

Praktikum Informatik 1

Einführung in die C++ Programmierung (SS 2016)



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
Hilfreiche Einstellungen und Tricks für Eclipse	4
1 Eclipse-Umgebung – Das erste eigene Programm	7
1.1 Motivation	7
1.2 Beschränkungen	7
1.3 Qualifikationsziele	8
1.4 Einführung in Eclipse	8
1.5 Einführung in CDT	8
1.5.1 Erstellen des ersten Projektes	9
1.5.2 Compilieren	10
1.5.3 Ausführen	11
1.6 Testen und Debuggen	11
1.6.1 Erzeugung von debugfähigem Code	12
1.6.2 Debugger mit graphischer Menüführung	12
1.6.3 Funktionen eines Debuggers	12
1.6.4 Bedienung des Debuggers in Eclipse	14
1.7 Referenzen und Anmerkungen	16
1.8 Aufgaben	17
1.8.1 Rekursive Berechnung der Fibonacci-Zahlen	17
1.8.2 Iterative Berechnung der Fibonacci-Zahlen	17
2 Datenstrukturen und Operatoren	19
2.1 Theorie	19
2.1.1 Datenobjekte und Datentypen	19
2.1.2 Operatoren	21
2.1.3 Typumwandlungen	23
2.1.4 Datenstrukturen	24
2.1.5 Komplexe Datenstrukturen	28
2.1.6 Blöcke	30
2.1.7 Gültigkeitsbereiche	31
2.1.8 Referenzen und Zeiger	33
2.1.9 Speicherbereich und Sichtbarkeit	34
2.1.10 Dynamische Speichernutzung	34
2.2 Aufgaben	35
2.2.1 Datentypen und Typumwandlung	35
2.2.2 Strukturen	36
2.2.3 Felder	36

Inhaltsverzeichnis

3 Ablauf- und Kontrollstrukturen	39
3.1 Theorie	39
3.1.1 Zielsetzung und Einordnung	39
3.1.2 Funktionen	39
3.1.3 Einfache Kontrollstrukturen	45
3.1.4 Weitere Kontrollstrukturen: Schleifen	48
3.1.5 Testgetriebene Entwicklung	53
3.1.6 Übersichtlicher und verständlicher Code	54
3.1.7 Kurzeinführung in Doxygen	55
3.2 Aufgaben	57
3.2.1 Vorbereitung	58
3.2.2 Basisfunktionen	58
3.2.3 Die Ablaufsteuerung	60
4 Einführung in Klassen	63
4.1 Objektorientierte Programmierung	63
4.1.1 Objekte	63
4.1.2 Klassen	64
4.2 Klassen in C++	64
4.2.1 Methoden und Attribute	65
4.2.2 Der This-Zeiger	67
4.2.3 Konstruktoren und Destruktoren	68
4.2.4 Initialisierungslisten	72
4.2.5 Zugriffsbeschränkung	75
4.3 Parameter	77
4.3.1 Konstante Parameter und Call By Reference	77
4.3.2 Defaultparameter	77
4.4 Konstante Methoden	78
4.5 Statische Attribute und Methoden	79
4.5.1 Statische Attribute	79
4.5.2 Statische Methoden	80
4.6 Include-Wächter	81
4.7 Vorausdeklaration	82
4.8 Aufgaben	83
4.8.1 Die Klasse Vektor	84
4.8.2 Rotationsmatrix	84
4.8.3 Wie weit entfernt ist der Horizont?	84
5 Dynamische Datenstrukturen und Dokumentation	87
5.1 Qualifikationsziele	88
5.2 Praktische Anwendungen	88
5.2.1 Einfach verkettete Liste	88
5.2.2 Doppelt verkettete Liste	89
5.2.3 Warteschlange	90
5.2.4 Stapelspeicher	90
5.3 Zur Bearbeitung der Aufgaben	91
5.3.1 Anpassung eines Quellcodes an eine Code Convention	91
5.3.2 Einfache Studentendatenbank	92
6 Objektorientierte Techniken	93

6.1	Template	93
6.1.1	Template-Funktionen	94
6.1.2	Template-Klassen	94
6.2	Überladen	95
6.2.1	Überladen von Funktionen	96
6.2.2	Überladen von Operatoren	96
6.2.3	Überladung der Streamoperatoren	98
6.3	STL – Standard Template Library	99
6.3.1	Vektor	100
6.4	Aufgaben	101
6.4.1	Template	101
6.4.2	STL	102
7	Vererbung und Polymorphie	103
7.1	Vererbung — Hierarchien im Klassenkonzept	103
7.2	Polymorphismus	106
7.2.1	Späte Bindung	107
7.3	Abstrakte Klassen und Interface-Klassen	109
7.4	Virtuelle Destruktoren	111
7.5	Aufgaben	111
7.5.1	Ein einfacher Taschenrechner	112
8	GUI-Programmierung mit <i>Qt</i>	113
8.1	Qt Referenz	113
8.1.1	Beispielprogramm	114
8.1.2	Signals und Slots	114
8.1.3	QString	115
8.1.4	QMessageBox	115
8.1.5	QPushButton	116
8.1.6	QRadioButton	117
8.1.7	QLabel	119
8.1.8	QLineEdit	119
8.1.9	QMenu	121
8.1.10	QGraphicsView	124
8.1.11	QMainWindow	125
8.1.12	QDialog	125
8.2	Aufgaben	125
8.2.1	Eine neue Applikation	126
8.2.2	Datenstruktur und Darstellung	127
8.2.3	Karte bearbeiten	129
8.2.4	Karte einlesen	130
8.2.5	Einen Weg suchen	130
8.2.6	Wahlpflichtaufgaben	131
8.3	Zusammenfassung	132
Anhang		133
Literaturverzeichnis		137
Index		139

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 oder Template–Verwendung kennenlernen, die für effiziente Lösungen erforderlich sind. Um Ihnen die Installation der Entwicklungsumgebung auch auf privaten Rechnern zu vereinfachen, bietet dieses erste Kapitel eine Installationsanleitung sowie ein Testverfahren zur Verifikation, ob die Installation der einzelnen Komponenten gelungen ist.

Noch vor dem ersten Versuch, und damit auch vor dem ersten Termin zum betreuten Programmieren, sollten Sie sich im Groben mit der Entwicklungsumgebung vertraut machen und die Ihnen gegebenen Hinweise verinnerlichen.

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 01 bis 06 den ersten Versuchsblock dar. Die Versuche 07 und 08 definieren den zweiten Versuchsblock. Werden die Fragen während eines Testates korrekt beantwortet, erhalten Sie hierüber eine Bestätigung. Die Qualität der Antworten und der Implementierung entscheiden dabei über die Bewertung des Testats. Das Ergebnis wird auf der Testatkarte, welche Ihnen ausgehändigt wird, sowie zusätzlich in einer zentralen Liste festgehalten.

Abnahme und Bewertung eines Aufgabenblocks

Sie müssen sich für die Testate über die Umfragen "Anmeldung zum Testat 1" und "Anmeldung zum Testat 2" im L²P anmelden. Der Anmeldeschluss liegt am 27. Mai 2016 für das erste Testat bzw. am 24. Juni 2016 für das zweite Testat. Eine spätere Anmeldung ist aus organisatorischen Gründen nicht möglich. Unentschuldigtes Fernbleiben nach verbindlicher Anmeldung führt zum Ausschluss vom Praktikum.

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

Erfolgreiche Praktikumsteilnahme

- 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: Korrekte Umsetzung; gute Kommentierung; gute Erklärung des Quellcodes
- 2 Punkte: Kleine Fehler bei der Umsetzung; Erklärung des Quellcodes ist befriedigend
- 1 Punkt: 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) können beide Testate je einmal wiederholt werden.

Sind Sie Ihrer 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.

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. In diesem Fall wird das Testat mit 0 Punkten bewertet. Eine Wiederholung ist einmalig möglich.

Semesterübersicht

Um Ihnen einen Überblick über das Semester zu ermöglichen, sind in dem Kalender auf Seite 3 alle Termine dieses Praktikums eingetragen. Die Wochen, an denen Testattermine vergeben werden, sind in der Übersicht mit "T1" und "T2" für das erste bzw. das zweite Testat gekennzeichnet. Die Angabe zu den Versuchen (V1 bis V8) hinter den Terminen gibt eine grobe Orientierung zum Arbeitsaufwand der einzelnen Versuche. Bitte beachten Sie auch die Ausweichtermine für die Gruppen, deren Termine auf vorlesungsfreie Tage fallen. Alle Angaben hierbei ohne Gewähr.

	April	Mai	Juni	Juli
1	Fr Semesterbeginn	1 So Maifeiertag	1 Mi T1 - Mi1, Mi2, Mi3 (V6)	1 Fr
2	Sa	2 Mo Mo1, Mo2, Mo3 (V3)	2 Do	2 Sa
3	So	3 Di Di2, Di3 (V3)	3 Fr	3 So
4	Mo	4 Mi Mi1, Mi2, Mi3 (V3)	4 Sa	4 Mo T2 - Mo1, Mo2, Mo3 (V8)
5	Di	5 Do Christi Himmelfahrt	5 So	5 Di T2 - Di1, Di2, Di3 (V8)
6	Mi	6 Fr	6 Mo T1 - Mo1, Mo2, Mo3 (V7)	6 Mi T2 - Mi1, Mi2, Mi3 (V8)
7	Do	7 Sa	7 Di T1 - Di1, Di2, Di3 (V7)	7 Do
8	Fr	8 So	8 Mi T1 - Mi1, Mi2, Mi3 (V7)	8 Fr
9	Sa	9 Mo Mo1, Mo2, Mo3 (V4)	9 Do	9 Sa
10	So	10 Di Di1, Di2, Di3 (V4)	10 Fr	10 So
11	Mo	11 Mi Mi1, Mi2, Mi3 (V4)	11 Sa	11 Mo T2 - Mo1, Mo2, Mo3
12	Di	12 Do	12 So	12 Di T2 - Di1, Di2, Di3
13	Mi	13 Fr	13 Mo T1 - Mo1, Mo2, Mo3 (V7)	13 Mi T2 - Mi1, Mi2, Mi3
14	Do	14 Sa Einführungsvorlesungsbeginn	14 Di T1 - Di1, Di2, Di3 (V7)	14 Do
15	Fr	15 So Pfingstsonntag	15 Mi T1 - Mi1, Mi2, Mi3 (V7)	15 Fr
16	Sa	16 Mo Pfingstmontag	16 Do	16 Sa
17	So	17 Di Pfingst-, Exkursionswoche	17 Fr	17 So
18	Mo	18 Mi Pfingst-, Exkursionswoche	18 Sa	18 Mo
19	Di	19 Do Pfingst-, Exkursionswoche	19 So	19 Di
20	Mi	20 Fr Pfingst-, Exkursionswoche	20 Mo Mo1, Mo2, Mo3 (V8)	20 Mi
21	Do	21 Sa	21 Di Di1, Di2, Di3 (V8)	21 Do Vorlesungsende
22	Fr	22 So	22 Mi Mi1, Mi2, Mi3 (V8)	22 Fr
23	Sa	23 Mo Mo1, Mo2, Mo3 (V5)	23 Do	23 Sa
24	So	24 Di Di1, Di2, Di3 (V5)	24 Fr Anmeldeschluss Testat 2	24 So
25	Mo	25 Mi Mi1, Mi2, Mi3 (V5)	25 Sa	25 Mo
26	Di	26 Do Fröhleichnam	26 So	26 Di
27	Mi	27 Fr Anmeldeschluss Testat 1	27 Mo T2 - Mo1, Mo2, Mo3 (V8)	27 Mi
28	Do	28 Sa	28 Di T2 - Di1, Di2, Di3 (V8)	28 Do
29	Fr	29 So	29 Mi Dies ab 13.00	29 Fr
30	Sa	30 Mo T1 - Mo1, Mo2, Mo3 (V6)	30 Do T2 - Mi1, Mi2, Mi3 (V8)	30 Sa
		31 Di T1 - Di1, Di2, Di3 (V6)		31 So

Beachten Sie die Ausweichtermine für Dies und vorlesungsfreie Tage.
Über kurzfristige Änderungen werden Sie über Campus und im L2P informiert.

Hilfreiche Einstellungen und Tricks für Eclipse

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

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

Hier können Sie auch zusätzlich das Interval festlegen, in dem Eclipse automatisch geöffnete Dateien im Hintergrund sichert. Geben Sie die Anzahl Minuten unter *Workspace save interval* an.

Tabulatorgröße

Zur besseren Übersicht und um eine Editor übergreifende Darstellung zu gewährleisten, bietet es sich an die Tabulatorgröße auf 4 Leerzeichen einzustellen.

Gehen Sie hierzu abermals in die Eclipse-Einstellungen:

Window → Preferences → General → Editors → TextEditors

Tragen Sie im zugehörigen Feld *Displayed tab width* "4" ein.

Optional können Sie zusätzlich den Punkt *Insert spaces for tabs* auswählen.

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* und *ok*.

Importfunktion

In vielen Versuchen im Laufe des Praktikums bekommen Sie vorgefertigte Projekte, die um eigenen Code ergänzt werden sollen. Um diese Vorlagen zu nutzen, wird die Import-Funktion von Eclipse genutzt. Gehen Sie hierzu wie folgt vor:

- *File → Import*
- *General → Existing Projects Into Workspace*
- Wählen Sie dort bei *Select root directory* das Vorlagenverzeichnis des Versuchs aus.
- Wählen Sie die Option *Copy Projects into Workspace* aus.

Bei einigen Vorlagen handelt es sich nicht um vorgefertigte Projekte, sondern nur um Dateien, zu erkennen an den fehlenden *.project* und *.cproject* Dateien. Solche Dateien können nur in ein bestehendes Projekt importiert werden. Gehen Sie hierzu wie folgt vor:

Erstellen Sie ein neues Projekt

- *File → New → C++ Project*

- Tragen Sie den Projektnamen ein und wählen Sie *Empty Project* und *MinGW GCC*, dann *ok*.
- Im Windows-Explorer ins 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*.

Die Codevorlagen finden Sie entweder im L²P oder verwenden Sie den folgenden Pfad zum Importverzeichnis im CIP-Pool.

P:\UserGrp\PI1\Vorlagen

Perspektive wiederherstellen

In Ecplise 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.

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

Window → Perspective → ResetPerspective

1 Eclipse-Umgebung – Das erste eigene Programm

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 um so 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 Beschränkungen

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.

1.3 Qualifikationsziele

In diesem Versuch werden alle Schritte aufgezeigt, um mittels Eclipse und dem Plugin CDT C bzw. C++ Programme zu entwickeln. Diese Schritte umfassen den vollen Programmierprozess:

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

Sowohl Eclipse als auch CDT sind auf jedem Rechner des Rechner-Pools installiert.

1.4 Einführung in Eclipse

Alle Projekte und die dazugehörigen Einstellungen und Dateien werden in dem *workspace* gespeichert; das dazugehörige Verzeichnis fragt Eclipse bei jedem Programmstart ab, siehe 1.1. Bestätigen Sie die Standardeinstellung einfach mit Ok.

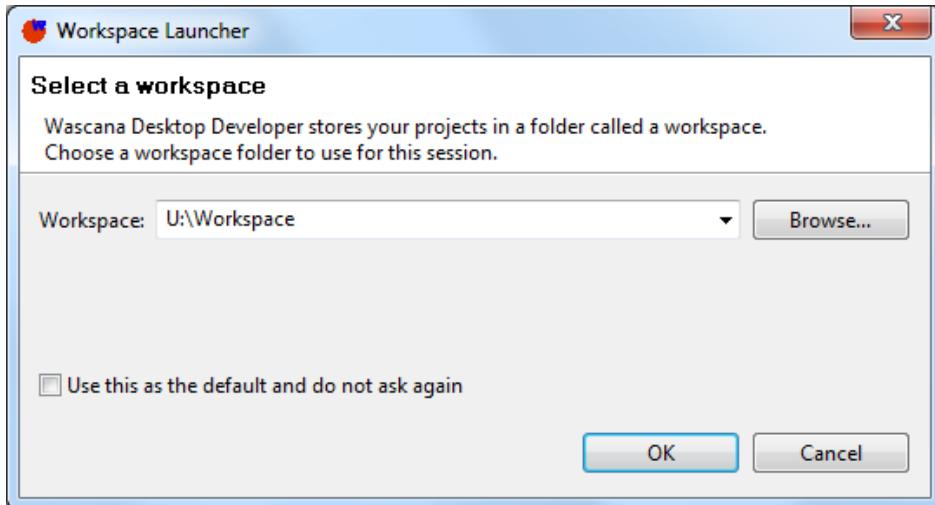


Abbildung 1.1: Einstellen des Verzeichnisses des Workspace

Im darauf folgenden Willkommen-Bildschirm wählen Sie direkt den Pfeil rechts aus: *Workbench - Go to the workbench*.

Da Eclipse ursprünglich für die Programmierung von Java konzipiert wurde, muss evtl. zuerst eine andere **Perspektive** geöffnet werden. Eine Perspektive beschreibt, welche Elemente (z.B. Editor, Projektübersicht, Statusanzeige) wo auf der Arbeitsoberfläche erscheinen. Um zur C/C++ - Perspektive zu wechseln, wählen Sie im Menu unter Window: Open Perspective: Other und dann C/C++ aus. Schon kann das erste C/C++ Projekt begonnen werden!

1.5 Einführung in CDT

Die soeben geöffnete Perspektive aktiviert die “Eclipse C/C++ Development Tools (CDT)”. Dazu gehört

- Ein Editor, der speziell an C/C++ angepasst ist, d. h. er unterstützt Syntax-Hervorhebung, automatisches Ergänzen uvm.
- Eine vollständige Integration der Projektverwaltung, also eine Zusammenfassung aller Dateien eines Projektes und automatisches Kompilieren und Linken des gesamten Projektes.
- Eine einfache und schnelle Navigation durch das geschriebene Programm durch einen internen Index von allen Deklarationen, Definitionen, Variablen, Instanzen und Funktionsaufrufen.
- Ein integrierter Debugger, mit dem der Programmablauf direkt aus Eclipse auf Fehler untersucht werden kann.

