt.

Die Funktionen ermöglichen es einem Anwender⁴, mit der Klasse zu kommunizieren. Man kann dabei unterscheiden zwischen *Aufträgen* und *Anfragen*. Die Methode *versuchBestanden* stellt einen Auftrag dar - in diesem Fall besteht der Auftag darin, die Information, dass ein Student einen Versuch bestanden hat, zu verarbeiten. Die Methode *gesamtPunktzahl* hingegen ist eine Anfrage, da hier aus der Klasse eine Information abgefragt wird. Diese Methode hat einen Rückgabewert. Die Methode *init* hat eine Sonderstellung und wird später noch genauer behandelt.

Eine Klasse besteht in C++ in der Regel aus zwei Dateien. Zum einen aus einer so genannten *Header-Datei*, die die Endung „.h“ besitzt. Sie enthält die *Definitionen* der Variablen und die *Deklarationen* der Methoden. Eine beispielhafte *Header-Datei* haben Sie schon in der Klasse *Student* gesehen.

Die andere Datei enthält die *Definitionen* der Methoden, oft auch *Implementierung* genannt, also die Funktionen an sich. Die Implementierung der Klasse enthält den eigentlichen Quelltext und wird mit der Dateiendung „.cpp“ gekennzeichnet. Eine beispielhafte Implementierung zur Klasse *Student* sehen Sie im folgenden Codeausschnitt:

```

1 #include "Student.h"
2
3 void Student::init(int pMatNr, std::string pName)
4 {
5     /* eine beispielhafte Implementierung der Funktion 'init' */
6     erreichtePunktzahl = 0;
7     this->matNr = pMatNr;
8     bestanden = 0;
9     this->name = pName;
10 }
11
12 void Student::versuchBestanden(int versuchsNr, int punktzahl)
13 {
14     /* Hier folgt die Implementierung der Funktion 'versuchBestanden'. */
15 }
16
17 int Student::gesamtPunktzahl()
18 {
19     return erreichtePunktzahl;
20 }
```

Listing 4.2: Die Klasse Student (Datei *Student.cpp*)

Die Anweisung im Listing 4.2 in Zeile 1 bewirkt, dass diese Zeile durch den Inhalt der Header-Datei an dieser Stelle ersetzt wird.

Bei einem Aufruf des Compilers werden nur die jeweiligen „.cpp“-Dateien einzeln übersetzt, da nur sie den Programmcode enthalten. Durch das Einfügen der „.h“-Dateien weiß der Compiler, wie er die Ausdrücke in dieser Datei zu interpretieren hat. Trifft er z.B. auf *erreichtePunktzahl*, weiß er, was dieser Ausdruck verkörpert.

In Zeile 3 signalisiert der sogenannte *Scope-Operator* „*Student::*“, dass es sich bei der folgenden Funktion um eine Memberfunktion der Klasse *Student* handelt.

Ohne diesen *Scope-Operator* würde der Compiler versuchen, eine globale Funktion gleichen Namens zu erstellen, was in der Regel zu einem Fehler führt.

Auf die Attribute und Methoden von Objekten wird mit dem „.“-Operator (Punktoperator) zugegriffen. Im folgenden Beispiel wird in Zeile 6 ein Objekt *heinz* erstellt. In Zeile 9 wird für *heinz* die Funktion *init()* aufgerufen und in Zeile 12 *versuchBestanden()*. Man braucht also immer ein konkretes Objekt, um auf Member der Klasse zugreifen zu können.

Ähnlich ist es beim Objekt *maria*, nur das es sich hier um einen Zeiger auf das Objekt *maria* handelt. Dieser Zeiger muss dereferenziert werden, um das Objekt zu erhalten. Hierzu bietet C++ zwei Möglichkeiten, den „*“-Operator oder den „->“-Operator. Den „*“-Operator haben Sie bereits in Kapitel 2.1.4 kennengelernt.

In Zeile 7 wird zunächst ein Objekt *maria* vom Typ *Zeiger auf Objekt Student* erzeugt, dann wieder im Heapspeicher mit *new* ein entsprechendes Objekt angelegt und die Adresse dem Zeiger *maria* zu-

⁴Mit „Anwender“ kann hier auch ein Programmierer gemeint sein, der unseren Code für sein Projekt weiterverwendet.

gewiesen.

In den Zeilen 10 und 13 sehen Sie den beispielhaften Gebrauch des „->“-Operators.

```

1 || #include "Student.h"
2 || int main()
3 || {
4 ||     /* Erstelle zwei Objekte der Klasse Student.
5 ||         Maria ist ein Zeiger auf einen Studenten */
6 ||     Student heinz;
7 ||     Student *maria = new Student();
8 ||     /* initialisiere */
9 ||     heinz.init(123456, "Heinz");
10 ||    maria->init(123457, "Maria");
11 |
12 ||    heinz.versuchBestanden(1, 2);
13 ||    maria->versuchBestanden(1, 3);
14 |
15 ||    delete maria;
16 ||    return 0;
17 || }
```

Listing 4.3: Zugriff auf die Methoden der Klasse *Student*

4.2.2 Konstruktoren und Destruktoren

Die Benutzung der *init()*-Methode zur Initialisierung der Membervariablen ist unelegant und fehleranfällig. Der Programmierer ist nicht verpflichtet, die *init()*-Methode aufzurufen, doch das Objekt ist nur sinnvoll, wenn die *init()*-Methode unmittelbar nach dem Anlegen der Instanz ausgeführt wird. C++ bietet besondere Methoden an, die bei jeder Erzeugung eines Objekts automatisch aufgerufen werden. Diese Methoden nennt man **Konstruktoren**.

Konstruktoren sind dadurch gekennzeichnet, dass sie denselben Namen wie die Klasse tragen und zusätzlich keinen Rückgabetyp definieren, da der Rückgabetyp der Klasse selbst entspricht.

Ein Konstruktor schafft die Voraussetzungen, um ein Objekt in einen sinnvollen Anfangszustand zu versetzen. Hier werden Attribute initialisiert, Ressourcen angefordert und was sonst noch alles nötig ist. Da es unterschiedliche, sinnvolle Anfangswerte für ein Objekt geben kann, kann es für eine Klasse auch mehrere Konstruktoren geben. Sie müssen sich aber in der Anzahl der Parameter und/oder den Datentypen der Parameter voneinander unterscheiden (weiteres hierzu in Abschnitt 7.1.1). Welcher Konstruktor dann aufgerufen wird, bestimmt der Compiler anhand der Parameterliste bzw. -typen bei der Definition des Objekts.

Wie im UML-Diagramm in Abbildung 4.3 zu sehen ist, sind im folgenden Codeabschnitt neue Memberfunktionen hinzugekommen, die keine Rückgabewerte haben. Dies wird im Folgenden genauer erklärt.

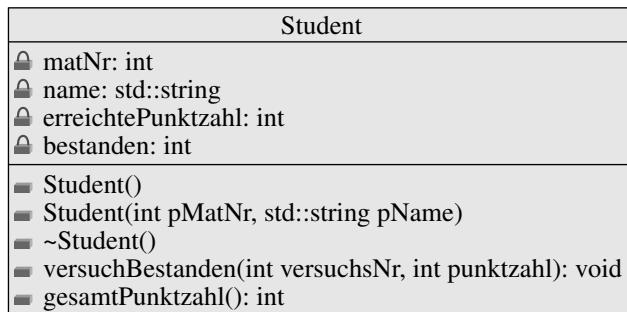


Abbildung 4.3: UML-Diagramm zur Klasse *Student*

```
1 || class Student
```

4 Einführung in Klassen

```

2  {
3  private:
4      int matNr;
5      std::string name;
6      int erreichtePunktzahl;
7      int bestanden;
8
9  public:
10     /* Konstruktoren */
11     Student();
12     Student(int pMatNr, std::string pName);
13
14     /* Destruktor */
15     ~Student();
16
17     void versuchBestanden(int versuchsNr, int punktzahl);
18     int gesamtPunkzahl();
19 }
20
21 //eine beispielhafte Implementierung der beiden Konstruktoren
22
23 Student::Student()
24 {
25     erreichtePunktzahl = 0;
26     matNr = 0;
27     bestanden = 0;
28     name = "";
29 }
30
31 Student::Student(int pMatNr, std::string pName)
32 {
33     erreichtePunktzahl = 0;
34     this->matNr = pMatNr;
35     bestanden = 0;
36     this->name = pName;
37 }
38
39 // Implementierung des Destruktors
40
41 Student::~Student()
42 {
43 }
44 }
```

Listing 4.4: Deklaration von Konstruktoren innerhalb einer Klasse

Der Codeausschnitt zeigt die Klasse *Student* mit zwei verschiedenen Konstruktoren. Der *Standardkonstruktor* (auch Default-Konstruktor) in Zeile 11 besitzt keine Parameter. Er ruft zu allen Membervariablen die Standardkonstruktoren auf. In C++ gelten auch die einfachen Datentypen wie *int* oder *double* als Klasse und werden über einen entsprechenden Konstruktor erzeugt. Häufig werden Standarddatentypen vom Programmierer im Funktionsrumpf noch mit 0 o.ä. (z.B. einer Interpretation für „leer“) belegt. Ist kein Konstruktor in der Klasse definiert, erzeugt C++ den Standardkonstruktor automatisch – dann ist der Funktionsrumpf natürlich leer, und es werden nur die Standardkonstruktoren aller Attribute aufgerufen. Wird bei der Erzeugung eines Objektes kein anderer Konstruktor angegeben, verwendet der Compiler den Standardkonstruktor zur Instanzierung. Wenn Sie also nur den Standardkonstruktor benötigen, müssen Sie diesen nicht extra implementieren. In Zeile 3 im Listing 4.5 wird der Standardkonstruktor benutzt. Zusätzlich hat die Klasse noch einen Konstruktor, der die Attribute für Matrikelnummer und Namen mit den übergebenen Werten initialisiert. Die Konstruktoren werden bei der Erzeugung der Objekte direkt aufgerufen, wie im Listing 4.5 zu sehen ist. Die *init()*-Methode ist nun nicht mehr nötig.

```

1  int main()
2  {
3      Student jim;
4      Student heinz(123456, "Heinz");
5 }
```

```

6  || Student* maria = new Student(123457, "Maria");
7  ||
8  ||     jim.versuchBestanden(2, 2)
9  ||     heinz.versuchBestanden(1, 2);
10 ||    maria->versuchBestanden(1, 3);
11 ||
12 || delete maria;
13 || return 0;
14 |

```

Listing 4.5: Aufruf von Konstruktoren und Destruktoren

Hinweis: Sobald in einer Klasse nur Konstruktoren mit Übergabeparametern definiert sind und kein Standardkonstruktor, wird der Compiler auch keinen Standardkonstruktor erzeugen. Von dieser Klasse kann dann kein Default-Objekt (ohne Übergabeparameter) instanziert werden. Der Compiler wird dann einen Fehler melden.

Ein Beispiel dazu werden Sie im übernächsten Kapitel bei den *Initialisierungslisten* kennenlernen.

Ein **Destruktor** ist das genaue Gegenteil eines Konstruktors. So wie der Konstruktor ausgeführt wird, während eine Instanz erzeugt wird, wird der Destruktor ausgeführt, wenn eine Instanz gelöscht wird. Dieser wird gewöhnlich dazu genutzt, um Ressourcen, die nach dem Lebensende der jeweiligen Instanz nicht mehr benötigt werden, freizugeben und sonstige Aufräumarbeiten zu erledigen.

Für jede Klasse kann und soll es genau einen Destruktor geben. Aus diesem Grunde weist der Compiler Klassen, die keinen Destruktor definieren, automatisch einen Standarddestruktur mit leerem Anweisungsteil zu. Die Definition eines eigenen Destruktors ist ähnlich zu einer Definition eines Konstruktors. Der Destruktor wird durch eine Tilde (~) vor dem Destruktornamen gekennzeichnet und hat nie einen Übergabeparameter (siehe Klasse *Student*). Im Listing 4.5 wird in Zeile 12 der Destruktor von Maria aufgerufen.

4.2.3 Der this-Zeiger

Im Listing 4.2 wurde in Zeile 9 ein sogenannter *this-Zeiger* verwendet.

Innerhalb von Klassenmethoden kann auf alle Elemente der Klasse zugegriffen werden, egal, ob sie nach außen hin sichtbar sind oder nicht (*private*). Syntaktisch befinden sich die Methoden im Namensbereich der entsprechenden Klasse. Werden mehrere Objekte (Instanzen) einer Klasse gebildet, so wie im vorherigen Beispiel, bekommt jedes Objekt eine Kopie der Attribute (Variablen) der Klasse. Die Methoden (Funktionen) der Klasse stehen aber nur einmal im Arbeitsspeicher und werden von allen Instanzen der Klasse gemeinsam benutzt.

Damit die Methoden wissen, auf welchem Datensatz (Objekt) sie arbeiten sollen, fügt der Compiler automatisch in jede Methode als ersten Parameter einen Zeiger vom Typ der Klasse ein und benutzt diesen bei jedem Zugriff auf die Attribute. Ein Beispiel soll dies verdeutlichen:

```

1  || class Einfach
2  || {
3  ||     private:
4  ||         int wert;
5  ||     public:
6  ||         void setWert(int j);
7  || };
8
9  void Einfach::setWert(int j)
10 {
11     wert = j;
12 }
13
14 int main()
15 {
16     Einfach einfach1;
17     einfach1.setWert(5);
18 }

```

Aus dem Aufruf der Methode `einfach1.setWert(5)` wird durch den Compiler `setWert(&einfach1, 5)`. Der Compiler hat einen Zeiger vom Typ `Einfach` als erstes Argument in den Methodenaufruf eingefügt. Damit dieser Aufruf funktionieren kann, muss auch die Methode `setWert` entsprechend modifiziert werden. Der Scopeoperator zeigt dabei an, für welche Klasse diese Modifikation stattfindet.

```
1 void setWert(Einfach* this, int j)
2 {
3     this->wert = j;
4 }
```

So sieht die Methode nach der Compilierung aus. Aufgerufen werden kann sie nur, wenn auch ein Objekt der Klasse `Einfach` bereits existiert, sonst gibt es keinen Zeiger für das erste Argument und damit auch keinen Zeiger auf die Attribute (dadurch wird sie quasi zu einer Memberfunktion). Der `this`-Zeiger steckt im Namen des Objekts, mit dem die Methode aufgerufen wird. Dem Programmierer steht die Nutzung des `this`-Zeigers, wie Sie schon gesehen haben, auch zur Verfügung und kann notwendig sein, wenn es bei Attributen und Übergabeparametern zu Namenskollisionen kommt, wie in Listing 4.2 Zeile 9.

Weil dies bei fast allen Memberfunktionen so ist, übernimmt der Compiler diese interne Umwandlung automatisch. Der Programmierer muss sich nicht darum kümmern und spart einiges an Tipparbeit. Er muss diese Tatsache jedoch im Hinterkopf haben.

4.2.4 Initialisierungslisten

Eine weitere Möglichkeit, die Attribute einer Klasse bei der Instanzierung zu initialisieren, stellen *Initialisierungslisten* dar. In Listing 4.6 sehen Sie den Aufbau einer solchen Liste anhand der Klasse `Student`.

Nach dem Funktionskopf des Konstruktors, aber noch vor dem Funktionsrumpf, der mit der ersten öffnenden, geschweiften Klammer beginnt, werden die Elemente der Initialisierungsliste eingefügt. Die Liste wird durch einen Doppelpunkt vom Funktionskopf getrennt.

```
1 Student::Student(int pMatNr, std::string pName)
2     : matNr(pMatNr), name(pName), bestanden(0), erreichtePunktzahl(0)
3 {
4 }
```

Listing 4.6: Initialisierungsliste

Der Name des zu initialisierenden Attributs wird zusammen mit dem entsprechenden Parameter des Konstruktors in der Liste aufgeführt. Statt eines Parameters kann auch ein für dieses Attribut gültiger Wert benutzt werden (hier `bestanden(0)`). Mehrere Initialisierungen werden durch Kommata getrennt.

Initialisierungslisten dienen dazu, dem Compiler mitzuteilen, welchen Konstruktor er mit welchem Wert bei der Erstellung der jeweiligen Attribute verwenden soll. Die Attribute werden mit dem Wert *initialisiert*, daher der Name Initialisierungsliste.

Das folgende Beispiel[1] soll veranschaulichen, warum es nötig sein kann, Initialisierungslisten zu verwenden.

```
1 class Point
2 {
3 public:
4     Point(int pX, int pY);
5
6 private:
7     int x;
8     int y;
9 };
10
11 class Circle
12 {
13 public:
```

```

14 |     Circle(int pX, int pY, double pRadius);
15 |
16 |     private:
17 |         Point middle;
18 |         double radius;
19 |     };
20 |
21 | Point::Point(int pX, int pY)
22 | {
23 |     this->x = pX;
24 |     this->y = pY;
25 | }
26 |
27 | Circle::Circle(int pX, int pY, double pRadius)
28 | {
29 |     this->middle = Point(pX,pY);
30 |     this->radius = pRadius;
31 | }

```

Listing 4.7: Fehlende Initialisierungsliste

Es gibt zwei Klassen, *Point* und *Circle*. Die Klasse *Point* besitzt zwei Attribute vom Typ *int* und einen Konstruktor, der zwei Intergerwerte erwartet. Da alle Standarddatentypen auch über einen Standardkonstruktor verfügen, ist dies kein Problem. Der Compiler ruft für die beiden Attribute jeweils den Standardkonstruktor für Integer auf. Allerdings hat die Klasse *Point* selbst keinen Standardkonstruktor.

Die Klasse *Circle* hat ebenfalls zwei Attribute, eins vom Typ *double* und eins vom Typ *Point*. Klassen können durchaus auch wieder Attribute in anderen Klassen sein.

Beim Compilieren von *Circle* will der Compiler den Standardkonstruktor von *Point* verwenden, den es aber nicht gibt. Dies führt zu einer Fehlermeldung, da *Circle* nicht angelegt werden kann. Dazu muß man wissen, dass alle Attribute schon beim Compilieren angelegt werden. Der Methodenrumpf des Konstruktors wird erst danach ausgeführt, dazu muss das Attribut *middle* schon existieren.

Eine Initialisierungsliste löst dieses Problem, indem sie dem Compiler mitteilt, welchen Konstruktor er für *Point* mit welchen Werten benutzen soll.

```

1 | class Point
2 | {
3 | public:
4 |     Point(int pX, int pY);
5 |
6 | private:
7 |     int x;
8 |     int y;
9 | };
10 |
11 class Circle
12 {
13 public:
14     Circle(int pX, int pY, double pRadius);
15 |
16 private:
17     Point middle;
18     double radius;
19 };
20 |
21 Point::Point(int pX, int pY) : x(pX), y(pY)
22 {
23 }
24 |
25 Circle::Circle(int pX, int pY, double pRadius) :
26     middle(pX,pY), radius(pRadius)
27 {
28 }
29 |
30 int main()

```

```
31 || t
32 ||     Circle kreis(4, 5, 2);
33 || }
```

Listing 4.8: Korrigierte Initialisierungsliste

Beide Konstruktoren verfügen über eine Initialisierungsliste, wobei sie bei *Point* nicht nötig wäre. Der Konstruktor von *Circle* erhält die Koordinaten eines Punktes, die der Compiler jetzt benutzt, um das Attribut *Point middle* mit dem richtigen Konstruktor aus der Initialisierungsliste mit den entsprechenden Werten anzulegen, hier 4 und 5.

Konstante Attribute, deren Werte bei jedem neu erzeugten Objekt(Instanz) verschieden sein können, erfordern zwingend einen Eintrag in der Initialisierungsliste, da sie nur an dieser Stelle einmalig initialisiert werden können. Wären also die beiden Attribute von *Point* konstant, dann bräuchte *Point* ebenfalls eine Initialisierungsliste.

4.2.5 Zugriffsbeschränkung

Mit der Einführung der Klassen wurde auch ein Konzept zur Definition von Zugriffsbeschränkungen auf Methoden und Attribute in Klassen eingeführt.

Alle in einer Klasse definierten Elemente (Member) können ohne Einschränkungen von den anderen Elementen (z.B. Methoden, Konstruktoren etc.) aus derselben Klasse verwendet werden, während der Zugriff von außerhalb durch die Zugriffsspezifizierer geregelt wird. C++ kennt drei Zugriffsspezifizierer: den öffentlichen (*public*), den geschützten (*protected*) und den privaten (*private*) Teil einer Klasse. Die Erläuterungen dazu finden Sie in der Tabelle 4.1:

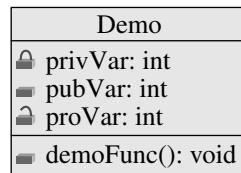
Tabelle 4.1: Zugriffsspezifizierer für Klassenelemente

Spezifizierer	Erläuterung
<code>public</code>	uneingeschränkt zugänglich von jeder externen Stelle
<code>protected</code>	zugänglich innerhalb der eigenen Klasse sowie sämtlicher abgeleiteten Klassen (siehe auch Abschnitt 8.6)
<code>private</code>	zugänglich nur innerhalb der eigenen Klasse. Dies ist auch der Standardzugriff für Elemente, die ohne Spezifizierer deklariert wurden.

Wie bereits erwähnt wurde, sind bei Klassen alle Member, die keinem Zugriffsspezifizierer zugeordnet sind, automatisch *private*.

Der Zugriffsspezifizierer *protected* spielt erst bei abgeleiteten Klassen eine große Rolle und wird erst in Kapitel 8.6 genauer erläutert.

Der folgende Codeausschnitt demonstriert den Effekt der Zugriffsspezifizierer beim Zugriff von innerhalb der Klasse und beim Zugriff über ein Objekt der Klasse. Die Zugriffsbeschränkung ist ein Hilfsmittel, um interne Daten vor dem Zugriff von außen zu schützen. So kann man beispielsweise verhindern, dass ungültige *privVar* und *proVar* von *außen* modifiziert werden. Nach dem bereits erwähnten *Delegationsprinzip* ist es in der Objektorientierten Programmierung wünschenswert, Attribute nicht nach außen sichtbar zu machen.

Abbildung 4.4: UML-Diagramm zur Klasse *Demo*.

```

1 #include <iostream>
2
3 class Demo
4 {
5     private:
6         int privVar;
7
8     public:
9         int pubVar;
10
11    protected:
12        int proVar;
13
14    public:
15        void demonFunc();
16    };
17
18 void Demo::demonFunc()
19 {
20     /* Zugriff auf Elemente der eignen Klasse */
21     privVar = 2; //ok
22     pubVar = 2; //ok
23     proVar = 2; //ok
24 }
25
26 int main()
27 {
28     Demo obj;
29     obj.privVar = 2; //Fehlermeldung
30     obj.pubVar = 2; //ok
31     obj.proVar = 2; //Fehlermeldung
32
33     return 0;
34 }
  
```

Listing 4.9: Zugriffsbeschränkung am Beispiel der Klasse *Demo*

4.3 Parameter

4.3.1 Konstante Parameter und Call By Reference

Wie Sie bereits in Versuch 1 erfahren haben, kann man mit dem Schlüsselwort *const* angeben, dass eine Variable nicht verändert werden kann. Dies ist auch für die Parameter in Funktionen möglich. Alle Parameter, die eine Funktion nicht verändern soll, werden auf *const* gesetzt.

Was man unter *Call By Reference* zu verstehen hat, wurde ebenfalls in Versuch 2 eingeführt. Das folgende Beispiel verdeutlicht das Prinzip:

```
1 || void insertInList(Student& teilnehmer);
```

Diese Funktion fügt einen Studenten in eine (hier nicht weiter spezifizierte) Liste ein. Würde man in dieser Funktion *Call By Value* benutzen, dann müsste das gesamte Objekt *teilnehmer* kopiert

werden, was bei manchen Objekten recht umfangreich sein kann. Wenn diese Kopie dann in eine Liste eingetragen wird, und diese Liste länger besteht, gibt es Probleme, denn die Kopie des Objekts wird am Ende der Funktion wieder gelöscht.

Außerdem kann die Funktion die Daten des Studenten absichtlich oder unabsichtlich ändern. Und weil es eine Referenz ist, ändert sich dadurch auch der äußere Wert.

Man kann diese beiden Techniken kombinieren: Übergibt man als Parameter eine Referenz oder einen Zeiger als Konstante, so nennt man dies *Const-Maskierung*.^[1] Im Beispiel würde dies folgendermaßen aussehen:

```
1 || void insertInList(const Student& teilnehmer);
```

Hierbei ist zu beachten, dass das Objekt, auf das sich die Referenz bezieht, selbst nicht konstant sein muss. Dies gilt nur innerhalb der Funktion, in die der Parameter als Konstante übergeben wurde.

Möchte man ein konstantes Objekt per Referenz an eine Funktion übergeben, muss dieser Parameter konstant sein. Die Verwendung konstanter Parameter hat den Vorteil, dass man einen besseren Überblick darüber bekommt, wann Daten verändert werden.

Zum Schluss sei noch erwähnt, dass man auch konstante Zeiger als Parameter verwenden kann. Dies sähe dann zum Beispiel so aus:

```
1 || void insertInList(const Student* teilnehmer);
```

4.3.2 Defaultparameter

Manchmal ist es erwünscht, einen Parameter in einer Methode nicht immer explizit setzen zu müssen. Wenn man die Beispielklasse *Student* um eine Membervariable ergänzt, welche das aktuelle Fachsemester enthält, ist diese in den meisten Fällen „2“.

Die neue Deklaration der Klasse sieht wie folgt aus:

```
1 || class Student
2 {
3     private:
4         int matNr;
5         std::string name;
6         int erreichtePunktzahl;
7         int fachsemester;
8         int bestanden;
9
10    public:
11        /* Konstruktor */
12        Student(int pMatNr, std::string pName, int pFachsemester = 2);
13        /* Destruktor */
14        ~Student();
15
16        /* Methoden */
17        void versuchBestanden(int versuchsNr, int punktzahl);
18        int gesamtPunktzahl();
19 };
```

Listing 4.10: Defaultparameter (Deklaration)

An der Implementierung der Klasse ändert sich nur die Initialisierung der Membervariable:

```
1 || Student::Student(int pMatNr, std::string pName, int pFachsemester):
2     matNr(pMatNr), name(pName), fachsemester(pFachsemester), bestanden(0),
3     erreichtePunktzahl(0)
4 {
```

Listing 4.11: Defaultparameter (Verwendung)

Beachten Sie, dass der Defaultparameter nur in der Deklaration der Methode angegeben werden darf. Ebenfalls ist zu beachten, dass rechts neben dem ersten Defaultparameter nur weitere Defaultparameter stehen dürfen.

Ein Aufruf der Methode kann nun entweder mit einem Wert für die Variable *fachsemester*, oder ohne diesen Parameter erfolgen. Im zweiten Fall wird der Defaultwert angenommen.

```

1 int main()
2 {
3     /* erstelle zwei Objekte der Klasse Student */
4     Student heinz(123456, "Heinz");
5     Student* maria = new Student(123457, "Maria", 4);
6     /* ... */
7 }
```

Listing 4.12: Defaultparameter (Verwendung)

4.4 Konstante Methoden

Wie bei Variablen gibt es die Möglichkeit, Methoden als konstant zu definieren. Da Klassen einen Datentyp definieren, von dem Objekte erstellt werden, ist es natürlich auch möglich, konstante Objekte der Klasse zu erzeugen. Funktionsaufrufe dieses Objektes wären nicht möglich, da nicht sichergestellt ist, dass sich das Objekt auch nicht verändert. Hier kommen *konstante Methoden* ins Spiel. Eine solche Methode kann Daten aus einem Objekt lesen, aber nicht schreiben. Den Unterschied zwischen konstanten und nicht konstanten Methoden verdeutlicht folgendes Beispiel:

```

1 int main()
2 {
3     const Student john(123458, "John");
4     john.versuchBestanden( 1, 3);
5     int result = john.gesamtPunktzahl();
6 }
```

Listing 4.13: Der Unterschied zwischen konstanten und nicht konstanten Methoden

Beide Funktionsaufrufe sind verboten und werden vom Compiler mit einem Fehler quittiert. Der Aufruf der zweiten Funktion ist aber ungefährlich, da er nur einen lesenden Zugriff darstellt, er sollte explizit erlaubt werden. Dazu wird die Methode *gesamtPunktzahl* als konstante Methode auswiesen. Eine konstante Methode wird, genau wie ein konstantes Attribut, durch ein zusätzliches *const* definiert:

```

1 class Student
2 {
3     /* ... */
4     int gesamtPunktzahl() const;
5 };
```

In konstanten Methoden können wiederum nur konstante Methoden der Klasse aufgerufen werden, um sicherzustellen, dass keine Daten verändert werden.