Im Folgenden werden die Vorteile des CDT gegenüber einem „einfachen“ Editor anhand der Erstellung eines kleinen Programmes zur Berechnung der Fakultät $n!$ gezeigt.

1.5.1 Erstellen des ersten Projektes

Um mit Hilfe von Eclipse eine ausführbare Datei zu erstellen, legen Sie zunächst ein neues *C++ Project* an, indem Sie unter dem Menü *File* → *New* die Option *C++ Project* wählen. Geben Sie in dem Wizard den Projektnamen („Fakultaet“) 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 Wizard¹ mit *Finish* und in der Projektübersicht links sollte das leere C/C++ Projekt *Fakultaet* 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 leerer 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/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 fakultaet(int n)
4 {
5     if (n <= 1)
6     {
7         return 1;
8     }
9     else
10    {
11        return (n * fakultaet(n-1));
12    }
13 }
```

¹Eclipse 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.

```

14
15 int main(int argc, char **argv)
16 {
17     int fak[10];
18     std::cout << "Fakultaeten:" << std::endl;
19     for (int i=0; i<10 ; i++)
20     {
21         fak[i]=fakultaet(i);
22         std::cout << "Die Fakultaet von " << i << " ist " << fak
23             [i] << std::endl;
24     }
25     return 0;
}

```

Die Zeilen 1 und 2 werden in späteren Versuchen erklärt und können hier ignoriert werden (siehe Kapitel 2.1.2). Das Programm berechnet mit Hilfe von Rekursion die Fakultäten von 0 bis 9.

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

1.5.2 Compilieren

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

Durch die Auswahl *Project → Build All* 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 Fakultaet ****

make all
Building file: ../main.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"main.d" -MT"main.d"
    -o"main.o" "../main.cpp"
Finished building: ../main.cpp

Building target: Fakultaet

```

```
Invoking: GCC C++ Linker
g++ -o"Fakultaet" ./main.o
Finished building target: Fakultaet
```

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
      -MMD -MP -MF"main.d" -MT"main.d" -o"main.o" "../main.cpp"
```

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

```
g++ -o"Fakultaet" ./main.o
```

sämtliche benötigten Object-Dateien des Projektes zur Datei *Fakultaet* 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!

1.5.3 Ausführen

Falls Ihr Programm fehlerfrei ist, können Sie es mit einem Klick der linken Maustaste auf das Projekt *Fakultaet* im linken Bereich und der Auswahl *Run As → Run Local C/C++ Application* ausführen. Im unteren Bereich sollten Sie jetzt unter der Konsole die richtige Ausgabe von *Fakultaet* sehen können:

```
Fakultaeten:
Die Fakultaet von 0 ist 1
Die Fakultaet von 1 ist 1
Die Fakultaet von 2 ist 2
Die Fakultaet von 3 ist 6
Die Fakultaet von 4 ist 24
Die Fakultaet von 5 ist 120
Die Fakultaet von 6 ist 720
Die Fakultaet von 7 ist 5040
Die Fakultaet von 8 ist 40320
Die Fakultaet von 9 ist 362880
```

1.6 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 getester 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 Programmentwicklung 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.

1.6.1 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 kompiliert 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 Menu 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++ -O0 -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 verlangsamen den normalen Programmablauf. Daher sollte man bei der Erzeugung der finalen Version (*Release*-Version) eines Programms auf diese Informationen verzichten.

1.6.2 Debugger mit graphischer Menüführung

Der Debugger ist in CDT integriert, besitzt jedoch eine andere Perspektive als die zur C/C++ Entwicklung.

Der integrierte Debugger wird ähnlich wie das Ausführen des Programms mit einem Klick der linken Maustaste auf den Projektnamen, dann aber mit der Auswahl *Debug As → Debug Local C/C++ Application* gestartet. Da der Debugger eine andere Perspektive als die zur C/C++ Entwicklung besitzt, sollte die folgende Frage, ob diese geöffnet werden soll, mit *Yes* beantwortet werden.

1.6.3 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. Bei einigen Debuggern können diese Werte sogar manuell verändert werden. Bei Eclipse ist dies derzeit noch nicht möglich.

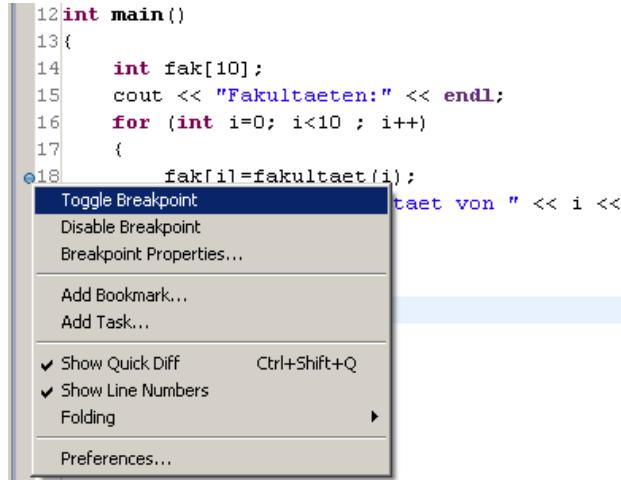


Abbildung 1.2: Hinzufügen eines Haltepunktes mit der rechten Maustaste oder Doppelklick auf die angezeigte Stelle.

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.

Call-Stack

In C/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.

1.6.4 Bedienung des Debuggers in Eclipse

Anhand des Fakultät-Beispielprogramms soll die Bedienung der Debug-Funktionen in der Eclipse-Umgebung erläutert werden.

Die Debug-Perspektive bietet separate Fenster für den Quellcode, Variablen und Call-Stack. Um die Programmausführung im Debug-Modus zu starten, klicken Sie auf das Käfer-Symbol (Markierung 1 in der Abbildung 1.3). Aufgrund des gesetzten Haltepunktes stoppt der Debugger die Ausführung in Zeile 18 des Programms.

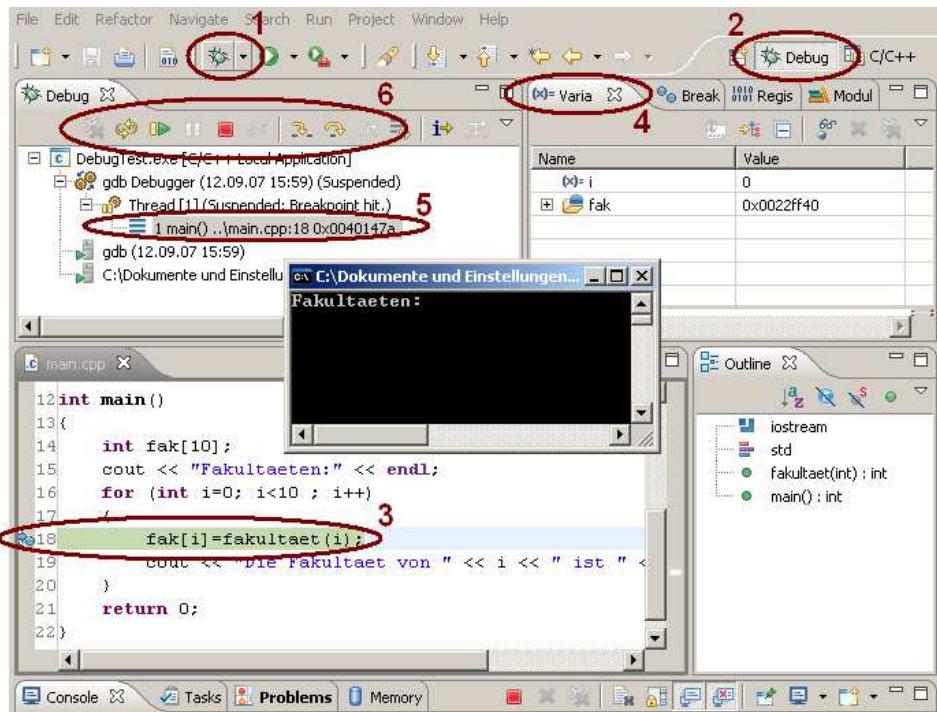


Abbildung 1.3: Das Fenster der Debug-Perspektive.

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

- Markierung 3 zeigt, dass der Programmablauf in Zeile 18 angehalten wurde.
- Markierung 4 zeigt das Fenster, in dem die Werte lokaler und globaler Variablen überprüft werden können. Die Variable *i* zeigt wie erwartet den Wert 0. Der Reiter *Breakpoints* dient zur Aktivierung bzw. Deaktivierung vorhandener Haltepunkte.
- Markierung 5 zeigt den Inhalt des Call-Stacks. Bisher ist nur die *main*-Funktion mit der Position "Zeile 18" abgelegt. Beim anschließenden Aufruf des Unterprogramms *fakultaet(i)* wird auch diese Funktion auf dem Stack angezeigt.
- Markierung 6 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.

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 *for*-Schleife einmal durchlaufen, bevor der Debugger das Programm wieder in Zeile 18 anhält. Die Zählvariable *i* hat nun den Wert 1, siehe Abbildung 1.5.

(x)= Variables		Breakpoints	Registers	Modules
Name	Value			
(x)= i	1			
+ Fak	0x0022ff40			

Abbildung 1.5: Anzeige des Wertes der Zählvariable *i*

Durch Klick auf den *Step-Over*-Button (Markierung 4, Tastenkombination F6) wird die Funktion *fakultaet(1)* ausgeführt. Nach weiteren Schritten ist das Programm wieder in Zeile 18 angelangt, wobei *i* jedesmal hochgezählt wird.

Im Gegensatz zu *Step-Over* ermöglicht es der *Step Into*-Button (Markierung 3, Tastenkombination F5) in die Funktion zu verzweigen. Wie in Abbildung 1.6 zu sehen, befindet sich das Programm nun in Zeile 6. Der Call-Stack zeigt damit nun auch die Funktion *fakultaet()* an. Die Variabeldarstellung zeigt den übergebenen Parameter *n* mit dem Wert 2 an. Der *Step-Return*-Button (Markierung 5, Tastenkombination F7) ermöglicht die sofortige Ausführung aller Anweisungen des Unterprogramms und Rücksprung zur aufrufenden Funktion. Das Debuggen kann jederzeit durch Klick auf den *Terminate*-Button (Markierung 2, Tastenkombination Strg+F2) beendet werden.

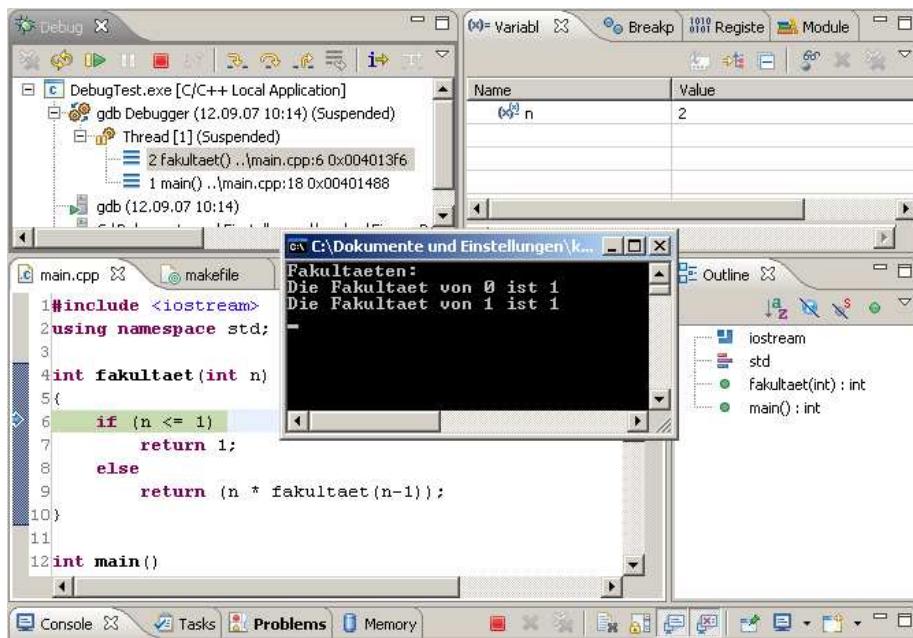


Abbildung 1.6: Debug-Sicht nach *Step Into*.

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 18.

1 Eclipse-Umgebung – Das erste eigene Programm

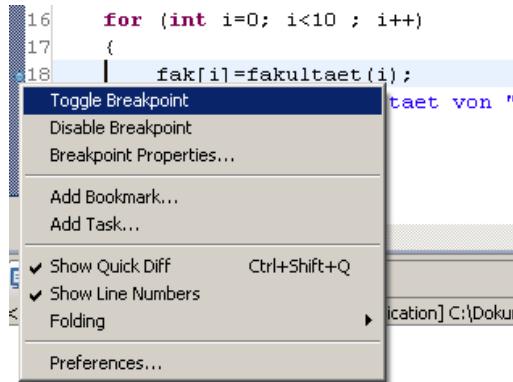


Abbildung 1.7: Entfernen eines Haltepunktes mit Hilfe der rechten Maustaste.

Nun wird ein neuer Haltepunkt in Zeile 21 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 21. Das Variablen-Fenster zeigt nun, wie das Feld *fak[]* mit den Fakultäten von 0 bis 9 gefüllt ist (Abbildung 1.8).

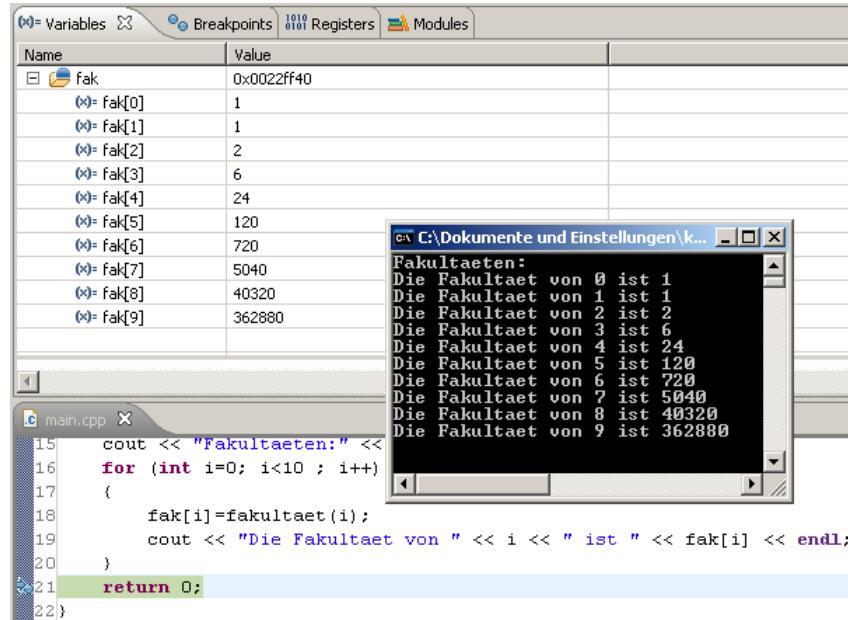


Abbildung 1.8: Neuer Haltepunkt am Ende des Programms.

1.7 Referenzen und Anmerkungen

Die bisherigen Schritte geben eine kurze Einführung in Eclipse bzw. in das CDT, ohne jedoch alle Möglichkeiten genau zu erklären. Die beste Methode, sich mit der IDE auseinanderzusetzen, wird das regelmäßige Benutzen und Ausprobieren der verschiedenen Optionen sein. 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 CDT. Ein sehr guter Einstiegspunkt ist der *C/C++ Development User Guide*[6]. Sehr gut ist auch die

Einführung von IBM [7]. Sie enthält ein schönes Beispiel, welches etwas über unser *Fakultaet* hinausgeht, ohne zu komplex zu sein. Allerdings verwendet es sehr stark die objektorientierten Möglichkeiten von C++, welche erst im Verlauf des Praktikums eingeführt werden.

Sehenswert ist auch das “Webinar”, also eine Online-Präsentation von Doug Schaefer [8], dem Hauptentwickler der CDT, auch wenn es sich nicht um die aktuellste Version von CDT behandelt.

1.8 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, ...

1.8.1 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.

1.8.2 Iterative Berechnung der Fibonacci-Zahlen

1. Laden Sie die Codevorlage aus dem L2P herunter. Dieser ZIP-Datei beinhaltet bereits die Vorlagen für alle weiteren Versuche. Speichern Sie diese Datei in einem eigenen Ordner auf Ihrem *U*-Laufwerk (z.B. Vorlagen) und entpacken Sie sie dort (nicht in den *workspace* von *eclipse*).
2. Importieren Sie das Projekt *Versuch01Teil2* (siehe Seite 4).
3. 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.
4. Vergleichen Sie für $n = 42$ die Geschwindigkeit von Ihrem Programm mit dem aus dieser Aufgabe. Gibt es einen Unterschied? Wenn ja, warum?
5. Zusatzaufgabe: Ab der 47-ten Fibonacci-Zahl geben beide Programme ein falsches Ergebnis aus. Warum? Wie kann man das beheben? Was ist die größte mit diesem Programm berechenbare Fibonacci-Zahl?

2 Datenstrukturen und Operatoren

Im ersten Versuch haben Sie die Entwicklungsumgebung *Eclipse* kennengelernt. In diesem Versuch werden Sie einen genaueren Blick auf Variablen, ihre Datentypen und ihre Gültigkeitsbereiche werfen und die Geheimnisse der Umwandlung von Datentypen erfahren. Sie werden lernen, wie man Variablen über Operatoren miteinander verknüpft und welche Möglichkeiten es gibt, um zusammengehörende Daten geeignet abzuspeichern. Darauf hinaus werden Sie Funktionen kennenlernen, mit denen bestimmte Teilaufgaben einfach gekapselt und auf diese Weise wiederholt aufgerufen werden können.

Ziel dieses Versuches ist es, Ihnen die grundsätzlichen Werkzeuge der Programmiersprachen *C* bzw. *C++* mit auf den Weg zu geben. Bei diesen grundsätzlichen Elementen unterscheiden sich diese beiden Sprachen nämlich noch so gut wie nicht. Die Unterschiede (oder genau genommen die Erweiterungen von *C++*) werden Sie erst in den späteren Versuchen kennen lernen. Am Ende dieses Versuches sollten Sie wissen, was

- Datentypen
- Operatoren
- Felder und Strukturen

sind und wie man diese einsetzen kann.

2.1 Theorie

2.1.1 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.

Variablen

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 auf 32-Bit-Architekturen. Beachten Sie, dass der tatsächliche Wertebereich system- und prozessorabhängig ist. Die Standard-Headerdatei "limits" definiert die tatsächlichen Wertebereiche.

Datenyp	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
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

Zum Typ *char* ist zu sagen, dass er dem Abspeichern eines einzelnen Zeichens aus dem Zeichensatz der Maschine dient. Da aber jedem Zeichen über den *ASCII-Code* ein bestimmter Wert der Länge 1 byte zugeordnet ist, entspricht ein Zeichen einer Zahl zwischen -128 und 127. Dem Großbuchstaben A ist beispielsweise der ASCII-Code 65 zugeordnet. Es ist daher egal, ob Sie schreiben *zeichen* = 'A' oder *zeichen* = 65, wenn *zeichen* vom Datentyp *char* ist.

Deklarationen

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 *Deklaration*. Die einfachste Vereinbarung beginnt mit dem Schlüsselwort des Datentyps, gefolgt von dem Namen der Variable:

```
// int value; // Deklaration 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 unterbinden. In diesem Praktikum sind Sie daher dazu angehalten, allen Variablen, die Sie einführen, auch einen Wert zuzuweisen.

```
// int value = 2014; // Deklaration 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 deklarieren, um potentielle Fehlerquellen zu vermeiden und leicht Kommentare einzufügen zu können. Die Variablen werden durch Kommas getrennt:

```
// mehrere Deklarationen vermeiden!!!
char letter1 = 'A', letter2, letter3 = 'P';
```

Beachten Sie, dass reines C die Deklaration einer Variablen nur am Anfang einer *Funktion* oder eines *Blocks* erlaubt. C++ ist hier weitaus anarchischer und erlaubt die Deklaration praktisch überall im Programm, auch mitten in einer Funktion oder in einem Block. Welchen Stil man bevorzugt, ist Geschmackssache. Manche Programmierer finden es übersichtlicher, alle verwendeten Variablen direkt am Anfang der Funktion im Überblick zu haben, andere

bevorzugen die Deklaration einer Variablen erst unmittelbar vor (oder sogar praktisch mit!) ihrer ersten Benutzung, so dass man die Variable erst dann sieht, wenn man sie wirklich braucht. Dies ist bei der Programmierung in C++ üblich.

Sogenannte Definitionen unterscheiden sich von Deklarationen. Eine Deklaration legt die Eigenschaft einer Variablen (also Datentyp, Speicherbedarf usw.) fest, eine Definition sorgt darüber hinaus dafür, dass Speicherplatz bereitgestellt wird. In allen Quelldateien für ein Programm darf es nur eine Definition für eine Variable geben.

Konstanten

Manche Dinge ändern sich nie, z.B. die Zahl π . Stellt man einer Variablen Deklaration 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 Deklaration immer initialisiert werden muss:

```
const double PI = 3.1415; // Deklaration der Konstanten PI
const double quote;      // Fehler: uninitalisierte Konstante
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 deklarieren!

2.1.2 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

2 Datenstrukturen und Operatoren

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 Negationsoperator wird in C++-Programmen häufig verwendet, um den Wert einer Variablen auf Null zu prüfen. Die Anweisung

```
!ok
```

ist äquivalent zu

```
ok == 0
```

Beachten Sie auch, dass in C++ ein logischer Ausdruck wahr ist, wenn er ungleich Null ist (und nicht, wenn er gleich eins ist). Also ist

```
ok
```

äquivalent zu

```
ok != 0
```

und eben nicht zu

```
ok == 1
```

Inkrement- und Dekrement-Operator

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

```
int value = 0;  
value++; // value ist jetzt 1  
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

```
value1 = value2++;
```

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

```
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`¹ oder von `cin` in den Speicher einer Variablen geschickt. Dazu muss zunächst die benötigte Funktionalität geladen werden:

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

Danach kann mit `cout` z.B. ein Text ausgegeben werden:

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

oder `cin` wartet auf eine Eingabe von einem `int`.

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

2.1.3 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.

```
|| int value1 = 42;
|| double value2 = 42.42;
|| // Zwischenwert von value1 wird zu double
|| 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:

```
|| int value1;
|| double value2 = 42.42;

|| 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:

¹Wie 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.

```

int value1;
double value2 = 42.42;

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.

2.1.4 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, die versehentlich weggeworfene Adresse der neuen Freundin wiederzufinden. Sie dürfen Ihren Müllhaufen daher guten Gewissens als Datenstruktur *Heap* betrachten.

Im Folgenden werden wir zwei einfache Datenstrukturen vorstellen, mit denen Sie in dieser und den folgenden Übungen arbeiten werden: dem *Feld* und der *Struktur*.

Arrays

Ein *Array* oder *Feld* 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 Variablenamen einfach die gewünschte Größe des Feldes, eingeschlossen in eckige Klammern, hinzu. Auf die gleiche Weise wird auch auf ein bestimmtes Element des Feldes zugegriffen:

```
// Deklaration eines Feldes mit 10 int-Werten
array[3] = 5; // Dem Element mit Index 3 den Wert 5 zuweisen
std::cout << array[1]; // Ausgeben des Elementes 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:

```
// Deklaration eines Feldes mit 17 Zeilen und 16 Spalten
std::string weh[17][16];
```

Beachten Sie, dass die Indizierung mit der Notation $feld[i][j]$ und nicht mittels $feld[i,j]$ 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 Deklaration initialisiert werden. Die Initialisierungsliste ist in geschweifte Klammern eingeschlossen, die Werte für die einzelnen Feldelemente werden durch Kommata voneinander getrennt:

```
// Initialisierung eines Feldes
int array[10] = { 2, 3, 8, 2, 5, 6, 2, 4, 10, 3 };
```

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

```
// Initialisierung eines zweidimensionalen Feldes
int array[2][5] = { { 2, 3, 8, 2, 5 }, // 1. Zeile
                    { 6, 2, 4, 10, 3 } }; // 2. Zeile
```

String (Zeichenkette)

Ein *String (Zeichenkette)* ist die Darstellung einer Folge von Zeichen. Er ist ein Spezialfall eines *Arrays*, nämlich ein eindimensionales *Array von Zeichen (char)*. In jedem Element des Arrays wird genau ein Zeichen abgelegt. Alle Elemente zusammen ergeben die Zeichenkette, z.B. einen Satz. Damit eignen sich Zeichenketten zum Speichern von Texten, Meldungen, usw.. Dabei kann mit dem Array auf jeden Buchstaben einzeln zugegriffen werden.

Damit im Programm das Textende eindeutig erkannt werden kann, wird nach dem letzten Zeichen der Zeichenkette noch ein weiteres Element mit dem Zeichen \0 (ASCII-Wert: 0) gespeichert. Das Array muss also immer ein Element mehr haben, als die Zeichenkette lang ist. Wird dieses nicht beachtet, wird das Programm fehlerhaft arbeiten oder sogar abstürzen. Strings können wie gewöhnliche Arrays initialisiert werden. Am Beispiel würde das so aussehen:

```
char text[ ] = {
    'D', 'i', 'e', 's', ' ', 'i', 's', 't', ' ', 'e', 'i', 'n', ' ', 'T', 'e', 'x',
    't', '!', '\0'
};
```

Dadurch wird ein Array mit 19 Elementen (18 Textzeichen und 1 Abschlusszeichen) erzeugt. Damit die Initialisierungen von Zeichenketten nicht immer buchstabenweise erfolgen müssen, werden die Texte einfach in Anführungsstriche gesetzt. Dadurch lässt sich das obige Array einfacher initialisieren:

```
char text[ ] = "Dies ist ein Text!";
```

Der Compiler setzt nun durch die Anführungsstriche automatisch das Abschlusszeichen an den Text heran, und man braucht sich an dieser Stelle nicht mehr darum zu kümmern. Probleme könnten jedoch auftreten, wenn eine Zeichenkette zum Einlesen von Texten benutzt wird. Genau wie bei den Arrays wird weder vom Compiler noch zur Laufzeit getestet, ob die Zeichenkette für den eingegebenen Text gross genug ist! Im Normalfall werden die im Speicher hinter dem Array abgelegten Variablen überschrieben, falls ein Text über die Grösse des Arrays hinweg eingelesen wird. Hierbei könnte das Programm oder sogar der Rechner abstürzen! Die Arrays sollten zum Einlesen von Zeichenketten also ausreichend dimensioniert werden. Etwas komfortabler ist der Datentyp string aus der C++-Standartbibliothek. Ein String sieht damit wie folgt aus:

```
#include <string>
std::string text = "Dies ist ein Text!";
char letter = text[11]; // letter beinhaltet das Zeichen 'n'
```

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:

```
|| struct Mitglied
|| {
||     string name;
||     int nr;
||     string strasse;
||     int telefon;
|| };
```

Eine so definierte 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 Variablen-deklaration aus? Wie vorher auch:

```
|| Mitglied hans;      // Deklaration 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:

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

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

```
|| Mitglied peter;
|| peter = hans;
```

ist äquivalent zu

```
|| Mitglied peter;

|| peter.name = hans.name;
|| peter.nr = hans.nr;
|| peter.strasse = hans.strasse;
|| peter.telefon = hans.telefon;
```

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:

```
|| Mitglied verein[100];           // Deklaration 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:

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

2.1.5 Komplexe Datenstrukturen

Der Aufzählungstyp enum

Der *enum* (*enumerated*) Datentyp wird immer dann verwendet, wenn eine Menge *logisch zusammengehöriger Konstanten* definiert werden soll. Er wird oft auch als *Aufzählungsdatentyp* bezeichnet. Durch Aufzählen lassen sich benutzereigene Typen schaffen. Aufzählungstypen oder auch Enumerationstypen verbessern die Lesbarkeit der Programme. Man kann sich Aufzählungstypen als eine Art Definition von vielen Konstanten vorstellen. Durch die Zusammenfassung zu einem Aufzählungstyp wird ausgedrückt, dass die Konstanten miteinander verwandt sind. Eine enum Anweisung hat die folgende Syntax:

```
enum [Typname]
{
    Aufzählung
} [Variablenliste];
```

Nach dem Schlüsselwort *enum* folgt ein Name, der den enum-Datentyp eindeutig kennzeichnet. Man kann dann zu einem beliebigen Zeitpunkt über diesen Namen entsprechende enum-Variablen mit diesem enum-Datentyp definieren.

Ähnlich wie bei einer Union kann hier optional der Typname oder die Variablenliste weggelassen werden, jedoch erscheint es meist sinnvoll, nur die Variablenliste auszulassen. Das Weglassen des Typnamens ist nur dann sinnvoll, wenn der Aufzählungstyp nur für eine Variablendefinition benötigt wird. Man spricht dann von einer *anonymen Typdefinition*. Zwischen den folgenden geschweiften Klammern werden die *Bezeichner (Konstanten)* aufgezählt, die zu diesem Typ gehören. Die Bezeichner sind durch Kommata getrennt und können nach der Definition nicht mehr verändert werden.

Einer enum-Variablen kann dann im Verlaufe des Programms nur eine der innerhalb des enum-Blocks definierten Konstanten zugewiesen werden. Bei der Definition eines Aufzählungstyps wird dem ersten Bezeichner der Wert 0 zugewiesen. Jede weitere Konstante erhält den Wert ihres Vorgängers, erhöht um eins. Die Bezeichner werden also automatisch fortlaufend durchnummeriert. Einer enum-Konstante kann aber auch innerhalb des enum-Blocks explizit ein bestimmter Wert zugewiesen werden. Hierzu wird nach dem Konstantennamen der Zuweisungsoperator angegeben, gefolgt von dem gewünschten Wert. Dies wird durch folgendes Beispiel verdeutlicht:

Beispiel:

```
enum Schalter
{
    DISABLED = 1,
    FOCUS ,
    PRESSED = FOCUS + 10
} schalter1;
```

Wenn alle danach folgenden Konstanten keinen erneuten expliziten Wert erhalten, werden sie fortlaufend um eins erhöht. Für die *enum*-Konstanten sind nur Ganzzahlwerte zugelassen. Wie das obige Beispiel zeigt, ist es erlaubt, eine bereits in der Liste definierte Konstante zur Berechnung des Wertes einer weiteren Konstante zu verwenden. Hier besitzt zum Beispiel die Konstante *DISABLED* den Wert 1, *FOCUS* den Wert 2 und *PRESSED* den Wert 12. Wie jede andere Variable kann auch eine *enum*-Variable bei ihrer Definition initialisiert werden. Bei den Aufzählungstypen ist zu beachten, dass eine implizite Typumwandlung zu den ganzen Zahlen möglich ist, aber nicht umgekehrt. Dies wird wiederum durch folgendes Beispiel verdeutlicht:

```
enum Wochentag
{
    Sonntag,
    Montag,
    Dienstag,
    Mittwoch,
    Donnerstag,
    Freitag,
    Samstag
};

Wochentag werktag, feiertag, heute = Dienstag;
werktag = Montag;           // richtig
feiertag = Sonntag;         // richtig
int wert = Freitag;         // richtig (implizite Typumwandlung)
heute = 4;                  // falsch (keine Typumwandlung möglich)
wert = Montag + Dienstag;   // richtig
heute = Montag + Dienstag;  // falsch
heute++;                   // falsch
```

Union

Union ist *struct* sehr ähnlich. Der wesentliche Unterschied jedoch liegt darin, dass die Elemente einer *union* nicht hintereinander im Speicher liegen, wie das bei *struct* der Fall war, sondern alle an der gleichen Stelle im Arbeitsspeicher. Man könnte eine *union* auch als *Entweder-Oder-Verbund* bezeichnen, da in ihr mehrere Variablen zusammengefasst werden, von denen nur eine gebraucht wird. Alle Elemente beginnen an derselben Stelle im Speicher. Das heisst, dass falls die Elemente unterschiedlich groß sind, enden sie dementsprechend an verschiedenen Stellen. Es kann also immer nur eines der vereinbarten Elemente verwendet werden. Alle anderen werden dabei überschrieben. Daher nimmt eine *union* so viel Speicher ein, wie das größte seiner Elemente benötigt.

Eine Union wird dazu benutzt, um unterschiedliche Elemente zu speichern, wenn man sich sicher ist, dass immer nur eine Variante benutzt wird. Der Aufbau, die Syntax und der Zugriff auf Elemente des Typs *union* ähneln sehr stark denen von *structs*:

```
union [Typname]
{
    Aufbau
} [Variablenliste];
```

Das folgende Beispiel zeigt, wie ein *char*-Element mit einem *short*-Element überlagert wird. Hierbei belegt das *char*-Element genau 1 Byte, während das *short*-Element 2 Byte belegt.

2 Datenstrukturen und Operatoren

```
union Ziffer
{
    char c_zeichen ; // 1 Byte
    short s_zahl ;   // 1 Byte + 1 Byte
} ziffer1;
```

Der Zugriff auf einzelne Variablen erfolgt hier wieder mit dem Element-Auswahl-Operator, dem Punkt, im Speziellen wird durch `ziffer1.c_zeichen = 7`; dem Element `c_zeichen` der Variable `ziffer1` der Wert 7 zugewiesen. Da sich aber `c_zeichen` und das erste Byte von `s_zahl` auf derselben Speicheradresse befinden, werden nur die 8 Bit des Elements `c_zeichen` verändert. Dabei bleiben die nächsten 8 Bit, welche benötigt werden, wenn Daten vom Typ `short` in die Variable `ziffer1` geschrieben werden, unverändert. Falls nun versucht wird, auf ein Element zuzugreifen, dessen Typ sich vom Typ des Elements unterscheidet, auf das zuletzt geschrieben wurde, ist das Ergebnis nicht immer definiert.

Ähnlich der Struktur kann man Instanzen einer Union direkt bei der Deklaration oder in einer extra Zeile erstellen.

```
union Wort
{
    char var1;
    float wert1;
    ...
} wort1;
Wort wort2, wort3;
```

Wenn auf den Wert `wert1` zugegriffen wird, wird der Wert genommen, der gerade an der Speicherstelle steht:

```
wort2.wert1 = 17.32;
```

Es ist aber auch möglich, den Namen der union sowie jegliche Variablen dieses Typs wegzulassen. Dann braucht man keine Elemente mehr auszuwählen. Es handelt sich dann um eine freie union:

```
union
{
    char var1;
    float wert1;
    ...
};
```

2.1.6 Blöcke

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

```
{
    Anweisung;
    Anweisung;
    ...
}
```

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*² zusammenzufassen.

2.1.7 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 deklariert wurde.** Mit „Block“ ist hierbei alles gemeint, was in geschweiften Klammern steht, insbesondere auch Funktionen.

In einem Block können Sie beliebige Variablen deklarieren, 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::
14            endl;
15    }
16    // Ende des Blockes
17
18    // Jetzt kommt ein Fehler
19    std::cout << "Nochmal : Der Kreisumfang beträgt : " << umfang
        << std::endl;
}
```

Beim Versuch, dieses Programm zu kompilieren, meldet der Compiler bei der zweiten Bildschirmausgabe die undecklarierte Variable *umfang*. Sie 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, genausowenig wie zwei Eintrittskarten mit dem Namen „Eintrittskarte“ für zwei verschiedene Vergnügungsparks irgendetwas miteinander zu tun haben:

²Kontrollstrukturen lernen Sie in dem nächsten Versuch kennen

2 Datenstrukturen und Operatoren

```
1 #include <iostream>
2
3 int main()
4 {
5     // Beginn des 1. Blockes
6     {
7         int wert = 4;           // Deklaration der Variablen wert
8         // wert: 4
9         std::cout << "Der Wert beträgt : " << wert << std::endl;
10    }
11
12    // Beginn des 2. Blockes
13    {
14        // Deklaration einer anderen Variablen wert
15        int wert=2;
16        // wert: 2
17        std::cout << "Der Wert beträgt : " << wert << std::endl;
18    }
19    return 0;
20 }
```

Manchmal müssen Sie in einem Vergnügungspark für ein Karussell eine extra Eintrittskarte kaufen. Bei der Ticketkontrolle müssen Sie dann diese vorzeigen, die Karte für den gesamten Park ist bei diesem speziellen Karussell nicht als Eintrittskarte gültig. Dies gilt auch für gleiche Namen in verschachtelten Blöcken. Es gilt jeweils die innerste Deklaration, die weiter außen deklarierten Variablen behalten zwar ihren Wert, sind jedoch im inneren Block nicht sichtbar:

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

2.1.8 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.

```
int zahl;
int* zeiger = NULL;
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 der Adresse `NULL` belegt werden. Eine alternative Zeigerinitialisierung wäre:

```
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:

```
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 Deklaration 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/C++-intern über Zeiger realisiert ist, lässt es sich mit Hilfe eines zusätzlichen Zeigers manipulieren.

```
int feld[5] = { 3, 6, 9, 12, 15 };
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 Programmes zu unerwünschten und nicht nachvollziehbaren Fehlern führen.

2.1.9 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 deklariert 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 Programmiersprachen C/C++ unterscheiden 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. Bei einem erneuten Aufruf des Blockes hat eine statische Variable den Wert, den sie beim vorherigen Verlassen des Blockes hatte. Da statische Variablen während der gesamten Ausführung des Programms existieren und entsprechenden Speicherplatz benötigen, sollte man auf ihre Verwendung möglichst verzichten.
- **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.

2.1.10 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.³

Bei dieser so genannten dynamischen Programmierung ist der C/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 werden⁴. Vergisst der Programmierer, nicht mehr benötigte Datenelemente zu löschen, dann bleibt

³In C werden hier die Operatoren „malloc“ und „free“ benutzt.

⁴Die Programmiersprachen Java oder .NET besitzen automatische Speicherbereinigungen (garbage collection), die dynamisch allokierten Speicherplatz frei geben, sobald die Verweise nicht mehr existieren.

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.

```
int* intZeiger = new int;
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 allokierte 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 *NULL* zuweisen (*intZeiger = NULL;*).

2.2 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 der Eclipse Praktikumsumgebung zu bearbeiten, importieren Sie diese wie auf Seite 4 beschrieben.

2.2.1 Datentypen und Typumwandlung

Bauen Sie nacheinander Folgendes in der Datei *Variablen.cpp* ein⁵:

1. Lassen Sie den Benutzer zwei ganze Zahlen (*iErste* und *iZweite*) eingeben, speichern Sie die Summe in *iSumme* und das Ergebnis der Division von *iErste* durch *iZweite* in *iQuotient*. Geben Sie die Ergebnisse mit einer entsprechenden Meldung auf dem Bildschirm aus⁶.
2. Speichern Sie nun die Ergebnisse auch in den Variablen *dSumme* und *dQuotient* vom Typ *double*⁷ 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*.⁸ Geben Sie am Ende des Programms beide Formen des Namens aus.
5. Legen Sie für die folgenden Unterpunkte einen eigenen Block⁹ an.

⁵Erst ein Feature testen, dann das nächste einbauen.

⁶Hier und im Folgenden sollten Sie bei der Ausgabe einen entsprechenden Hinweis für den Benutzer ausgeben, damit dieser weiß, was sich hinter der Ausgabe von welchen Werten verbirgt.

⁷Hier sollen die Werte aus den ursprünglichen Werten neu berechnet werden.

⁸Nutzen Sie den + -Operator für Strings, der zwei Strings aneinander hängt.

⁹Ein Block besteht aus einer öffnenden und einer schließenden geschweiften Klammer({ }).

2 Datenstrukturen und Operatoren

- a) Legen Sie ein Feld *iFeld* aus Ganzzahlen mit 2 Elementen an und geben Sie den Elementen die Werte 1 und 2. Mit welchen Indizes kann man die Elemente des Feldes ansprechen?¹⁰
- b) Erzeugen Sie ein Feld aus 3 mal 3 Elementen mit den Werten 1 bis 9 mit dem Namen *spielfeld*. Mit welchen Indizes kann man die Elemente des Feldes ansprechen?
- c) Geben Sie die Elemente beider Felder aus.
- d) 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. Ändern Sie Punkt 6 so ab, dass die berechneten Zahlen die Position der Buchstaben im Alphabet unabhängig von Groß- und Kleinschreibung angeben. Berechnen Sie diese Zahl und geben Sie sie aus¹¹.

2.2.2 Strukturen

Deklarieren Sie in der Datei *Strukturen.cpp* 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 seinen 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.

2.2.3 Felder

Beim Caesar-Code oder auch Caesar-Verschlüsselung handelt es sich um eines der einfachsten und ältesten Verschlüsselungsverfahren der Welt. Angeblich wurde es von Julius Caesar selbst für militärische Korrespondenzen genutzt. Aufgrund seiner Einfachheit eignet es sich zwar gut für anschauliche Zwecke, wird in der Praxis jedoch nur noch zum Schutz vor unabsichtlichem Lesen eingesetzt (z.B beim Verfahren ROT13¹²), 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. Meistens handelt es sich dabei um eine einfache Verschiebung des Alphabets um eine feste Anzahl Stellen N (z.B N=4: A wird zu E). Es ist allerdings auch möglich die Zeichen des Geheimalphabets willkürlich zu wählen. Dabei muss jedoch gewährleistet sein, dass die

¹⁰Ausgabe im Programmcode

¹¹Benutzen Sie u.a. den modulo Operator(%).

¹²Weitere Informationen finden Sie unter <https://de.wikipedia.org/wiki/ROT13>

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 Caesar 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. Hierbei werden Sie auf Kontrollstrukturen (if, while etc.), die im nächsten Kapitel eingeführt werden, gezielt verzichten.

1. Erstellen Sie ein neues Projekt.
2. Legen Sie die Lookup-Tabelle für die Verschlüsselung an. In dieser wird jede Majuskel 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 2x26 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 Reihenfolge¹³.
3. Entwerfen Sie eine Funktion, die als Parameter ein Zeichen übergeben bekommt und unter Zuhilfenahme der Lookup-Tabelle das entsprechende Geheimzeichen ausgibt. Nutzen Sie hierzu die Eigenschaft der Zeichendarstellung vom Typ *char* aus und berechnen Sie daraus die entsprechende Indizierung.
4. Testen Sie Ihre Funktion, in dem Sie ihr Programm derart erweitern, dass der Benutzer ein Wort bestehend aus 5 Großbuchstaben eingeben kann. Geben Sie das Wort in Klartext und anschließend verschlüsselt in der Konsole aus.
5. Schreiben Sie analog zu ihrer Verschlüsselungsfunktion eine Entschlüsselungsroutine. Dazu benötigen Sie eine weitere Lookup-Tabelle.
6. Entschlüsseln Sie nun die Benutzereingabe und geben Sie sie in der Konsole aus.

Zusatzaufgabe:

Sie haben nun ein Programm geschrieben, das zwei Schlüssel enthält. Der eine wird zum Verschlüsseln ihrer Nachricht genutzt, der andere zum Entschlüsseln. Um die mechanische Komplexität gering zu halten, wurde die Verschlüsselungsmaschine Enigma¹⁴ so konstruiert, dass sowohl zum Verschlüsseln als auch zum Entschlüsseln derselbe Schlüssel genutzt werden konnte.

- Wie müssen Sie Ihren Schlüssel verändern, um ein vergleichbares Verhalten zu erzielen?
- Ändern Sie Ihr Programm so, dass bei Eingabe eines korrekt verschlüsselten Wortes der Klartext ausgegeben wird und umgekehrt.
- Was bedeutet die Änderung in Hinsicht auf die kombinatorische Vielfalt möglicher Verschlüsselungen? Stellt diese Änderung einen Vor- oder Nachteil für die Sicherheit Ihrer Verschlüsselung dar?

¹³Stellen Sie sicher, dass jeder Buchstabe nur ein mal vorhanden ist.

¹⁴Weitere Informationen unter [https://de.wikipedia.org/wiki/Enigma_\(Maschine\)](https://de.wikipedia.org/wiki/Enigma_(Maschine))

3 Ablauf- und Kontrollstrukturen

Alle Programme, die Sie im vorherigen Versuch geschrieben haben, bestanden aus einer Folge von Anweisungen, die beginnend mit der ersten Zeile des Programmcodes der Reihe nach ausgeführt wurden. Eine solche Programmstruktur ist nur zur Lösung sehr einfacher Aufgaben anwendbar. In diesem Versuch werden Sie Kontrollstrukturen kennenlernen, mit denen Sie im Programmcode verzweigen oder bestimmte Teile des Codes mehrfach ausführen können. Dies kommt auch einem klaren und leicht lesbaren Programmtext zugute. Im Aufgabenteil werden Sie Ihr neues Wissen anwenden und das Spiel *Reversi* programmieren.

Nachdem diese Mechanismen anhand einfacher Beispiele bekannt gemacht worden sind, sollen sie dabei helfen, ein erstes komplexeres Programm zu gestalten. Konkret thematisiert dieser Versuch:

- Kennenlernen von Funktionen.
- Einsatz von Schleifen und ihre Handhabung.
- Umsetzung des Wissens hin zu einem komplexeren Programm.
- Überprüfung des eigenen Programmes durch Tests

Die gestellte Aufgabe ermöglicht 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 ein Teil der Bewertung bei den Testaten darstellt und ab jetzt bei **jedem** Versuch erwartet wird.

3.1 Theorie

3.1.1 Zielsetzung und Einordnung

Durch diesen Versuch sollen Sie lernen, mit Hilfe von Kontrollstrukturen in einem 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. All dies zusammen mit dem Einsatz von Funktionen 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*.

3.1.2 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.

3 Ablauf- und Kontrollstrukturen

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

```
|| 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:

```
|| double f(double x, double y)
{
    double z = sqrt(x * x + y * y);
    return z;
}
```

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. In obigem 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     // Kuerzere Implementierung als oben:
7     return sqrt(x * x + y * y);
8 }
9
10 int main()
11 {
```

¹In der Literatur wird manchmal zwischen den Begriffen Argument und Parameter unterschieden. Wir verwenden hier beide Begriffe synonym.

```

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 }
```

Das Programm bindet zunächst ein paar Header ein und implementiert die Funktion $f(x, y)$. Die Funktion `main()`, die wir im Folgenden auch synonym mit dem Wort Hauptprogramm bezeichnen werden, 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 (hier 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 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

3 Ablauf- und Kontrollstrukturen

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:

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

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:

```
// Funktionsdefinition ohne Argumente
double pi(void)
{
    return 3.14159265;
}
int main()
{
    double value;
    // Funktionsaufruf mit leerer Argumentliste
    value = 2 * pi();
    return 0;
}
```

Call by value

Funktionsparameter werden in C++ normalerweise als Wert übergeben (Call by value). Für das Beispiel aus Abschnitt 3.1.2 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 3.1.

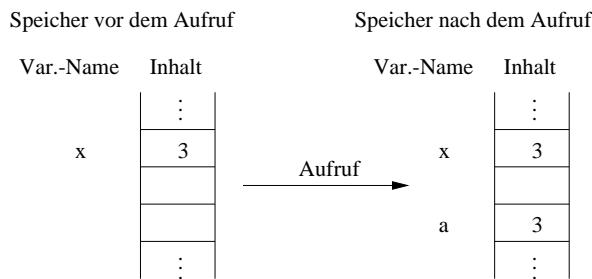


Abbildung 3.1: Call by value

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/C++* auch Call by reference realisieren. Hierzu werden in *C* Zeiger verwendet.

Zeiger sind recht umständlich in der Handhabung. *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 3.2.

Ein Funktionsargument kennzeichneten 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.

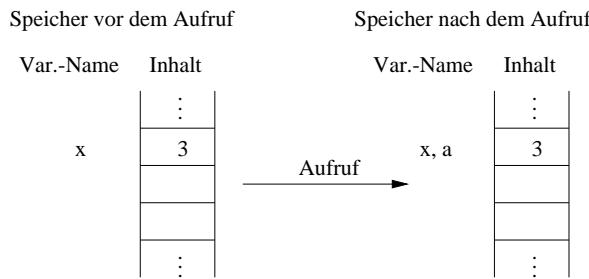


Abbildung 3.2: Call by reference

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 (*a*) 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
```

3 Ablauf- und Kontrollstrukturen

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

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

ändert, wird auch *x* inkrementiert:

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

Hinweis: Vertiefend wird in 2.1.8 auf die Thematik „Referenzen und Zeiger“ eingegangen. Zum jetzigen Zeitpunkt sollen Ihnen die bereits genannten Informationen bzgl. Referenzen und Zeiger genügen.

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 andern(int array[2][2])  
5 {  
6     array[0][0] = 2;  
7 }  
8  
9 void feldAusgeben(const int array[2][2])  
10 {  
11     std::cout << array[0][0] << " " << array[0][1] << std::endl;  
12     std::cout << array[1][0] << " " << array[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     andern(werte);  
24     // Ausgabe:  
25     std::cout << "Das Feld ist nun " << std::endl;  
26     feldAusgeben(werte);  
27     return 0;  
28 }
```

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

```
Das Feld lautet
```

```
1 2
```

```
3 4
```

```
Das Feld ist nun
```

```
2 2
```

```
3 4
```

3.1.3 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 behandeln wir die *if-else*-Verzweigung und ihre Abwandlungen 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.

```
if (expression)
{
    statement1;
}
else
{
    statement2;
}
```

Der Ausdruck *expression* kann jeder Ausdruck sein, der als Ergebnis die Wahrheitswerte *true* (entspricht ungleich Null) oder *false* (entspricht Null) liefert. Trifft die Bedingung zu (hat also *expression* einen Wert ungleich 0), so wird die Anweisung *statement1* ausgeführt. Ist die Bedingung nicht erfüllt (hat also *expression* den Wert 0), 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 Folge von Anweisungen, handeln kann.²

***else-if*-Anweisung**

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

²Beachten Sie weiterhin, dass ein abschließendes Semikolon immer fester Bestandteil einer einzelnen Anweisung ist.

3 Ablauf- und Kontrollstrukturen

```
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 werden wir dies nicht weiter explizit erwähnen.

Beispiel

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

```

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 *konstanten* Wert, so kann auch die *switch*-Anweisung verwendet werden, um in Abhängigkeit des Wertes im Programmcode zu verzweigen:

```

switch (expression)
{
    case constValue1 : statement1;
    case constValue2 : statement2;
    default: statement3;
}
```

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 void umwandlung(char& c)
4 {
5     switch (c)
6     {
7         // x wird durch * ersetzt
8         case 'x':
9             c = '*';
10            break;
11        // Vokale werden durch ? ersetzt
12        case 'a':
13        case 'e':
14        case 'i':
15        case 'o':
16        case 'u':
```

```

17     c = '?';
18     break;
19 // Alle anderen werden durch Leerzeichen ersetzt
20 default:
21     c = ' ';
22     break;
23 }
24 }
25
26 int main(void)
27 {
28     char zeichen = char();
29     std::cout << "Geben Sie ein Zeichen ein : ";
30     std::cin >> zeichen;
31     umwandlung(zeichen);
32     std::cout << "Ihre Eingabe wurde umgewandelt zu '" << zeichen
33     << "'" << std::endl;
34 }
```

Das Programm erwartet die Eingabe eines Buchstabens und wandelt diesen entweder in ein '*', '?' oder Leerzeichen um:³

```
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 Einsprungmarke 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/C++. Achten Sie also beim Verwenden einer *switch*-Anweisung besonders darauf!

3.1.4 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.