Um Fehler zu vermeiden, ist es sinnvoll, wann immer möglich eine Funktion als konstant zu deklarieren.

4.5 Include-Wächter

Am Anfang dieses Kapitels wurden in Listing 4.1 folgende Zeilen nicht erklärt:

```

1 #ifndef STUDENT_H
2 #define STUDENT_H
3 /* ... */
4 #endif
```

Listing 4.14: Ausschnitt aus Listing 4.1

4 Einführung in Klassen

In C++ darf eine Klasse, eine Aufzählung oder Ähnliches in einer Übersetzungseinheit (also in einer Datei) nur einmal definiert sein.

Durch die Header-Dateien sind aber Konstrukte möglich, in denen diese Regel missachtet wird:

Datei ‘grandfather.h’

```
1 || class Grandfather
2 || {
3 ||     int member;
4 || }
```

Datei ‘father.h’

```
1 || #include "grandfather.h"
2 || /* ... */
```

Datei ‘child.cpp’

```
1 || #include "grandfather.h"
2 || #include "father.h"
3 || /* ... */
```

In der Datei „child.cpp“ ist die Datei „grandfather.h“ einmal direkt und einmal über die Datei „father.h“ eingebunden. Dies verstößt gegen die *One-Definition-Rule*, da die Klasse Grandfather offensichtlich zweimal definiert ist. Abhilfe schaffen sogenannte *Include-Wächter*.

Betrachten Sie Listing 4.14:

- In Zeile 1 wird abgefragt, ob ein Makro mit dem Namen STUDENT_H definiert ist. Fortgefahren wird nur im Fall, dass dieses Makro noch nicht definiert ist.
- In Zeile 2 wird dieses Makro definiert.
- Zeile 4 markiert das Ende der If-Abfrage.

Beim ersten Einbinden der Datei wird also ein Makro definiert, welches verhindert, dass die Datei noch einmal eingebunden wird. In der Datei „child.cpp“ taucht also die Definition der Klasse „Grandfather“ nur einmal auf. Daher der Rat an Sie für die Zukunft:

- Fangen Sie *jede* Klassendefinition, die Sie erstellen, mit einem Include-Wächter an.
- Wählen Sie den Makronamen *eindeutig* und klar zur Klasse gehörend.

4.6 Aufgaben

In den nachfolgenden Aufgaben wird die Klasse „Vektor“ erstellt. Diese repräsentiert einen 3-dimensionalen Vektor.

Rotationsmatrix

Eine Rotationsmatrix beschreibt eine Drehung um eine Achse. Wenn Sie einen Vektor mit dieser Matrix multiplizieren, erhalten Sie einen Vektor, der um den eingestellten Winkel gedreht ist. Ein Beispiel für die Rotation um die z-Achse:

$$\vec{y} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_x \\ x_y \\ x_z \end{pmatrix}$$

Die Klasse Vektor

Erstellen Sie ein neues Projekt mit dem Namen "Versuch04" und importieren Sie die zur Verfügung gestellten Codevorlagen. Sie finden ein Grundgerüst für die Klasse **Vektor** vor, welche 3 private Membervariablen x, y und z enthält sowie die Deklarationen der Funktionen. Die Funktion *ausgabe()* ist bereits implementiert. Testen Sie die Funktionen, nachdem Sie diese implementiert haben, einzeln, indem Sie in der Datei main.cpp einige geeignete Vektoren erstellen und verwenden. Für die Funktion *ausgabe()* ist das schon geschehen. Die Klasse soll zusätzlich folgende public-Funktionen besitzen:

- Eine Funktion, in der die Länge des Vektors berechnet und zurückgegeben wird.
- Eine Funktion, die zwei Vektoren subtrahiert und den Ergebnisvektor zurückgibt.
- Eine Funktion, die das Skalarprodukt zweier Vektoren berechnet und das Ergebnis zurückgibt.
- Eine Funktion, die den Winkel zwischen zwei Vektoren in Grad berechnet und zurückgibt.
- Eine Funktion, die einen Vektor um die z-Achse rotiert. Übergeben Sie der Funktion dazu einen Winkel in *Radian*.

Benutzen Sie zur Implementierung die vorgegebenen Funktionsdeklarationen. Implementieren Sie keine weiteren Funktionen innerhalb der Klasse *Vektor*. Benutzen Sie die vorhandenen, um die folgende Aufgabe zu lösen. Es gibt mehrere Möglichkeiten, dies zu realisieren.

Denken Sie daran, alle Funktionen und übergebenen Variablen, soweit möglich, als konstant zu definieren.

Wie weit entfernt ist der Horizont?

Berechnen Sie die Entfernung, die ein Mensch von 1,70 Meter Körpergröße, der auf einer 555,7 Meter hohen Plattform steht, auf einer (ideal runden) Erde sehen kann, bevor die Erdkrümmung dies verhindert. Betrachten Sie Abbildung 4.5 und nutzen Sie die Klasse *Vektor* und die von Ihnen in Aufgabe 4.6 implementierte Funktion, um diese Aufgabe zu lösen. Entwickeln Sie eine iterative Lösung für dieses Problem und verwenden Sie dabei nicht den Satz des Pythagoras oder die trigonometrischen Funktionen. Beachten Sie, dass bei der Berechnung mit Fließkommazahlen ganzzahlige Werte aufgrund der großen Genauigkeit, also der Anzahl der Nachkommastellen, kaum zu erreichen sind. Der Radius der ideal runden Erde soll in dieser Aufgabe 6371 Kilometer betragen.

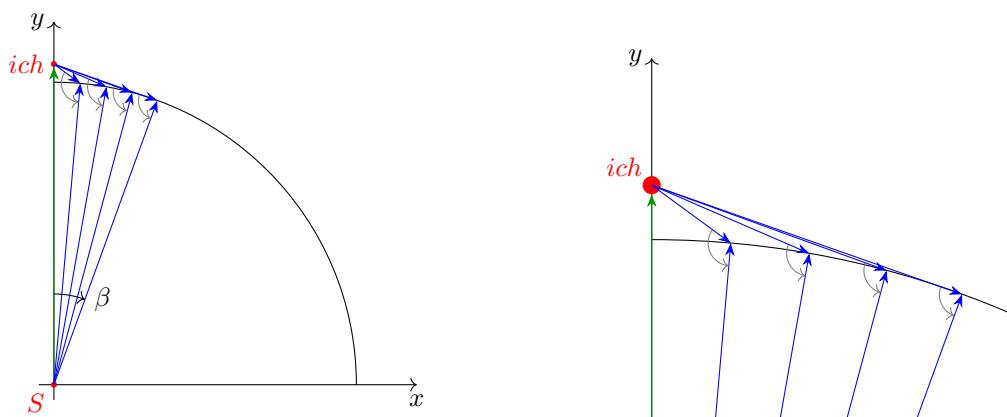


Abbildung 4.5: Die Grafik veranschaulicht das Problem. Versuchen Sie unter Zuhilfenahme der Grafik den Sachverhalt mathematisch abzuleiten.

Implementieren Sie innerhalb Ihrer Berechnungsschleife einen Zähler, der die Anzahl der notwendigen Schritte ermittelt.

Wie groß ist die Distanz, wenn der Mensch auf einer 555,7 Meter hohen Plattform steht?

Wie groß ist der Winkel β ?

Ihre Ausgabe soll folgendes Aussehen haben:

```
Sie können 84.2774 Km weit sehen.  
Sie sind 557.4000 Meter hoch.  
Der Winkel beträgt 0.7579 Grad.  
Anzahl Schritte: 1322751
```

Damit Ihre Werte in dieser Form erscheinen, stehen Ihnen durch das Einbinden der *iomanip*-Bibliothek eine Reihe sogenannter *Input-Output-Manipulatoren* zur Gestaltung Ihrer Ausgabe zur Verfügung. An dieser Stelle seien kurz diejenigen vorgestellt, die Sie zur Lösung der Aufgabe benötigen. Für alle Möglichkeiten der formatierten Ausgabe sei auf die C++ Referenz verwiesen.

```
1 || std::cout << std::fixed;
```

Dies bewirkt, dass Gleitkommazahlen nicht mehr in der Exponentdarstellung, sondern Nachkommiform dargestellt werden, und das so lange, bis diese wieder aufgehoben wird.

```
1 || std::cout << std::setprecision(n);
```

Dies legt fest, wie viele signifikante Stellen n bei den Gleitkommazahlen dargestellt werden. Es wird dabei intern gerundet, die eigentliche Ausgangszahl wird nicht verändert.

Freiwillige Zusatzaufgabe

Wie Sie gesehen haben, müssen Sie Ihre Schrittweite sehr klein wählen, damit Sie den korrekten Punkt am Horizont finden. Ist Ihre Schrittweite zu groß gewählt, laufen Sie über den Punkt hinaus. Die Wahl der richtigen Schrittweite kann einiges an Zeit in Anspruch nehmen und hat deutliche Auswirkungen auf die Laufzeit des Programms.

Implementieren Sie eine Intervallschachtelung, die in der Lage ist, festzustellen, wenn Sie über den Zielpunkt hinausgelaufen sind. Ändern Sie dann die Rotationsrichtung und verkleinern Sie die Schrittweite. Führen Sie dieses Schema solange durch, bis der Zielpunkt gefunden ist.

Realisieren Sie einen zusätzlichen Zähler innerhalb der Schleife, der die Schritte in die jeweilige Richtung ermittelt und ausgibt.

Experimentieren Sie mit der anfänglichen Schrittweite.

Mögliche Ausgabe:

```
Zu weit vorwärts gedreht. Ändere Schrittweite von 0.010000000000 zu -0.001000000000  
Winkel: 89.677662219824      Schritte in diesem Durchlauf: 1  
Zu weit rückwärts gedreht. Ändere Schrittweite von -0.001000000000 zu 0.000100000000  
Winkel: 90.013150748057      Schritte in diesem Durchlauf: 7  
Zu weit vorwärts gedreht. Ändere Schrittweite von 0.000100000000 zu -0.000010000000  
Winkel: 89.995859244649      Schritte in diesem Durchlauf: 3  
Zu weit rückwärts gedreht. Ändere Schrittweite von -0.000010000000 zu 0.000001000000  
Winkel: 90.000431719017      Schritte in diesem Durchlauf: 8  
Zu weit vorwärts gedreht. Ändere Schrittweite von 0.000001000000 zu -0.000000100000  
Winkel: 89.999973230298      Schritte in diesem Durchlauf: 8  
Zu weit rückwärts gedreht. Ändere Schrittweite von -0.000000100000 zu 0.000000010000  
Winkel: 90.000001877717      Schritte in diesem Durchlauf: 5
```

```
Sie können 84.2774452568 Km weit sehen.
```

```
Sie sind 557.3999999997 Meter hoch.
```

```
Der Winkel beträgt 0.7578810694 Grad.
```

```
Anzahl Schritte: 34
```

5 Dynamische Datenstrukturen

Algorithmen greifen in der Regel auf Datenobjekte zu und verändern diese gemäß den Anforderungen des Benutzers. Oft ist jedoch zum Zeitpunkt der Implementierung nicht bekannt, welche Menge an Datenobjekten die Algorithmen zur Laufzeit verarbeiten müssen. Verwendet der Programmierer statisch angelegte Datenobjekte, so muss er den Speicherplatzbedarf bereits während der Implementierung festlegen. Damit bleibt ihm nur die Möglichkeit, die maximal anfallende Datenmenge zu schätzen. Der dabei reservierte Speicherplatz überschreitet damit jedoch in der Regel den tatsächlich benötigten Platz. Zudem besteht die Gefahr, dass der Platz bei einer zu geringen Schätzung nicht ausreicht und der Programmablauf abgebrochen werden muss.

Zur Lösung dieses Problems bietet die Sprache C++ die Möglichkeit, so genannte dynamische Datenstrukturen einzusetzen, deren Größe erst zur Laufzeit festgelegt und auch wieder verändert werden kann. Anders als bei statischen Datenobjekten werden sie während der Implementierung über eine Adresse angesprochen, die auf ihren zur Laufzeit festgelegten Ort im Arbeitsspeicher verweist. Das Datenelement *Zeiger* speichert diese Adresse. Dies ermöglicht die Entwicklung von leistungsfähigen und effizienten Algorithmen, kann jedoch bei unsachgemäßer Verwendung zu Programmfehlern führen.

Bekanntschaft mit Zeigern haben Sie bereits in Abschnitt 2.1.4 gemacht.

5.1 Qualifikationsziele

In diesem Versuch werden die Grundlagen für dynamische Datenstrukturen aufgezeigt. Nach dem Studium dieses Versuches besitzen Sie die folgenden Kenntnisse:

- Eigenschaften der drei Speicherklassen von C++.
- Einsatz von Zeigern.
- Einsatz von dynamischen Datenstrukturen am Beispiel von einfach und doppelt verketteten Listen.
- Verwendung der Speicherverfahren FIFO und LIFO am Beispiel einer Warteschlange und eines Stapspeichers.

5.2 Praktische Anwendungen

5.2.1 Einfach verkettete Liste

Zum Speichern von mehreren Datenelementen gleichen Typs bietet sich zunächst ein Feld an. Der Vorteil eines Feldes ist, dass der Zugriff auf ein beliebiges Element immer gleich ist. Dem steht jedoch der Nachteil gegenüber, dass die Größe eines Feldes während der Programmierung festgelegt werden muss und nicht geändert werden kann. So kann es vorkommen, dass es entweder zu groß konzipiert wird (Speicherplatzverschwendungen) oder zu klein ist und nicht alle aufkommenden Datenelemente fassen kann.

Eine Lösung dieses Problems ist der alternative Einsatz einer verketteten Liste. Ihre Datenelemente werden dynamisch zur Laufzeit erzeugt und vollständig auf dem Heap gespeichert. Eine leere Liste besteht zunächst nur aus zwei Zeigern (*front* und *back*), die den Wert *nullptr* (Nullzeiger) enthalten. Dies ist ein spezieller Zeiger, aber ein Zeiger, und nicht der Wert null. Ein Listenelement (*Listenelement*) besteht immer aus den Daten selbst (*data*) und einem Zeiger (*next*), der auf das in der Liste nachfolgende Listenelement zeigt.

```

1 || class ListenElement
2 || {
3 || | private:
4 || | | ListenElement* next;
5 || | | int data;
6 || };

```

Beim Einfügen eines neuen Listenelementes wird mit dem Operator *new* zuerst der Speicher auf dem Heap allokiert und dann die Daten initialisiert (hier: Integerwert festlegen). Schließlich werden dem *next*-Zeiger des vorigen Elementes und dem Abschluss-Zeiger *back* die Adresse des neu angelegten Elementes zugewiesen. Die einfach verkettete Liste kann nur in einer Richtung durchlaufen werden, da die *next*-Zeiger nur in eine Richtung zeigen (siehe Abbildung 5.1). Der *next*-Zeiger des letzten Elements ist immer ein *nullptr*, so dass dies eine Abbruchbedingung darstellt, um Iterationen über der Liste zu beenden.

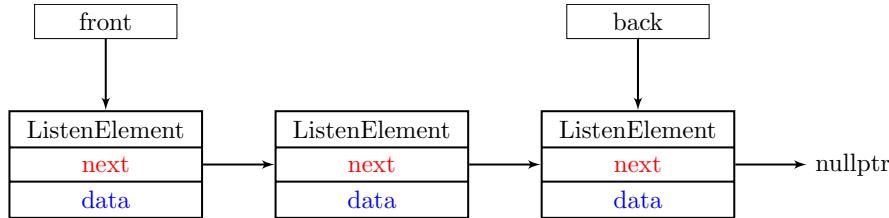


Abbildung 5.1: Schema einer einfach verketteten Liste.

5.2.2 Doppelt verkettete Liste

Möchte man auf das vorletzte Element einer einfach verketteten Liste zugreifen, so bleibt nur die Möglichkeit, mit dem *front*-Zeiger beim ersten Element zu starten, um die Liste elementweise bis zum vorletzten Element zu durchlaufen. Dieses Problem kann gelöst werden, indem die Liste durch Hinzufügen eines weiteren Zeigers (*prev*) auch in die andere Richtung verkettet wird. In diesem Fall spricht man von einer doppelt verketteten Liste (siehe Abbildung 5.2). Ein weiterer Vorteil von doppelt verketteten Listen ist das effizientere Löschen eines Elements mit einem bestimmten Wert. Da der Wert zunächst gesucht werden muss, sind bei einer einfach verketteten Liste zwei Vorgänge nötig. Zunächst muss die Liste durchlaufen werden, bis das zu löschenende Element gefunden wurde. Der Zeiger verweist dann auf dieses Element. Zum Löschen muss er aber das vorhergehende Element referenzieren, damit dessen *next*-Zeiger verändert werden kann. Dies erfordert bei der einfach verketteten Liste einen weiteren Durchlauf, während die doppelt verkettete Liste es erlaubt, das Vorgängerelement über den *prev*-Zeiger zu referenzieren.

```

1 || class ListenElement
2 || {
3 || | private:
4 || | | ListenElement* next;
5 || | | ListenElement* prev;
6 || | | int data;
7 || };

```

Der *prev*-Zeiger des ersten Elements ist, wie der *next*-Zeiger des letzten Elements, ein *nullptr*. So lassen sich beide Enden der Liste terminieren.

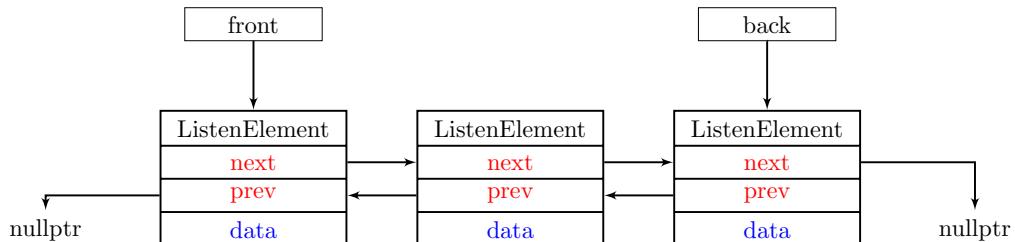


Abbildung 5.2: Schema einer doppelt verketteten Liste.

5.2.3 Warteschlange

Die Warteschlange (englisch: *queue*) ist ein so genannter FIFO-Speicher. FIFO steht für „*First in, first out*“ und besagt, dass die in der Warteschlange abgelegten Datenelemente in der gleichen Reihenfolge ausgelesen werden, in der sie gespeichert worden sind. Häufige Anwendungen von Warteschlangen sind Pufferspeicher bei der Datenübertragung. Abbildung 5.3 stellt eine solche Warteschlange dar und verdeutlicht, wie am linken Ende neue Elemente hinzugefügt und am rechten Ende bereits gespeicherte Elemente entnommen werden. Dazu bieten Warteschlangen die beiden Grundoperationen *pushBack* zum hinten Hinzufügen und *popFront* zum vorne Entfernen von Elementen. Die Implementierung erfolgt als verkettete Liste. Üblicherweise wird zusätzlich eine Funktion zur Verfügung gestellt, die anzeigt, ob die Datenstruktur leer ist, und eine Funktion, die die Daten des ersten Elements ausliest.

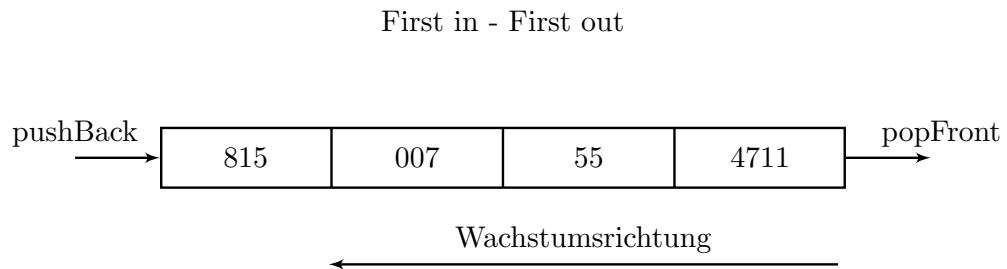


Abbildung 5.3: Schema einer Warteschlange/Queue/Puffer.

5.2.4 Stapelspeicher

Im Gegensatz zur Warteschlange ist der Stapelspeicher (englisch: *stack*) eine LIFO-Datenstruktur („*Last in, first out*“). Dessen Elemente werden in der umgekehrten Reihenfolge entnommen, in der sie der Datenstruktur hinzugefügt wurden. Ein Stapelspeicher wird typischerweise als ein Turm von Datenelementen visualisiert (siehe Abbildung 5.4), auf dem lediglich auf das oberste Element etwas abgelegt oder dieses heruntergenommen werden kann. Die Operationen zum Hinzufügen und Entnehmen werden als *push* und *pop* bezeichnet und operieren nur auf dem jeweils obersten Element. Auch hier bietet sich die Implementierung als verkettete Liste an: Die Operation *push* fügt ein neues Element hinzu, indem sie ein neues Listenelement erzeugt und dieses vorne an die Liste anhängt. Mit *pop* wird das vorderste Element wieder entnommen.

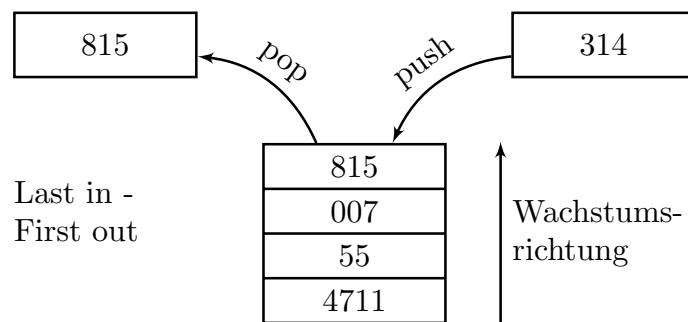


Abbildung 5.4: Schema eines Stapelspeichers/Stacks.

5.3 Aufgaben

5.3.1 Einfache Studentendatenbank

Gegeben ist eine einfache Studentendatenbank, die als einfach verkettete Liste implementiert ist. Sie besteht aus einem Hauptprogramm, das bereits eine textbasierte Menüsteuerung enthält, und drei

5 Dynamische Datenstrukturen

Klassen, *Liste*, *ListenElement* und *Student*. Die Liste besteht aus einer einfach verketteten Liste aus Listenelementen. Die Listenelemente enthalten die Daten des jeweiligen Studenten sowie einen Zeiger auf das nächste Listenelement. Die Klasse *Liste* hat bereits mehrere Funktionen zum Bearbeiten der Liste, *pushBack* fügt ein Listenelement am Ende hinzu, *popFront* entfernt vorne ein Listenelement, *dataFront* liest die Daten des ersten Studenten aus, *empty* prüft, ob die Liste leer ist, und *ausgabeVorwaerts* gibt die Liste von vorne nach hinten aus.

- Erstellen Sie ein neues Projekt mit dem Namen *Versuch05*.
- Importieren Sie die Codevorlagen.
- Erweitern Sie die Liste derart, dass eine doppelt verkettete Liste vorliegt. Ändern Sie dazu auch die Methoden *pushBack()* und *popFront()*.
- Fügen Sie einen weiteren Menüpunkt *Datenbank in umgekehrter Reihenfolge ausgeben* hinzu und implementieren Sie diesen.
- Als weiteren Menüpunkt implementieren Sie *Datenelement löschen*. Es soll nach der Matrikelnummer gefragt und dann der passende Eintrag dazu gelöscht werden. Ein erfolgreicher Löschvorgang soll durch die Ausgabe der Daten des Studenten signalisiert werden. Ist die Liste leer oder wurde der Student nicht gefunden, soll eine entsprechende Mitteilung ausgegeben werden.
- Erstellen Sie einen sechsten Menüpunkt *Datenelement vorne hinzufügen*, für den Sie die passende Methode implementieren.

6 Standard Template Library – STL

Dieses Kapitel stellt Ihnen die Standard Template Library vor. Im Theoriteil sollen Sie sich einen Überblick über diese Klassenbibliothek verschaffen. Im Umfang dieses Praktikums kann nicht der komplette Umfang behandelt werden. Daher sei auf die Dokumentation der STL verwiesen¹.

Weiterhin soll noch auf die Referenz über allgemeine Template-Klassen hingewiesen werden, die das Grundgerüst für die STL bilden².

6.1 Templates

Es kommt häufig vor, dass eine bestimmte Funktionalität innerhalb eines Programms immer wieder benötigt wird, wie z.B. die Überprüfung bei der Reversi-Aufgabe, ob ein Zug gültig ist. Da ist es zweckmäßig, diesen Teil der Implementierung in einer eigenen Funktion zu kapseln und bei Bedarf dann immer wieder aufzurufen. Da die Funktionen nur bei der Reversi-Aufgabe Sinn machten, waren Rückgabewert und Übergabeparameter eindeutig festgelegt.

Es gibt daneben aber noch Funktionen, die für verschiedene Datentypen als Parameter in ihrer Funktionalität gleich sind. Normalerweise müssten Sie dann für jeden Datentyp (z.B. *int* oder *double*) eine eigene Funktion schreiben. Ein Beispiel soll dies verdeutlichen. Es sollen zwei Werte mittels der Funktion *swap()* vertauscht werden.

```
1 void swap(int& x, int& y)
2 {
3     int tempwert = x;
4     x = y;
5     y = tempwert;
6 }
7 }
```

Die Werte werden hier als Referenz übergeben, eine lokale Kopie wäre sinnlos.

Wenn Sie dieselbe Funktionalität für den *double*-Datentyp brauchen, müssen Sie diese Funktion noch mal schreiben und *int* gegen *double* tauschen. Bei noch mehr Datentypen entsprechend mehr.

Templates (*Schablonen*) bieten Ihnen die Möglichkeit, sich bei der Implementierung nicht auf einen bestimmten Datentyp festlegen zu müssen. Stellen Sie sich das so ähnlich vor wie eine Schablone zur Herstellung von Kleidung. Sie können die Schablone (*Template*) auf irgendeinen Stoff (*Datentyp*) legen und diesen dann ausschneiden und erhalten immer dieselbe Form. Selbst wenn ein Kunde seinen eigenen Stoff mitbringt, funktioniert Ihre Schablone immer noch. Genau so funktionieren Templates in C++. Und damit andere ihre eigenen Datentypen benutzen können, werden Templates entsprechend allgemein formuliert.

Die obige Funktion *swap()* sieht als Template dann folgendermaßen aus.

```
1 template <class T>
2 void swap(T& x, T& y)
3 {
4     T tempwert = x;
5     x = y;
6     y = tempwert;
7 }
8
9
10 int main()
```

¹<http://www.cplusplus.com/reference/stl/>

²http://en.cppreference.com/w/cpp/language/class_template

```

11 || {
12 |     int a = 5;
13 |     int b = 7;
14 |     swap<int>(a, b);
15 |     double c = 10.5;
16 |     double d = 0.5;
17 |     swap<double>(c, d);
18 |     return 0;
19 }

```

Das Schlüsselwort *template* teilt dem Compiler mit, um was für eine Art von Funktion es sich hier handelt und das zuerst das Vorkommen aller T's durch einen konkreten Datentyp ersetzt werden muß. Der Aufruf in der *main()*-Funktion erfolgt dann mit der entsprechenden Typangabe nach dem Funktionsnamen und vor der Parameterliste. Durch diese allgemeine Formulierung können dem Template auch eigene Klassen als Datentyp übergeben werden, z.B. die Klasse *Vektor* aus Aufgabe 4 oder die Klasse *Student* aus Aufgabe 5, auch die würden dann vertauscht.

Auf die gleiche Art können Sie auch selbst *template*-Klassen erstellen, die zur Zeit der Implementierung auf keinen bestimmten Datentyp festgelegt sind.