³Beachten Sie hier wiederum die Übergabe des Parameters per call by reference, eine Übergabe per call by value hätte keinen Effekt auf die Ausgabe.

for-Anweisung

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

```
for (expr1; expr2; expr3)
{
    statement;
}
```

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:

```
expr1;
solange expr2 erfüllt ist
{
    statement;
    expr3;
}
```

Beispielsweise haben Sie ein Array mit 10 Einträgen, die sie ausgeben wollen:

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

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

```
int array [10][10];
for (int y = 0; y < 10; y++)
{
    for (int x = 0; x < 10; x++)
    {
        array [x][y] = x;
    };
}
```

In diesem Fall wird also in jedem Schleifendurchlauf der ersten Schleife, die zweite einmal vollständig durchlaufen. Insgesamt wird demnach jedes der 100 Felder des Arrays 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/C++ die *while*-Schleife, bei der vor jedem Durchlauf überprüft wird, ob eine weitere Wiederholung erwünscht wird:

```
||| while (expression)
||| {
|||     statement;
||| }
```

Zunächst wird die Bedingung *expression* bewertet. Ist *expression* ungleich 0 (also *true*), wird *statement* ausgeführt. Anschließend wird *expression* erneut bewertet. Dies wiederholt sich, bis *expression* den Wert 0 hat, also *false* ist. 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 Fibunacci-Zahlen bis 100 aus:

```
1 #include <iostream>
2
3 int main()
4 {
5     int prevFib = 0;
6     int curFib = 1;
7     std::cout << 0 << std::endl << 1 << std::endl;
8     while (curFib <= 100)
9     {
10         int newFib = prevFib + curFib;
11         prevFib = newFib;
12         curFib = prevFib;
13         std::cout << curFib << std::endl;
14     }
15 }
```

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

```
||| for (expr1; expr2; expr3)
||| {
|||     statement;
||| }
```

ist (bis auf eine kleine Ausnahme, die wir hier vernachlässigen) identisch zu:

```
||| expr1;
||| while (expr2)
||| {
|||     statement;
|||     expr3;
||| }
```

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:

```
do
{
    statement;
}
while (expression);
```

Zunächst wird das *statement* ausgeführt. Anschließend wird die Bedingung *expression* bewertet. Ist ihr Wert ungleich 0, wird *statement* erneut ausgeführt. Dies wiederholt sich so lange, bis *expression* gleich 0 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 secret, guess;
8
9     srand(time(NULL)); // initialize random seed
10    secret = rand() % 10 + 1; // random number between 1 and 10
11
12    do
13    {
14        std::cout << "Guess a number between 1 and 10:" << std::endl
15        ;
16        std::cin >> guess;
17    }while(secret != guess);
18    std::cout << "Congratulations!" << 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(void)
4 {
5     int zahl = 0;
6     // unendliche Schleife starten
7     while (true)
```

3 Ablauf- und Kontrollstrukturen

```
8  {
9      // Eingeben einer Zahl
10     std::cout << "Geben Sie eine Zahl zwischen 1 und 10 ein : "
11     ;
12     std::cin >> zahl;
13     // Zahl innerhalb des Wertebereichs?
14     if (zahl >= 1 && zahl <= 10)
15     {
16         break;                                // verlassen von while
17     }
18     // Fehlermeldung
19     std::cout << "Das war wohl nichts, versuchen Sie " << "es
20     nochmal!" << std::endl;
21 }
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 << endl;
12 }
```

```

13
14 int main(void)
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.

3.1.5 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 diese Test möglich weit automatisiert sind, 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

Code Convention

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.

3.1.6 Übersichtlicher und verständlicher Code

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.
- 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 in Funktionen vermeiden.
- Mit Kommentaren in Funktionen nicht übertreiben.
- Zeiger und Referenzen stehen direkt hinter dem Datentyp.

Beispiele

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

```
int p(int f, int g){ int h = f*f+g*g;
return h; }

/*
Die Funktion Pythagoras realisiert den Satz des Pythagoras.
Sie bekommt a und b übergeben und liefert c^2 zurück.
*/
```

```

int pythagoras(int a, int b)
{
    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.

3.1.7 Kurzeinführung in Doxygen

Kommentare dienen nicht nur zur Beschreibung einzelner Codezeilen, sondern auch ganzer Funktionen. Daher muss Code, der wiederverwendet werden soll, kommentiert werden. Dabei sollte die Dokumentation der Funktionsweise von Schnittstellen einer Klasse im Vordergrund stehen. Insbesondere bei der *Black Box Reuse* ist eine saubere Kommentierung nötig, da die Implementierung selbst nicht eingesehen werden kann. Wurden alle Teile eines Programms ausführlich kommentiert und dokumentiert, erleichtert dies die spätere Kommerzialisierung einzelner Codeteile in separaten Bibliotheken. 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.

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:

```

/**  
 * ... text ...  
 */

```

- Blockkommentare die mit einem “!” gekennzeichnet sind:

```

/*!  
 * ... text ...  
 */

```

- C++-Zeilenkommentar die mit “//” anfangen:

```

///... Erste Zeile  
///... Zweite Zeile

```

- C++-Zeilenkommentar die mit “//!” anfangen:

```

//!... Erste Zeile  
//!... Zweite Zeile  
//!... 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
- 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:

```
/*! \brief Brief description.  
*          Brief description continued.  
*  
* Detailed description starts here.  
*/
```

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

```
/*! \brief Brief description.  
*          Brief description continued.  
*  
* Detailed description starts here.  
* Detailed description continued.  
*  
* \return Beschreibung des Rückgabewerts der Funktion.  
* \param x Beschreibung des Parameters x.  
* \param y Beschreibung des Parameters y.  
*/
```

Ein Programmierer der Ihre Funktion benutzt 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 ersten Zeile stehen muss. Eine Ausführliche Übersicht aller Befehle finden Sie unter <http://www.stack.nl/~dimitri/doxygen/manual/commands.html>

Hinweis: 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. Es ist jedoch sinnvoll, ausschließlich die Funktionen im Header mit Doxygen zu kommentieren und in der Implementierung normale C++-Kommentare zum Dokumentieren von internen Code-Details zu verwenden.

Anlegen von Doxy-Files

Um zu einem Eclipseprojekt eine Doxygen-Doku zu erstellen, brauchen Sie nur ein neues Doxyfile hinzufügen (mittels *Ctrl+N → Doxyfile*). Durch Doppelklick auf das Doxyfile können Sie die Doku konfigurieren. Sie sollten *Scan recursively* aktivieren und als Mode *all entities* auswählen. Zur besseren Navigation können Sie als *Output Format* die Option *with frames and navigation tree* einstellen. Mit dem Kontextmenüeintrag zu dem Projekt *Build Documentation* können Sie die Dokumentation erstellen. Sie wird in mehreren html-Dateien im Projekt abgelegt.

Hinweis: Im Verlauf des Praktikums sollten Sie sich immer mal wieder die Zeit nehmen, Ihren Sourcecode zu kommentieren oder Informationen mittels Doxygen zu hinterlegen, die Ihnen als wichtig erscheinen und bei den Testaten weiterhelfen. Bitte beachten Sie, dass die Kommentierung Teil der Bewertung ist und von nun an bei jedem Versuch vorgenommen werden muss.

3.2 Aufgaben

Die heutige 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. Importieren Sie hierzu bitte das vorgegebene Projekt aus dem zentralen Vorlagenverzeichnis.

Im L²P steht eine compilierte Demoversion zur Verfügung. Anhand dieser können Sie später nachvollziehen, ob Ihr Programm korrekt funktioniert. Dazu lassen Sie KI gegen KI spielen und vergleichen das Ergebnis der Demoversion mit Ihrem eigenen.

Spielregeln

Die Spielregeln von *Reversi* sind recht einfach zu verstehen, nehmen dem Spiel aber nichts seiner strategischen Komplexität. *Reversi* wird auf einem 2-dimensionalen Spielfeld gespielt, wie es in Abb. 3.3 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 3.3 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 Wikipedia⁴.

⁴<http://de.wikipedia.org/wiki/Reversi>

3.2 Aufgaben

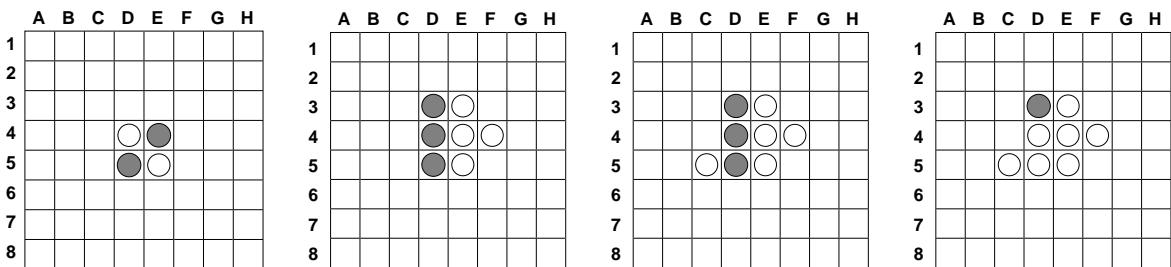


Abbildung 3.3: Startaufstellung und Beispiel

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

3.2.1 Vorbereitung

Importieren der Vorlage

Zu diesem Versuch ist ein Eclipse-cpp-Projekt vorgegeben.

1. Laden sie die Codevorlage aus dem L²P runter und importieren Sie diese in Ihren Workspace.

Das Projekt umfasst alle benötigten Dateien. Dabei stellen *Reversi_KI.h* und *Reversi_KI.cpp* eine komplett vor implementierte 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 *main.cpp* implementiert wird. Wir haben in diesen Dateien die Programmstruktur bereits vorgegeben, sodass Sie nur noch die Funktionsrümpfe implementieren müssen. Natürlich können Sie zusätzliche Funktionen dort einfügen, wo es Ihnen sinnvoll erscheint. Zur Implementierung der 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 run_full_test(void)
```

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

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 Funktion *show_field()* ist als Beispiel in den Vorlagen bereits vorhanden. Sie gibt das Spielfeld auf der Konsole aus. Nutzen Sie diese als Orientierung bei der Implementierung der weiteren Funktionen.

Wer ist der Gewinner?

Nach Abschluss des Spiels wird es notwendig sein den Gewinner zu ermitteln. Die Funktion dazu werden Sie als erstes implementieren. Sie werden diese mit einer geeigneten Testfunktion überprüfen und entsprechend dokumentieren. Anschließend werden Sie die restlichen Basisfunktionen in der gleichen Weise bearbeiten.

1. Die Funktion `int winner(const int field[SIZE_Y][SIZE_X])` zählt alle Felder die Spieler 1, sowie alle Felder die Spieler 2 besetzt hat und liefert anschließend den Gewinnercode:

- 0: falls das Spiel unentschieden ist,
- 1: falls Spieler 1 gewinnt,
- 2: falls Spieler 2 gewinnt.

Implementieren Sie die Funktion in der Datei `main.cpp`. Nutzen Sie dabei Schleifen, die auf der Seite 48 eingeführt werden.

Funktionstest

Nun werden Sie sicherstellen, dass die Funktion `winner()` korrekt funktioniert. Hierzu werden Sie mehrere Felder und die entsprechenden Gewinnercodes vorbereiten. Sie werden außerdem eine Testfunktion schreiben, die ein Feld-Code-Paar bewertet, und anschließend diese auf alle vorbereiteten Felder anwenden.

1. Vervollständigen Sie erst die Testfunktion

```
bool test_winner(const int field[SIZE_Y][SIZE_X], const int winner_code)
```

in der Datei `test.cpp`. Wenden Sie in dieser Testfunktion die Funktion `winner()` auf das übergebene Spielfeld an. Vergleichen Sie anschließend das Ergebnis des Funktionsaufrufs mit dem erwarteten Ergebnis, das im zweiten Parameter übergeben wird. Geben Sie zurück ob das berechnete Ergebnis dem erwarteten entspricht.

2. Ergänzen Sie das multidimensionale Array `winner_matrix` um drei von Ihnen entworfene Spielfelder.
3. Tragen Sie die zugehörigen Gewinnercodes in der gleichen Reihenfolge in das Array `winner_code` ein.
4. Rufen sie nun mithilfe der Funktion `run_full_test()` für jedes Feld-Code-Paar die Testfunktion auf und geben Sie auf der Konsole aus, ob Ihre Funktion in allen Fällen korrekt arbeitet.

Projektdokumentation

In diesem Versuch werden Sie Doxygen zur Dokumentation des Projekts nutzen. Doxygen wurde auf der Seite 55 eingefügt⁵.

1. Erstellen Sie ein neues Doxyfile.
2. Dokumentieren Sie die Funktion `winner()` und die Funktion `test_winner()`. Orientieren Sie sich dabei an der bereits vorgegebenen Dokumentation der Funktion `show_field()`.

⁵Nutzen Sie als Referenz auch die Dokumentation von Doxygen unter: <http://www.doxygen.org>

3.2 Aufgaben

3. Erstellen Sie die Dokumentation durch Klick auf das Doxygen Symbol (blaues @). Sie finden die erstellte Dokumentation in dem Unterordner *html*.

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.

Weitere Basisfunktionen

1. Überprüfen Sie, ob der Spieler *player* an der Position (*pos_x*, *pos_y*) einen Zug durchführen darf in der Funktion:

```
bool turn_valid(const int field[SIZE_Y][SIZE_X],  
                const int player,  
                const int pos_x,  
                const int pos_y)
```

2. Überprüfen Sie die Funktion *turn_valid()*. Rufen Sie diese hierzu in der Funktion *test_turn_valid()* auf. Erzeugen Sie entsprechende Ausgaben und geben Sie zurück, ob sich die Funktion wie erwartet verhält. Lassen Sie die Testfunktion wiederum über die vorgegebenen Testfälle innerhalb der Funktion *run_full_test()* laufen. Stellen Sie sicher, dass alle Fälle wie erwartet funktionieren.
3. Führen Sie den Zug eines Spielers innerhalb der Funktion *execute_turn()* aus. Diese Funktion hat eine starke Ähnlichkeit mit der Funktion *turn_valid()*. Nutzen Sie Teile ihres bestehenden Quellcodes und erweitern Sie diese hier geschickt.
4. Stellen Sie sicher, dass die Funktion *execute_turn()* korrekt funktioniert mithilfe der Testfunktionen *test_execute_turn()* und *run_full_test()*. Auch hier sind einige Testfälle gegeben. Erweitern Sie diese bei Bedarf.
5. Implementieren Sie die Funktion *possible_turns()*. Diese Funktion berechnet die Anzahl möglicher Züge eines Spielers und gibt diese zurück. Schreiben Sie entsprechende Testfunktionen, die sicherstellen dass die Funktion korrekt funktioniert.

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 diese 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 *human_turn()* führt einen menschlichen Zug aus. Hierbei wird der Benutzer erst nach seinem Zug gefragt. Dieser kann das gewünschte Feld in der Form A1 für die obere linke Ecke und H8 für die untere rechte Ecke auf der Konsole eingeben. Aus den Benutzereingaben

werden die zugehörigen Arrayindizes bestimmt. Dann wird mithilfe der Funktion *turn_valid()* geprüft, ob der Zug gültig ist und, falls ja, die Funktion *execute_turn()* aufgerufen.

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

1. Implementieren Sie eine Ablaufsteuerung für zwei menschliche Spieler.

- Lassen Sie abwechselnd den Spieler 1 und den 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:

```
int main(void)
{
    int player_type[2] = { HUMAN, HUMAN };
    game(player_type);
    return 0;
}
```

Der Computer spielt mit

Die Funktion *computer_turn()*, in der Datei *Reversi_KI*, führt analog zu *human_turn()*, 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 *game()* so, dass auch der Computer einen Zug machen kann. Der Spielertyp wird jeweils mit dem zweistelligen Feld *player_type* per Parameter übergeben. Nutzen Sie diesen um zu entscheiden ob der jeweilige Spieler menschlich, oder künstlich ist. Fügen Sie bei Bedarf weitere Basisfunktionen hinzu, um einfache Aufgaben zu kapseln.
2. Testen Sie ihre Implementierung, indem Sie die Spielertypen im Hauptprogramm variieren. Es ist möglich, zwei Computer gegeneinander spielen zu lassen⁶.

Zusatzaufgaben

- Fragen Sie vor dem 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.

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 von einander funktionieren und somit auch umfassend getestet werden können. Sie haben diesen Funktions-Pool anschließend genutzt und schrittweise ein Programm mit einer komplexen Ablaufsteuerung, einer Benutzerschnittstelle und einer künstlichen Intelligenz implementiert.

⁶Die korrekte Lösung dieser Partie finden Sie zur Kontrolle in der Vorlage des Versuchs.

3.2 Aufgaben

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 *OOP* 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 *OOP*-Prinzip aufbaut. Literatur und Quellen zu weiterführenden Techniken und Konzepten im Bereich der *OOP* finden Sie im Literaturverzeichnis: [1, 2, 3, 4].

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 *OOP* und der Linguistik zieht, dann entsprechen die Objekte allen Substantiven der deutschen Sprache.

Man könnte auch sagen, dass Objekte in der *OOP* 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 *OOP* auch Instanz genannt, die Begriffe werden oft synonym verwendet.

Bleiben wir beim Beispiel des Fahrkartenautomaten. Hieran kann man die grundlegenden Gedanken hinter der *OOP* gut nachvollziehen [4]:

- Es existieren viele verschiedene Geräte mit demselben Bauplan.
- Wichtig ist, was das Gerät „kann“, und nicht, wie es dies tut.
- Das Innere 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 teilen wir Objekte, die ähnliche Eigenschaften oder Funktionalitäten haben, in spezielle Gruppen ein und klassifizieren sie somit. Diese Klassifizierung hilft uns dabei, unsere komplexe Umwelt zu abstrahieren, damit sie uns leichter verständlich ist.

Ähnlich verhält es sich in der *OOP*. 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 darstellen.¹ In der *OOP* stellen Klassen also Systeme dar und mehrere Klassen bilden ebenfalls wieder ein System.

4.2 Klassen in C++

Klassen funktionieren in C++ wie Strukturen, haben aber ein eigenes Schlüsselwort. Variablen innerhalb von Klassen nennt man *Attribute* und Funktionen innerhalb von Klassen *Methoden*. Beides zusammen wird unter dem Begriff Member zusammengefasst.