Eine Besonderheit der STL ist, dass alle ihre Klassen und Funktionen als Templates implementiert sind. So verbirgt sich hinter dem *vector*-Container beispielsweise die folgende *Template*-Definition:

```

1 | template<class T> class std::vector
2 | {
3 | //Membervariablen und Memberfunktionen
4 | };
5 |
6 | int main()
7 | {
8 |     // vector mittels Template für den Datentyp double
9 |     std::vector<double> myvector;
10 |
11 |     // weiter im Programm
12 |     return 0;
13 }

```

6.2 Standard Template Library – STL

Effiziente Datenstrukturen und Algorithmen sind in der Regel schwierig zu implementieren. Viele Programme verwenden (meist aus Unkenntnis) schlechte, langsame und specheraufwendige Lösungen, obwohl in der Forschung oder in der Praxis längst viel bessere und effizientere Implementierungen bekannt sind. Um diese Lücke zu schließen, verwenden moderne Programmiersprachen sogenannte *Container*-Klassen, die moderne und effiziente Implementierungen bieten. Der Anwender kann einfach seine zu verwaltenden Objekte in den entsprechenden *Container* füllen und sich nur auf die eigentlichen Ziele seines Programms konzentrieren. Die *STL – Standard Template Library* ist ein spezieller Teil der C++-Standardbibliothek, die verschiedene *Container*-Klassen und Algorithmen zur Verfügung stellt.

Zur STL gehören:

- **Container:** In diesen Containern werden die Daten verwaltet. Sie sind ähnlich strukturiert wie die Ihnen bisher bekannten Datenstrukturen, aber die Daten darin können dynamisch mitwachsen. Es gibt verschiedene Typen von Containern, die jeweils für bestimmte Zwecke optimiert sind, bzw. effiziente Implementierungen häufig verwendeter Datenstrukturen darstellen.

Container	
<vector>	eindimensionales Feld
<list>	doppelt-verkettete Liste
<deque>	»double ended« Queue
<queue>	Queue
<stack>	Stack
<map>	Assoziatives Feld
<set>	Menge
<bitset>	Feld von Booleschen Werten

- **Iteratoren:** Eine typische Anwendung bei den meisten Containerarten besteht darin, Element für Element durch den Container zu iterieren und zu navigieren. Dies wird üblicherweise durch Definition einer zur Containerart passenden Iteratorklasse erreicht. Iteratoren sind den Zeigern verwandt und dienen dem Zugriff auf die Elemente, die in Containern abgespeichert sind.
- **Algorithmen (Funktionen):** In der *STL* stehen eine Reihe von Algorithmen (Funktionen) zur Verfügung, die mit Hilfe von Iteratoren auf den Elementen der verschiedenen Container operieren können. Sie stellen eine allgemeine Erweiterung der Funktionalität der Container-Klasse dar.
- Es gibt noch eine Reihe von **Hilfsklassen und -funktionen** wie Funktionsobjekte, Adapter, etc., die momentan aber nicht von Interesse sind.

6.2.1 Die Klasse *vector*

Nachfolgend wird als Beispiel für einen Standardcontainer die Klasse *vector* beschrieben. Sofern nicht anders angegeben, gelten die hier gemachten Aussagen für jeden Standardcontainer. Allerdings ist der *STL*-Container *vector* den Ihnen bekannten Arrays sehr ähnlich. Im Unterschied zu einem gewöhnlichen Array kann ein *vector* allerdings dynamisch wachsen und stellt Ihnen zusätzliche Funktionen zur Bearbeitung der im Container enthalten Elemente zur Verfügung.

1. Zuerst muss die Headerdatei *vector* inkludiert werden, in der das *vector*-Template deklariert ist.

```
1 || #include <vector>
```

2. Danach wird ein *vector*-Container beispielweise für string-Daten erzeugt.

```
1 || std::vector<std::string> gartenCenter;
```

3. Es wird ein passender Iterator benötigt, um auf die Daten im Container zugreifen zu können. Es gibt auch einen Iterator, der in umgekehrter Reihenfolge auf den Container zugreift. Dieser Iterator heißt *reverse_iterator*.

```
1 || std::vector<std::string>::iterator it;
```

Wenn nur Daten aus dem Container gelesen werden sollen, aber nicht verändert werden, empfiehlt es sich, einen konstanten Iterator zu verwenden, der mit *const_iterator* bezeichnet wird. Auch hierzu gibt es einen Iterator, der den Container in umgekehrter Reihenfolge durchläuft (*const_reverse_iterator*).

4. Jede Containerart besitzt eigene, spezifische Memberfunktionen. Der Container *vector* bietet z.B. die foldenden Memberfunktionen, um ein Element am Ende hinzuzufügen bzw. zu löschen:

```
1 || push_back(element); // hängt ein neues Element am Ende an
2 || pop_back();          // löscht das letzte Element
```

Eine ausführliche Anleitung zur Benutzung der *C++*-Referenz finden Sie im folgenden Kapitel unter *Nutzung der C++ Referenz*. Eine gedruckte Version wäre sicherlich schwer zu tragen. Das folgende Beispiel soll Ihnen dabei helfen, einen Überblick über die Vorgehensweise der Containererzeugung zu bekommen.

```
1 || #include <iostream>
2 || #include <vector>
3 || #include <string>
4
5 int main()
6 {
7     std::vector<std::string> gartenCenter;
8     std::vector<std::string>::iterator it;
9     std::vector<std::string>::reverse_iterator revIt;
10
11    gartenCenter.push_back("Rosen");
12    gartenCenter.push_back("Heckenpflanzen");
13    gartenCenter.push_back("Bambus");
14    gartenCenter.push_back("Blütenstauden");
```

6 Standard Template Library – STL

```
15     gartenCenter.push_back("Rhododendron");
16     gartenCenter.push_back("Nadelgehölze");
17
18     //Ausgeben
19     for (it = gartenCenter.begin(); it != gartenCenter.end(); it++)
20     {
21         std::cout << *it << std::endl;
22     }
23
24     for (revIt = gartenCenter.rbegin(); revIt != gartenCenter.rend(); revIt++)
25     {
26         std::cout << *revIt << std::endl;
27     }
28
29     // Die Anzahl der Elemente im Container wird zurückgegeben.
30     std::cout << gartenCenter.size() << std::endl;
31
32     return 0;
33 }
```

Dieses Beispielprogramm produziert folgende Ausgabe:

```
Rosen
Heckenpflanzen
Bambus
Bluetenstauden
Rhododendron
Nadelgehoelze
Nadelgehoelze
Rhododendron
Bluetenstauden
Bambus
Heckenpflanzen
Rosen
6
```

Schauen Sie nun das folgende Beispielprogramm an. Die Funktionalität ist die gleiche wie beim vorherigen Programm, aber die Implementierung unterscheidet sich deutlich.

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4
5 using namespace std;
6
7 int main()
8 {
9     vector<string> gartenCenter;
10
11     gartenCenter.push_back("Rosen");
12     gartenCenter.push_back("Heckenpflanzen");
13     gartenCenter.push_back("Bambus");
14     gartenCenter.push_back("Bluetenstauden");
15     gartenCenter.push_back("Rhododendron");
16     gartenCenter.push_back("Nadelgehoelze");
17
18     //Ausgeben
19     for (int i = 0; i < gartenCenter.size(); i++)
20     {
21         cout << gartenCenter[i] << endl;
22     }
23
24     for (auto revIt = gartenCenter.rbegin(); revIt != gartenCenter.rend(); revIt++)
25     {
26         cout << *revIt << endl;
```

```

27 }
28
29 for (string str : gartenCenter)
30 {
31     cout << str << endl;
32 }
33
34 // Die Anzahl der Elemente im Container wird zurückgegeben.
35 cout << gartenCenter.size() << endl;
36
37 return 0;
38 }
```

Dieses Programm benutzt gleich mehrere, weitere Möglichkeiten, die C++ bietet.

- *using namespace std;* bewirkt, dass der Compiler bei der Suche nach Funktionen oder Variablen den Namensraum *std* mit einbezieht. *cout* z.B. ist ein Objekt vom Typ *ostream*, das dort definiert ist. Auch die Klasse *string* ist dort definiert. Daher brauchen Sie den Namensraum nicht mehr explizit anzugeben. Dies kann jedoch auch zu Konflikten führen, sobald Sie selber eine Funktion schreiben, die zufälligerweise denselben Namen hat wie eine in *std*. Dann weiß der Compiler nicht, welche gemeint ist.
- In Zeile 21 wird kein Iterator benutzt, um auf die Elemente in *vector* zuzugreifen, sondern ein Index wie bei einem Feld. Dies ist möglich, da die Klasse *vector* die einzelnen Elemente hintereinander wie in einem Feld ablegt. Daher ist der Zugriff wie bei einem Feld möglich. Bei allen anderen Containern ist diese Zugriffsart nicht definiert. Außerdem findet keine Bereichsüberprüfung statt.
- *auto* bewirkt, dass der Compiler selbst bestimmt, welchen Typ eine Variable hat. Hier ist *revIt* vom Type *reverse_iterator*. Obwohl Sie *auto* fast überall benutzen können, ist es ratsam, dies nur an Stellen zu machen, an denen es auch für Außenstehende sofort ersichtlich ist, um welchen Datentyp es sich handelt.
- Zeile 29 stellt eine weitere Möglichkeit dar, eine *for* Schleife zu benutzen, die allerdings etwas anders funktioniert. Die Variable *str* ist vom selben Typ wie die Elemente in *vector*, also kein Iterator. Der Variablen werden nacheinander alle Elemente von *vector* zugewiesen. Es handelt sich dabei um eine Kopie, der Wert in *vector* kann nicht geändert oder gelöscht werden. Außerdem funktioniert diese Art der *for* Schleife nur vorwärts, nicht rückwärts. Die Funktionalität ist also eingeschränkt. Diese *for* Schleife kann bei allen Containern verwendet werden.

Gerade bei der Verwendung von Iteratoren zum Durchlaufen eines Containers, hier *vector*, kann *auto* die Übersichtlichkeit erhöhen, weil dadurch der Kopf der Schleife kürzer wird. Es ist üblich, Iteratoren in der *for* Schleife selbst zu definieren, nicht davor. Das vermeidet eine gewisse Fehleranfälligkeit, wenn Sie nicht daran denken, dass der Iterator bereits existiert, und ihn erneut definieren. Wird er in der Schleife definiert, wird er am Ende der Schleife wieder gelöscht.

```

|| for (std::vector<std::string>::iterator it = gartenCenter.begin(); it != gartenCenter.
||         end(); it++)
|| for (auto it = gartenCenter.begin(); it != gartenCenter.end(); it++)
```

6.3 Einfache Datei Ein- und Ausgabe

Sobald Sie ein Programm beenden, gehen alle Werte in Ihren Variablen verloren. Bei der Studentenliste aus Aufgabe 5 gehen alle geänderten Angaben verloren und Sie müssen diese Änderungen erneut eingeben. Um dies zu vermeiden, stellt C++ die Klassen der *stream*-Bibliotheken zur Verfügung, mit deren Hilfe z.B. in Dateien geschrieben werden kann.

6.3.1 Die *stream*-Klassen

In C++ ist ganz allgemein die Ein- und Ausgabe mittels *streams* implementiert. Abstrakt gesehen ist ein Stream eine Sequenz von *Bytes*, auf die seriell zugriffen werden kann, lesend und schreibend. *Streams* können fast beliebig groß sein. Es gibt auch die Möglichkeit des wahlfreien Zugriffs auf *streams*, indem ein Zeiger innerhalb des *streams* positioniert werden kann, um an einer bestimmten Position zu lesen oder zu schreiben, darauf soll hier aber nicht näher eingegangen werden. Es wird zwischen sogenannten *input-streams*, z.B. Tastatur, Datei oder Netzwerk, und *output-streams*, z.B. Monitor, Drucker, Netzwerk oder Datei, unterschieden. Manche *streams* können sowohl Eingabe- als auch Ausgabestreams sein, z.B. Dateien oder Netzwerke. Das schöne an *streams* ist, dass Sie nur eine Möglichkeit kennen müssen, um Daten mit unterschiedlichen Geräten einzulesen oder auszugeben. Zwei dieser *stream*-Objekte haben Sie bereits kennengelernt, *std::cin* und *std::cout*, die durch das Einbinden der Bibliothek *iostream* automatisch angelegt werden. *std::cin* ist dabei normalerweise mit der Tastatur verbunden, alle Eingaben werden dabei in *std::cin* gespeichert und bei Bedarf durch den Operator » eingelesen. Genauso verhält es sich bei *std::cout*, nur werden hier Ihre Daten auf dem Monitor ausgegeben. Um mit Dateien zu arbeiten gibt es die Bibliothek *fstream*, hier werden aber nicht automatisch Objekte angelegt. Sie müssen Objekte dieser Klasse selbst erstellen und anschließend mit einer Datei verbinden. Durch

```
|| #include <fstream>
```

wird die entsprechende Bibliothek eingebunden, analog zu *<iostream>*.

6.3.2 Daten in eine Datei schreiben

Sie können durch die *stream*-Bibliotheken Daten genauso in eine Datei schreiben wie auf den Monitor. Ein kleines Beispiel soll das verdeutlichen:

```
1 #include <iostream>
2 #include <fstream>
3
4 int main()
5 {
6     // Die Daten eines Studierenden
7     int matNr = 12345;
8     std::string name = "Eva Mustermann";
9     std::string geburtsdatum = "14.3.1998";
10
11    // Erstellen eines Objekts der Klasse ofstream
12    std::ofstream ausgabe;
13
14    // Öffnen einer Datei, in die geschrieben werden soll
15    ausgabe.open("studenten.txt");
16
17    // Daten in die Datei schreiben
18    ausgabe << matNr << std::endl;
19    ausgabe << name << std::endl;
20    ausgabe << geburtsdatum << std::endl;
21
22    // Schließen der Datei
23    ausgabe.close();
24
25    return 0;
26 }
```

Nachdem die Bibliothek *fstream* eingebunden wurde, werden einige Variablen erstellt, die dann in die Datei *ausgabe* geschrieben werden sollen. In Zeile 12 wird ein Objekt vom Typ *ofstream* mit dem Namen *ausgabe* erzeugt und in Zeile 15 mit der Datei *studenten.txt* verbunden. Der Name der Datei ist frei wählbar. Wenn die Datei noch nicht existiert, wird sie erzeugt. Wenn sie existiert, wird ihr Inhalt überschrieben. *ofstream* ist eine Klasse, die von der Klasse *ostream* abgeleitet ist. In dieser Klasse ist auch der Ausgabeoperator *<<* für alle Standarddatentypen definiert, so wie Sie es auch von *cout* kennen. Jetzt werden die Werte der Variablen in die Datei geschrieben. Das *std::endl* ist dabei

wichtig, damit die einzelnen Werte durch ein Trennzeichen unterscheidbar sind. Ohne ein solches Trennzeichen stünden die Werte direkt hintereinander, also *12345Eva Mustermann14.3.1998*. Dies würde beim Einlesen Probleme bereiten. *std::endl* fügt einen Zeilenvorschub ein (*linefeed*, '\n'). Der Zeilenvorschub gehört zur Gruppe der sogenannten *Whitespaces*, zu der auch das Freizeichen ' ' oder der Tabulator '\t' gehören. Dazu später noch mehr beim Einlesen aus Dateien. Zuletzt wird die Datei in Zeile 23 geschlossen. Dies geschieht auch automatisch, sobald der Gültigkeitsbereich verlassen wird. Die Datei hat jetzt folgenden Inhalt:

```
1 || 12345
2 || Eva Mustermann
3 || 14.3.1998
```

6.3.3 Daten aus einer Datei lesen

Das Lesen aus Dateien geschieht ähnlich dem Schreiben, allerdings gilt es hier, einige Besonderheiten zu beachten. Hier ein Beispiel zum Lesen aus einer Datei:

```
1 || #include <iostream>
2 || #include <fstream>
3 |
4 || int main()
5 || {
6 ||     // Die Daten eines Studierenden
7 ||     int matNr = 0;
8 ||     std::string vorname = "";
9 ||     std::string nachname = "";
10 ||    std::string geburtsdatum = "";
11 |
12 ||    // Erstellen eines Objekts der Klasse ifstream
13 ||    // Kurzform, die Datei wird beim Erstellen des Objekts direkt geöffnet
14 ||    std::ifstream eingabe("studenten.txt");
15 |
16 ||    // Daten aus der Datei lesen
17 ||    eingabe >> matNr;
18 ||    eingabe >> vorname;
19 ||    eingabe >> nachname;
20 ||    eingabe >> geburtsdatum;
21 |
22 ||    // Ausgabe der Daten auf dem Bildschirm
23 ||    std::cout << matNr << std::endl;
24 ||    std::cout << vorname << ' ' << Nachname << std::endl;
25 ||    std::cout << geburtsdatum << std::endl;
26 |
27 ||    // Datei wird automatisch geschlossen
28 |
29 ||    return 0;
30 || }
```

Es werden wieder einige Variablen erstellt, die die Werte eingelesener Daten aufnehmen. In Zeile 14 wird ein Objekt vom Typ *ifstream* erzeugt und die Datei, erster Parameter als String, geöffnet. Beim Einlesen der Werte mit dem Eingabeoperator >> gilt es zu beachten, dass aus dem Stream *eingabe* nur solange gelesen wird, bis ein *Whitespace* gefunden wird, also ein '\n', ein ' ' oder ein '\t'. Es gibt noch mehr Zeichen, die zur Gruppe der *Whitespaces* gehören, hier aber im Moment nicht interessieren (siehe hierzu die Referenz). Das gefundene *Whitespace* verbleibt aber im Stream. 12345 wird gelesen und in *matNr* gespeichert. 12345 ist nicht mehr in *eingabe*. Das erste Zeichen in *eingabe* ist nun '\n'. Solange Sie den Eingabeoperator >> benutzen, ist dies kein Problem, da führende *Whitespaces* überlesen und aus dem Stream entfernt werden. Beim nächsten Lesen wird dann aber nur *Eva* eingelesen, da danach ein ' ' folgt, also ein *Whitespace*. Hier ist also die Aufteilung in Vorname und Nachname notwendig. Würde dies nicht beachtet, also wie beim Schreiben nur in *name* eingelesen, dann landet der Nachname in der Variablen *geburtsdatum*.

Dieses Vorgehen funktioniert also nur, solange die einzulesenden Daten immer dasselbe Format haben. Wenn Sie die Datei von Anderen bekommen und schon vorher ein Programm zur Auswertung schreiben wollen, Sie aber nur Folgendes wissen:

- Die Eingabedatei enthält keine Leerzeilen.
- Jeder Datensatz besteht aus 3 Zeilen.
 - Matrikelnummer als Integer
 - Name als String
 - Geburtstag als String

Nehmen Sie an, in der Datei *studenten.txt* befände sich ein weiterer Studierender mit dem Namen *Jan David Berg*, also eine Person mit zwei Vornamen. Dann käme das Programm durcheinander, da jetzt der Nachname als Geburtstag eingelesen wird. Anschließend würde versucht, den Geburtstag in ein Integer umzuwandeln. Das gibt einen Fehler.

In der STL gibt es jedoch eine Funktion, die es ermöglicht, ganze Zeilen einzulesen, *std::getline(istream& is, string& str, char delim)*. Der erste Parameter ist eine Referenz auf den Eingabestream, der zweite Parameter ist eine Referenz auf den String, in den die Zeile eingelesen werden soll, der dritte Parameter ist das Begrenzungszeichen. Wenn kein Begrenzungszeichen angegeben wird, ist es '\n', also *newline*. Im Gegensatz zum Eingabeoperator *>>* entfernt *std::getline()* auch das Begrenzungszeichen am Ende. Eine Besonderheit bei *std::getline()* ist, das führende *Whitespaces* nicht ignoriert und verworfen werden. Beginnt eine Zeile mit einem Leerzeichen, so bleibt dieses im Ausgabestring erhalten. Ist das erste Zeichen ein '\n', wird dies als leere Zeile interpretiert. Der übergebene String ist dann leer und erst dann würde '\n' entfernt. Dies ist durchaus sinnvoll, wenn auch leere Zeilen beachtet werden sollen, etwa wenn ein Artikel eingelesen wird, der ja durchaus auch Leerzeilen enthalten kann. Leerzeilen würden dann durch zwei oder mehr aufeinander folgende '\n' repräsentiert.

Das Programm zum Einlesen der Studierendendatei sieht dann so aus:

```

1 #include <iostream>
2 #include <fstream>
3
4 int main()
5 {
6     // Die Daten eines Studierenden
7     int matNr = 0;
8     std::string name = "";
9     std::string geburtsdatum = "";
10
11    // Erstellen eines Objekts der Klasse ifstream
12    std::ifstream eingabe("studenten.txt");
13
14    // Daten aus der Datei lesen
15    eingabe >> matNr;
16    eingabe.ignore(1, '\n');
17    std::getline(eingabe, name);
18    // Alternativ std::getline(eingabe, geburtsdatum);
19    eingabe >> geburtsdatum;
20
21    // Ausgabe der Daten auf dem Bildschirm
22    std::cout << matNr << std::endl;
23    std::cout << name << std::endl;
24    std::cout << geburtsdatum << std::endl;
25
26    return 0;
27 }
```

In Zeile 16 wird die Klassenfunktion *ignore()* verwendet, um das führende *Whitespace* '\n', das nach dem Auslesen der Matrikelnummer noch im Stream verblieben ist, auszulesen und wegzuerwerfen. In Zeile 19 kann alternativ auch *std::getline()* verwendet werden, damit das abschließende '\n' entfernt wird, falls anschließend noch ein weiterer Datensatz eingelesen werden soll, z.B. eine Adresse, die selbst wieder Leerzeichen enthalten kann.// Die Datei wird zum Programmende automatisch geschlossen.

6.3.4 Die Datei-Flags

Beim Öffnen einer Datei kann noch genauer spezifiziert werden, in welchem Modus das passieren soll. Dies geschieht mit Hilfe sogenannter Datei-Flags, die in der Basisklasse *ios* definiert sind. Basisklassen

und Vererbung werden in Kapitel 8 näher erläutert. Es gibt folgende Datei-Flags:

```
ios::in Datei wird zum Lesen geöffnet, Standard bei ifstream
ios::out Datei wird zum Schreiben geöffnet, Standard bei ofstream
ios::app Daten werden an das Ende der Datei geschrieben
ios::ate Internen Zeiger an das Ende der Datei positionieren
ios::trunc Der Dateinhalt wird gelöscht
ios::binary Die Datei wird als Binärdatei geöffnet, z.B. eine Bilddatei
```

Es gibt noch eine Reihe weiterer Flags, auf die hier nicht weiter eingegangen wird. Näheres können Sie der Referenz entnehmen. Eine *ofstream* wird standardmäßig nur mit dem Flag `ios::out` geöffnet, d.h. der alte Inhalt der Datei wird überschrieben. Wenn die alten Daten ergänzt werden sollen, müssen Sie zusätzlich das Flag `ios::app` verwenden. Sobald Sie jedoch mehr Flags als den Standard benutzen wollen, müssen Sie alle Flags explizit angeben. Die Flags werden als zweiter Parameter an den Konstruktor übergeben. Mehrere Flags werden mittels des | Operator verbunden.

Der Befehl zum Öffnen einer Datei zum Schreiben und Anhängen sieht dann so aus:

```
|| std::ofstream ausgabe("studenten.txt", ios::out | ios::app);
```

6.3.5 Fehler abfangen

Beim Umgang mit Dateien kann es zu Fehlern kommen, die, wenn möglich, behandelt werden. Bei dem Versuch, eine Datei zu öffnen, können Fehler auftreten, weil die Datei nicht existiert, ein externer Dateispeicher im Moment nicht zur Verfügung steht oder weil Sie nicht das Recht haben, eine Datei zu lesen oder zu verändern.

Die *fstream* Objekte besitzen vier sogenannte Zustandsbits, die den Status der Streams signalisieren und über Memberfunktionen abgefragt werden können. Diese Funktionen sind `good()`, `eof()`, `fail()` und `bad()`. Die gesetzten Bits bedeuten:

- `good` signalisiert, dass der Stream in Ordnung ist.
- `eof` signalisiert, dass das Ende des Stream erreicht wurde.
- `fail` signalisiert, dass beim Lesen ein Fehler aufgetreten ist, der Stream selbst aber noch in Ordnung ist. Dies kann passieren, wenn z.B. ein Integer eingelesen werden soll, aber Buchstaben in dem Wert sind.
- `bad` signalisiert, dass der Stream grundsätzlich nicht mehr in Ordnung ist oder das Werte verlorengegangen sind. Der Stream kann nicht mehr verwendet werden.

Die Bits werden gesetzt, sobald einer der Zustände eingetreten ist, ihr Wert ist dann `true`. Nach dem Öffnen einer Datei kann damit z.B. überprüft werden, ob dies erfolgreich war.

```
1 || std::ofstream ausgabe("studenten.txt", ios::out | ios::app);
2 || if (!ausgabe)
3 || {
4 ||     std::cout << "Fehler beim Öffnen der Datei!";
5 ||     exit(1);
6 || }
```

In Zeile 2 wird nur überprüft, ob die Datei geöffnet wurde. Falls nicht, wird eine Fehlermeldung ausgegeben und das Programm beendet.

Beim Lesen aus einer Datei soll z.B. solange gelesen werden, bis das Dateiende erreicht ist. Sind keine weiteren Daten in der Datei, wird das `eof`-Bit gesetzt. Dies geschieht aber erst, wenn der Versuch, Daten zu lesen, gescheitert ist. Ein Beispiel:

```
1 || #include <iostream>
2 || #include <fstream>
3 |
4 || int main()
```

```

5  {
6      // Die Daten eines Studierenden
7      int matNr = 0;
8      std::string name = "";
9      std::string geburtsdatum = "";
10
11     // Erstellen eines Objekts der Klasse ifstream
12     std::ifstream eingabe("studenten.txt");
13     if (!eingabe)
14     {
15         std::cout << "Fehler beim Öffnen der Datei!";
16         exit(1);
17     }
18
19     // Daten aus der Datei lesen
20     eingabe >> matNr;
21
22     while (!eingabe.eof())
23     {
24         eingabe.ignore(1, '\n');
25         std::getline(eingabe, name);
26         std::getline(eingabe, geburtsdatum);
27
28         // Ausgabe der Daten auf dem Bildschirm
29         std::cout << matNr << std::endl;
30         std::cout << name << std::endl;
31         std::cout << geburtsdatum << std::endl;
32
33         // Einlesen des nächsten Datensatzes, um Dateiende zu überprüfen
34         eingabe >> matNr;
35     }
36
37     std::cout << "Es sind keine weiteren Daten vorhanden." std::endl;
38
39     return 0;
40 }
```

6.4 Aufgaben

In den folgenden Aufgaben soll die doppelt verkettete Liste durch die Klasse *vector* der STL ersetzt werden. Des weiteren sollen die Daten der Studierenden aus einer Datei eingelesen werden und nach der Bearbeitung auch wieder in einer Datei gespeichert werden können. Für die Bearbeitung der Aufgaben soll das Projekt aus Versuch 5 verwendet werden. Erstellen Sie dazu ein neues Projekt *Versuch06*. Markieren Sie die Dateien *main*, *Student.cpp/.h*, die anderen Dateien werden nicht mehr benötigt. Ziehen Sie die Dateien nicht mit der Maus in *Versuch06*, denn dann werden sie in *Versuch05* gelöscht. Drücken Sie die rechte Maustaste, dann *Copy*. Markieren Sie *Versuch06*, rechte Maustaste, dann *Paste*. Vergewissern Sie sich, dass die Dateien in beiden Projekten vorhanden sind.

6.4.1 Nutzung der C++-Referenz

Für die Lösung der nachfolgenden Aufgaben ist es sehr hilfreich, sich mit dem Umgang der C++-Referenz vertraut zu machen. Einige Beispiele sollen Ihnen dies vermitteln.

1. Öffnen Sie in einem Browser die Seite www.cplusplus.com. Sie befinden sich auf der Startseite der C++-Referenz. Hier finden Sie z.B. Tutorials, eine Einführung in C++ oder ein Forum.
2. Sie suchen Informationen zur Klasse *vector*. Tippen Sie unten rechts, nicht oben auf der Seite, *vector* in das Suchfeld und klicken Sie auf *search*. Sie sind jetzt auf der Hilfeseite zu *vector*.
3. Als erstes sehen Sie die Deklaration der Klasse *vector*. Es folgt eine Beschreibung der Klasse, eine Darstellung der internen Funktionsweise und Besonderheiten. Der erste Satz teilt Ihnen mit, dass es sich um einen Container handelt, der die Werte wie ein Feld speichert, sich aber zur Laufzeit in der Größe ändern kann (was bei normalen Feldern nicht geht).

4. Scrollen Sie jetzt etwas runter. Lassen Sie sich nicht dadurch verwirren, dass Sie noch nicht alles verstehen, was Ihnen an Information angeboten wird, z.B. der Abschnitt *Container properties* oder *Member types*, das ist normal.
5. Scrollen Sie so weit nach unten, bis der Abschnitt *Member functions* auftaucht. Dieser Teil ist der eigentlich Interessante, da hier die Funktionen beschrieben werden, die Ihnen diese Klasse zur Verfügung stellt. Sie können auf jede dieser Funktionen klicken und erhalten dann nähere Informationen zur Benutzung, meistens mit Beispielen.
6. Klicken Sie auf die erste Funktion, den Konstruktor (*constructor*), Sie wollen ja wissen, wie Sie ein Objekt dieser Klasse erstellen können. Achten Sie bei den Beschreibungen darauf, das der Reiter *C++11* ausgewählt ist.
7. Sie sehen jetzt alle Konstruktoren, die die Klasse zur Verfügung stellt. Für Sie sind im Moment erst mal nur die ersten drei von Interesse. Der erste Konstruktor ist der *Default*-Konstruktor, also derjenige, der benutzt wird, wenn Sie ein Objekt ohne Parameterübergabe anlegen. Der Konstruktor hat hier zwar einen Übergabeparameter, aber dieser hat einen Defaultwert, der benutzt wird, wenn kein Parameter übergeben wird. Dasselbe gilt für die anderen Konstruktoren. Beim zweiten Konstruktor können Sie die Anfangsgröße festlegen, und beim Dritten sowohl Anfangsgröße als auch Anfangswert.
8. Am besten scrollen Sie einfach zu dem Beispiel runter, dann sehen Sie direkt, wie man Konstruktoren benutzen kann. Dabei müssen Sie nicht alle Beispiele verstehen. Es reicht völlig, sich zunächst mit den gängigen Arten zu beschäftigen. Das Schöne an vielen Beispielen in dieser Referenz ist, dass Sie die Beispiele direkt selbst ausprobieren können, ohne sie in eine eigene IDE kopieren zu müssen. Wenn Sie oben rechts neben dem Beispiel ein kleines Zahnrad sehen, können Sie darauf klicken und es öffnet sich eine Seite mit dem Quellcode des Beispiels. Diesen Code können Sie direkt compilieren und laufen lassen, indem Sie zuerst links unten *C++11* anwählen und dann rechts auf *Run* klicken. Sie können den Quellcode auch beliebig verändern, um auszuprobieren, wie sich das Verhalten ändert.

Diese Seite funktioniert unabhängig von der Referenz und Sie können sie immer benutzen, wenn Sie mal schnell etwas testen wollen (URL: www.cpp.sh). Außerdem können Sie sich eine *URL*-Adresse Ihrer Seite geben lassen, über die Sie Ihr Beispiel jederzeit wieder aufrufen können, wenn Sie daran weiterarbeiten wollen.

9. Kehren Sie auf die Hauptseite von *vector* zurück und schauen Sie sich die restlichen Funktionen an. Die Namen sind bereits beschreibend genug, damit man eine Vorstellung hat, was die Funktion tut. Im Gegensatz zu einer Liste verfügt die Klasse *vector* nicht über eine Funktion, mit der Sie direkt am Anfang ein Element einfügen können. Dazu müssen Sie die Funktion *insert* nutzen. Diese erwartet als ersten Parameter einen konstanten Iterator und als zweiten Parameter den Wert, der eingefügt werden soll. Dies sähe dann z.B. so aus

```
|| meinvector.insert(meinvector.begin(), meinWert);
```

10. Wichtig sind auch die Iteratoren, mit denen Sie Ihren Vektor durchlaufen können, z.B. *begin()*. Wie Sie gerade gesehen haben, erhalten Sie einen Iterator auf das erste Element. Mit *end()* erhalten Sie einen Iterator, der hinter das letzte Element Ihres Vektors zeigt. Dadurch können Sie ihn dazu benutzen, um in einer Schleife eine Endbedingung festzulegen. Sehen Sie sich dazu das kleine Beispiel in der Referenz an.
11. Sie können auf die Elemente Ihres Vektors entweder mit den eckigen Klammern `[]`, wie bei einem Feld, zugreifen, oder über die Funktion *at()*. Der Unterschied ist, das bei `[]` keine Bereichsüberprüfung stattfindet, Sie also selbst für den Index verantwortlich sind. Bei *at()* findet diese Überprüfung statt, was allerdings deutlich langsamer ist als der Zugriff über `[]`. Sind Sie außerhalb des Bereichs, gibt es zur Laufzeit eine Fehlermeldung (*exception*), die vom Programm dann entsprechend behandelt werden kann. Darauf wird in diesem Praktikum aber nicht näher eingegangen.
12. Die Funktion *erase()* erlaubt es Ihnen, innerhalb Ihres Vektors ein beliebiges Element oder einen Bereich von Elementen zu löschen. Dadurch müssen aber alle Elemente dahinter entsprechend nach vorne verschoben werden, was recht aufwendig ist. Außerdem verlieren alle Iteratoren, die

auf Elemente zeigen, die hinter dem gelöschten Element liegen, ihre Gültigkeit, besonders der `end()`-Iterator. Daher müssen Sie, sofern Sie in einer Schleife durch Ihren Vektor laufen, diesen Durchlauf beenden (`break;`), da ja alle weiteren Elemente verschoben wurden, auch das letzte.

13. Durch die bereitgestellten Beispiele haben Sie gesehen, wie Sie auf einen Vektor zugreifen können. Die restlichen Funktionen können Sie sich bei Bedarf näher ansehen.
14. Kehren Sie jetzt auf die Hauptseite der C++-Referenz zurück, indem Sie oben links auf das Logo klicken. In der STL gibt es nicht nur Klassen, sondern auch viele hilfreiche, fertige Funktionen. Ein Beispiel dafür ist die Funktion `find()`. Tippen Sie in das Suchfeld der Hauptseite `find` ein. Beachten Sie hierbei, dass Sie für die Suche in der C++-Referenz weder die Funktionsparameter noch die Klammern eintippen müssen, sondern nur den Funktionsnamen, hier `find`. Es öffnet sich die entsprechende Seite. Bereits in der Deklaration zu Beginn der Seite können Sie erkennen, dass `find()` als Parameter zwei Iteratoren, die einen zusammenhängenden Bereich darstellen, erwartet, sowie den Wert, der gefunden werden soll. Der Rückgabewert ist wieder ein Iterator, der auf das Element verweist, sofern es gefunden wurde. Wenn nicht, wird der zweite Parameter zurückgegeben. Daher sollte er so gewählt sein, das er hinter das letzte Element verweist.
15. An dem Beispiel können Sie erkennen, dass dies auch bei Feldern funktioniert. `myints+4` entspricht dabei dem ersten Element hinter dem Feld. Dieser Iterator fungiert aber nur als Abbruchbedingung. Es wird nicht auf diese Stelle im Speicher zugegriffen, das gewährleistet die Funktion. In dem Kasten etwas oberhalb des Beispiels ist das Verhalten nochmal erklärt.
16. Die Funktion `find()` benutzt die Operatorfunktion `operator==()`. Diese Funktion ist für die Standarddatentypen implementiert. Damit das auch mit selbstdefinierten Datentypen (z.B. Klassen wie `student`) möglich ist, muss der Vergleichsoperator entsprechend implementiert (*überladen*) sein. Wie dies funktioniert, erfahren Sie im folgenden Kapitel.
17. Wenn Sie bei `find()` auf die linke Seite schauen, sehen Sie im oberen Kasten, dass die Funktion `find()` in der Bibliothek `algorithm` zu finden ist. In dem Kasten darunter sehen Sie all die Funktionen, die in `algorithm` auch noch enthalten sind, z.B. `sort()`, die für den nächsten Versuch noch ganz interessant sein kann.

Die Punkte 14 bis 17 beziehen sich auf Aufgabe 7, sollen aber hier im Rahmen der Einführung schon erwähnt werden, damit Sie wissen, wo Sie Informationen finden können. Die Funktion `find()` können Sie in Aufgabe 6 noch nicht benutzen.

Die Benutzung der C++-Referenz ist äußerst hilfreich und nicht schwierig. Sie finden dort vieles, das Sie direkt für die Implementierung Ihrer eigenen Anwendungen nutzen können. Durch sinnvolle Funktionsnamen wird das Auffinden von Informationen deutlich erleichtert. Beherzigen Sie das für Ihre eigenen Funktionen, andere werden es Ihnen danken.

6.4.2 STL

Ersetzen Sie in der Funktion `main.cpp` die Funktionalität mit Funktionen, die die STL bereits zur Verfügung stellt. Dabei spielt die Reihenfolge der Implementierungen keine Rolle, Sie können also zunächst die bereits vorhandenen Funktionen anpassen oder zuerst das Einlesen der Daten aus einer Datei implementieren, das ist Ihnen überlassen.

1. Benutzen Sie die `push` und `pop` Funktionen der Klasse `vector`, um Elemente am Ende hinzuzufügen oder zu löschen.
2. Benutzen Sie die Funktionen `erase()` und `insert()` der Klasse `vector`, um Elemente an anderen Stellen zu löschen oder einzufügen.
3. Benutzen Sie einen `iterator` bzw. einen `reverse_iterator`, um alle Elemente der Liste auszugeben.
4. Fügen Sie einen weiteren Menüpunkt *Datenelement vorne loeschen* ein und implementieren Sie diese Funktion mit der entsprechenden Ausgabe.
5. Erweitern Sie das Menü um den Punkt *Daten aus einer Datei einlesen* und implementieren Sie diese Funktionalität. Kommentieren Sie dazu die zu Beginn der `main`-Funktion vorhandene Routine zum Einlesen der Daten der Studierenden samt der Abfrage zum automatischen Füllen aus.

Die Daten werden ab jetzt ausschließlich aus einer Datei eingelesen. Fragen Sie nach dem Namen der Datei, aus der die Daten eingelesen werden sollen.

Durch diese Funktionalität wäre es auch möglich, bereits eingelesene Daten erneut einzulesen. Daher ist zu bedenken, was mit bereits eingelesenen Daten passieren soll. Sie können einfach alle bisherigen Daten löschen, dies ist am Einfachsten. Die Klasse *vector* stellt hierzu die Funktion *clear()* zur Verfügung. Sie könnten die einzulesenden Daten aber auch mit den bereits vorhandenen Daten vergleichen (z.B. anhand der Matrikelnummer) und nur die übernehmen, die noch nicht vorhanden sind. Oder Sie könnten den Benutzer fragen, was passieren soll, wenn schon Daten vorhanden sind. Sie sehen, es ist genau zu überlegen, wie Sie hier vorgehen wollen.

Hier soll es reichen, wenn Sie die Daten in *vector* einfach löschen, bevor Sie Daten einlesen. In den Vorlagen finden Sie eine Textdatei, *studierende.txt*, die Sie entweder in Ihr Projekt importieren oder in Ihren *workspace* kopieren können.

6. Fügen Sie einen weiteren Menüpunkt *Daten in eine Datei sichern* ein und implementieren Sie diesen. Fragen Sie, in welche Datei die Daten gesichert werden sollen. Ist die Datei bereits vorhanden, soll sie überschrieben werden.
7. Achten Sie bei allen Funktionsaufrufen darauf, dass die Liste nicht leer ist. Dies kann sonst zu einem nicht definierten Verhalten führen, meistens zu einem Programmabsturz.

Hinweis: Da es sich bei den Ausgabefunktionen um konstante Methoden handelt, müssen Sie konstante Iteratoren verwenden (siehe Seite 83).

Nach Abschluss des Versuches soll Ihr Programm dazu in der Lage sein, neue Studierende dynamisch in den Vektor aufzunehmen und an beliebiger Stelle wieder löschen zu können. Außerdem soll es alle Studierende im Vektor ausgeben können, sowohl von Anfang bis Ende, als auch in umgekehrter Reihenfolge. Die Daten der Studierenden sollen aus einer Datei eingelesen, als auch in einer Datei gesichert werden können.

7 Überladung

7.1 Theorie

Normalerweise darf eine Funktion oder eine Methode nur eine einzige Definition haben, damit der Compiler entscheiden kann, welchen Code er compilieren soll. Nun wird diese Einschränkung etwas aufgeweicht. Unter C++ ist es möglich, unter gewissen Randbedingungen mehrere Funktionen oder Methoden (z.B. Überladen der Konstruktoren im Kapitel 4.2.2) mit gleichem Namen zu definieren. Diese 'Mehrfach-Definitionen' werden auch als Überladen bezeichnet. Überladen ist nicht nur für Funktionen, sondern auch für Operatoren möglich, denn auch Operatoren sind in C++ als Funktionen implementiert.

7.1.1 Überladen von Funktionen

Funktionen und Methoden lassen sich in C++ überladen. Das bedeutet, dass der gleiche Name für unterschiedliche Implementierungen verwendet werden kann. Wenn mehrere Funktionen den gleichen Namen haben, so müssen sich diese irgendwie unterscheiden, damit beim Aufruf der Funktion ersichtlich wird, welche Funktion gemeint ist. Dazu müssen sich die Signaturen der Funktionen in mindestens einem der folgenden Punkte unterscheiden:

- Die Funktionen besitzen eine unterschiedliche Anzahl von Parametern.

```
|| void p (int i) { ... }
  || void p (int i, int j) { ... }
```

- Die Datentypen der Parameter sind unterschiedlich.

```
|| void p (int i) { ... }
  || void p (float j) { ... }
```

Zu beachten ist, dass der Rückgabewert einer Funktion kein Entscheidungskriterium darstellt. Der Compiler meldet einen Fehler, wenn sich zwei Funktionen nur in ihrem Rückgabewert unterscheiden.

7.1.2 Überladen von Operatoren

Per Überladung kann eine Großzahl der C++-Operatoren an die Verwendung mit Operanden von selbst definierten Daten- bzw. Klassentypen angepasst werden. Man hat beispielsweise eine Klasse *Student* zur Identifizierung eines Studenten implementiert. In diesem Fall kann man z.B. die Vergleichsoperatoren (<, >, == ...) für Operanden vom Klassentyp überladen, um Objekte der Klasse einfach mit Hilfe der Operatoren vergleichen zu können.

Folgende Überlegungen spielen bei der Überladung eines Operators eine Rolle:

- Soll die Operatorfunktion in der Klasse definiert werden?
- Oder soll sie im globalen Dateibereich außerhalb der Klasse definiert werden?

Bei Überladung innerhalb der Klasse ist der erste Operand automatisch das aktuelle Objekt der Klasse, für das die Operatorfunktion aufgerufen wird.

```
1 /* Überladung unärer Operatoren im Klassenbereich */
2 Rückgabewert Klassenname::operator <op>()
3 // <op> ∈ !, ++, --, ...
4 {
5 }
```

7 Überladung

```
6  /* Überladung binärer Operatoren im Klassenbereich */
7  Rückgabetyp Klassenname::operator <op>(Typ operand)
8      // <op> ∈ +, -, <, >, ==, ...
9  {
10 }
11 }
```

Aus

```
|| objektA == objektB;
```

wird intern der Funktionsaufruf

```
|| objektA.operator==(objektB);
```

objektA dient hier als Objekt einer Klasse und liefert damit den *this*-Zeiger (siehe Kapitel 4.2.3). Damit dieser Funktionsaufruf funktioniert, muß die Funktion *operator==()* innerhalb der Klasse, von der *objektA* ein Objekt ist, implementiert sein. Dies gilt für fast alle Operatoren.

Das folgende Beispiel zeigt die Definition und Implementierung einer Klasse *Student*, für die der Vergleichsoperator *==* überladen wurde, um zwei Objekte anhand der Matrikelnummer auf Gleichheit prüfen zu können.

```
1 class Student
2 {
3     public:
4         Student(int pMatNr, const std::string& pName, const std::string& pVorname);
5         bool operator==(const Student& student);
6         int getMatNr();
7
8     private:
9         int matNr;
10        std::string name;
11        std::string vorname;
12    };
1
1 Student::Student(int pMatNr, const std::string& pName, const std::string& pVorname) :
2     matNr(pMatNr), name(pName), vorname(pVorname)
3 {
4 }
5
6 bool Student::operator==(const Student& student)
7 {
8     if (matNr == student.matNr)
9         return true;
10    else
11        return false;
12 }
13
14 int Student::getMatNr()
15 {
16     return matNr;
17 }
```

Bei Überladung außerhalb der Klasse werden alle benötigten Operanden als Parameter übergeben.

```
1 /* Überladung unärer Operatoren */
2 Rückgabetyp operator <op>(Klassentyp operand)
3     // <op> ∈ !, ++, --, ...
4 {
5 }
6
7 /* Überladung binärer Operatoren */
8 Rückgabetyp operator <op>(Typ operand1, Typ operand2)
9     // <op> ∈ +, -, <, >, ==, ...
10
11 }
```

Der Vergleichsoperator *==* wird wie folgt außerhalb der Klasse *Student* überladen:

```

1 || bool operator==(Student& student1, Student& student2)
2 || {
3 || // direkter Zugriff auf die Membervariablen nicht möglich
4 || if ( student1.getMatNr() == student2.getMatNr() )
5 ||     return true;
6 || else
7 ||     return false;
8 || }

```

Der Vergleich von zwei Instanzen kann schließlich folgendermaßen durchgeführt werden:

```

1 | int main()
2 | {
3 |     Student peter(222222, "Lustig", "Peter");
4 |     Student max(223344, "Muster", "Max");
5 |     if (peter == max)
6 |     {
7 |         std::cout << "Max und Peter sind gleich" << std::endl;
8 |     }
9 |     else
10 |    {
11 |        std::cout << "Max und Peter sind nicht gleich" << std::endl;
12 |    }
13 | }

```

Wenn es keinen besonderen Grund gibt, soll die Operatordefinition möglichst innerhalb einer Klasse stattfinden. Manchmal ist eine Definition innerhalb der Klasse nicht möglich, wenn z.B. der erste Operand des Operators ein Klassenobjekt einer Klasse darstellt, auf deren Implementierung man keinen Zugriff hat. Dazu gehören die Objekte `std::cout` der Klasse `ostream` und `std::cin` der Klasse `istream` in Verbindung mit den Eingabe- und Ausgabeoperatoren » und « (siehe 7.1.4).

7.1.3 Algorithmen der STL

Für sich betrachtet ist ein Container nicht besonders interessant. Um wirklich sinnvoll zu sein, muss ein Container Basisoperationen wie das Abfragen der Größe, das Iterieren, Kopieren, Sortieren und Suchen nach Elementen unterstützen. Die Standardbibliothek bietet glücklicherweise Algorithmen und Funktionen, um die fundamentalen und allgemeingültigen Anforderungen von Anwendern an Container zu erfüllen. Sie befinden sich alle im Namensbereich `std` und werden in `<algorithm>` bzw. in `<functional>` deklariert.

```

|| #include <algorithm>
|| #include <functional>

```

Für eine vollständige Übersicht sei auch hier wieder auf die Dokumentation verwiesen.

Sie haben bisher in diesem Kapitel gelernt, die Vergleichsoperatoren zu überladen. Dies ist bei fast allen nicht primitiven Typen nötig, damit die Operatoren wie erwartet funktionieren. Wurden diese korrekt überladen, sind die Algorithmen nun in der Lage, die Objekte innerhalb eines Containers zu vergleichen, und somit bestimmten Operationen zu unterziehen. Als Beispiel soll das frühere Beispiel zur Verwendung der Container um einen Sortieralgorithmus modifiziert werden. Bitte beachten Sie, dass ein `string` eine Klasse darstellt und die gängigen Operatoren bereits überladen wurden, siehe hierzu auch 7.1.2.

```

1 | #include <iostream>
2 | #include <vector>
3 | #include <string>
4 | #include <algorithm>
5 |
6 | int main()
7 | {
8 |     vector<string> gartenCenter;
9 |
10 |     gartenCenter.push_back("Rosen");
11 |     gartenCenter.push_back("Heckenpflanzen");
12 |     gartenCenter.push_back("Bambus");

```

7 Überladung

```
13 || gartenCenter.push_back("Bluetenstauden");
14 || gartenCenter.push_back("Rhododendron");
15 || gartenCenter.push_back("Nadelgehoelze");
16
17 // Die Pflanzen werden lexikographisch aufsteigend sortiert
18 std::sort(gartenCenter.begin(), gartenCenter.end());
19
20 //Ausgeben
21 for (auto it = gartenCenter.begin(); it != gartenCenter.end(); it++)
22 {
23     std::cout << *it << std::endl;
24 }
25
26 for (auto revIt = gartenCenter.rbegin(); revIt != gartenCenter.rend(); revIt
27    ++)
28 {
29     std::cout << *revIt << std::endl;
30 }
31
32 }||
```

Dieses Beispielprogramm produziert folgende Ausgabe:

```
Bambus
Bluetenstauden
Heckenpflanzen
Nadelgehoelze
Rhododendron
Rosen
Rosen
Rhododendron
Nadelgehoelze
Heckenpflanzen
Bluetenstauden
Bambus
```

7.1.4 Überladung der Streamoperatoren

Die Eingabe- und Ausgabeoperatoren werden oft auch *Streamoperatoren* genannt. Sehr häufig überlädt man die Streamoperatoren `>>` und `<<`, um die Daten von Objekten selbst definierter Klassen auf bequeme Weise einlesen und ausgeben zu können.

Die Besonderheit dieser Überladung liegt darin, dass der linke Operand (erster Parameter der Operatorfunktion) ein Objekt der Klasse *ostream* für `std::cout` bzw. ein Objekt der Klasse *istream* für `std::cin` darstellt, die beide von der Klasse *iostream* abgeleitet sind. Dies bedeutet, dass die Operatorüberladung, wenn sie innerhalb einer Klasse implementiert werden soll, in der Klasse *ostreams* bzw. *istream* erfolgen müsste. Das geht aber nicht, weil der Quellcode zu dieser Bibliothek nicht zur Verfügung steht.

Wie schon beim Vergleichoperator `==` wird aus

```
|| std::cout << Zahl;           // siehe folgendes Beispiel
```

intern der Funktionsaufruf

```
|| std::cout.operator<<(Zahl);
```

Auch hier fügt der Compiler wieder den *this*-Zeiger vom Typ *ostream* als ersten Parameter ein. Das bedeutet aber, das es in der Klasse *ostream* eine Operatorfunktion

```
|| std::ostream& operator<<(Zahl);
```

geben müsste. Das wird nicht der Fall sein, denn die Klasse *Zahl* wurde ja gerade erst implementiert. Wenn der Compiler keine entsprechende Funktion innerhalb der Klasse findet, durchsucht er den globalen Bereich.

Die Überladung muss daher im globalen Bereich erfolgen. Da die Funktion außerhalb des Klassenbereichs definiert ist (zu erkennen daran, dass sie keinen *scope*-Operator vor dem Namen hat), hat sie auch keinen Zugriff auf die *private*- und *protected*- Datenelemente. Der zweite Übergabeparameter ist jedoch ein Objekt der Klasse und wird benutzt, um eine Ausgabefunktion aufzurufen, die innerhalb der Klasse implementiert ist und somit Zugriff auf alle Member hat.

Sei als Beispiel das folgende Programm mit der Klasse *Zahl* gegeben:

```

1 #include <iostream>
2
3 class Zahl
4 {
5 public:
6     Zahl(int pWert);
7     void ausgabe(std::ostream& out);
8 private:
9     int wert;
10};
11
12 Zahl::Zahl(int pWert)
13 {
14     this->wert = pWert;
15 }
16
17 // Klassenfunktion fuer die Ausgabe
18 void Zahl::ausgabe(std::ostream& out)
19 {
20     out << wert;           //
21 }
22
23 // globale Funktion fuer operator<<()
24 std::ostream& operator<<(std::ostream& out, Zahl& zahl)
25 {
26     zahl.ausgabe(out);      // Aufruf der Klassenfunktion ausgabe()
27                                         // out ist eine Referenz fuer std::cout
28     return out;
29 }
30
31 int main()
32 {
33     Zahl zahl1 = Zahl(42);
34     Zahl zahl2 = Zahl(21);
35
36     std::cout << zahl1 << std::endl;
37
38     std::cout << zahl1 << " " << zahl2 << std::endl;
39
40     return 0;
41 }
```

So produziert dieses eine Ausgabe von:

```
42
42 21
```

Offensichtlich kapselt hier die Klasse *Zahl* nur sehr simpel einen Integer. Interessant ist allerdings, dass mit der Überladung des *<<*-Operators nun diese Klasse direkt in *std :: cout* geschrieben werden kann und eine definierte Ausgabe produziert. Dies funktioniert dadurch, dass der Operator *<<* für *std::ostream* als linken Operand und *Zahl* als rechten Operand global überladen wurde. Das eigentliche Schreiben in *ostream* übernimmt allerdings die Klasse selber in ihrer Ausgabe-Funktion. Diese bekommt, wie im Code zu sehen, einen *ostream* übergeben (im Anwendungsfall das Objekt *cout*) und kann nun in diesen *ostream* schreiben (im Beispiel die private Variable *wert*).

7.2 Aufgaben

7.2.1 Operatorenüberladung

Zur Lösung dieses Versuchs soll der Code aus Versuch 6 erweitert werden. Erstellen Sie ein neues Projekt *Versuch07* und kopieren Sie den Code von Versuch 6 in dieses.

Um weitere Funktionalitäten der STL nutzen zu können, muss Ihre *Student*-Klasse verschiedene Objekte vergleichen können. Hierzu müssen die entsprechenden Operatoren überladen werden.

1. Überladen Sie die Operatoren `<`, `>`, `==` innerhalb Ihrer *Student*-Klasse. Diese sollen die Studenten anhand ihrer Matrikelnummern vergleichen.
2. Ändern Sie die Ausgabe-Funktion der *Student*-Klasse. Diese bekommt einen Parameter vom Typ `ostream&`, in den die Ausgabe mit dem `<<`-Operator geschrieben wird.
3. Überladen Sie den Ausgabe Operator `<<` außerhalb Ihrer *Student*-Klasse, damit einzelne Objekte direkt ausgegeben werden können.

Hinweis: Beachten Sie, dass innerhalb der STL-Algorithmen die Templates als `const` definiert sind. Daher müssen alle Funktionen, die von der STL genutzt werden, auch als `const` definiert sein. Dies betrifft insbesondere die Überladung von Operatoren.

7.2.2 STL - Funktionalität

Mit obigen Änderungen ist die STL nun in der Lage, Studenten zu vergleichen und es können weitere Funktionalitäten genutzt werden.

1. Erweitern Sie ihr Programm um eine Sortierfunktion. Verwenden Sie dabei `std::sort` der STL. Geben Sie die Liste nach dem Sortieren aus. Erweitern Sie Ihre Menüsteuerung entsprechend.
2. Suchen Sie im Menüpunkt 5 *Datenelement loeschen* einen Studenten mit der Funktion `find()`, geben Sie seine Daten aus und löschen Sie ihn anschließend. Geben Sie die restlichen Elemente wieder aus.

8 Vererbung und Polymorphie

Das Thema der Vererbung und Polymorphie ist äußerst komplex und kann daher im Rahmen dieses Praktikums nur angerissen werden. Weiterführende Literatur und Quellen finden Sie im Literaturverzeichnis: [1, 2, 3, 4]

8.1 Vererbung — Hierarchien im Klassenkonzept

Bisher haben Sie Klassen nur als Gruppierung bzw. Klassifizierung von Objekten verwendet. In der objektorientierten Programmierung ist es aber mindestens genauso wichtig, Beziehungen zwischen Klassen herstellen zu können. Wenn mehrere Klassen sich sehr ähnlich (oder im groben gleich) verhalten, sich aber im Detail unterscheiden, dann ist es sinnvoll, die Gemeinsamkeiten in einer eigenen Klasse zusammenzufassen. Es entsteht eine Oberklasse – Unterkasse Beziehung, die durch die sogenannte **Vererbung** realisiert wird.

Sehen Sie sich dazu einmal das in Abbildung 8.1 dargestellte Diagramm an. Die **Oberklasse** (auch **Basisklasse** genannt) *AllgemeineKlasse* besitzt ein Attribut und eine Methode. Die **Unterkasse** *SpezielleKlasse* **erbt** nun diese beiden Member von ihrer Oberklasse. Das heißt, sie stehen auch in Objekten der Klasse *SpezielleKlasse* zur Verfügung, obwohl sie nicht neu implementiert werden müssen und auch im UML-Diagramm kein zweites Mal auftauchen. Die Unterkasse fügt jetzt selbst noch ein Attribut und eine Methode hinzu, sie ist also spezieller als ihre Oberklasse. Die Vererbung wird im UML-Diagramm durch einen speziellen Pfeil ausgedrückt[3].

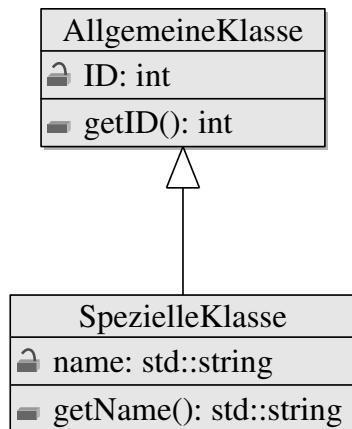


Abbildung 8.1: UML-Diagramm zur Beziehung zwischen Ober- und Unterkasse

Das folgende Programm, das mit den Angestellten eines Unternehmens arbeitet, soll dies verdeutlichen. Es definiert eine Klasse *Employee*, die allgemein einen Angestellten des Unternehmens repräsentiert. Zu jedem Angestellten des Unternehmens soll sowohl der Vorname als auch der Nachname und das Datum der Einstellung¹ gespeichert werden. Folgender Codeausschnitt zeigt die Definition der Klasse *Employee*:

```
1 || class Employee
2 || {
3 ||     public:
```

¹ Date ist eine erfundene Klasse, die ein Datum speichern kann. Sie wird hier nicht implementiert.