```

1 class Lampe
2 {
3 private:
4     /* Attribute */
5     int gewicht;
6     std::string farbe;
7
8 public:
9     /* Methoden */
10    void anschalten();
11    void ausschalten();
12 };

```

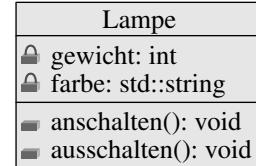


Abbildung 4.1: Die Klasse *Lampe* und ihr UML-Diagramm.

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* vier Member besitzt, von denen jeweils zwei Methoden bzw. Attribute sind.

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

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 *Praktikumsteilnehmer*, 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 wäre es zum Beispiel möglich, eine Datenbank mit allen Praktikumsteilnehmern zu erstellen. Wie in der Beispieldokumentation vom Anfang des Kapitels besteht diese Klasse auch aus Attributen und Methoden.³ Sie enthält Variablen für die Matrikelnummer, den Namen des Studenten und dessen erreichte Punktzahl.

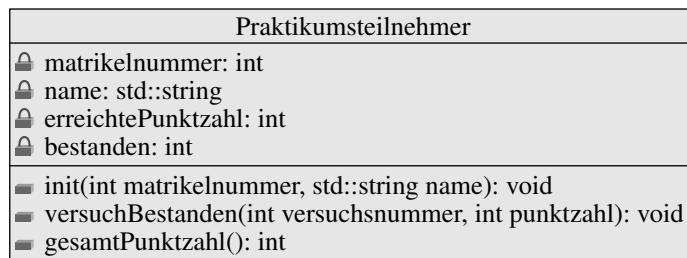


Abbildung 4.2: UML-Diagramm zur Klasse *Praktikumsteilnehmer*

```

#ifndef PRAKTIKUMSTEILNEHMER_H
#define PRAKTIKUMSTEILNEHMER_H

class Praktikumsteilnehmer
{
private:
    int matrikelnummer;
    std::string name;
    int erreichtePunktzahl;
    int bestanden;

public:
    /* Initialisiert das Objekt,
     * auf jeden Fall nach der Erzeugung eines Objekts aufrufen!
     */
  
```

³Die Befehle in den Zeilen 1, 2 und 18 werden in Abschnitt 4.6 erläutert und sind hier nur der Vollständigkeit halber erwähnt.

4 Einführung in Klassen

```
void init(int matrikelnummer, std::string name);
void versuchBestanden(int versuchsnummer, int punktzahl);
int gesamtPunktzahl();
};

#endif
```

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

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 Praktikumsteilnehmer 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, der nicht *void* ist. Die Methode *init* hat eine Sonderstellung und wird später noch genauer behandelt.[4]

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 *Deklarationen* der Variablen und Methoden. Eine beispielhafte *Header*-Datei haben Sie schon in der Klasse *Praktikumsteilnehmer* gesehen.

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

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

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

20 || }

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

Was ist hier passiert?

Da die Implementation der Klasse in einer separaten Datei erfolgt, weiß der Compiler nicht, welche Methoden und Attribute es in der Klasse gibt. Dazu muß man wissen, daß der Compiler nur die jeweiligen „.cpp“-Dateien einzeln übersetzt, da nur sie den Programmcode enthalten. Trifft er z.B. auf *erreichtePunktzahl*, weiß er nicht, was dieser Ausdruck verkörpert, er kennt die Deklaration an dieser Stelle nicht.

Um für Klarheit zu sorgen, geschieht im Listing 4.2 in Zeile 1 ein so genannter *include* der Header-Datei, der Inhalt dieser Datei wird an dieser Stelle eingefügt. Nun weiß der Compiler zwar, was es für Funktionen geben soll, für die Implementation muss aber nun klargestellt werden, dass die folgende Funktion auch zur gewünschten Klasse gehört.

Dies geschieht in Zeile 3 durch einen *Scope-Operator*, der den Zusammenhang mit einer bestimmten Klasse herstellt. „*Praktikumsteilnehmer*::“ drückt somit aus: Die nachfolgende Funktion gehört zur Klasse *Praktikumsteilnehmer*.

Ebenso wie bei *structs* wird auf Attribute und Methoden von Objekten mit dem „.“-Operator zugegriffen. Alternativ kann man natürlich auch die in Kapitel 2.1.8 kennengelernten Zeiger verwenden. Beides wird im folgenden Codeausschnitt beispielhaft gezeigt:

```

1 #include "Praktikumsteilnehmer.h"
2 int main(int argc, char* argv[])
3 {
4     /* Erstelle zwei Objekte der Klasse Praktikumsteilnehmer.
5        Maria ist ein Zeiger auf einen Praktikumsteilnehmer */
6     Praktikumsteilnehmer heinz;
7     Praktikumsteilnehmer *maria = new Praktikumsteilnehmer();
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 *Praktikumsteilnehmer*

Heinz wird als *Praktikumsteilnehmer* deklariert und Maria als Zeiger auf einen *Praktikumsteilnehmer*. Daher kann auf die Member der entsprechenden Klasse mithilfe des . bzw. -> Operators zugegriffen werden.

4.2.2 Der This-Zeiger

Im Listing 4.2 wurde in Zeile 9 ein sogenannter *This-Zeiger* verwendet. Was es mit diesem Zeiger auf sich hat, wird im Folgenden erläutert.

Innerhalb von Methoden kann auf alle Elemente der Klasse zugegriffen werden — egal, ob sie nach außen hin sichtbar sind oder nicht. 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 bzw. jede Instanz eine Kopie der Attribute der Klasse. Die Methoden der Klasse stehen aber nur einmal im Arbeitsspeicher und werden von allen Instanzen der Klasse gemeinsam benutzt.

Wie kommen nun die Methoden an die Daten ihres Objektes? Dazu erhält jede Methode einen zusätzlichen Parameter vom Compiler, den der Programmierer innerhalb dieser Methode verwenden kann. Dieser Parameter — der *This-Zeiger* — stellt einen Zeiger auf das Objekt dar, für das die Methode aufgerufen wurde.[1]

Nach einem Aufruf von zum Beispiel

```
|| heinz.init(123456, "Heinz");
```

zeigt der This-Zeiger in der aufgerufenen *init()*-Methode auf das Objekt „heinz“.

Bei jedem Aufruf einer nicht-statischen Methode einer Klasse wird dieser Zeiger automatisch als erster Parameter an die Memberfunktion übergeben, und er wird innerhalb von Methoden implizit verwendet. Das bedeutet, dass Zuweisungen ohne This-Zeiger denselben Effekt haben wie solche mit This-Zeiger⁵.

4.2.3 Konstruktoren und Destruktoren

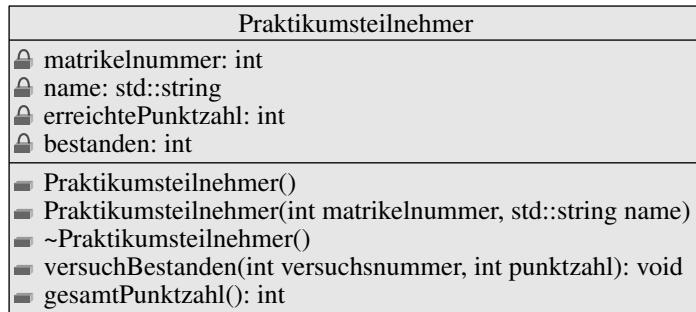
Die Benutzung der *init()*-Methode zur Initialisierung der Membervariablen ist unelegant und fehleranfällig. Der Programmierer ist nicht verpflichtet, die *init()*-Methode aufzurufen, doch das Objekt ist nur sinnvoll, wenn die *init()*-Methode unmittelbar nach dem Anlegen der Instanz ausgeführt wird.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

In der Objektwelt von C++ ist der Konstruktor eine Funktion, die für das Erzeugen eines Objektes aufgerufen wird. 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 6.2.1). 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.

⁵Vorausgesetzt, es gibt keine Namenskollisionen, wie in Listing 4.2 Zeile 9.

Abbildung 4.3: UML-Diagramm zur Klasse *Praktikumsteilnehmer*

```

1  class Praktikumsteilnehmer
2  {
3      private:
4          int matrikelnummer;
5          std::string name;
6          int erreichtePunktzahl;
7          int bestanden;
8
9      public:
10     /* Konstruktoren */
11     Praktikumsteilnehmer();
12     Praktikumsteilnehmer(int matrikelnummer, std::string name);
13
14    /* Destruktor */
15    ~Praktikumsteilnehmer();
16
17    void versuchBestanden(int versuchsnummer, int punktzahl);
18    int gesamtPunktzahl();
19 };
20
21 //eine beispielhafte Implementierung der beiden Konstruktoren
22
23 Praktikumsteinehmer::Praktikumsteilnehmer()
24 {
25     erreichtePunktzahl = 0;
26     matrikelnummer = 0;
27     bestanden = 0;
28     name = "";
29 }
30
31 Praktikumsteinehmer::Praktikumsteilnehmer(int matrikelnummer,
32                                         std::string name)
33 {
34     erreichtePunktzahl = 0;
35     this->matrikelnummer = matrikelnummer;
36     bestanden = 0;
37     this->name = name;

```

Listing 4.4: Deklaration von Konstruktoren innerhalb einer Klasse

Der Codeausschnitt zeigt die Klasse *Praktikumsteilnehmer* mit zwei verschiedenen Konstruktoren. Der *Standardkonstruktor* (auch Default-Konstruktor) besitzt keine Parameter. Er ruft zu allen Membervariablen die Standardkonstruktoren auf. 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. Dies ist im Listing 4.5 in Zeile 3 der Fall. 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(int argc, char* argv[])
2 {
3     Praktikumsteilnehmer jim;
4     Praktikumsteilnehmer heinz(123456, "Heinz");
5
6     Praktikumsteilnehmer* maria = new Praktikumsteilnehmer(123457,
7                 "Maria");
8
9     jim.versuchBestanden(2, 2);
10    heinz.versuchBestanden(1, 2);
11    maria->versuchBestanden(1, 3);
12
13    delete maria;
14    return 0;
}

```

Listing 4.5: Aufruf von Konstruktoren und Destruktoren

Hinweis: Wenn in einer Klasse kein Default-Konstruktor definiert ist, sondern nur Konstruktoren mit Übergabeparametern, so kann von dieser Klasse kein Default-Objekt (ohne Übergabeparameter) instanziert werden. Der Compiler wird dann einen Fehler melden.

Ein *Destruktor* ist das genaue Gegenteil eines Konstruktors. So wie dieser 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 Standarddestrukturator 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 *Praktikumsteilnehmer*). Im Listing 4.5 wird in Zeile 12 der Destruktor von Maria und in Zeile 13 der von Jim und

Heinz aufgerufen. Dies geschieht jeweils automatisch zum Ende der Gültigkeit der Variablen. Ein Verwendungsbeispiel für Destruktoren finden Sie nachfolgend in Form eines binären Suchbaums:

```

1  class BTree
2  {
3  public:
4      int data;
5
6  private:
7      BTree *left;           //linke Verzweigung im Baum
8      BTree *right;          //rechte Verzweigung im Baum
9
10 public:
11     BTree( int data );    //Konstruktor (mit Parameter)
12     ~BTree();             //Destruktor
13     bool insert( int key ); //Deklaration der Einfügefunktion
14 };
15
16 BTree::BTree( int data )
17 {
18     this->data = data;
19     left = nullptr;        //Kennzeichnung, dass der Baum hier zu
20     right = nullptr;       //Ende ist durch sinnvolle Initiali-
21                               //sierung im Konstruktor (der Wert
22                               //wäre sonst zufällig)
23 }
24
25 BTree::~BTree()
26 {
27     delete left;           //Löschen der Unterknoten, die sonst
28     delete right;          //unzugreifbar im Speicher verbleiben
29 }
30
31 bool BTree::insert( int key )
32 {
33     if ( key < data )      //links einfügen
34     {
35         if ( left == nullptr ) //unterster Knoten?
36         {
37             left = new BTree( key ); //neue Instanz erzeugen
38             return true;           //Wert in Baum eingetragen
39         }
40         else
41         {
42             return left->insert( key ); //rekursiv bis left=nullptr
43         }
44     }
45     else if ( key > data ) //wie links einfügen, nur
46         rechts

```

```

46  {
47      if ( right == nullptr )
48      {
49          right = new BTee( key );
50          return true;
51      }
52      else
53      {
54          return right->insert( key );
55      }
56  }
57  return false;           //Wert konnte nicht eingetragen werden
58 }
```

Listing 4.6: Ein binärer Suchbaum mit Einfügefunktion

4.2.4 Initialisierungslisten

Es gibt noch eine weitere Möglichkeit, die Attribute einer Klasse bei der Instanzierung zu initialisieren, ähnlich der *init*-Methode aus Listing 4.2 oder durch den Konstruktor:

```

Praktikumsteilnehmer::Praktikumsteilnehmer(int matrikelnummer,
                                             std::string name):
    matrikelnummer(matrikelnummer),
    name(name),
    erreichtePunktzahl(0),
    bestanden(0)
{}
```

Listing 4.7: Initialisierungsliste

Es fällt auf, dass die Klassenattribute hier nicht innerhalb der Methode belegt wurden. Stattdessen sind hinter dem Funktionskopf — getrennt durch einen Doppelpunkt — neue Befehle hinzugekommen. Diesen speziellen Bereich hinter dem Funktionskopf nennt man *Initialisierungsliste*. Im Folgenden soll erklärt werden, was Initialisierungslisten ausmacht, und warum sie für die Objektorientierte Programmierung mit C++ wichtig sind.

Die Syntax einer Initialisierungsliste dürfte an obigem Beispiel bereits deutlich werden. Die Initialisierung wird bei der *Implementierung* des Konstruktors zwischen Kopf und Rumpf — also direkt über der ersten geschweiften Klammer — eingefügt. Der Name des zu initialisierenden Elements wird zusammen mit dem entsprechenden Parameter des Konstruktors in der Liste aufgeführt. Dies entspricht, wie später noch genauer erklärt wird, einem Konstruktorauftruf. Mehrere Initialisierungen werden durch Kommas getrennt. Die Initialisierungsliste wird vom Methodenkopf durch einen Doppelpunkt getrennt.

Warum ist nun eine Initialisierungsliste überhaupt notwendig, es besteht doch auch die Möglichkeit, die Initialisierung im Methodenrumpf vorzunehmen? Um dies zu verstehen, ist ein genauerer Blick auf die Programmiersprache C++ notwendig: Innerhalb des Methodenblocks können nur Variablen verändert werden, die bereits eine Speicherstelle besitzen und somit bereits instanziert wurden. Die Initialisierungsliste definiert den Zeitpunkt, zu dem die Attribute

der Klasse tatsächlich instanziert werden. Dies geschieht dadurch, dass Konstruktoren der entsprechenden Klassen aufgerufen werden. Wenn für ein Element in der Initialisierungsliste kein Konstruktor explizit angegeben wird, wird der Standardkonstruktor verwendet. Übrigens haben in C++ alle Standarddatentypen einen Standardkonstruktor. Wenn es sich bei einem der Attribute um ein Objekt einer Klasse ohne Standardkonstruktor handelt, ist der Code ohne Initialisierungsliste *nicht kompilierbar*.[1]

Im folgenden Beispiel[1] wird das Problem einer leeren Initialisierungsliste deutlich:

```

1 class Point
2 {
3 public:
4     Point(int x, int y);
5     int x;
6     int y;
7 };
8
9 class Circle
10 {
11 public:
12     Circle(const Point &centre, double radius);
13
14 private:
15     Point centre;
16     double radius;
17 };
18
19 Point::Point(int x, int y)
20 {
21     this->x = x; this->y = y;
22 }
23
24 Circle::Circle(const Point &centre, double radius)
25 {
26     this->centre = centre;
27     this->radius = radius;
28 }
```

Listing 4.8: Fehlende Initialisierungsliste

Die Initialisierungslisten der Klassen *Point* und *Circle* sind leer und rufen deswegen jeweils alle Standardkonstruktoren auf. Bei den Standarddatentypen *int* und *double* ist dies kein Problem, aber die Klasse *Point* besitzt keinen Standardkonstruktor. Dieses Programm könnte also nicht kompiliert werden. Deswegen werden wir es nun korrigieren:

```

1 class Point
2 {
3 public:
4     Point(int x, int y);
5     int x;
6     int y;
7 };
```

```

8
9 class Circle
10 {
11 public:
12     Circle(const Point &centre, double radius);
13
14 private:
15     Point centre;
16     double radius;
17 };
18
19 Point::Point(int x, int y) : x(x), y(y)
20 {
21 }
22
23 Circle::Circle(const Point &centre, double radius) :
24     centre(centre.x, centre.y), radius(radius)
25 {
26 }
```

Listing 4.9: Korrigierte Initialisierungsliste

Aufgrund der oben genannten Problematik und auch aus Effizienzgründen ist es anzuraten, *wenn immer möglich* Initialisierungslisten zu verwenden.

Konstante Attribute erfordern zwingend einen Eintrag in der Initialisierungsliste, da sie nur an dieser Stelle einmalig initialisiert werden können.

Auch die Attribute einer Oberklasse können über die Initialisierungsliste durch Aufruf des entsprechenden Konstruktors der Oberklasse initialisiert werden. Im folgenden Beispiel wurden in Zeile 20 sowie 24f Initialisierungslisten verwendet, um die Attribute mit den übergebenen Parametern zu initialisieren[1]:

```

1 class Form
2 {
3 public:
4     Form(unsigned int colour);
5
6 protected:
7     unsigned int colour;
8 };
9
10 class Circle : public Form
11 {
12 public:
13     Circle(const Point &centre, double radius, unsigned int colour
14         );
15
16 private:
17     Point centre;
18     double radius;
19 };
```

```

19
20 Form::Form(unsigned int colour) : colour(colour)
21 {
22 }
23
24 Circle::Circle(const Point &centre, double radius, unsigned int
25   colour) :
25   Form(colour), centre(centre.x, centre.y), radius(radius)
26 {
27 }

```

Listing 4.10: Initialisierung von Basisklassenelementen

4.2.5 Zugriffsbeschränkung

Mit der Einführung der Klassen wurde auch ein Konzept zur Definition von Zugriffsbeschränkungen auf Methoden und Attribute in Klassen eingeführt. Dies wird nun genauer erläutert.

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 7.1)
<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*. Hier liegt auch der einzige Unterschied zwischen Strukturen und Klassen in C++ : Bei Strukturen sind Member standardmäßig *public*.

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.

4 Einführung in Klassen

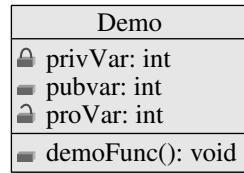


Abbildung 4.4: UML-Diagramm zur Klasse *Demo*.