```

4     Employee(std::string, std::string, Date&);
5     void print() const;
6
7 private:
8     std::string firstName;
9     std::string familyName;
10    Date hiringDate;
11 };
12
13 Employee::print() const
14 {
15     std::cout << firstName << " "
16             << familyName
17             << " has been hired on "
18             << hiringDate.tostr()
19             << std::endl;
20 }
```

Listing 8.1: Definition der Klasse *Employee*

Das Programm soll nun weiter zwischen *Manager* und *Worker* unterscheiden. Da sowohl *Manager* als auch *Worker* Angestellte, also *Employees* des Unternehmens sind, soll das Modell diese Beziehung repräsentieren. Hierzu wird das Konzept der Vererbung von Klassen verwendet. *Manager* und *Worker* sind Spezialisierungen der Klasse *Employee*. Die Beziehung der Vererbung wird durch einen Doppelpunkt gekennzeichnet.

```
|| class <KlassenName>:<Zugriffsspezifizierer> <BasisklassenName>
```

Die obige Syntax der Vererbung zeigt an, dass eine Klasse alle Elemente einer anderen Klasse (ihrer Basisklasse) erbt. Durch die Zugriffsspezifizierer kann festgelegt werden, ob die Zugriffsrechte der geerbten Elemente unverändert übernommen (*public*) oder nach außen beschneidet (*protected* und *private*) werden sollen. Die wichtigste und häufigste Form der Vererbung ist die *public*-Vererbung von einer Basisklasse. Sie werden in diesem Praktikum ausschließlich diese Form verwenden. Wenn man keinen Zugriffsspezifizierer angibt, werden alle Elemente als *private* vererbt.

Folgender Codeausschnitt zeigt die Definition der Klasse *Manager*, die *public* von *Employee* erbt. Zusätzlich zu den Daten eines *Employees* wird zu einem Manager eine Liste von Mitarbeitern gespeichert, die für den Manager arbeiten. Außerdem wird ein Level gespeichert, das die Stufe in der Hierarchie repräsentiert. Die Klasse *Worker* erbt öffentlich von *Employee* und erhält zunächst einen Konstruktor, aber keine neuen Daten. Abbildung 8.2 gibt einen Überblick über die drei Klassen, ihre Member sowie ihre Beziehung untereinander. Der Pfeil von *Manager* nach *Worker* stellt dabei eine Assoziation zwischen diesen beiden Klassen dar. Dies hat mit Vererbung nichts zu tun und soll nur die Unternehmensstruktur darstellen.

```

1 class Manager : public Employee
2 {
3     public:
4         Manager(std::string, std::string, Date&);
5         void addWorker(Worker*);
6         void setLevel(int);
7
8     private:
9         std::list<Worker*> group;
10        int level;
11 }
```

Listing 8.2: Die Klasse *Manager*

```

1 class Worker : public Employee
2 {
3     public:
4         Worker(std::string, std::string, Date&);
5 }
```

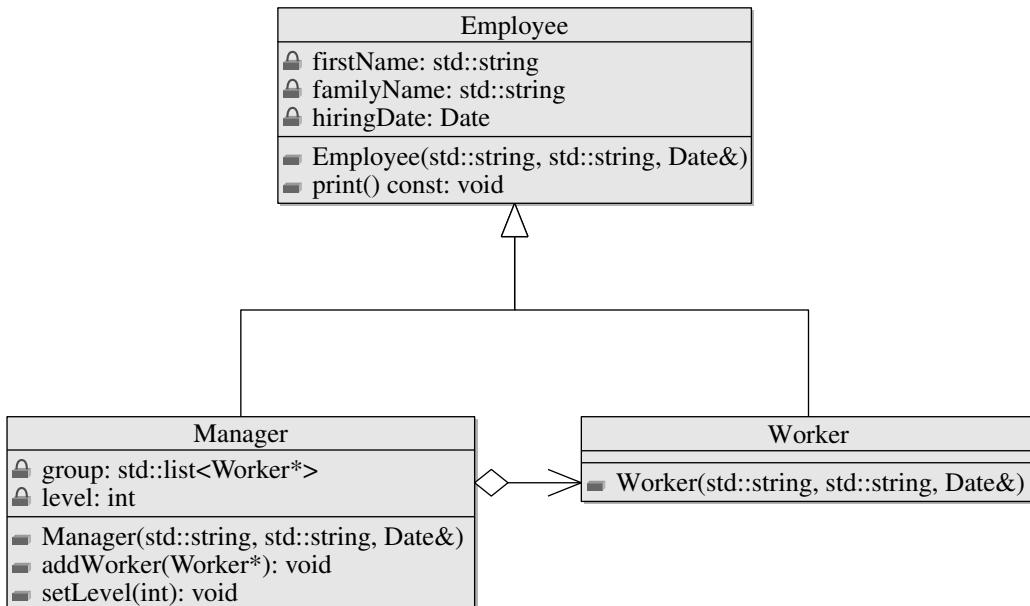
Listing 8.3: Die Klasse *Worker*

```

1 int main()
2 {
3     Date heute(1,1,1970);
4     Worker fritz("Fritz", "Outgoer", heute);
5     Manager klaus("Klaus", "Leiter", heute);
6     klaus.setLevel(2);
7     klaus.addWorker(&fritz);
8
9     klaus.print();
10    fritz.print();
11 }

```

Listing 8.4: Aufruf von vererbten Methoden

Abbildung 8.2: UML-Diagramm zu *Employee*, *Manager* und *Worker*.

Nun können Sie schon das erste Programm mit den Klassen schreiben. Das Listing 8.4 zeigt, wie zunächst eine Instanz der Klasse `Worker` und danach eine Instanz der Klasse `Manager` erzeugt wird. Anschließend wird für den `Manager` `klaus` ein Level gesetzt und `Worker` `fritz` wird `Manager` `klaus` zugewiesen. Schließlich wird die Methode `print` von beiden Instanzen aufgerufen. Dies ist möglich, da beide Klassen diese Methode von der Basisklasse erben. Es ist hier also nicht nötig, die `print`-Methode in den abgeleiteten Klassen noch einmal zu implementieren.

8.2 Polymorphismus

In diesem Abschnitt wird ein weiteres Konzept der objektorientierten Programmierung, die *Polymorphie*, eingeführt. Polymorphie bedeutet im Kontext der Objektorientierung, dass sich verschiedene Objekte beim Aufruf derselben Methode unterschiedlich verhalten können.

Betrachten Sie erneut das Beispiel aus dem letzten Abschnitt. Die abgeleitete Klasse `Manager` besitzt mehr Attribute als ihre Oberklasse `Employee`. Die Ausgabemethode `print()`, die sie von der Oberklasse erbt, kann diese jedoch nicht ausgeben, da die Attribute in der Basisklasse nicht bekannt sind. Das folgende Beispiel zeigt, wie die Ausgabefunktion in der Unterkategorie `Manager` angepasst werden kann, um auch auszugeben, auf welchem Level der Unternehmenshierarchie er sich befindet und wie groß die Gruppe ist, für die er verantwortlich ist.

```

1 class Manager : public Employee
2 {
3     public:
4         /* ... */
5         void print() const
6
7     private:
8         std::list<Worker*> group;
9         int level;
10    };
11
12 void Manager::print() const
13 {
14     Employee::print();
15     std::cout << "--> Manager at level " << level
16             << " with " << group.size() << " Worker(s)"
17             << std::endl;
18 }

```

Listing 8.5: Die Ausgabefunktion wird in der abgeleiteten Klassen angepasst

Die Funktion *print()* ruft zuerst die Funktion *print()* der Klasse *Employee* auf und gibt danach die spezifischen Daten der Klasse *Manager* aus. Wenn man nun Objekte der beiden Klassen erstellt, kann man mit einem Aufruf der Funktion *print()* die Daten ausgeben.

Damit wird ein *Polymorphismus* erzeugt, da sich zwei Objekte beim Aufruf derselben Funktion unterschiedlich verhalten.

8.2.1 Späte Bindung

Häufig ist zur Übersetzungszeit eines Programms noch gar nicht klar, welchen Typ ein Objekt später haben wird.

Alle Angestellten des Unternehmens sollen in einer Liste gespeichert werden. Man wählt also eine Liste, die Zeiger auf *Employees* speichert und initialisiert sie mit den bereits bekannten Angestellten.

```

1 int main()
2 {
3     std::list<Employee*> register;
4
5     Date heute(1,1,1970);
6     Worker *workerFritz
7         = new Worker("Fritz", "Outgoer", heute);
8     Manager *managerKlaus
9         = new Manager("Klaus", "Leiter", heute);
10    managerKlaus->setLevel(2);
11    managerKlaus->addWorker(workerFritz);
12
13    register.push_back(workerFritz);
14    register.push_back(managerKlaus);
15
16    return 0;
17 }

```

Listing 8.6: In diesem Programm wird eine Liste von Angestellten erstellt.

Es handelt sich also um eine Liste, die Elemente von Typ *Employee** erwartet, aber einen *Worker** und einen *Manager** speichert. Es ist tatsächlich immer möglich, dass Zeiger auf Objekte von Unterklassen dort abgelegt werden können, wo Zeiger auf Objekte einer Oberklasse erwartet werden.

Nun soll die soeben erstellte Liste durchlaufen und für alle Elemente die *print()*-Methode aufrufen werden. Die *main* Funktion sieht dann folgendermaßen aus:

```

1 int main()
2 {
3     /* ... */

```

```

4     std::list<Employee*>::Iterator it;
5
6     for (it = register.begin(); it != register.end(); it++)
7     {
8         (*it)->print();
9     }
10
11 }
12 }
```

Listing 8.7: Die Liste kann in einer Schleife durchlaufen werden.

Kompiliert man das Programm und führt es aus, dann sieht die Ausgabe folgendermaßen aus:

```
Fritz Outgoer has been hired on 1.01.1970
Klaus Leiter has been hired on 1.01.1970
```

Es wurde bei beiden Elementen in der Liste die Funktion *print()* der Klasse *Employee* aufgerufen, obwohl keines der Objekte vom entsprechenden Typ ist. Die Erweiterung der *print()* Methode funktioniert also nur, wenn die Variable auch mit Typ *Manager** oder *Worker** angelegt wurde.

Um möglichst dynamischen Code erstellen zu können, sollte der obige Ansatz zum erwarteten Ergebnis führen. Zur Laufzeit soll die richtige Methode gefunden und ausgeführt werden, basierend auf dem Typ des entsprechenden Objekts. Dieses Konzept nennt sich *späte Bindung* und lässt sich folgendermaßen definieren:

„Objektorientierte Systeme sind in der Regel in der Lage, die Zuordnung einer konkreten Methode zum Aufruf einer Operation erst zur Laufzeit eines Programms vorzunehmen. Dabei wird abhängig von der Klassenzugehörigkeit des Objekts, auf dem die Operation aufgerufen wird, entschieden, welche Methode verwendet wird.“

Diese Fähigkeit, eine Methode dem Aufruf einer Operation erst zur Laufzeit zuzuordnen, wird späte Bindung genannt. Dies röhrt daher, dass für die Zuordnung der Methode der spätest mögliche Zeitpunkt gewählt wird, um die Methode an den Aufruf einer Operation zu binden.“ [3]

In C++ wird diese Fähigkeit über *virtuelle Methoden* zur Verfügung gestellt. Kennzeichnet man eine Methode mit dem Schlüsselwort *virtual*, so wird sie spät gebunden. Es empfiehlt sich, Methoden in Ober- und Unterklasse einheitlich virtuell zu definieren, auch wenn dies nur in der Basisklasse zwingend notwendig ist. Dies bietet die Möglichkeit, sein Modell später einfach zu erweitern und sorgt für Übersichtlichkeit.

In UML-Diagrammen werden virtuelle Methoden kursiv dargestellt.

Die notwendige Änderung in der Klasse *Employee* ist das Deklarieren der Methode *print()* als virtuell.

```

1 class Employee
2 {
3     /* ... */
4     virtual void print() const
5     /* ... */
6 };
7
8 class Manager : public Employee
9 {
10    /* ... */
11    virtual void print() const
12    /* ... */
13 };
14
15 Manager::print() const
16 {
17     Employee::print();
18     std::cout << "--> Manager at level " << level
19     << " with " << group.size() << " Worker(s)"
```

```
20 }           << std::endl;
21 }
```

Listing 8.8: Implementierung einer virtuellen Ausgabefunktion

Wenn Sie das Programm kompilieren und ausführen, zeigt es das erwartete Verhalten mit folgender Ausgabe:

```
Fritz Outgoer has been hired on 1.01.1970
Klaus Leiter has been hired on 1.01.1970
--> Manager at level 2 with 1 Worker(s)
```

Es ist nun möglich, alle Angestellten in einer einzigen Liste des Datentyps der Basisklasse zu verwalten. Der Mechanismus der *späten Bindung* ist an vielen Stellen der OOP sehr nützlich.

8.3 Initialisierungslisten

In Kapitel 4.2.4 haben Sie bereits Initialisierungslisten kennengelernt. Wie Sie im Folgenden sehen werden, können diese Initialisierungslisten auch dazu genutzt werden, um Attribute einer Basisklasse durch Aufruf des entsprechenden Konstruktors der Basisklasse zu initialisieren. Im folgenden Beispiel wurden in Zeile 20 sowie 24f. Initialisierungslisten verwendet, um die Attribute mit den übergebenen Parametern zu initialisieren[1]. Beachten Sie, dass im Gegensatz zu Kapitel 4.2.4 die Klasse *Circle* von der Klasse *Form* abgeleitet ist und die Basisklasse *Form* nur über die Initialisierungsliste korrekt initialisiert werden kann, siehe Zeile 24 und 25.

```
1 class Form
2 {
3     public:
4         Form(unsigned int pColor);
5
6     protected:
7         unsigned int color;
8     };
9
10    class Circle : public Form
11    {
12        public:
13            Circle(const Point &pCentre, double pRadius, unsigned int pColor);
14
15        private:
16            Point centre;
17            double radius;
18    };
19
20    Form::Form(unsigned int pColor) : color(pColor)
21    {
22    }
23
24    Circle::Circle(const Point &pCentre, double pRadius, unsigned int pColor) :
25        Form(pColor), centre(pCentre.x, pCentre.y), radius(pRadius)
26    {
27    }
```

Listing 8.9: Initialisierung von Basisklassenelementen

8.4 Abstrakte Klassen

In den vorherigen Abschnitten haben Sie sich intensiv mit Vererbung und weiterführenden Techniken der objektorientierten Programmierung auseinander gesetzt.

Vererbung in der objektorientierten Programmierung hat dabei keine Verwandtschaft zur biologischen Vererbung. Die zentrale Frage in der objektorientierten Programmierung ist vielmehr „*Ist X ein Y?*“. Also zum Beispiel „*Ist ein Manager ein Employee?*“ oder auch „*Ist ein Löwe ein Säugetier?*“.[1]

Nun ist „Säugetier“ aber ein abstrakter Begriff, jedes Tier hat auch noch zusätzliche Eigenschaften, die es zu einem konkreten Lebewesen machen. Ein *reines* Säugetier lässt sich also nicht finden. Der Begriff beschreibt Eigenschaften, die nur bei konkreten Tierarten — wie Löwen — anzutreffen sind. Von diesen abstrakten Begriffen macht unsere Sprache regen Gebrauch, denn sie sind ein weiterer Schritt der Klassifizierung und helfen somit bei der Abstraktion.

Ebenso wichtig wie abstrakte Begriffe für unsere Sprache sind *abstrakte Klassen* für die objektorientierte Programmierung. Und was dieser Begriff bedeutet, haben Sie an obigem Beispiel bereits gesehen. In abstrakten Klassen werden, wie Sie es bereits kennen, Gemeinsamkeiten zusammengefasst. Allerdings wird es niemals Objekte dieser Basisklassen geben können, die nicht auch Objekte einer Unterkasse sind.

Um dieses Konzept in Zusammenhang mit der Programmierung in C++ zu bringen, ein Beispiel: Es sei ein Programm gegeben, das für zweidimensionale Grafikobjekte Klassen zur Verfügung stellt und den Flächeninhalt der Objekte berechnen kann. Dieses Programm ist in Abbildung 8.3 dargestellt.[2]

Das Beispiel enthält Klassen für Kreise, Ellipsen, Vierecke und Dreiecke. Gemeinsam ist all diesen Objekten, dass sie über Koordinaten definiert werden können (z.B. hat eine Ellipse zwei Brennpunkte und ein Dreieck drei Eckpunkte). Zusätzlich enthalten die Klassen noch für die entsprechende Form charakteristische Elemente (wie z.B. den Abstand der Brennpunkte vom Ellipsenrand).

Die gemeinsame Basisklasse *GraphObject* beinhaltet die gemeinsamen Attribute und definiert Methoden, die alle Unterklassen implementieren sollen. Die Methode *calcArea()* zum Beispiel, die den Flächeninhalt berechnet, sollte in jeder Klasse implementiert werden. Sie wird aber in jeder Unterkasse anders aussehen. Die Klasse *GraphObject* ist abstrakt, es würde keinen Sinn ergeben, von dieser Klasse Objekte zu erzeugen, oder die Methode zur Berechnung des Flächeninhalts bereits hier zu implementieren. Trotzdem beinhaltet die Basisklasse die virtuelle Methode *calcArea()*, da sie ja eine Gemeinsamkeit all ihrer Unterklassen ist. In der objektorientierten Programmierung nennt man solche Methoden, die immer auf die Implementierungen in den Unterklassen verweisen, *abstrakte Methoden*.

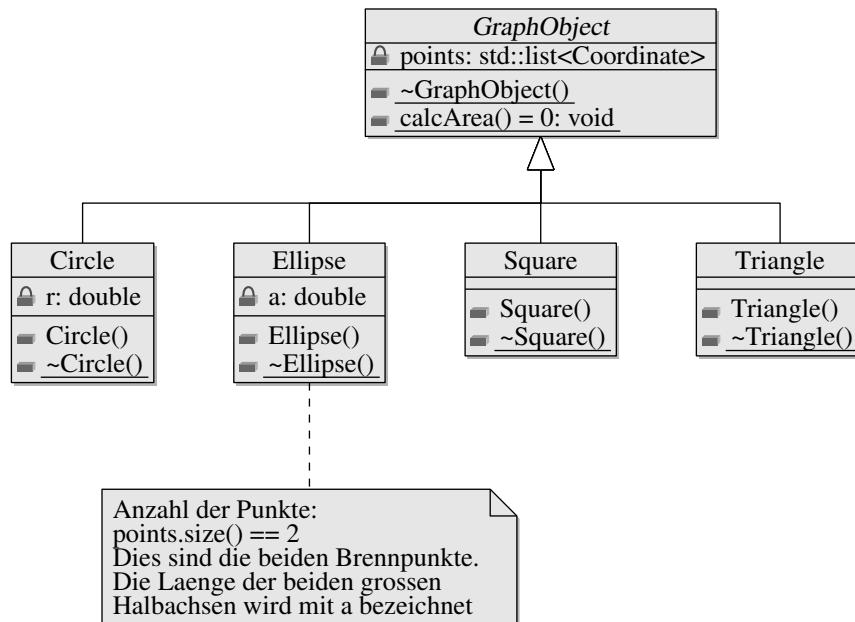


Abbildung 8.3: Ein Beispielprogramm für 2D Grafik

Von abstrakten Klassentypen gibt es keine direkten Objekte. Alle Objekte müssen hier zugleich Objekte einer Unterkasse sein. Für abstrakte Klassentypen gibt es in C++ kein eigenes Schlüsselwort.

In C++ nennt man abstrakte Methoden auch *rein virtuelle Methoden*. Diese Methoden sind ebenfalls spätgebunden und benötigen in der Klasse, in der sie als rein virtuell gekennzeichnet sind, keine Implementierung. Gekennzeichnet werden sie durch ein `=0` hinter der entsprechenden Deklaration. Im Beispiel der 2D Grafikobjekte sähe das dann folgendermaßen aus:

```

1 || class GraphObject
2 || {
3 ||     public:
4 ||         /* ... */
5 ||         virtual void calcArea() = 0;
6 ||         /* ... */
7 || };

```

Listing 8.10: Eine rein virtuelle Funktion

In C++ sind alle Klassen, die *mindestens eine* rein virtuelle Methode besitzen, automatisch abstrakt. Sie werden also nicht zusätzlich gekennzeichnet. Das bedeutet andersherum aber auch, dass der Programmierer für jede abstrakte Klasse mindestens eine Methode finden muss, die nicht implementiert wird. Was man tun kann, wenn dies nicht zum erdachten Programm passt, ist unter anderem Thema des nächsten Abschnitts.

8.5 Virtuelle Destruktoren

Es ist anzuraten, Destruktoren immer virtuell zu definieren. Hat man es nämlich mit polymorphen Strukturen zu tun, dann muss es möglich sein, den Destruktor der richtigen Unterklasse auch über einen Zeiger auf die Basisklasse zu finden. Häufig stellen Basisklassen keine echte Funktionalität bereit, weswegen die Implementierung dieses Destruktors meistens leer bleiben wird. Trotzdem ist es wichtig, ihn nicht zu vergessen und zu implementieren, denn der Compiler würde nur den normalen Destruktor hinzufügen, der aber dann nicht virtuell ist.

Destruktoren können in C++ natürlich auch rein virtuell deklariert werden. Dies ist zum Beispiel dann sinnvoll, wenn man eine abstrakte Klasse definieren möchte, die aber keine abstrakten - also in C++ virtuellen - Methoden besitzt. Allerdings *muss* auch ein rein virtueller Destruktor eine - eventuell leere - Implementierung haben.

Denken Sie auch daran, die Implementierung virtueller Destruktoren in der `.cpp` Datei vorzunehmen, obwohl dies innerhalb der Klassendefinition bequemer wäre. Ansonsten wird die Methode vom Compiler eventuell falsch behandelt [1].

8.6 Zugriffsspezifizierer `protected`

Wenn eine Klasse von einer Basisklasse abgeleitet wird, so erbt sie deren Attribute und Methoden, aber in Abhängigkeit der Zugriffsspezifizierer der Basisklasse.

Bisher haben Sie ausschließlich die Zugriffsspezifizierer *public* und *private* benutzt. Das war auch kein Problem, da bisher keine weiteren Klassen von einer Basisklasse abgeleitet wurden. Und die Klassenmethoden haben ja Zugriff auf private Methoden und Attribute. Werden diese jetzt vererbt, gibt es in der abgeleiteten Klasse keine Attribute oder Methoden, die in der Basisklasse als *private* deklariert sind. Das kann gewünscht sein.

```

1 || class Baseclass
2 || {
3 ||     public:           // nur Standardkonstruktor, muss nicht implementiert werden
4 ||         void print();
5 ||     private:
6 ||         std::string name;
7 ||         double einkommen;
8 ||     };
9 |
10 || class Subclass : public Baseclass

```

```

11 || {
12 || public:
13 ||     Subclass()      // Standardkonstruktor mit Initialisierung im Methodenrumpf
14 || {
15 ||     name = "Walter"; // Fehler, das Attribut gibt es nicht
16 ||     einkommen = 53000.0; // Fehler, das Attribut gibt es nicht
17 || }
18 || }
```

Listing 8.11: Private Vererbung

Der Zugriffsspezifizierer *protected* verhält sich in der Basisklasse selbst genau wie *private*. Von außen kann nicht auf die Member zugegriffen werden, während innerhalb der Klasse deren Methoden Zugriff haben.

Verwenden Sie in Ihrer Basisklasse aber den Zugriffsspezifizierer *protected*, dann erbt die abgeleitete Klasse diese Attribute und Methoden, und zwar wieder als *protected*.

```

1 class Baseclass
2 {
3     public:          // nur Standardkonstruktor, muss nicht implementiert werden
4         void print();
5     protected:
6         std::string vorname;
7     private:
8         double einkommen;
9     };
10
11 class Subclass : public Baseclass
12 {
13     public:
14         Subclass()      // Standardkonstruktion mit Initialisierung im Methodenrumpf
15     {
16         vorname = "Walter";      // ist okay, vorname ist protected
17         einkommen = 53000.0;    // Fehler, das Attribut gibt es nicht
18     }
19 }
```

Listing 8.12: Protected Vererbung

8.7 Aufgaben

Bedingtes Compilieren

Wenn Sie in die Vorlage der Datei *Buecherei.cpp* schauen, werden Sie im Code die zusätzlichen Anweisungen `#if`, `#else` und `#endif` sehen.

Anweisungen, die mit `#` beginnen, haben Sie bereits in den vorhergehenden Aufgaben benutzt. Dabei handelt es sich aber nicht um Anweisungen für den eigentlichen Compiler, sondern um Anweisungen an den sogenannten *Preprocessor*. Dieser *Preprocessor* ist Bestandteil eines jeden C++-Compilers. Er bereitet die von Ihnen erstellte Datei mit der Implementierung für den Compiler vor. Er erstellt die *Include-Wächter* und ersetzt die *Includes* durch den Text in den Headerdateien. Aber er fügt nur den reinen Text ein, ohne weitere Überprüfung.

Damit Sie das Programm schon zu Beginn kompilieren können, ohne dass die abgeleiteten Klassen existieren, dürfen einige Abschnitte in Ihrem Code noch nicht berücksichtigt werden. Durch die `#if` - `#else` - `#endif`-Anweisungen wird dies möglich. Diese Anweisungen verhalten sich wie die in Ihrem Code. Mit der Anweisung `#define UNTERKLASSENVORHANDEN false` wird eine Variable `UNTERKLASSENVORHANDEN` angelegt und mit `false` initialisiert. Die `#if`-Anweisung fragt diese Variable ab und entscheidet dann, welcher Codeabschnitt berücksichtigt wird. Nur dieser Teil ist dann in der Textdatei vorhanden, die dem Compiler übergeben wird, so dass dieser den anderen Code nicht sieht.

Wenn Sie dann die fehlenden Klassen implementiert haben, setzen Sie `UNTERKLASSENVORHANDEN` auf `true` und der Compiler wird diese dann berücksichtigen.

Aufgabenstellung

Familie Mustermann gehört die Bücherei Bücherwurm. Um auf lange Sicht mit der aufkommenden Konkurrenz mitzuhalten, möchte die Bücherei Bücherwurm ihr Angebot erweitern. Zusätzlich zu Büchern sollen zukünftig auch Magazine und DVDs ausgeliehen werden können. Um einen Überblick zu haben, welche Medien in der Bücherei zur Verfügung stehen, existiert bereits eine Verwaltungssoftware. Im Zug der Erweiterung des Angebots muss die Verwaltungssoftware angepasst werden.

Die Verwaltungssoftware soll in der Lage sein, verschiedene Medien in einer Datenbank zu speichern. Jedes Medium soll durch eine eindeutige ID-Nummer gekennzeichnet sein und einen Titel besitzen. Das Programm soll in der Lage sein, eine Liste aller Medien auszugeben, Medien auszuleihen und entsprechend auch zurückzugeben. Außerdem sollen neue Medien der Datenbank hinzugefügt werden können.

Für die Realisierung der Verwaltungssoftware sind Ihnen bereits einige Klassen und Funktionen vorgegeben:

- Jedes Medium in der Bücherei wird durch eine Instanz der Klasse *Medium* abgebildet. Innerhalb der Klasse *Medium* werden eine eindeutige ID sowie der Titel des Mediums gespeichert, wobei die zugehörige ID automatisch im Konstruktor durch eine statische Variable erzeugt wird (Statische Variablen und statische Funktionen werden im nächsten Kapitel 9.3, Seite 122 näher beschrieben). Darüber hinaus wird zu jedem Medium die Information gespeichert, ob das Medium ausgeliehen wurde oder nicht. Ist das Medium ausgeliehen, so wird auch das Datum der Ausleihe und die Person gespeichert, an die das Medium ausgeliehen wurde. Außerdem bietet jedes Medium die Möglichkeit, ausgeliehen zu werden und wieder an die Bücherei zurückzugeben zu werden. Hierfür sind die Funktionen *ausleihen()* und *zurueckgeben()* vorgesehen. Mit der Funktion *ausgabe()* können alle Informationen zu einem Medium ausgegeben werden. Die prinzipielle Funktionsweise dieser Klasse und ihrer Funktionen ist Ihnen bereits vorgegeben.
- Um die Informationen zu speichern, an welche Person ein Medium ausgeliehen wurde, existiert die Klasse *Person*. Hier wird der Name und das Geburtsdatum der Person gespeichert.
- Die Klasse *Datum* ist zur Bearbeitung von Datumsobjekten vorgesehen. Diese werden sowohl für das Geburtsdatum als auch für das Datum der Ausleihe benötigt. Innerhalb der Klasse *Datum* finden Sie Beispiele für die Operatorüberladung. Für genauere Erklärungen der einzelnen Funktionen und Operatoren lesen Sie die Kommentare im Quellcode bitte sorgfältig.
- In der Datei *Buecherei.cpp* finden Sie die Implementierung der Verwaltungssoftware. Die Datenbank ist in Form eines *std::vector<Medium*>* realisiert. Darüber hinaus bietet die *main*-Funktion eine bereits vorgegebene Menüsteuerung für die Verwaltungssoftware. Bevor Sie mit der Aufgabe beginnen, betrachten Sie die einzelnen Punkte der Menüsteuerung und machen Sie sich klar, welche Funktionen für die Aufgabe genutzt werden und von welchem Objekt diese aufgerufen werden.

Hinweis: Die Klassen *Person* und *Datum* sind bereits fertig implementiert, hier müssen Sie nichts anpassen. An der Klasse *Medium* werden Sie im Laufe der Aufgabe kleine Veränderungen vornehmen.

8.7.1 Erstellen der Unterklassen

Um eine Unterscheidung zwischen den verschiedenen Medien treffen zu können, die in der Bücherei zur Verfügung stehen, sollen Sie drei weitere Klassen implementieren, eine für Bücher, eine für Magazine und eine für DVDs. Da sowohl Bücher als auch Magazine und DVDs einen Titel und eine eindeutige ID haben, nutzen Sie überall die Klasse *Medium* als Basisklasse.

- Beginnen Sie mit der Klasse *Buch*. Ein Buch soll zusätzlich zum Titel auch noch einen Autor speichern. Legen Sie hierzu eine Klasse *Buch* an (*File → New → Class*)
 1. Leiten Sie die Klasse *Buch* von der Klasse *Medium* ab.
 2. Legen Sie zum Speichern des Autors eine neue Membervariable an.

3. Der Konstruktor soll dabei folgendes Format haben `Buch(std::string initTitel, std::string initAutor)`. Implementieren Sie den Konstruktor und beachten Sie, dass die Klasse *Medium* keinen Standardkonstruktor hat (siehe Kapitel 8.3).
 4. Da ein Buch neue Membervariablen besitzt, die ausgegeben werden sollen, überladen Sie die Funktion `ausgabe()`. Es sollen alle Informationen ausgegeben werden. Hierzu bestehen verschiedene Möglichkeiten, nutzen Sie entweder die *protected member* oder die Funktion `ausgabe()` der Basisklasse *Medium*. Anschließend geben Sie noch den Autoren des Buches aus. Am Ende der Aufgabenstellung finden Sie beispielhafte Ausgaben, an denen Sie sich orientieren sollen.
- Als nächstes sollen Sie die Klasse *Magazin* realisieren. Diese speichert zusätzlich zum Titel das Erscheinungsdatum der Ausgabe sowie die Sparte ab. Legen Sie die Klasse *Magazin* an. Gehen Sie analog zum Buch vor. Der Konstruktor soll folgendes Format haben `Magazin(std::string initTitel, Datum initDatumAusgabe, std::string initSparte)`. Für die Implementierung der `ausgabe()`-Funktion orientieren Sie sich an den beispielhaften Ausgaben.
 - Abschließend implementieren Sie nach dem gleichen Schema die Klasse *DVD*. Eine DVD besitzt im Gegensatz zu Büchern und Magazinen eine Altersfreigabe und ein Genre. Der Konstruktor der Klasse *DVD* hat folgendes Format `DVD(std::string initTitel, int initAltersfreigabe, std::string initGenre)`.
 - Setzen Sie nun oben in der Datei *Buecherei.cpp* den Wert `UNTERKLASSENVORHANDEN` von `false` auf `true` und kompilieren Sie ihren Code.
 - Lassen Sie sich die Datenbank ausgeben. Was stellen Sie fest?
 - Öffnen Sie die Datei *Medium.h* und schreiben Sie vor die Funktion `void ausgabe() const` ein `virtual`, sodass dort `virtual void ausgabe() const` steht. Kompilieren Sie ihren Code erneut und lassen sich die Datenbank ausgeben.

8.7.2 Einführung von Ausleihbeschränkungen

Sehen Sie sich die Funktion `virtual bool ausleihen(Person person, Datum ausleihdatum)` in der Klasse *Medium* genauer an. Ein Medium kann nur ausgeliehen werden, wenn es in der Bücherei vorhanden ist, also nicht gerade von einer anderen Person ausgeliehen wurde. Die Bücherei möchte zusätzliche Beschränkungen für das Ausleihen von Magazinen und DVDs einführen.

- Ein Magazin darf nicht ausgeliehen werden, wenn es sich um die neueste Ausgabe handelt. Gehen Sie davon aus, dass alle Magazine immer zu Beginn eines neuen Monats erscheinen.
- Eine DVD darf nur von Personen ausgeliehen werden, die die Altersbeschränkungen erfüllen.

Ihre Aufgabe ist es nun, diese Ausleihbeschränkungen zu realisieren. Machen Sie sich zunächst Gedanken, welche Funktionen in welchen Klassen überladen werden müssen, um die gewünschte Funktionalität zu realisieren.

- Beginnen Sie mit der Ausleihbeschränkung für die Magazine.

 1. Überladen Sie hierzu die Funktion `bool ausleihen(Person person, Datum ausleihdatum)` in der Klasse *Magazin*.
 2. Überprüfen Sie zunächst, ob die Ausleihbeschränkung erfüllt ist oder nicht. Wenn Sie den Subtraktionsoperator für zwei Objekte vom Typ *Datum* benutzen, erhalten Sie den Abstand zwischen diesen beiden Daten als Anzahl ganzer Monate. Nutzen Sie dies, um zu überprüfen, ob es sich um die aktuellste Ausgabe handelt.
 3. Geben Sie eine entsprechende Meldung aus, wenn es sich um die neueste Ausgabe des Magazins handelt.
 4. Wenn es sich nicht um die neueste Ausgabe des Magazins handelt, nutzen Sie die `ausleihen()`-Funktion der Basisklasse *Medium*.

5. Wenn das Ausleihen erfolgreich war, geben sie `true` zurück, wenn das Ausleihen nicht erfolgreich war, geben Sie `false` zurück.
- Anschließend realisieren Sie die Ausleihbeschränkung für alle weiteren, relevanten Klassen. Gehen Sie analog zu der Ausleihbeschränkung für Magazine vor, indem Sie die Funktion `bool ausleihen(Person person, Datum ausleihdatum)` überladen. Nutzen Sie den Subtraktionsoperator, wenn nötig, und geben Sie gegebenenfalls eine Meldung aus.
- Überprüfen Sie die Funktionsweise der Ausleihbeschränkungen.

8.7.3 Einführen einer rein virtuellen Funktion

Um die Software vor fehlerhafter Benutzung zu schützen, soll die Klasse *Medium* zu einer abstrakten Klasse gemacht werden. Somit verhindern Sie, dass Instanzen vom Typ *Medium* erzeugt werden. Eine Klasse wird zu einer abstrakten Klasse, sobald diese mindestens eine rein virtuelle Funktion besitzt, siehe Kapitel 8.4. Überlegen Sie, welche Funktion Sie rein virtuell machen können. Beachten Sie, dass eine rein virtuelle Funktion keine Implementierung besitzen muss, diese aber dennoch haben kann. Überprüfen Sie, ob sich die Funktionalität ihres Codes durch Einführen der rein virtuellen Funktion geändert hat. Versuchen Sie abschließend, eine Instanz vom Typ *Medium* der Bücherei hinzuzufügen.

8.7.4 Operatorüberladung zur Ausgabe aller ausgeliehenen Medien

Überladen Sie den Ausgabeoperator der Klasse *Medium*. Es sollen alle Informationen eines Mediums ausgegeben werden, die auch von der Ausgabefunktion *Medium* ausgegeben werden. Nutzen Sie hierfür die bereits implementierte *ausgabe*-Funktion. Bitte beachten Sie, dass hierfür eine Änderung der Signatur der Funktion notwendig ist, so wie Sie es in Kapitel 7.1.4 kennengelernt haben.

Implementieren Sie anschließend einen neuen Menüpunkt für die Bibliothek, der es ermöglicht, alle ausgeliehenen Medien auszugeben. Nutzen Sie für die Ausgabe der Medien den Ausgabeoperator. Ist kein Medium aus der Bücherei ausgeliehen, soll eine entsprechende Meldung erscheinen. Um die neue Funktionalität zu realisieren, wird es notwendig sein, neue Funktionen zu implementieren. Entscheiden Sie selbstständig, welche Funktionen notwendig sind.

8.7.5 Beispielhafte Ausgaben

Hier finden Sie beispielhaft Ausgaben, an denen Sie sich orientieren können:

Beispielhafte Ausgabe für ein Buch:

```
ID: 0
Titel: Tom Sawyers Abenteuer und Streiche
Status: Medium ist zurzeit nicht verliehen.
Autor: Mark Twain
```

Beispielhafte Ausgabe für eine Zeitschrift:

```
ID: 1
Titel: c't - Magazin für Computertechnik
Status: Medium ist zurzeit nicht verliehen.
Ausgabe: 01/04/2018
Sparte: Computerzeitschrift
```

Beispielhafte Ausgabe für eine DVD:

ID: 2

Titel: Der unsichtbare Dritte

Status: Medium ist zurzeit nicht verliehen.

FSK: ab 12 Jahre

Genre: Action, Krimi

9 GUI-Programmierung mit *Qt*

In den bisherigen Versuchen hat die gesamte Kommunikation zwischen Nutzer und Programm nur mithilfe einer Konsole stattgefunden. Da die Handhabung bei größeren und komplexeren Programmen über die Konsole nicht praktikabel ist, verwendet man eine grafische Benutzeroberfläche, auch GUI (*Graphical User Interface*) genannt. Eine GUI vereinfacht zwar die Kommunikation zwischen Programm und Benutzer deutlich, ist aber auch in der Erstellung aufwändiger. In diesem Versuch werden Sie lernen, wie man eine solche GUI erstellt.

Sie werden in diesem Versuch die Grundzüge der *Qt*-Bibliothek kennenlernen, welche bereits eine große Menge fertiger Elemente einer grafischen Benutzeroberfläche bereitstellt. Des Weiteren werden Sie in diesem Versuch eigenständig ein größeres Softwareprojekt erstellen und dabei alle bekannten Konzepte aus den vorhergehenden Versuchen erneut anwenden.

- Grundlagen der GUI-Programmierung
- Grundlagen von *Qt*
- Erstellung eines größeren Softwareprojekts

9.1 Qt

Bei *Qt*¹ handelt es sich um eine *C++*-Klassenbibliothek, mit der plattformübergreifend grafische Benutzeroberflächen programmiert werden. Allerdings liefert *Qt* auch viele andere Funktionen mit. So werden neue Datentypen eingeführt, die dank moderner Umsetzung im Vergleich zu den konventionellen Datentypen in *C++* eine bessere Handhabung bieten. Ein Beispiel dafür ist die Klasse *QString* als Alternative zur Klasse *std::string*. *Qt* ergänzt die nativen Fähigkeiten von *C++* sehr sinnvoll und ist eine hervorragend strukturierte und - nach einer Trainingsphase - auch sehr effizient nutzbare Basis zur Realisierung grafischer Benutzeroberflächen. *Qt* wird unter einer Open Source Lizenz veröffentlicht: Der Quellcode ist einsehbar und man darf *Qt* in eigenen Projekten nutzen.

Qt unterstützt alle aktuellen Betriebssysteme sowohl für Desktop- als auch für Mobile-Applikationen. Darüber hinaus gibt es für viele weiterverbreitete Programmiersprachen eine API, sodass man *Qt* nicht nur in Verbindung mit *C++* nutzen kann.

Neben der Klassenbibliothek bietet *Qt* mit dem *Qt-Creator* eine IDE, ähnlich Eclipse, an, die speziell auf die Programm- und GUI-Entwicklung mit *Qt* zugeschnitten ist. Zum Layouten grafischer Oberflächen ist der sogenannte *Qt-Designer* integriert, der es dem Benutzer erlaubt, die Gestaltung der Oberfläche per *Drag-and-Drop* durchzuführen und die Eigenschaften und das Verhalten der einzelnen Elemente zu konfigurieren.

Darüber hinaus kann der *Qt-Designer* bereits einen großen Teil des Rahmencodes automatisch generieren.

Sie werden hier umfassend von diesen Möglichkeiten Gebrauch machen und überrascht sein, wie hilfreich das ist.

Selbstverständlich kann man die *Qt*-Bibliothek auch ohne diese Werkzeuge nutzen und *Qt*-basierte Programme komplett von Hand erstellen. Dies setzt allerdings eine genauere Kenntnis dieser Bibliothek voraus. Für die ersten Versuche und einen Überblick über die Möglichkeiten ist die *Qt-Creator* IDE sehr hilfreich.

¹ausgesprochen wie das englische Wort *cute*

9.1.1 Eine erste Applikation

Am einfachsten lässt sich die Benutzung des *Qt-Creators* anhand einer Applikation erläutern. Außerdem sieht man, welchen Rahmen die IDE dem Benutzer bei der Erstellung eines Projekts bereits zur Verfügung stellt.

Starten Sie den *Qt-Creator*. Sie sehen den sogenannten Willkommensbildschirm. Hier können Sie auswählen, ob Sie ein neues Projekt anlegen wollen oder ein bereits existierendes geöffnet werden soll.

Wählen Sie *Neues Projekt*. Es öffnet sich ein Fenster, in dem Sie nähere Angaben zu Ihrem Projekt auswählen können.

Wählen Sie links *Anwendung*, in der Mitte *Qt-Widgets-Anwendung*. Klicken Sie unten rechts auf auswählen. In dem neu erscheinenden Fenster nennen Sie Ihr Projekt *Streetplanner*, dann weiter. Im folgenden Fenster können Sie eine Plattform angeben, für die Sie Ihr Projekt entwickeln wollen, hier gibt es im Moment nur eine Auswahl, *Desktop*, daher einfach weiter. Im darauffolgenden Fenster wird die Klasse angezeigt, die *Qt-Creator* anlegen wird, hier *MainWindow*, abgeleitet von *QMainWindow*. Es wird auch eine Formulardatei erzeugt mit Endung *.ui*. Diese beinhaltet die Beschreibung der grafischen Oberfläche. Dann weiter. Es folgt eine kurze Zusammenfassung des Projekts, die Sie bestätigen. Damit ist die Erstellung der Grundlage für Ihr erstes Projekt abgeschlossen, Sie sollten sich im Editor befinden und verfügen über ein bereits lauffähiges Projekt.

Links neben dem Editorfenster befindet sich ein Fenster, in dem, wie bei Eclipse, Ihr Projekt und die dazugehörigen Dateien angezeigt werden. Ganz links befindet sich die Hauptsteuerungsleiste für den *Qt-Creator*. Im Moment sind Sie im Editiermodus, d.h. Sie können Ihre Dateien bearbeiten. Es wurden bereits vier Datei für Sie erstellt, allerdings ohne große Funktionalität, aber lauffähig.

Die Symbole in der ganz linken Leiste bewirken:

- Willkommen - hier können neue Projekte angelegt werden oder ältere geöffnet werden
- Editieren - hier bearbeiten Sie Ihre Dateien und den Quellcode
- Design - hier wird angezeigt, wenn Sie sich im *Qt-Designer*-Modus befinden
- Debug - schaltet in die Debuggeransicht um, ohne diesen zu starten
- Projekte - hier werden Ihnen Daten zu Ihrem Projekt angezeigt und können auch verändert werden
- Hilfe - öffnet ein lokales Hilfefenster, in dem Sie nach Beschreibungen suchen können

Weiter unten in derselben Leiste befinden sich die im Moment wichtigsten Symbole:

- Monitor - legt die Projektart, die erzeugt werden soll, fest. Default ist Debugger
- Grüner Pfeil - compiliert das aktive Projekt und führt es aus
- Grüner Pfeil mit Käfer - compiliert das aktive Projekt und startet den Debugger
- Hammer - compiliert das aktive Projekt, ohne es zu starten

In der Menüleiste oben finden Sie viele Funktionen, die Ihnen aus Eclipse bereits bekannt sein sollten. Wenn Sie die Datei *mainwindow.ui* durch einen Doppelclick öffnen, wechselt der *Qt-Creator* automatisch in den Designmodus. Diese Datei kann nur im Designmodus bearbeitet werden und beschreibt das Aussehen und das Verhalten Ihrer grafischen Oberfläche.

In der Mitte sehen Sie den *Editor*. Hier können Sie die Elemente Ihrer Oberfläche anordnen.

Links sehen Sie jetzt alle Elemente, die Ihnen zur Gestaltung zur Verfügung stehen.

Rechts oben neben dem Hauptfenster befindet sich eine hierachische Liste der Elemente, die bereits zu Ihrer grafischen Oberfläche gehören, im Moment ein Hauptfenster mit zentralem Anzeigefenster, eine Menüleiste, eine Toolleiste sowie eine Statusleiste.

Darunter befindet sich ein Fenster, das die aktuellen Eigenschaften des ausgewählten Elements beschreibt. Diese Eigenschaften können hier verändert werden.

Sie verlassen den Designmodus, indem Sie ganz links auf *Editieren* klicken. Im Hauptfenster erscheint eine XML-Datei, die Ihre Oberfläche beschreibt und beim Compilieren in C++-Code übersetzt wird. Diese Datei kann nicht per Hand editiert werden. Schliessen Sie die Datei, indem Sie im Editorfenster neben dem Dateinamen auf das *X* klicken.

Wenn Sie jetzt Ihr Programm starten, indem Sie ganz links auf den grünen Pfeil klicken, erscheint ein einfaches Fenster ohne weitere Funktion.

9.2 Klassen und Funktionen in Qt

Im Folgenden sollen einige Klassen und Funktionsweisen von *Qt* vorgestellt werden, die verschiedene Teile der Benutzeroberfläche realisieren. Für eine detailliertere Beschreibung wird auf die gut lesbare Dokumentation des *Qt*-Projekts [6] verwiesen.

Die vorgestellten Klassen werden in der anschließenden Aufgabenstellung benötigt. Dort wird auch erläutert, wie diese Elemente im Designermodus angelegt und konfiguriert werden können. Einige Aspekte werden Ihnen dennoch erst richtig klar werden, wenn Sie beginnen, Ihre eigene Oberfläche zu gestalten.

9.2.1 Signals und Slots

Qt nutzt das so genannte Signal-Slot-Konzept, welches es ermöglicht, dass verschiedene Objekte miteinander kommunizieren. Beispielsweise kann so eine definierte Funktion aufgerufen werden, wenn auf eine Schaltfläche gedrückt wird.

Dieses Konzept besteht aus zwei Teilen:

Signal Ein Signal ist eine Botschaft, die beim Eintritt eines bestimmten Ereignisses, z.B. von einem grafischen Element oder auch vom eigenen Programm, gesendet wird.

Slot Ein Slot ist grundsätzlich eine *normale* Funktion, die mit einem Signal verknüpft ist, und dann als Empfänger für ein Signal genutzt werden kann.

Es ist möglich, mehrere Signale mit einem Slot zu verknüpfen, oder mehrere Slots mit einem Signal. Wenn ein Signal mit einem Slot verbunden wurde, wird die Slot-Funktion jedes Mal ausgeführt, nachdem das entsprechende Signal ausgegeben (*emittiert*) wurde.

9.2.2 QApplication

QApplication ist eine Klasse, die für die allgemeine Verwaltung der Applikation von der Initialisierung bis zum Beenden des Programms sorgt und in jeder *Qt*-Anwendung genau einmal instanziiert werden muss. Die Klasse *QApplication* erhält als Parameter dieselben Argumente, die auch der main-Funktion übergeben wurden.

Sie finden ein Objekt dieser Klasse in Ihrem soeben erstellten Programm in der Datei *main.cpp*. Der *Qt-Creator* hat dieses Objekt bereits für Sie angelegt.

Mit der Methode *exec()* startet die sogenannte *Ereignis-Hauptschleife*. Diese Schleife wartet auf Ereignisse, z.B. in Form von Benutzereingaben, um diese dann an die zugehörigen Funktionen zur Verarbeitung weiterzuleiten. Durch den intern verwalteten Aufruf der *QApplication*-Methode *quit()* (beispielsweise durch einen Klick auf den unter dem Betriebssystem Windows bekannten *Schließen*-Button) wird das Programm beendet.

9.2.3 QDebug

Die Klasse *QDebug* stellt Funktionen bereit, mit Hilfe derer während der Entwicklungsphase Informationen auf die Konsole ausgegeben werden können, oder in eine Datei oder ein Gerät. Wenn das Programm fertig entwickelt ist, werden diese Ausgaben nicht mehr benötigt, daher der Name.

Um *QDebug* nutzen zu können, binden Sie diese Klasse mittels `#include` ein, wie Sie es bereits aus den vorherigen Versuchen kennen.

Ein Beispiel zur Verwendung von *qDebug()* sehen Sie im folgenden Abschnitt *QString*. Sie werden sehen, dass *qDebug()* sehr ähnlich zu *std::cout* funktioniert, jedoch mehr Möglichkeiten bietet.

9.2.4 QString

Die Klasse *QString* ist dem *std::string* sehr ähnlich. Die grundlegenden Funktionen werden alle auch von *QString* unterstützt und um zusätzliche Funktionen ergänzt.

QString stellt z.B. eine Memberfunktion *arg()* zur Verfügung, die einen String zur Laufzeit in einen anderen String einfügen kann. Die Memberfunktionen *toInt()* und *toDouble()* wandeln einen String in die entsprechenden Zahlen um, wenn das möglich ist. Um aus einer Zahl einen String zu erzeugen, steht die Funktion *QString::number()* zur Verfügung. Die Funktionen *QString::fromStdString(const std::string& str)*² und *toStdString()* ermöglichen eine Konvertierung zwischen den beiden Stringtypen. Ähnliche Konvertierungen stehen auch in anderen Klassen zur Verfügung und werden benötigt, wenn auf Funktionen der *STL* zugegriffen werden soll, die bestimmte Datentypen als Parameter erwarten.

Beispiele:

```

1 #include <QString>
2 #include <QDebug>
3 .
4 .
5 QString str1 = "rot";
6 QString str2 = "blau";
7 QString str3 = QString("Das Buch ist vorne %1, hinten %2.").arg(str1).arg(str2);
8 qDebug() << str3;
```

Das Buch ist vorne rot, hinten blau.

```

1 #include <QString>
2 #include <QDebug>
3 .
4 .
5 int a;
6 QString str1 = "5";
7 QString str2 = "hallo";
8 bool ok;
9 a = str1.toInt(&ok);    \\ ok wird auf true gesetzt
10 if (!ok)
11 {
12     qDebug() << "Fehler: keine Zahl!";
13 }
14 else
15 {
16     qDebug() << "Zahl: " << a;
17 }
18
19 a = str2.toInt(&ok);    \\ ok wird auf false gesetzt
20 if (!ok)
21 {
22     qDebug() << "Fehler: keine Zahl!";
23 }
24 else
25 {
26     qDebug() << "Zahl: " << a;
27 }
```

Zahl: 5
Fehler: keine Zahl!

9.2.5 QMessageBox

Die Klasse *QMessageBox* öffnet einen modalen Dialog, mit dem der Benutzer informiert oder befragt werden kann. Modal bedeutet, dass das Programm so lange wartet, bis der Benutzer die Meldung quittiert hat. Eine *QMessageBox* hat standardmäßig nur eine Schaltfläche, die mit Ok beschriftet ist. Mit

²Hierbei handelt es sich um eine statische Funktion, da hier kein Objekt von *QString* zur Verfügung steht.

der Funktion `setStandardButtons(StandardButton button)` kann festgelegt werden, ob weitere Schaltflächen hinzugefügt werden sollen. Die Funktion `setText(const QString& text)` setzt den anzuseigenden Text, während mit der Funktion `setInformativeText(const QString& text)` ein zusätzlicher, erklärender Kommentar hinzugefügt werden kann. Die Funktion `setDefaultButton(StandardButton button)` legt fest, welcher Button vorausgewählt ist. Angezeigt wird die MessageBox mit der Funktion `exec()`, die einen Ganzahlwert zurückgibt, der angeigt, auf welchen Knopf geklickt wurde.

Im folgenden Beispiel wird eine `QMessageBox` erstellt, die einen Speicherdialog anzeigt, der anschließend ausgewertet wird. In der Auswertung wird eine neue `QMessageBox` erzeugt, die nur einen OK-Knopf besitzt und ausgibt, welcher Knopf angeklickt wurde.