```
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;    //Fehlernachricht
30     obj.pubVar = 2;    //ok
31     obj.proVar = 2;    //Fehlernachricht
32
33     return 0;
34 }
```

Listing 4.11: Zugriffsbeschränkung am Beispiel der Klasse *Demo*

4.3 Parameter

4.3.1 Konstante Parameter und Call By Reference

Wie Sie bereits in Versuch 2 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:

```
|| void insert_in_list(Praktikumsteilnehmer &teilnehmer);
```

Diese Funktion fügt einen Praktikumsteilnehmer 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.

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

```
|| void insert_in_list(const Praktikumsteilnehmer &teilnehmer);
```

Die Verwendung konstanter Parameter hat zwei Vorteile. Man bekommt einen besseren Überblick darüber, wann Daten verändert werden, und es besteht die Möglichkeit, auch konstante Objekte sinnvoll zu verarbeiten.

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

```
|| void insert_in_list(Praktikumsteilnehmer * const teilnehmer);
```

4.3.2 Defaultparameter

Manchmal ist es erwünscht, einen Parameter in einer Methode nicht immer explizit setzen zu müssen. Wenn wir zum Beispiel unsere Beispielklasse *Praktikumsteilnehmer* um eine Membervariable ergänzen, welche das aktuelle Fachsemester enthält, ist diese in den meisten Fällen „2“.

Die neue Deklaration der Klasse sieht wie folgt aus:

```
1 class Praktikumsteilnehmer
2 {
3     private:
4         int matrikelnummer;
5         std::string name;
6         int erreichtePunktzahl;
7         int fachsemester;
8         int bestanden;
9
10    public:
11        /* Konstruktor */
12        Praktikumsteilnehmer(int matrikelnummer, std::string name, int
13                             fachsemester = 2);
14        /* Destruktor */
```

```

14     ~Praktikumsteilnehmer();
15
16     /* Methoden */
17     void versuchBestanden(int versuchsnummer, int punktzahl);
18     int gesamtPunktzahl();
19 };

```

Listing 4.12: Defaultparameter (Deklaration)

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

```

Praktikumsteilnehmer::Praktikumsteilnehmer(int matrikelnummer,
                                             std::string name, int fachsemester):
    matrikelnummer(matrikelnummer),
    name(name),
    erreichtePunktzahl(0),
    fachsemester(fachsemester),
    bestanden(0)
{
}

```

Listing 4.13: Defaultparameter (Verwendung)

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

```

int main(int argc, char* argv[])
{
    /* erstelle zwei Objekte der Klasse Praktikumsteilnehmer */
    Praktikumsteilnehmer heinz(123456, "Heinz");
    Praktikumsteilnehmer* maria = new Praktikumsteilnehmer
        (123457, "Maria", 4);
    /* ... */
}

```

Listing 4.14: 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     Praktikumsteilnehmer const john;
4     john.versuchBestanden( 1, 3 );
5     int result = john.gesamtPunktzahl();
6 }
```

Listing 4.15: Der Unterschied zwischen konstanten und nicht konstanten Methoden

Beide Funktionsaufrufe sind verboten und werden von 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 werden wir nun die Methode *gesamtPunktzahl* als konstante Methode ausweisen. Eine konstante Methode wird, genau wie eine Konstante, durch ein zusätzliches *const* definiert:

```

1 class Praktikumsteilnehmer
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 Statische Attribute und Methoden

4.5.1 Statische Attribute

Im Listing 4.1 zur Nutzung der Klasse *Praktikumsteilnehmer* stellen Sie fest, dass die „Matrikelnummer“ vom Benutzer der Klasse vergeben werden muss. Dies ist sowohl aufwändig als auch fehleranfällig.

Es gibt im C++ Klassenkonzept jedoch die Möglichkeit, in einer Klasse ein *statisches* Attribut anzulegen, das in jedem *Objekt* der Klasse gleich ist. Statische Attribute stellen sozusagen versteckte globale Variablen dar und existieren unabhängig davon, ob Objekte der Klasse instanziert wurden.

Damit bestände die Möglichkeit, die erstellten Objekte zu zählen und die Matrikelnummern dementsprechend zu setzen:

```

1 class Praktikumsteilnehmer
2 {
3     private:
4         static int objectCounter;
5         int matrikelnummer;
6         std::string name;
7         int erreichtePunktzahl;
8         int bestanden;
9
10    public:
```

```

11  Praktikumsteilnehmer(std::string name, int fachsemester = 2);
12  ~Praktikumsteilnehmer();
13  /* ... */
14 };
15
16 int Praktikumsteilnehmer::objectCounter = 0;
17
18 Praktikumsteilnehmer::Praktikumsteilnehmer(std::string name, int
19   fachsemester) :
20   matrikelnummer(++objectCounter), name(name),
21   erreichtePunktzahl(0), fachsemester(fachsemester),
22   bestanden(0)
23 {
24 }
25
26 /* ... */

```

Listing 4.16: Deklaration und Verwendung eines statischen Attributs

4.5.2 Statische Methoden

Genauso, wie es statische Attribute gibt, gibt es auch statische Methoden. Diese sind auch aufrufbar, ohne dass eine Instanz der Klasse existieren muss (was für die übrigen Methoden gilt). Damit das möglich ist, verwenden statische Methoden keinen *This-Zeiger*. 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 wurde ein statisches Attribut *objectCounter* angelegt und initialisiert. Um auf dieses Attribut zugreifen zu können, auch wenn noch keine Instanz von *Praktikumsteilnehmer* existiert, kann eine statische Methode benutzt werden. Die Deklaration und Implementierung dieser statischen Methode sieht wie folgt aus:

```

1 class Praktikumsteilnehmer
2 {
3 private:
4 static int objectCounter;
5 int matrikelnummer;
6 std::string name;
7 int erreichtePunktzahl;
8 int bestanden;
9
10 public:
11 Praktikumsteilnehmer(std::string name, int fachsemester = 2);
12 ~Praktikumsteilnehmer();
13
14 static int getobjectCounter();
15 /* ... */
16 };

```

```

17
18 int Praktikumsteilnehmer::objectCounter = 0;
19
20 Praktikumsteilnehmer::Praktikumsteilnehmer(std::string name, int
21 fachsemester) :
22 matrikelnummer(++objectCounter), name(name), erreichtePunktzahl
23 (0), fachsemester(fachsemester), bestanden(0)
24 {
25 }
26
27 int Praktikumsteilnehmer::getobjectCounter()
28 {
29     return objectCounter;
30 }
31 /* ... */

```

Listing 4.17: Deklaration und Verwendung eines statischen Attributs

4.6 Include-Wächter

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

```

#ifndef PRAKTIKUMSTEILNEHMER_H
#define PRAKTIKUMSTEILNEHMER_H
/* ... */
#endif

```

Listing 4.18: Ausschnitt aus Listing 4.1

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'
class Grandfather
{
    int member;
};

Datei 'father.h'
#include "grandfather.h"
/* ... */

Datei 'child.cpp'

#include "grandfather.h"
#include "father.h"
/* ... */

```

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 4.18:

- In Zeile 1 wird abgefragt, ob ein Makro mit dem Namen PRAKTIKUMSTEILNEHMER_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.

4.7 Vorausdeklaration

Wenn zwei Klassen einen Zeiger auf die jeweils andere Klasse als Membervariable enthalten sollen, braucht man eine sogenannte Vorausdeklaration. Warum dies der Fall ist, soll das folgende Beispiel verdeutlichen:

```
#include "Engine.h"

class Car
{
public:
    Car(Engine* engine);
    ~Car();
    void drive();
    void startEngine();
private:
    Engine* engine;
};
```

car.h

```
#include "car.h"

class Engine
{
public:
    Engine(Car* car);
    ~Engine();
    void start();
    void increaseRPM();
    void decreaseRPM();
private:
    Car* car;
```

```
|| } ;
```

engine.h

Wie zu sehen ist, speichern beide Klassen einen Zeiger auf die jeweils andere. Wenn der Compiler nun beginnt, die Klasse *Car* zu übersetzen, kennt er *Engine* noch nicht. Auch die *include* Anweisung hilft nicht, da dann in der *Engine*-Klasse wieder *Car* unbekannt ist. Um dieses Problem zu lösen, hilft es nur, explizit dem Compiler zu sagen, dass es sich bei *Car* und *Engine* um Klassen, also Datentypen handelt.

Dies erreicht man dadurch, dass man eine sogenannte Vorausdeklaration durchführt. Dazu wird zum Beispiel in der Datei *car.h* die Zeile

```
|| class Engine;
```

hinzufügt. Auf unser gesamtes Beispiel angewendet, sieht das dann folgendermaßen aus:

```
|| class Engine;
|| class Car
|| {
|| public:
||     Car(Engine* engine);
||     ~Car();
||     void drive();
||     void startEngine();
|| private:
||     Engine* engine;
|| };
```

car.h

```
|| class Car;
|| class Engine
|| {
|| public:
||     Engine(Car* car);
||     ~Engine();
||     void start();
||     void increaseRPM();
||     void decreaseRPM();
|| private:
||     Car* car;
|| };
```

engine.h

Es ist wichtig zu beachten, dass dieser Lösungsansatz nur funktioniert, wenn die Verknüpfung über Zeiger geschieht.

4.8 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:

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

4.8.1 Die Klasse Vektor

Importieren Sie das Projekt „Versuch04“ aus dem Vorlagenverzeichnis in Ihren **Workspace**. Sie finden ein Grundgerüst für die Klasse **Vektor** vor, welche 3 private Membervariablen x, y und z enthält. Die Klasse soll folgende public-Funktionen besitzen:

- 3 Funktionen, die die Werte von x, y bzw. z zurückgeben.
- Eine Funktion, in der die Länge des Vektors berechnet und zurückgegeben wird.
- Eine Funktion, die einen Vektor als Parameter bekommt und diesen anschließend vom **this**-Zeiger subtrahiert.
- Eine Funktion, die zwei Vektoren addiert und einen neuen Vektor zurückgibt.
- Eine Funktion, die die Orthogonalität zweier Vektoren prüfen kann.
- Eine Funktion, die die Werte des Vektors ausgibt.

Denken Sie daran, alle Funktionen und übergebenen Variablen soweit möglich als konstant zu definieren. Testen Sie alle Funktionen, indem Sie in der Datei main.cpp einige Vektoren erstellen und verwenden.

4.8.2 Rotationsmatrix

Implementieren Sie in die Klasse **Vektor** eine Funktion, die einen Vektor um die z-Achse rotiert. Übergeben Sie der Funktion dazu einen Winkel in Radian.

4.8.3 Wie weit entfernt ist der Horizont?

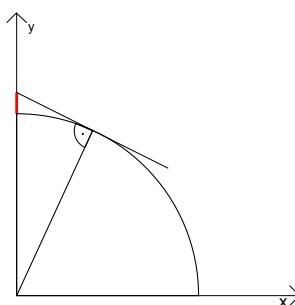


Abbildung 4.5: Die Grafik veranschaulicht das Problem. Versuchen Sie unter Zuhilfenahme der Grafik den Sachverhalt mathematisch abzuleiten.

In dieser Aufgabe soll die Entfernung berechnet werden, die ein Mensch von 1,80 Meter Körpergröße 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 bei der Aufgabe 4.8.2 implementierte Funktion, um diese Aufgabe zu lösen. Verwenden Sie dabei nicht den Satz des Pythagoras und die trigonometrischen Funktionen. Beachten Sie, dass bei der Rechnung mit Fließkommazahlen ganze Werte aufgrund der großen Genauigkeit, also der Anzahl der Nachkommastellen, kaum zu erreichen sind.

Wie groß ist die Distanz, wenn der Mensch auf einer 500 Meter hohen Plattform steht?

Hinweis:

Der Radius der ideal runden Erde soll in dieser Aufgabe 6371 Kilometer betragen.⁶

⁶Die Sichtweite auf einem 830 Meter hohen Turm beträgt bspw. ca. 103 Kilometer. Verifizieren Sie dies!

5 Dynamische Datenstrukturen und Dokumentation

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 Programmwickler 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 bieten die Sprachen C und 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. Die Datenelemente „Zeiger“ und „Referenz“ speichern diese Adresse. Sie ermöglichen die Entwicklung von leistungsfähigen und effizienten Algorithmen, können jedoch bei unsachgemäßer Verwendung zu Programmfehlern führen.

Bekanntschaft mit Referenzen haben Sie bereits in Abschnitt 3.1.2 gemacht.

Neben der Betrachtung von programmiersprachenspezifischen Strukturen soll in diesem Kapitel ein Augenmerk auf die Struktur, Lesbarkeit und Dokumentation im Allgemeinen gelegt werden.

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. Des Weiteren werden Softwarepakete regelmäßig mit neuen Funktionen ausgestattet, damit sie sich von Konkurrenzprodukten unterscheiden und den Kunden zum Kauf einer neuen Programmversion anregen. Oftmals werden Funktionen auch auf Kundenwunsch ergänzt. Bei derartigen Anpassungen einer Software wird versucht, möglichst viel der alten Funktionalität weiter zu verwenden, um den Arbeitsaufwand gering zu halten. Zeitersparnisse bringen insbesondere

- eine vollständige und klar verständliche Dokumentation des Programmcodes
- auf die Funktion hinweisende Namen von Variablen und Funktionen
- gut lesbarer Code

Die Dokumentation umfasst dabei nicht nur die Kommentierung des Quelltextes, sondern eine ausführliche Beschreibung jedes einzelnen Bausteins des Programms. Diese 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.

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/C++.
- Einsatz von Zeigern und Referenzen.
- Einsatz von dynamischen Datenstrukturen am Beispiel von einfach und doppelt verketteten Listen.
- Verwendung der Speicherverfahren LIFO und FIFO am Beispiel einer Warteschlange und eines Stapelspeichers.

Darüber hinaus sollte Ihnen Folgendes geläufig sein:

- Programmieren mit Code Convention.
- Erstellung einer Dokumentation in Eclipse mit Doxygen.

Anmerkung: Bereits auf den ersten Seiten des Skriptes (Seite 54ff) wurde auf die Code Convention und die zu erstellende Dokumentation hingewiesen. Ab jetzt sollten Sie verstärkt auf diese Merkmale achten, da sie ein Teil der Bewertung bei den Testaten ist.

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 (Speicherplatzverschwendug) 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 (*head* und *tail*), die auf *NULL* zeigen. Ein Listenelement (*ListElem*) besteht immer aus den Daten selbst (*data*) und einem Zeiger, der auf das in der Liste nachfolgende Listenelement (*next*) zeigt.

```
class ListElem
{
private:
    ListElem* next;
    int data;
};
```

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 *tail* die Adresse des neu angelegten Elementes zugewiesen. Die einfache 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 zeigt immer auf *NULL*, so dass dies eine Abbruchbedingung darstellt, um Iterationen über der Liste zu beenden.

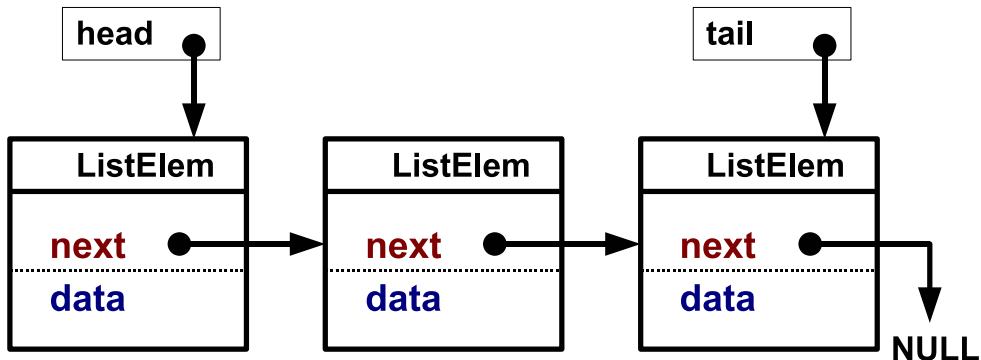


Abbildung 5.1: Schema einer einfach verketteten linearen Liste.

5.2.2 Doppelt verkettete Liste

Möchte man auf das vorletzte Element einer einfach verketteten Liste zugreifen, so bleibt nur die Möglichkeit, mit dem *head*-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 löschen 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.

```
class ListElem
{
private:
    ListElem* next;
    ListElem* prev;
    int data;
};
```

Der *prev*-Zeiger des ersten Elements zeigt wie der *next*-Zeiger des letzten Elements auf *NULL*. So lassen sich beide Enden der Liste terminieren.

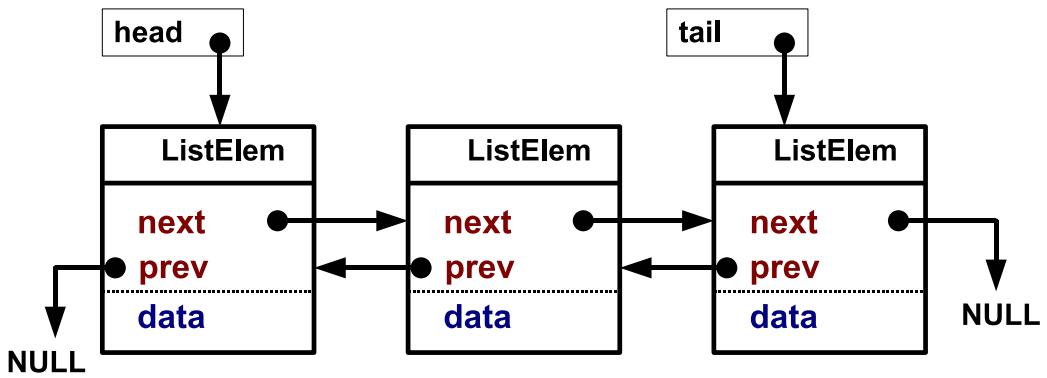


Abbildung 5.2: Schema einer doppelt verketteten Liste.