```

1 #include <QMessageBox>
2 .
3 .
4 QMessageBox msgBox;
5
6 msgBox.setText("The document has been modified.");
7 msgBox.setInformativeText("Do you want to save your changes?");
8
9 msgBox.setStandardButtons(QMessageBox::Save |
10                           QMessageBox::Discard |
11                           QMessageBox::Cancel);
12 msgBox.setDefaultButton(QMessageBox::Save);
13
14 int ret = msgBox.exec();
15
16 QMessageBox result;
17
18 switch (ret)
19 {
20 case QMessageBox::Save:
21     result.setText("Save was clicked.");
22     break;
23 case QMessageBox::Discard:
24     result.setText("Don't save was clicked.");
25     break;
26 case QMessageBox::Cancel:
27     result.setText("Cancel was clicked.");
28     break;
29 default:
30     result.setText("This should be impossible");
31     break;
32 }
33
34 result.exec();

```

Beispiel QMessageBox

9.2.6 QPushButton

Die Klasse `QPushButton` beschreibt eine einfache Schaltfläche (Button), die einen Text anzeigen kann und üblicherweise dazu verwendet wird, eine bestimmte Funktion auszuführen, nachdem sie geklickt wurde.

Wesentliche Attribute sind `size` und `text`, die die Größe des Buttons und den angezeigten Text bestimmen. Der Text kann aber nicht nur, wie oben schon beschrieben, über den Konstruktor gesetzt werden, sondern auch mithilfe der Funktion `setText(const QString& text)`. Weitere Funktionen, wie zum Beispiel das Setzen eines Shortcuts, können in der Dokumentation nachgelesen werden.

Das Signal, das bei einem Klick auf den Button emittiert wird, heißt `clicked()`.

9.2.7 QLabel

Die Klasse `QLabel` dient zur Anzeige von Text oder Bildern. Eine Interaktion mit dem Benutzer ist nicht vorgesehen.

9.2.8 QLineEdit

Die Klasse *QLineEdit* ist ein einzeiliger Texteditor, in den der Benutzer Text eingeben kann. Es stehen eine Reihe von Memberfunktionen zur Verfügung, z. B. für *Drag-and-Drop* oder *Copy-and-Paste*. Den vom Benutzer eingegebenen Text liest man mit der Funktion *text()* aus, die den Text in einem *QString* zurückgibt.

Im folgenden Beispiel soll ein eingegebener Text nach Klick auf den entsprechenden Button in einer *QMessageBox* ausgegeben werden:

```

1 #include <QPushButton>
2 #include <QLineEdit>
3 #include <QMessageBox>
4
5 QString str;
6 str = ui->lineEdit->text();
7
8 // Kopiert den Text in die MessageBox
9
10 QMessageBox msgBox;
11 msgBox.setText(str);
12 msgBox.exec();
```

Beispiel QLineEdit

9.2.9 QGraphicsView und QGraphicsScene

Ein *QGraphicsView* dient zum Anzeigen einer *QGraphicsScene*. Dabei sorgt das *QGraphicsView* unter anderem dafür, dass scrollen möglich wird.

QGraphicsScene stellt eine Zeichenfläche bereit. So stehen z.B. Funktionen zur Verfügung, die Ellipsen, Rechtecke oder Linien zeichnen. Die Syntax lautet wie folgt:

- `addRect(double x, double y, double w, double h, const QPen& pen = QPen(),
 const QBrush& brush = QBrush())`
- `addEllipse(double x, double y, double w, double h, const QPen& pen = QPen(),
 const QBrush& brush = QBrush())`
- `addLine(double x1, double y1, double x2, double y2,
 const QPen& pen = QPen())`

Bei einem Rechteck und einer Ellipse steht *x* für die X-Koordinate und *y* für die Y-Koordinate der oberen linken Ecke; *w* und *h* beschreiben die Breite und die Höhe. Die Ellipse wird hierbei in den so definierten, rechteckigen Zeichenbereich eingepasst. Bei einer Linie übergibt man zwei Punkte, zwischen denen die Linie gezeichnet wird. Für diese Positions- und Größenparameter werden Fließkommazahlen akzeptiert.

Die Klasse *QPen* beschreibt, wie die Linien gezeichnet werden. Die Farbe und die Breite werden dazu mit den Funktionen *setColor(const QColor &color)* und *setWidth(int width)* eingestellt. Es lässt sich auch festlegen, ob eine Linie gepunktet gezeichnet wird. *QBrush* beschreibt Typ und Farbe des „virtuellen Zeichenstiftes“. Hier wird die Farbe und das Füllverhalten im Konstruktor angeben. Beispielsweise würde *QBrush(Qt::red, Qt::SolidPattern)* ein damit gezeichnetes Rechteck vollständig rot ausfüllen.

Es besteht neben den obengenannten Funktionen noch die Möglichkeit, mithilfe der Funktion *addItem(QGraphicsItem *)* andere Elemente hinzuzufügen, die von der Klasse *QGraphicsItem* erben. Ein Beispiel dafür ist die Klasse *QGraphicsTextItem*, mit der ein Text an eine bestimmte Stelle im Zeichenbereich eingefügt werden kann. Die Position und der Text können im Konstruktor übergeben (z.B. *QGraphicsTextItem("Hallo Welt!")*) oder mit der Funktion *setPlainText(const QString &text)* direkt gesetzt werden. Die Position wird mit der Funktion *setPos(double x, double y)* gesetzt.

Im folgenden Beispiel wird eine gelbe Linie gezeichnet und ein roter Punkt gesetzt, der mit einem Text beschriftet wird.

```

1 #include <QGraphicsView>
2 #include <QGraphicsTextItem>
3 .
4 .
5 QGraphicsView view;           // Erstellen der QGraphicsView.
6 QGraphicsScene scene;         // Erstellen der QGraphicsScene.
7
8 // GraphicsScene der GraphicsView übergeben, um sie anzuzeigen.
9
10 view.setScene(&scene);
11
12 // Man kann keine einzelnen Punkte erstellen,
13 // sondern nur eine Ellipse.
14
15 scene.addEllipse(100, 100, 4, 4, QPen(Qt::red),
16                   QBrush(Qt::red, Qt::SolidPattern));
17 QPen pen;
18 pen.setWidth(5);
19 pen.setColor(Qt::yellow);
20 scene.addLine(10, 10, 190, 10, pen);
21
22 // Eine Klasse, um Text anzuzeigen
23
24 QGraphicsTextItem* text = new QGraphicsTextItem;
25 text->setPos(80, 70);           //Position des Textes
26 text->setPlainText("Hallo Welt!"); // Text
27 scene.addItem(text);

```

Beispiel QGraphicsScene

9.2.10 QMainWindow

Die *QMainWindow*-Klasse bietet ein Hauptfenster für die Applikation. Darin wird die programm-spezifische grafische Benutzeroberfläche eingebettet. Ein *QMainWindow* besteht aus einer Menüleiste (*QMenuBar*), einer horizontalen Leiste, auf der Statusinformationen angezeigt werden (*QStatusBar*), einer Leiste, auf der Kontrollelemente angeordnet werden (*QToolBar*) und einem Bereich, in dem die Hauptkomponenten der Benutzeroberfläche liegen (*Central Widget*).

9.2.11 QDialog

Dialoge sind eigenständige Fenster, die eine Interaktion mit dem Benutzer ermöglichen. Selbsterstellte Dialoge werden von *QDialog* abgeleitet. *QDialog* ist auch die Basisklasse aller in *Qt* zur Verfügung stehenden Dialogklassen, z.B. für *QMessageBox*. Dialoge werden als besondere Klasse angelegt, als sogenannte Formularklasse, und sie haben eine eigene Formulardatei mit Endung *ui*. Diese Datei kann im Designermodus bearbeitet werden, so dass das Aussehen und die Funktionalität von Dialogen mit wenig Aufwand wie gewünscht gestaltet werden kann.

Alle Dialoge lassen sich mit der Funktion *exec()* anzeigen. Die Handhabung wurde mit der Klasse *QMessageBox*, die von *QDialog* erbt, in vorhergehenden Beispielen gezeigt.

Standardmäßig gibt die *exec()*-Funktion eine 1 zurück, wenn der *ok*-Button geklickt wird, eine 0, wenn *cancel*.

Die Instanz des Dialogs existiert auch dann noch, wenn das Fenster geschlossen ist. Es können weiterhin alle Funktionen aufgerufen werden, die die Klasse zur Verfügung stellt. Dies kann interessant sein, wenn der Rückgabewert von *exec()* ausgewertet werden soll. Hat der Benutzer *ok* geklickt, wird z.B. eine Funktion Ihres Dialogs aufgerufen, die dann etwas macht, z.B. die Daten in den Eingabefeldern auswerten und dem Benutzer einen entsprechenden Wert zurückgibt. Der Benutzer kann ja nicht selbst auf die Eingabefelder zugreifen.

Gibt die *exec()*-Funktion eine 0 zurück, dann hat der Benutzer *cancel* geklickt, und es passiert z.B. nichts.

9.3 Statische Attribute und Methoden

Statische Attribute

Es gibt im C++ Klassenkonzept die Möglichkeit, in einer Klasse ein *statisches* Attribut anzulegen. Ein *statisches* Attribut gehört nicht zu einem Objekt der Klasse, sondern existiert unabhängig davon, ob Objekte der Klasse instanziert wurden. Dieses *statische* Attribut ist für alle Objekte gleich. Statische Attribute stellen sozusagen versteckte, globale Variablen dar und existieren unabhängig davon.

Statische Attribute werden außerhalb ihrer Klasse initialisiert, denn innerhalb ginge es nur, wenn ein Objekt der Klasse erstellt wird.

Im folgenden Beispiel sollen alle Praktikumsteilnehmer eine fortlaufende Nummer erhalten, und gleichzeitig eine fortlaufende Nummer, die für den Benutzernamen zum Anmelden im CIP-Pool verwendet werden soll. Um dies zu realisieren, benutzt man zwei *statische* Variablen.

```

1  class Praktikumsteilnehmer
2  {
3  private:
4      static int objectCounter;
5      static int accountCounter;
6      int id;
7      int accountId;
8      std::string name;
9      :
10     :
11
12 public:
13     Praktikumsteilnehmer(std::string pName);
14     ~Praktikumsteilnehmer();
15     /* ... */
16 };
17
18
19 // Anzahl der diesjährigen Praktikumsteilnehmer
20 int Praktikumsteilnehmer::objectCounter = 0;
21
22 // Der letzte Praktikumsteilnehmer vom letzten Jahr
23 int Praktikumsteilnehmer::accountCounter = 9330001;
24
25 Praktikumsteilnehmer::Praktikumsteilnehmer(std::string pName) :
26     id(++objectCounter), accountId(++accountCounter), name(pName)
27 {
28 }
29
30 /* ... */

```

Listing 9.1: Deklaration und Verwendung eines statischen Attributs

Statische Methoden

Genauso, wie es *statische* Attribute gibt, gibt es auch *statische Methoden*. Diese sind auch aufrufbar, ohne dass ein Objekt der Klasse existieren muss (was für die übrigen Methoden gilt). Damit diese Methoden aufrufbar sind, dürfen diese keinen *this*-Zeiger verwenden, da dieser nur existiert, sobald ein Objekt erzeugt wurde. Auch statische Methoden kann man als versteckt global bezeichnen, sie gehören aber zum Namensraum der Klasse. Statische Methoden können - da sie keinen *this*-Zeiger haben - nicht auf Member eines Objektes zugreifen. Der Zugriff auf statische Attribute und statische Methoden ist aber möglich.[1]

Im obigen Beispiel wurden zwei statische Attribute *objectCounter* und *accountCounter* angelegt und initialisiert. Um auf diese Attribute zugreifen zu können, auch wenn noch keine Instanz von *Praktikumsteilnehmer* existiert, kann eine statische Methode benutzt werden. Die Deklaration und Implementierung dieser statischen Methoden sieht wie folgt aus:

```

1  class Praktikumsteilnehmer
2  {
3  private:
4      static int objectCounter;
5      static int accountCounter;
6      int id;
7      int accountId;
8      std::string name;
9
10 public:
11     Praktikumsteilnehmer(std::string pName);
12     ~Praktikumsteilnehmer();
13
14     static int getObjectCounter();
15     static int getAccountNumber();
16     /* ... */
17 };
18
19 // Anzahl der diesjährigen Praktikumsteilnehmer
20 int Praktikumsteilnehmer::objectCounter = 0;
21
22 // Der letzte Praktikumsteilnehmer vom letzten Jahr
23 int Praktikumsteilnehmer::accountCounter = 9330001;
24
25
26 Praktikumsteilnehmer::Praktikumsteilnehmer(std::string pName) :
27     id(++objectCounter), accountId(++accountCounter), name(pName)
28 {
29 }
30
31 int Praktikumsteilnehmer::getObjectCounter()
32 {
33     return objectCounter;
34 }
35
36 int Praktikumsteilnehmer::getAccountNumber()
37 {
38     return accountCounter();
39 }
40
41 /* ... */
42
43 int main()
44 {
45     /*...*/
46
47     // Anzahl Praktikumsteilnehmer
48     int zaehler = Praktikumsteilnehmer::getObjectCounter();
49
50     // Letzter vergebener Account
51     int lastAccountNumber = Praktikumsteilnehmer::getAccountNumber();
52     /*...*/
53 }
```

Listing 9.2: Deklaration und Verwendung einer statischen Methode

Beachten Sie, dass statische Methoden nicht über ein Objekt der Klasse aufgerufen werden, da sie unter anderem keinen *this*-Zeiger haben. Wie im Beispiel zu sehen, werden sie mittels des *Scope-Operators* :: und dem Klassennamen aufgerufen. Ohne existierendes Objekt von *Praktikumsteilnehmer* hätte die Variable *zaehler* den Wert 0 und die Variable *lastAccountNumber* 9330001.

In der nachfolgenden Aufgabe werden Sie solche *statischen* Methoden benutzen. Die Klasse *Dijkstra* stellt Ihnen eine solche Methode zur Verfügung (siehe Vorlagen). Es ist sinnvoll, diese Methode in eine eigene Klasse zu kapseln, denn so kann der Dijkstra-Algorithmus leicht in anderen Programmen wiederverwendet werden, indem man dort die Klasse einbindet. Es ist aber nicht sinnvoll, ein Objekt dieser Klasse zu instanzieren. Es wird lediglich die Funktionalität zur Verfügung gestellt.

Auch die *Qt*-Klassenbibliothek stellt Ihnen innerhalb ihrer Klassen zahlreiche, statische Methoden zur

Verfügung. Die Klasse *QFileDialog* besitzt eine statische Methode, mit der Sie über ein Auswahlfenster eine Datei öffnen können, die Klasse *QDir* eine Methode, den aktuellen Pfad zu setzen, ohne ein Objekt dieser Klassen erzeugen zu müssen. Nähere Beschreibungen finden Sie in der *Qt* Referenz, deren Benutzung später im Aufgabenteil mit Beispielen erläutert wird.

9.4 Aufgaben

In diesem Versuch erstellen Sie ein Programm zur Darstellung einer einfachen Karte. Es wird dem Nutzer die Möglichkeit bieten, Städte und Straßen hinzuzufügen und darin den kürzesten Weg mittels des Dijkstra-Algorithmus zu finden. Im späteren Verlauf dieses Versuches entscheiden Sie selbst, wie Sie Ihr Programm weiter entwickeln.

Sie werden das Programm im Wesentlichen selbst implementieren, allerdings finden Sie im L²P auch für diesen Versuch Hilfestellungen, Schnittstellen-Vorgaben und Algorithmus-Implementierungen. Es wird im Verlauf des Versuchs darauf hingewiesen, wenn diese benötigt werden. Sie finden in den Vorlagen eine Textdatei, die Anregungen zur Gestaltung der Testfunktionen enthält und aus der Sie mit *Copy-and-Paste* kopieren können.

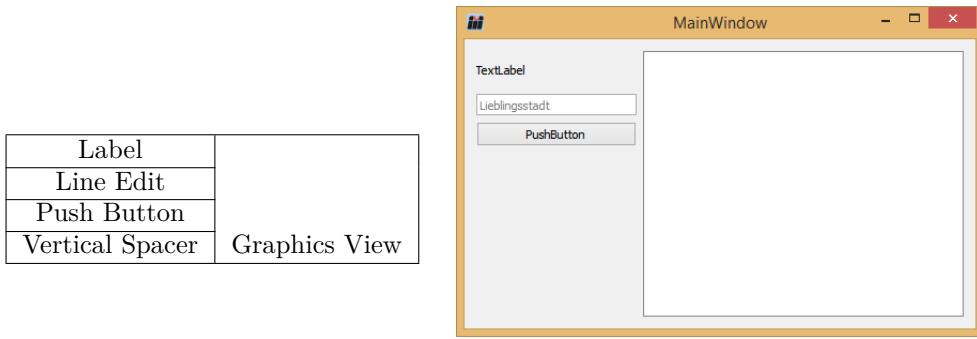
Ihr Projekt haben Sie zu Beginn dieses Kapitels bereits angelegt.

9.4.1 Eine neue Applikation

Das Grundgerüst für Ihr Projekt *Streetplanner* haben Sie ja bereits zu Beginn dieses Versuchs erstellt. Jetzt werden Sie Ihr noch leeres Fenster mit einfachen GUI-Elementen gestalten.

Einfache GUI-Elemente

1. Klicken Sie doppelt auf die Formulardatei *mainwindow.ui*. Es öffnet sich daraufhin der *Qt Designer* im Designermodus. Im Hauptbereich sehen Sie das Fenster, so wie es bei der Ausführung des Programmes aussehen wird. An der Seite sehen Sie die GUI-Elemente, die sich *auf das Fenster ziehen* lassen.
2. Ziehen Sie zunächst ein *Label* auf die linke Seite ihres Fensters. Platzieren Sie dann ein *Line Edit* und einen *Push Button* darunter. Danach ziehen Sie ein *Graphics View* auf die rechte Seite Ihres Fensters. Die Elemente sollen sich nicht überschneiden. Klicken Sie mit der rechten Maustaste auf eine freie Stelle in Ihrem Hauptfenster. In dem Kontextmenü fahren Sie auf *Layout* und wählen in dem sich öffnenden Fenster *Objekte tabellarisch anordnen* aus. Ihre Elemente sind jetzt etwas anders im Fenster angeordnet als zuvor.
3. Ziehen Sie jetzt einen *Vertical Spacer* auf Ihr Fenster. Eine blaue Linie zeigt Ihnen an, an welcher Stelle er eingefügt wird. Platzieren Sie ihn auf der linken Seite unter den PushButton. Der Spacer wird nun alle Elemente nach oben schieben, auch die GraphicsView. Um dies zu ändern, klicken Sie auf Ihre GraphicsView. Vergrößern Sie Ihr GraphicsView, indem Sie an dem blauen Steuerelement unten in der Mitte nach unten ziehen. Jetzt gehört der Spacer zur linken Seite Ihrer Tabelle (Grid) und schiebt nur noch die Elemente auf der linken Seite nach oben.
4. Mit den Tasten *Alt-Shift-r* lässt sich eine Vorschau erzeugen ohne Compilieren zu müssen. Das Vorschaufenster sieht aus wie später das Fenster in Ihrem Programm, und verhält sich auch so. Testen Sie Ihr Fenster, indem Sie es vergrößern und verkleinern. Ihr Fenster sollte jetzt folgender Tabelle ähnlich sehen.

**Abbildung 9.1:** Anordnung erster Elemente

5. Auf der rechten, oberen Seite werden die Elemente Ihres Fensters angezeigt. Wenn Sie auf eins der Elemente in dieser Liste klicken, werden darunter die Eigenschaften dieses GUI-Elemente angezeigt. Sie können auch auf das Element in Ihrem Hauptfenster klicken.

Diese werden Sie jetzt nutzen, um Ihre GUI-Elemente zu konfigurieren. Ihre GUI-Elemente sind als Grid (Tabelle) angeordnet, im Moment zwei Spalten (Column), rechts und links, und vier Zeilen (Rows), wobei die GraphicsView über alle vier Zeilen geht. Wie Sie gesehen haben, wird die Größe aller GUI-Elemente gleichmäßig verändert, wenn Sie die Größe des Fensters in der Vorschau ändern. Für die Elemente auf der linken Seite ist das nicht schön. Um dies zu ändern, klicken Sie auf centralWidget. Scrollen Sie rechts im Eigenschaftsfenster nach unten. Dabei sehen Sie, dass es sehr viele Einstellmöglichkeiten gibt. Unten finden Sie den Abschnitt *Layout* und dort die Eigenschaft *layoutColumnStretch* mit Wert 0,0. Ändern Sie diesen Wert auf 0,1. Dies bewirkt, dass jetzt nur noch Spalte 1 vergrößert wird. Spalten und Zeilen beginnen bei 0.

Darüber sehen Sie die Eigenschaft *layoutVerticalSpacing*, die den vertikalen Abstand zwischen Ihren GUI-Elementen bestimmt. Wenn Sie diesen Wert ändern, sehen Sie das sofort in Ihrem Fenster.

6. Klicken Sie auf *label*. Ändern Sie den Namen dieses Objekts, z.B. *label_eingabe*. Benutzen Sie einen sinnvollen Namen, denn dieser Name wird später in Ihrem Programm benutzt, um genau dieses GUI-Element anzusprechen. Scrollen Sie nach unten bis *text*. Hier können Sie den Text des Labels ändern, z.B. *Ihre Eingabe*.
7. Klicken Sie auf *lineEdit*. Ändern Sie wieder den Namen, z.B. *lineEdit_teste_was*. Weiter unten finden Sie *placeholderText*, setzen Sie ihn z.B. auf *Ihre Eingabe*. Darunter finden Sie noch *clearButtonEnabled*, setzen Sie dort ein Häckchen. Ihr LineEdit hat jetzt rechts einen Löschbutton, sobald ein Text eingegeben wird. In der Vorschau können Sie das testen.
8. Klicken Sie auf *pushButton*. Ändern Sie wieder den Name, z.B. *pushButton_teste_was*. Die Beschriftung können Sie bei *Text* ändern, z.B. *Teste was*. Etwas weiter oben gibt es die Eigenschaft *toolTip*. Wenn Sie hier einen Text eingeben, erscheint dieser, wenn die Maus über dem Button ist. Probieren Sie z.B. *Testet*, ob Ihre Eingabe eine Zahl oder ein Text ist. Im Vorschaufenster können Sie das überprüfen.

Diese sehr ausführliche Einführung in die Möglichkeiten der Oberflächengestaltung sollte Ihnen ein erstes Gefühl geben, was mit dem *Qt-Designer* möglich ist. Für die anderen Eigenschaften verwenden Sie die oben schon erwähnte Dokumentation von *Qt*. In den entsprechenden Klassen finden Sie diese Beschreibungen. Die Nutzung der *Qt*-Dokumentation wird in einem der folgenden Kapitel noch näher erläutert.

Starten Sie nun das Programm und testen Sie das Aussehen.

Das Programm wird interaktiv

Führen Sie zum Einstieg in die Programmierung mit *Qt* einige Tests durch.

1. Öffnen Sie den *Qt Designer* für das *MainWindow* erneut und wählen Sie im Kontextmenü des *Push Button* (den Button markieren, rechte Maustaste) den Menüpunkt *Slot anzeigen* aus. Die

erscheinende Liste zeigt alle Signale, die dieser Knopf aussenden kann. Wählen Sie das Signal `clicked()`: Dadurch wurde ein Slot (eine Funktion) angelegt und mit dem Signal verbunden, d.h. ab sofort wird bei jedem Klick auf den Knopf die als Slot deklarierte Funktion ausgeführt. Der Cursor sollte nun auch schon zu diesem Slot gesprungen sein.

2. Lassen Sie beim Klick auf den Knopf eine Nachricht auf der Konsole mit Hilfe von `qDebug()` ausgeben und testen Sie die Funktion (siehe Kapitel 9.2.3, Seite 117).
3. Geben Sie auch den Text aus, den der Benutzer in das *Line Edit* eintragen kann. Nutzen Sie hierbei die Methode `.arg()` der Klasse *QString* (siehe Kapitel 9.2.4, Seite 118). Im folgenden Kapitel ist genau beschrieben, wie Sie für diese und noch weitere Methoden in der *Qt*-Referenz die Informationen und Beschreibungen finden, die Sie benötigen.

Hinweis: Auf die GUI-Elemente wird über den Zeiger *ui* zugegriffen. Zum Beispiel ist `ui->lineEdit` ein Zeiger auf das GUI-Element *Line Edit* (siehe Kapitel 9.2.8, Seite 120).

4. Geben Sie, falls der Benutzer in das *Line Edit* eine Zahl eingetragen hat, die Zahl erhöht um 4 aus. Nutzen Sie dazu auch Methoden von *QString* (siehe folgendes Kapitel zur *Qt*-Referenz).

Nutzung der *Qt*-Referenz

Damit Sie bei den weiteren Aufgaben schneller Informationen über die *Qt*-Klassen und deren Memberfunktionen finden, soll das Vorgehen hier an einigen Beispielen erläutert werden.

1. Öffnen Sie in einem Browser die Seite doc.qt.io. Sie befinden sich auf der Startseite der *Qt*-Dokumentation. Hier finden Sie z.B. Tutorials und Beispiele.
2. Klicken Sie auf *Qt Reference Documentation*. Sie sehen eine Reihe an Auswahlmöglichkeiten, wählen Sie *All Classes*.
3. Sie suchen Informationen zur Funktion `arg()` sowie zu Funktionen, die einen String in eine Zahl umwandeln. Da alle Klassen mit *Q* beginnen, bezieht sich die Liste der Buchstaben auf den ersten nach *Q*. Klicken Sie auf *S*. Suchen Sie die Klasse *QString* und klicken darauf. Sie sind jetzt auf der Hilfeseite der Klasse *QString*.
4. Es gibt eine kurze Beschreibung über die Funktion der Klasse. Wenn Sie etwas nach unten scrollen, finden Sie den Abschnitt *Public Functions*. Dies sind die Memberfunktionen, die innerhalb der Klasse als *public* deklariert sind und von außerhalb aufrufbar sind. Als erstes erscheinen die vorhandenen Konstruktoren der Klasse. Sie sehen z.B. einen Standardkonstruktor.
5. Scrollen Sie, bis Sie die Funktion `arg()` sehen. Es gibt mehrere Funktionen mit diesem Namen, die sich jedoch in ihrer Parameterliste unterscheiden. Alle geben einen *QString* zurück. Um eine nähere Beschreibung der Funktion zu bekommen, klicken Sie auf das erste `arg()`. Sie sind jetzt bei der ausführlichen Beschreibung dieser Funktion. Als ersten Parameter erwartet diese Funktion eine Referenz auf einen *QString*, die weiteren Parameter können, müssen aber nicht angegeben werden, da sie über Default-Werte verfügen.
6. Sehen Sie sich am besten das Beispiel an, dieses veranschaulicht den Gebrauch von `arg()` sehr gut. Innerhalb eines *QString*s ersetzt diese Funktion das Zeichen `%1` durch den Rückgabewert der ersten `arg()`-Funktion, `%2` durch den zweiten.
7. Wenn Sie jetzt noch etwas weiter nach unten scrollen, werden die anderen `arg()`-Funktionen beschrieben. Sie unterscheiden sich hauptsächlich durch den ersten Übertragungsparameter. Wie Sie sehen, können das auch Zahlen sein, die dann in einen *QString* umgewandelt werden. Sehr praktisch für Ihre obige Aufgabe.
8. Gehen Sie zurück auf die Seite mit der Liste aller Funktionen. Scrollen Sie durch die Liste aller Memberfunktionen. Dabei sehen Sie all die Möglichkeiten, die diese Klasse zur Verfügung stellt, z.B. `isEmpty()`, `setNum()` und mehr.
9. Wenn Sie weiterscrollen, finden Sie eine Funktion `toInt()`. Klicken Sie auf diese, und sehen Sie sich die Beschreibung und das Beispiel an. Der erste Parameter ist vom Typ *bool* und zeigt an, ob die Umwandlung erfolgreich war. Ist der Wert *false*, dann waren es nicht nur Ziffern, die

vorkamen. Sie können diesen also nutzen, um zu entscheiden, ob eine Zahl oder ein Text im *QString* ist. Ähnliche Funktionen gibt es auch für *double*.

10. Gehen Sie zurück zu der Seite, auf der Sie die Klassen per Buchstaben auswählen können. Wählen Sie jetzt *L*, anschliessend *QLineEdit*. Sie suchen für Ihre Aufgabe eine Funktion, die Ihnen den eingegebenen Text als *QString* zurückgibt. Wenn Sie durch die Memberfunktionen scrollen, kommen also von vorne herein nur Funktionen in Frage, die einen *QString* zurückgeben. Davon gibt es nicht so viele. Die beiden Funktionen *displayText()* und *text()* geben den eingegebenen Text zurück, *text()* immer so, wie er eingegeben wurde, *displayText()* in Abhängigkeit von weiteren Einstellungen. Da Sie den Text wie eingegeben brauchen, wählen Sie *text()*. Die Funktion wird ohne Parameter aufgerufen und liefert Ihnen den Text als *QString*.
11. Viele Klassen erben von übergeordneten Klassen Funktionen. Innerhalb der Dokumentation der Klasse werden zunächst nur die Member angezeigt und beschrieben, die in der Klasse neu hinzukommen. Daher kann es sein, dass die Funktion, die Sie suchen, nicht angezeigt wird, da sie bereits in einer anderen Klasse beschrieben wurde, dies gilt z.B. für die Memberfunktionen *show* und *hide* zum Anzeigen und Verstecken von Fenstern. Da diese Funktionen in vielen Klassen sinnvoll sind, wurden sie bereits in der Klasse *QWidget* implementiert, von der viele Klassen erben, die Fenster benutzen. Wenn Sie alle Funktionen einer Klasse sehen wollen, können Sie am Beginn der Beschreibung der jeweiligen Klasse auf *List of all members, including inherited members* klicken, dann werden diese angezeigt. Je nach Klasse können dies viele sein.

Sie haben gesehen, dass der Gebrauch der *Qt*-Referenz erstens nicht schwer ist, und zweitens schnell hilfreiche Informationen liefern kann, die Sie bei der Implementierung Ihrer eigenen Anwendungen unterstützen. Diese Beispiele haben außerdem verdeutlicht, wie wichtig und sinnvoll es ist, den Funktionen aussagekräftige Namen zu geben, damit andere sie auch finden und nutzen können.

Das Programm wird kommunikativer

Normalerweise steht dem Benutzer keine Konsole für die Ausgabe von Informationen zur Verfügung. Dies wäre auch nicht intuitiv, schließlich handelt es sich um ein grafisches Programm.

Erweitern Sie Ihr Programm, indem Sie zusätzlich zur Ausgabe durch *qDebug()* eine grafische Ausgabe mittels der Klasse *QMessageBox* realisieren. Schauen Sie sich dazu nochmal das Beispiel im Abschnitt *QLineEdit* auf Seite 120 an.

Die GUI zeichnet

1. Legen Sie in der Klasse *MainWindow* ein Attribut vom Typ *QGraphicsScene* an.
2. Ein Attribut vom Typ *QGraphicsView* haben Sie bereits in Ihrer GUI angelegt. Übergeben Sie die *QGraphicsScene* im Konstruktor der Klasse *MainWindow* an das GUI-Element *Graphics View* (siehe Beispiel 9.2.9, Seite 120).
3. Fügen Sie beim Klick auf den Knopf zusätzlich ein zufällig positioniertes Rechteck in die Szene ein.

Hinweis: Sie generieren eine zufällige Zahl zwischen 0 und 9 mit `rand()*9/RAND_MAX`.

Die GUI erhält eine Menüleiste

1. Öffnen Sie den *Qt Designer* für das *MainWindow* erneut und klicken in der obersten Zeile auf *Geben Sie Text ein*. Geben Sie *File* ein. Darunter dann nacheinander *Open*, *Save*, *Trenner hinzufügen*, *Exit*. Klicken Sie auf *Open* und nehmen rechts im Eigenschaftenfenster das Häkchen bei *enabled* weg. Machen Sie dasselbe für den Menüeintrag *Save*. Dies bewirkt, dass die Einträge zwar sichtbar sind, aber ohne Funktion und ausgegraunt.

2. Klicken Sie auf *Exit* und setzen bei *shortcut* den Wert *Alt+E*, indem Sie diese Tastenkombination drücken.
3. Wechseln Sie im untersten Fenster in den *Aktionseditor*. Wählen Sie mit der rechten Maustaste die Aktion *Exit* aus und in dem sich öffnenden Kontextmenü den Punkt *Slot anzeigen*. Wählen Sie die Aktion *triggered()*. Bei Menüeinträgen heißt es *Aktion*, verhält sich aber so, wie Sie es von den Signalen beim Pushbutton her kennen. Sie sollten jetzt in der zugehörigen Funktion sein.
4. Rufen Sie hier nur die Funktion *close()* auf. Testen Sie Ihr Menü, auch den Shortcut.
5. Erstellen Sie ein weiteres Menü *Scene* mit dem Menüeintrag *Clear Scene* und dem Shortcut *Alt-s*.
6. Im Aktionseditor erstellen Sie wieder den entsprechenden Slot. Nutzen Sie die Memberfunktion *clear()* der Klasse *QGraphicsScene*, um den Inhalt Ihrer *scene* zu löschen.
7. Erstellen Sie ein Menü *Info* mit dem Eintrag *About*, der eine Messagebox öffnet und eine kurze Beschreibung ausgibt. Nutzen Sie hierfür die Memberfunktion *about* von *QMessageBox*. Die Funktion *exec()* benötigen Sie dann nicht. *about()* erwartet als ersten Parameter einen Zeiger auf das übergeordnete Fenster, also *this*.

Ihre grafische Oberfläche enthält nun einige wesentliche Elemente, welche Sie von anderen Oberflächen her kennen.

9.4.2 Datenstruktur und Darstellung

Ihr Programm beinhaltet im Moment eine Klasse *MainWindow*, die die GUI definiert und verwaltet. In dieser Teilaufgabe werden Sie Klassen zur Verwaltung und Visualisierung der Karte erzeugen.

Die Teilaufgabe wird Sie schrittweise durch die Implementierung leiten. Die Schritte sind so gewählt, dass sie immer sofort getestet werden können. Sie werden zu jedem Teilschritt einen Knopf auf dem Hauptfenster anlegen, um den entsprechenden Test durchzuführen. Behalten Sie diese Testfunktionen bis zum Versuchsende bei und führen Sie diese regelmäßig aus, um die korrekte Funktionalität der Teillösungen sicherzustellen.

Hinweis: In den Vorlagen zu diesem Versuch finden Sie eine Datei *TestFunktionen.txt*. Darin finden Sie Vorschläge, wie Sie Ihre Testfunktionen gestalten können. Für die später noch beschriebene *AbstractMap* können Sie den Test komplett übernehmen, nachdem Sie alle Funktionen implementiert haben.

Eine Klasse repräsentiert eine Stadt

Erstellen Sie eine neue Klasse *City*, welche die Eigenschaften einer Stadt repräsentieren soll. Implementieren Sie auch eine Funktion zur Darstellung der Klasse in der GUI.

1. Navigieren Sie über das Menü zu *Datei* → *Neu* → *C++* → *C++ Class* und erstellen Sie eine neue Klasse *City*. Eine Stadt wird durch einen Namen sowie durch ihre Koordinaten *X* und *Y* definiert. Erstellen Sie die nötigen Attribute und einen passenden Konstruktor.
Nutzen Sie für den Konstruktor die Signatur `city(QString name, int x, int y)`.
2. Implementieren Sie eine Funktion `draw(QGraphicsScene& scene)`, die es der Klasse *City* ermöglicht, sich selbst als einen roten Punkt zu zeichnen. Geben Sie in dieser Funktion auch eine Debug-Information auf die Konsole aus.
3. Fügen Sie dem Hauptfenster einen weiteren Knopf mit der Beschriftung *Test Draw City* hinzu.
Nutzen Sie im *Qt Designer* auch die Eigenschaft *objectName*. Es ist sinnvoll, den Namen zu ändern, bevor der Slot angelegt wird, da *Qt* den Namen für die entsprechende Slot-Funktion nutzt.
Erstellen Sie einen Slot, der bei jedem Klicken des Knopfs aufgerufen wird.
4. Erstellen Sie in diesem Slot zwei Städte und zeichnen Sie diese in Ihrem *Graphics View*.
5. Stellen Sie sicher, dass beim Drücken des Knopfs auf der Konsole die Nachricht erscheint und die Stadt auf dem *Graphics View* zu sehen ist.

6. Ergänzen Sie die *draw()* Funktion so, dass auch der Name der Stadt dargestellt wird (siehe Seite 120).

Eine Klasse für die Karte

In dieser Teilaufgabe erstellen Sie zur Verwaltung der gesamten Karte eine Klasse *Map*. Diese wird die Städte und Straßen enthalten und später eine Schnittstelle zu den weiterverarbeitenden Algorithmen bilden. Damit die Klasse zu den Algorithmen passt, werden Sie diese von einer rein abstrakten Klasse ableiten, die in den Vorlagen zur Verfügung steht.

1. Legen Sie mit Hilfe des *Qt Creators* eine neue Klasse mit dem Namen *Map* an.
2. Erstellen Sie in der *Map* ein Attribut, das die Zeiger auf alle Städte speichern kann. Benutzen Sie hierbei entsprechenden Containerklassen von *Qt*.
3. Schreiben Sie eine Funktion *addCity(City*)*, die einen Zeiger auf eine Stadt abspeichert. Geben Sie hierbei den Namen der neuen Stadt auf der Konsole aus. Ergänzen Sie bei Bedarf die Klasse *City* um entsprechende Zugriffsfunktionen.
4. Implementieren Sie eine Funktion *draw(QGraphicsScene& scene)*, die das Zeichnen aller Städte veranlasst.
5. Legen Sie die *Map* als ein privates Attribut in der Klasse *MainWindow* an. Stellen Sie sicher, dass das Anlegen der Karte, das Hinzufügen von Städten und das Zeichnen der Karte funktioniert, indem Sie einen weiteren Test-Knopf dem Hauptfenster hinzufügen. (Siehe dazu auch *TestFunktionen.txt*)
6. Kopieren Sie mit dem Windows-Explorer die Dateien *abstractmap.h* und *abstractmap.cpp* in das Projektverzeichnis. Wählen Sie im *Qt Creator* im Kontextmenü des Projektverzeichnisses den Punkt *Existierende Datei Hinzufügen* und fügen Sie diese Dateien dem Projekt hinzu.
7. Lassen Sie die Klasse *Map* von der Klasse *AbstractMap* erben. Passen Sie bei Bedarf die Funktion *addCity()* an, sodass die Signatur der Vorgabe durch *AbstractMap* entspricht. Gehen Sie sicher, dass die Testfunktion noch einwandfrei funktioniert.

Eine Klasse repräsentiert eine Straße

Vervollständigen Sie nun die Datenstruktur mit der Klasse für die Straßen. Die Straßen enthalten die Information, welche Städte sie verbinden, und lassen sich außerdem zeichnen. Erweitern Sie auch die Klasse *Map* so, dass sie die Straßen verwalten und zeichnen kann.

1. Erstellen Sie eine Klasse für eine Straße. Nutzen Sie für den Konstruktor die Signatur *Street(City*, City*)*. Implementieren Sie eine *draw*-Funktion. Ergänzen Sie bei Bedarf die Klasse *City* um Zugriffsfunktionen.
2. Testen Sie die neue Klasse mit einem weiteren Test-Knopf auf dem Hauptfenster.
3. Implementieren Sie in der Klasse *Map* eine Möglichkeit, auch Straßen zu speichern. In der Klasse *AbstractMap* ist eine virtuelle Funktion zum Hinzufügen von Straßen bereits enthalten. Entfernen Sie die entsprechenden Kommentarzeichen und implementieren Sie die Funktion in der Klasse *Map*.
4. Testen Sie die neuen Funktionen mit einem weiteren Test-Knopf auf dem Hauptfenster.
In der Dokumentation der Funktion *addStreet()* der Klasse *AbstractMap* ist vorgegeben, dass sich eine Straße nicht hinzufügen lässt, wenn die Städte, die sie verbindet, nicht in der Karte vorhanden sind. Zeigen Sie durch entsprechende Ausgaben in der Testfunktion, dass Ihre Implementierung sich auch an diese Vorgabe hält.

9.4.3 Karte bearbeiten

In dieser Teilaufgabe widmen Sie sich erneut der GUI. Sie werden dem Benutzer eine Möglichkeit geben Städte einzutragen. Hierzu werden Sie einen Dialog programmieren, der die notwendigen Informationen vom Benutzer erfragt und die entsprechende Stadt erzeugt. Anschließend werden Sie die Funktionen aus der vorangegangenen Teilaufgabe nutzen, um die Stadt der Karte hinzuzufügen und zu zeichnen.

GUI aufräumen

Fügen Sie eine Checkbox ein, die die Test-Knöpfe unsichtbar schalten kann.

1. Fügen Sie dem Hauptfenster das Element *Check Box* hinzu.
2. Nutzen Sie den Slot *clicked()*. Die Klasse *QWidget* stellt die Funktionen *hide()* und *show()* zur Verfügung. *QCheckBox* erbt von *QWidget*. Mehr finden Sie in der *Qt*-Referenz.

Hinweis: Der Status der *Checkbox* wird mit der Funktion *isChecked()* erfragt.

Neue Stadt hinzufügen

Erzeugen Sie einen Dialog, der vom Benutzer die Information über eine neue Stadt erfragt.

1. Erzeugen Sie einen Dialog mit dem *Qt Creator* unter *Datei* → *Neu*, dann unter *Dateien und Klassen*, nicht unter *Projekte*, *Qt* → *Qt-Designer-Formularklasse*, auf der nächsten Seite den ersten Eintrag *Dialog with Buttons Bottom*. Es wird auch eine *ui*-Datei erzeugt. Fügen Sie diesem Fenster GUI-Elemente hinzu, sodass der Nutzer die notwendigen Informationen eingeben kann. Für dieses Fenster bietet sich das Layout *Objekte in Formularlayout anordnen* an.
2. Fügen Sie einen weiteren Knopf mit der Beschriftung *Add City* zum Hauptfenster hinzu. Erzeugen Sie im entsprechenden *clicked()*-Slot eine Instanz des Dialoges. Lassen Sie diesen mit der Funktion *exec()* erscheinen. Lesen Sie in der *Qt*-Dokumentation [6] nach, wie der Rückgabewert der Funktion definiert ist. Testen Sie mit Hilfe einer Debug-Ausgabe, wie sich der Rückgabewert beim Bestätigen und beim Ablehnen des Dialogs verhält.
3. Ergänzen Sie die Dialog-Klasse um eine Funktion, die die Benutzerdaten auswertet, eine Stadt erzeugt, und einen Zeiger darauf zurückgibt. Geben Sie die eingegebenen Daten zusätzlich auf der Konsole aus.
4. Erweitern Sie den Slot des Knopfs *Add City*, sodass die vom Benutzer eingegebene Stadt auf der Karte erscheint.

9.4.4 Karte einlesen

In dieser Teilaufgabe werden Sie das Programm so erweitern, dass eine vorhandene Karte eingelesen werden kann. Die Methoden, um diese aus einer Datei einzulesen, werden für Sie bereitgestellt. Zuvor werden Sie jedoch das Programm so vorbereiten, dass verschiedene Datenquellen genutzt werden können. Dem Versuch ist eine *abstrakte Klasse MapIo* beigelegt, die die Anforderungen an eine *Map*-Quelle beschreibt. Im wesentlichen enthält diese die Funktion *fillMap(AbstractMap*)*, die der Karte Städte und Straßen hinzufügt. Außerdem liegt dem Versuch auch eine Implementierung bei, die eine fest vorgegebene Karte erzeugt.

1. Fügen Sie die Dateien *mapio.h*, *mapio.cpp*, *mapionrw.h* und *mapionrw.cpp* in das Projektverzeichnis ein, und fügen Sie diese dann dem Projekt hinzu.
2. Legen Sie einen Zeiger vom Typ *MapIo* als Attribut der Klasse *MainWindow* an.
3. Erzeugen Sie im Konstruktor von *MainWindow* eine neue Instanz der Klasse *MapIoNrw* und weisen Sie diese dem neuen Attribut zu.
4. Erzeugen Sie einen weiteren Knopf auf dem Hauptfenster mit der Beschriftung *Fill Map*. Lassen Sie in dessen *clicked*-Slot die Karte von der Klasse *MapIo* füllen und zeichnen.

9.4.5 Einen Weg suchen

In dieser Teilaufgabe werden Sie für den Benutzer die Möglichkeit programmieren, einen Weg zwischen zwei Städten zu finden. Sie werden erst die benötigten Schnittstellen-Funktionen für den Suchalgorithmus ergänzen und durch einen umfangreichen Test sichergehen, dass diese korrekt funktionieren. Anschließend binden Sie den bereitgestellten Suchalgorithmus ein. Zum Schluss ergänzen Sie die Benutzeroberfläche.

Vollständige AbstractMap

Implementieren Sie die abstrakten Funktionen von *AbstractMap* vollständig. Benutzen Sie dabei den vorgegebenen Test, um sicherzustellen, dass die Funktionen korrekt arbeiten.

1. Entfernen Sie nacheinander die übrigen Kommentarzeichen in der Klasse *AbstractMap* und ergänzen Sie jeweils die Funktionen in der Klasse *Map*. Implementieren Sie diese jedoch nur soweit, dass das Programm erstellt und ausgeführt werden kann. Geben Sie, wo es notwendig ist, eine 0 oder eine leere Liste zurück.
2. Erzeugen Sie einen weiteren Knopf auf dem Hauptfenster mit der Beschriftung *Test Abstract Map* und fügen Sie in dessen *clicked*-Slot den vorgesehenen Test aus *TestFunktionen.txt* ein. Gehen Sie sicher, dass sie den Test durchführen können. Dieser muss komplett durchlaufen werden und sollte aktuell noch auf Fehler in der Implementierung hinweisen.
3. Implementieren Sie nun schrittweise die Funktionen. In der Klasse *AbstractMap* ist das gewünschte Verhalten der Funktionen dokumentiert. Starten Sie, während Sie die Funktionen implementieren, immer wieder den Test, um den aktuellen Implementierungsstand der Funktionen zu überprüfen. Ergänzen Sie bei Bedarf die Klassen *City* und *Street* um weitere Zugriffs- und Hilfsfunktionen.

Einbinden des Dijkstra

Dem Versuch liegt eine Implementierung des Dijkstra-Algorithmus³ bei. Testen Sie diesen erst mit zwei von Ihnen festgelegten Städten. Ergänzen Sie anschließend die grafische Oberfläche so, dass der Benutzer nach Wegen zwischen beliebigen Städten suchen kann.

1. Binden Sie den Dijkstra-Algorithmus ein. Nutzen Sie die Implementierung in den Dateien *dijkstra.h* und *dijkstra.cpp*. Fügen Sie einen Test-Knopf hinzu, der für zwei von Ihnen festgelegte Städte den Dijkstra-Algorithmus evaluiert. Geben Sie den gefundenen Weg auf der Konsole aus.
2. Zeichnen Sie die Straßen, die zu dem Weg gehören, in rot als breite Linie. Erstellen Sie dazu die Funktion `drawRed(QGraphicsScene& scene)` in der Klasse *Street*.
3. Fügen Sie dem Hauptfenster GUI-Elemente hinzu, die für eine freie Wegsuche notwendig sind und implementieren Sie die Suche.

9.4.6 Wahlpflichtaufgaben

Entwickeln Sie das Programm weiter, indem Sie mindestens zwei der folgenden Aufgaben bearbeiten.

Benutzereingabe überprüfen

Überprüfen Sie vor dem Einfügen von Städten über den Dialog, ob der Benutzer die Daten korrekt eingegeben hat. Geben Sie ihm im Fehlerfall die Möglichkeit, diese zu korrigieren, indem Sie den Dialog nochmals öffnen.

Wie man die Eingabe von Zahlen überprüft, haben Sie ja schon mehrfach angewendet. Leider gibt

³Dieser Algorithmus ist in der Lage, in einem Graphen einen kürzesten Weg zwischen zwei Knoten zu bestimmen. Siehe Anhang

es für *QString* die für Standardstrings definierte Funktion *find_first_not_of()* nicht. Wenn Sie den *QString* benutzen, um einen Standardstring zu erstellen, können Sie das damit sehr wohl. Schauen Sie sich dazu auch das Beispiel in der Klasse *string* in der C++-Referenz an, dann ist alles klar. *QString* bietet entsprechende Funktionen zum umwandeln, die *Qt*-Referenz hilft.

Straßen einfügen

Implementieren Sie einen Dialog, der es dem Benutzer ermöglicht, auch Straßen einzufügen, analog zur Eingabe von Städten.

Vorschläge für die Städte

Nutzen Sie zur Eingabe der Städte für die Wegsuche das GUI-Element *Combo Box*. Füllen Sie dieses mit den Städten aus der Karte, sodass der Benutzer daraus auswählen kann. Die Städteleiste liegt Ihnen ja bereits vor. *QComboBox* bietet Ihnen verschiedene Möglichkeiten, die Box zu füllen. Schauen Sie sich das in der *Qt*-Referenz an.

Karte von der Festplatte Einlesen

Lesen Sie die Karte aus einer Datei. In den Dateien *mapiofileinput.h* und *mapiofileinput.cpp* steht Ihnen eine Implementierung des Interfaces *MapIo* zur Verfügung, welches erlaubt, eine Karte von einer Datei einzulesen. Ersetzen Sie auf Knopfdruck das Attribut *mapIo* des *MainWindow* durch eine neue Instanz dieser Klasse. Nutzen Sie die Klasse *QFileDialog*, um vom Nutzer zu erfragen, welche Dateien eingelesen werden sollen. Die Dateien mit Städten und Straßen liegen ebenfalls dem Versuch bei.

Kleine und Große Städte

Erzeugen Sie zwei neue Klassen *BigTown* und *SmallCity* und lassen Sie diese von der Klasse *City* erben. Nutzen Sie für die Konstruktoren die gleiche Signatur wie die der zuvor realisierten Stadt. Überschreiben Sie die *draw* Funktion und ändern Sie diese so ab, dass die Städte unterschiedlich gezeichnet werden. Denken Sie daran, die Funktion in *City* als virtuell zu deklarieren. Testen Sie die funktionierende Polymorphie mit geeigneten Testfunktionen. Aktivieren Sie in der Datei *mapiofileinput.cpp*, bzw. in der Datei *mapionrw.cpp* die *CITY_EXTENSION* und laden Sie die Karte erneut.

Langsame und schnelle Straßen

Erzeugen Sie zwei neue Klassen *StateRoad* und *Motorway*. Verfahren Sie weiter analog zu dem vorhergehenden Abschnitt.

Dijkstra-Algorithmus advanced

Der Dijkstra-Algorithmus ist nicht beschränkt auf die Fähigkeit, einen Weg nach dem Kriterium *kürzeste Strecke* zu finden, tatsächlich ist das Optimalitätskriterium frei wählbar. Erweitern Sie Ihr Programm derart, dass es möglich wird, neben dem kürzesten auch den schnellsten Weg, d.h. den mit der kürzesten Fahrzeit zu ermitteln. Dazu müssen Sie die Implementierung des Dijkstra-Algorithmus nicht ändern.

Hinweise:

Die Implementierung nutzt die Funktion `AbstractMap::get_length` um die Kosten einer Straße zu ermitteln.

Als Durchschnittsgeschwindigkeiten eignen sich 50 km/h für langsame und 130 km/h für schnelle Straßen.

9.5 Zusammenfassung

In diesem Versuch haben Sie mit Hilfe der *Qt*-Klassenbibliothek ein vollständiges Softwareprojekt erstellt. Dieses enthält eine komplexe Datenstruktur und Algorithmen, die darauf arbeiten. Weiterhin haben Sie eine intuitiv bedienbare, grafische Bedienoberfläche realisiert. Sie haben gelernt, mit dem *Qt Creator* umzugehen und vorgegebene Algorithmen bzw. Teilimplementierungen in Ihren Quellcode einzubinden. Dabei sind Sie in kleinen, stets validierbaren Schritten vorgegangen. Das finale Ergebnis bietet somit nicht nur ein funktionierendes Programm, sondern enthält auch den gesamten Test-Code, der jederzeit ausgeführt werden kann, um die Korrektheit des Progammus nach Änderungen zu belegen. Ihr Programm zeichnet sich zudem durch einen modularen Aufbau aus, der sicherstellt, dass weitere Ergänzungen leicht realisierbar sind.

Anhang

Im folgenden finden Sie die aus Grundgebiete der Informatik 1¹ bekannten Implementierungen zu Quicksort und Dijkstra

Quicksort

Erklärung zu den Variablen und Funktionen:

- A beschreibt das zu sortierende Array
- l linke Grenze des zu sortierenden Arrays
- r rechte Grenze des zu sortierenden Arrays
- v Pivotelement
- $exchange$ vertauscht zwei Werte miteinander

```
1 void quick_sort (int A[], int l, int r)
2 {
3     int k;
4     if (r <= l)
5         return;
6     k = partition(A, l, r);
7     quick_sort(A, l, k - 1);
8     quick_sort(A, k + 1, r);
9 }
10
11 int partition (int A[], int l, int r)
12 {
13     int i, j, k, v;
14     k = r;
15     v = A[k];
16     i = l;
17     j = r - 1;
18     while (1)
19     {
20         while (A[i] <= v && i < r)
21             i++;
22         while (A[j] >= v && j >= l)
23             j--;
24         if (i >= j)
25             break;
26         else
27             exchange(A[i], A[j]);
28     }
29     exchange(A[i], A[k]);
30     return i;
31 }
```

Dijkstra

Vorbemerkungen:

- Voraussetzung: Datenstruktur für Graphen (*graph*) und Mengen (*set*) verfügbar

¹Beispiele und Erklärung aus dem Grundgebiete Informatik 1 Skript von Herrn Professor Leupers

- *graph* unterstützt Zugriff auf Knoten und Kanten
- *set* unterstützt übliche Mengenoperationen
- Variablen:
 - *dist[v]* bezeichnet den aktuell min. Abstand von Knoten *v* zum Startknoten
 - *cost(v, w)* bezeichnet Gewichtung der Kante {v, w}
 - Mengen GREEN, YELLOW und RED bezeichnen grüne und gelbe Knoten sowie rote Kanten
- Rote Kanten bilden stets einen Baum von aktuell kürzesten Pfaden vom Startknoten aus

```

1 void shortest_path(graph G = (V = {v0,...,vn}, E))
2 {
3     set GREEN = {}, YELLOW = {v0}, RED = {};
4     dist[v0] = 0;
5     while (YELLOW != {}) /* solange noch unbearbeitete Knoten */
6     {
7         v = MinDist(YELLOW); /* v ∈ YELLOW mit minimalem dist-Wert */
8         Insert(v, GREEN); /* GREEN = GREEN ∪ {v} */
9         Delete(v, YELLOW); /* YELLOW = YELLOW \ {v} */
10
11        for (w ∈ Succ(v)) /* für alle Nachfolger w von v */
12        {
13            if (!(w ∈ YELLOW ∪ GREEN)) /* neuer Knoten erreicht */
14            {
15                Insert({v, w}, RED);
16                Insert(w, YELLOW);
17                dist[w] = dist[v] + cost(v, w);
18            }
19            else if (w ∈ YELLOW) /* w erneut erreicht */
20            {
21                if (dist[v] + cost(v, w) < dist[w])
22                {
23                    Insert({v, w}, RED);
24                    e = PreviousEdge(w); /* vorher rote Kante zu w */
25                    Delete(e, RED);
26                    dist[w] = dist[v] + cost(v, w);
27                }
28            }
29        }
30    }
31 }
```

Literaturverzeichnis

- [1] R. Schneeweiss, *Moderne C++ Programmierung*.
Springer-Verlag Berlin Heidelberg, 2012.
- [2] Axel Rogat, *Objektorientiertes Programmieren mit C++ und JAVA*.
Bergische Universität Wuppertal, 1997.
- [3] Bernhard Lahres und Gregor Rayman, *Objektorientierte Programmierung*.
Galileo Computing, 2009.
<http://openbook.galileocomputing.de/oop>
- [4] Wikipedia-Artikel zu Vererbung:
[https://de.wikipedia.org/wiki/Vererbung_\(Programmierung\)](https://de.wikipedia.org/wiki/Vererbung_(Programmierung))
- [5] C/C++ Development User Guide
http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.cdt.doc.user/concepts/cdt_o_home.htm
- [6] Qt Referenz,
<http://doc.qt.io/qt-6/index.html>