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 *enqueue* und *dequeue* zum Hinzufügen und Entfernen von Elementen. Die Implementierung erfolgt als verkettete Liste. Üblicherweise wird zusätzlich eine Funktion zur Verfügung gestellt, die anzeigt, ob die Datenstruktur leer ist.

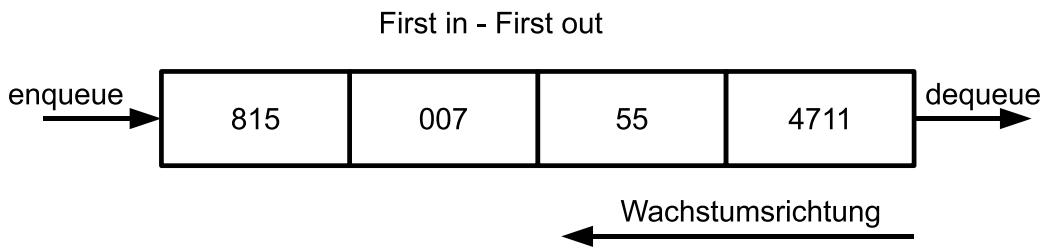


Abbildung 5.3: Schema einer Warteschlange/Queue/Puffer.

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.

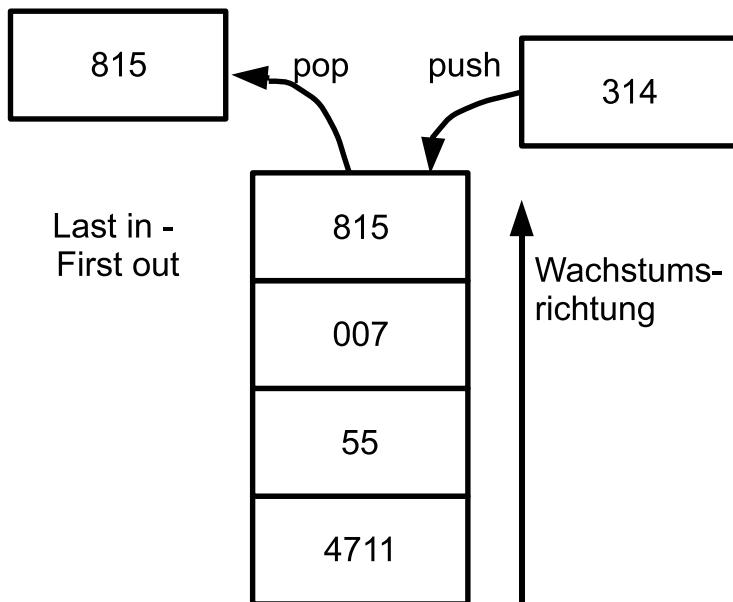


Abbildung 5.4: Schema eines Stapelspeichers/Stacks.

5.3 Zur Bearbeitung der Aufgaben

Für die Bearbeitung der Aufgaben wird Ihnen ein vorgefertigtes Codepackage zur Verfügung gestellt. Dieses soll mittels der Importfunktion importiert werden. Die Codefragmente sind den Aufgabenstellungen entsprechend zu ergänzen.

5.3.1 Anpassung eines Quellcodes an eine Code Convention

Es wird eine Struktur *Student* mit Attributen von Studenten zur Verfügung gestellt. Zudem gibt es eine Klasse *Stack*, die drei Funktionen und einen Konstruktor anbietet sowie eine Klasse *ListElem*, deren private Variablen über Setter und Getter erreichbar sind. Dieser Code ist nicht dokumentiert, schwer zu verstehen und fehlerhaft. Es ist bekannt, dass die Klasse *Stack* eine push, eine pop und eine Ausgabefunktion realisiert.

- Rücken Sie den Code wie in Kapitel 3.1.6 beschrieben ein.
- Benennen Sie die Variablen verständlich (*new_element*, *cursor*, *head*, *tail*, *next*, *data* sowie sinnvolle Namen für Funktionsparameter)
- Benennen Sie die Funktionen des Stacks nach ihrer Aufgabe in *push()*, *pop()* und *ausgabe()* um.
- Benennen Sie die Funktionen des Listenelements sinnvoll (*set...()*, *get...()*).
- Wieso ist die Ausgabenreihenfolge der Listenelemente anders als erwartet und wo liegt der Fehler?
- Implementieren Sie die fehlerhafte Methode derart, dass ein Stack vorliegt.
- Dokumentieren Sie die Klassen *Stack* und *ListElem*. Erstellen Sie dazu ein neues Doxygenfile mit den entsprechenden Einstellungen sowie eine HTML-Dokumentation. Bei Fragen lesen Sie bitte zuerst die Doxygen-Beschreibung im Kapitel 3.1.7.

5.3.2 Einfache Studentendatenbank

Ihre Aufgabe ist es nun, selbst Code nach dem bekannten Schema zu erstellen.

Gegeben ist die Klasse *List*, in der Studentendaten als einfach verkettete lineare Liste gespeichert werden. Die Struktur besitzt bereits eine Anhängemethode, eine Abhängemethode und eine Ausgabemethode, die die Liste von vorne nach hinten ausgibt. Im Hauptprogramm ist bereits eine textbasierte Menüsteuerung gegeben, um die Liste zu verwalten.

- Erweitern Sie die Liste derart, dass eine doppelt verkettete Liste vorliegt. Ändern Sie dazu auch die Methode *enqueue_head()*.
- Fügen Sie einen weiteren Menüpunkt „Datenbank in umgekehrter Reihenfolge ausgeben.“ hinzu.
- Als weiteren Menüpunkt implementieren Sie bitte „Datenelement löschen“. Es soll nach der Matrikelnummer gefragt und dann der passende Eintrag dazu gelöscht werden. Ein erfolgreicher Löschvorgang soll durch einen Hinweis signalisiert werden.
- Ergänzen Sie weiterhin einen sechsten Menüpunkt „Datenelement hinten ergänzen“, für den Sie die passende Methode implementieren.

6 Objektorientierte Techniken

Dieses Kapitel stellt Ihnen verschiedene weiterführende Techniken rund um die objektorientierte Programmierung vor. Nach dem Studium des Theorieteils und der Durchführung der praktischen Aufgaben

- können Sie Templates verwenden, um generische Programmierlösungen zu entwickeln.
- können Sie die Technik des so genannten *Überladens* verwenden, die nicht nur für Funktionen, sondern auch für Operatoren möglich ist.
- haben Sie sich einen Überblick über die **STL** (*Standard Template Library*) verschafft und sind in der Lage, eigenständige Recherchen in der Dokumentation der **STL** durchzuführen.

6.1 Template

Bei der Konzeption und Implementierung von Programmen kommt es häufig vor, dass gleichartige Funktionen für unterschiedliche Datentypen erforderlich sind. Stellen Sie sich vor, Sie wollen zwei Zahlen, die aus *int*-Werten bestehen, vergleichen und anschließend die größere Zahl ausgeben lassen. Sie schreiben dazu folgende Funktion:

```
|| int vergleich(int x, int y)
{  
    if (x > y)  
    {  
        return x;  
    }  
    else  
    {  
        return y;  
    }  
}
```

Aber sollen nun zwei *double*-Werte verglichen werden, erfordert dies die Implementierung einer neuen Funktion, die sich jedoch lediglich in der Signatur (d.h. den übergebenen Datentypen) von der ersten Implementierung unterscheidet. Der Teil des Codes, der die Funktionalität beinhaltet, wird dagegen direkt kopiert. Diese Duplizierung führt zu einem höheren Wartungsaufwand bei Adaptionen der Implementierung.

C++ stellt ein Konzept zur Verfügung, das die Trennung von Funktionalität und Datentyp erlaubt. Die so genannten *Templates* („Schablonen“) erlauben die Implementierung von generischen Klassen oder Funktionen, deren Datentyp erst bei der Instanziierung der Klasse oder beim Aufruf der Funktion festgelegt wird. Während der Erstellung der ausführbaren Datei legt der Compiler fest, welcher Datentyp Verwendung findet.

Syntaktisch beginnt jedes Template mit

```
|| template <class T>
```

wobei T als Platzhalter für den noch unbekannten Datentyp steht. Danach folgt die normale Definition der Klasse oder der Funktion, wobei T weiterhin als Platzhalter verwendet wird.

6.1.1 Template-Funktionen

Wenn Algorithmen unabhängig vom Datentyp implementiert werden sollen, verwendet man Template-Funktionen. Dieses Konzept verdeutlicht das folgende Beispiel, bei dem das Maximum von zwei Werten ermittelt wird:

```
template <class T>
const T& max (const T& x, const T& y)
{
    if (x > y)
    {
        return x;
    }
    else
    {
        return y;
    }
}
```

Es ist wichtig, dass der Operator „>“ für alle Datentypen definiert ist, die mit der Funktion zum Einsatz kommen (weiteres hierzu in Abschnitt 6.2.2). Das Template lässt sich nun wie folgt verwenden:

```
int main ()
{
    int a = 3;
    int b = 7;
    float c = 4.7;
    float d = 4.2;
    std::cout << "Maximum INT ist " << max(a,b) << std::endl;
    std::cout << "Maximum FLOAT ist " << max(c,d) << std::endl;
}
```

Anhand der Datentypen der jeweils übergebenen Variablen erkennt der Compiler, dass in Zeile 7 ganze Zahlen und Zeile 8 Gleitkommazahlen als Datentyp verwendet werden. Er erzeugt daher selbstständig zwei Funktionen, die sich im Datentyp unterscheiden. Der Entwickler muss so statt zwei nahezu identischer nur noch eine Funktion implementieren und pflegen.

6.1.2 Template-Klassen

Wenn Datenstrukturen unabhängig vom Typ der gespeicherten Elemente dargestellt werden sollen, verwendet man Template-Klassen. Dieses Konzept bietet sich insbesondere für Container-Strukturen wie Stacks, Listen oder Bäume an, die andere Objekte aufnehmen und verwalten. Für diese gibt es in der so genannten *Standard Template Library* bereits sehr effiziente Implementierungen. Diese werden wir noch im Abschnitt 6.3 behandeln. Eine Besonderheit von Templates ist, dass ihre Implementierung in der Header-Datei stehen muss. Das folgende Beispiel erläutert den Aufbau einer Template-Klasse. Es stellt eine LIFO-Datenstruktur (Stack) dar, deren Größe bei der Instanziierung mit dem Parameter *size* festgelegt werden

muss, z.B. `StackSpeicher<8, double> stack;`. Die Methoden `push` und `pop` erlauben das Einfügen und Entnehmen von Datenelementen:

```

template<int size, class T>
class StackSpeicher
{
    public:
        stackSpeicher();
        void push(const T& E);
        void pop(T& E);

    private:
        T space [size];
        int index;
};

/* Erzeugt einen leeren StackSpeicher */
template<int size, class T>
StackSpeicher<size, T>::stackSpeicher(): index(-1)
{
}

/* Fügt das Element E dem Stack hinzu */
template<int size, class T>
void StackSpeicher<size, T>::push(const T& E)
{
    if (index < (size-1))
    {
        index += 1;
        space[index] = E;
    }
}

/* Kopiert das oberste Element in den Parameter E */
template<int size, class T>
void StackSpeicher<size, T>::pop(T& E)
{
    if (index >= 0) // Stack nicht leer
    {
        E = space[index];
        index -= 1;
    }
}

```

6.2 Überladen

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 4.2.3) mit gleichem Namen zu definieren. Diese 'Mehrfach-Definitionen' werden auch als Überladen bezeichnet. Überladen ist nicht nur für Funktionen sondern auch für Operatoren möglich.

6.2.1 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.

6.2.2 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 aufgesetzt. In diesem Fall kann man z.B. die Vergleichsoperatoren (<, >, == ...) für Operanden vom Klassentyp überladen, um Objekte der Klasse einfach mit Hilfe der Operatoren manipulieren 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.

```
/* Überladung unärer Operatoren im Klassenbereich */
Rückgabewert operator <op>() // <op> ∈ !, ++, --, ...
{
    /* ... */
}

/* Überladung binärer Operatoren im Klassenbereich */
Rückgabewert operator <op>(Typ operand) // <op> ∈ +, -, ...
{
    /* ... */
}
```

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

```

class Student
{
public:
    Student(int matNr, const std::string &name, const std::string
            &vorname);
    bool operator == (const Student &student);
    int getMatNr();

private:
    int matNr;
    std::string name;
    std::string vorname;
};

Student::Student(int matNr, const std::string &name, const std::
                  string &vorname) :
    matNr(matNr), name(name), vorname(vorname)
{
}

bool Student::operator == (const Student &student)
{
    if (matNr == student.matNr)
        return true;
    else
        return false;
}

int Student::getMatNr()
{
    return matNr;
}

```

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

```

/* Überladung unärer Operatoren */
Rückgabetyp operator <op>(Klassentyp operand)
    // <op> ∈ !, ++, --, ...
{

/* Überladung binärer Operatoren */
Rückgabetyp operator <op>(Typ operand1, Typ operand2)
    // <op> ∈ +, -, ...
{
}

```

6 Objektorientierte Techniken

Der Vergleichsoperator `==` wurde wie folgt außerhalb der Klasse `Student` überladen:

```
bool operator == (Student student1, Student student2)
{
    // direkter Zugriff auf die Membervariablen nicht möglich
    if ( student1.getnMatNr() == student2.getnMatNr() )
        return true;
    else
        return false;
}
```

Der Vergleich von zwei Instanzen kann schließlich folgendermaßen durchgeführt werden:

```
int main()
{
    Student peter(222222, "Lustig", "Peter");
    Student max(223344, "Muster", "Max");
    if (peter == max)
    {
        std::cout<<"Max und Peter sind gleich"<<std::endl;
    }
    else
    {
        std::cout<<"Max und Peter sind nicht gleich"<<std::endl;
    }
}
```

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 kein Klassenobjekt darstellt. Dazu gehören beispielweise die Streamoperatoren `>>` und `<<`.

6.2.3 Überladung der Streamoperatoren

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) eine Instanz der Klasse `iostreams` darstellt, d.h. die Operatorüberladung müßte, wenn sie innerhalb einer Klasse implementiert werden soll, in der Klasse `iostreams` erfolgen. Das geht hier nicht, weil der Quellcode zu dieser Bibliothek nicht zur Verfügung steht. Die Überladung muß daher im globalen Bereich erfolgen. Da die Funktion `operator<<` außerhalb des Klassenbereichs definiert ist, hat sie auch keinen Zugriff auf die `private-` und `protected-` Datenelemente. Der zweite Übergabeparameter ist jedoch eine Instanz der eigenen Klasse und wird benutzt, um eine Ausgabefunktion aufzurufen, die innerhalb der Klasse implementiert ist und somit Zugriff auf alle Member hat.

```
ostream& operator << (ostream& out, Student& student)
{
    return student.ausgabe(out);
}
```

6.3 STL – Standard Template Library

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 Programmes 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 uns bisher bekannten Datenstrukturen, nur die Daten darin können dynamisch mitwachsen. Es gibt verschiedene Kategorien 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.. Aus Zeitgründen können wir diese nicht alle im Zuge des Praktikums einführen.

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:

```
template<class T> class std::vector
{
    //Membervariablen und Memberfunktionen
};
```

6.3.1 Vektor

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 uns bekannten Arrays sehr ähnlich. Im Unterschied zu einem gewöhnlichen Array kann ein *vector* allerdings dynamisch wachsen.

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

```
#include <vector>
```

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

```
vector<string> gartenCenter;
```

3. Jetzt brauchen wir einen passenden Iterator, um auf die Daten im Container zugreifen zu können.¹

```
typedef vector<string>::iterator itType;
itType it;
```

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

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

5. 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 ausführliche Dokumentation, Auflistung von Algorithmen und von Memberfunktionen der einzelnen Container folgen Sie bitte den Links in der Fußnote.² 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 #include <algorithm>
5 using namespace std;
6
7 int main()
```

¹Es gibt auch einen Iterator, der in umgekehrter Reihenfolge auf den Container zugreift. Dieser Iterator heißt *reverse_iterator*

²<http://www.cplusplus.com/reference/> oder <http://www.sgi.com/tech/stl/>

```

8  {
9      vector<string> gartenCenter;
10     typedef vector<string>::iterator iterTyp;
11
12     gartenCenter.push_back("Rosen");
13     gartenCenter.push_back("Heckenpflanzen");
14     gartenCenter.push_back("Bambus");
15     gartenCenter.push_back("Blütenstauden");
16     gartenCenter.push_back("Rhododendron");
17     gartenCenter.push_back("Nadelgehölze");
18
19     // Die Pflanzen werden lexikographisch absteigend sortiert
20     stable_sort(gartenCenter.begin(), gartenCenter.end(),
21                  greater<string>());
22
23     for (iterTyp it = gartenCenter.begin(); it != gartenCenter.
24           end(); it++)
25     {
26         cout << *it << endl;
27     }
28
29     // Die Anzahl der Elemente im Container wird zurückgegeben.
30     cout << gartenCenter.size() << endl;
31
32     return 0;
33 }
```

Dieses Beispielprogramm produziert folgende Ausgabe:

```

Rosen
Rhododendron
Nadelgehölze
Heckenpflanzen
Blütenstauden
Bambus
6
```

6.4 Aufgaben

Für die Bearbeitung der Aufgaben wird Ihnen ein vorgefertigtes Projekt zur Verfügung gestellt. Dieses soll mittels der Importfunktion importiert werden. Die Codefragmente sind den Aufgabenstellungen entsprechend zu ergänzen.

6.4.1 Template

Das Stack-Speicher-Template und die Student-Klasse werden bereits zur Verfügung gestellt und sollen nun erweitert werden:

1. Implementieren Sie einen Operator, der die Matrikelnummern der Objekte vergleicht um die Studenten-Klasse zu sortieren.

6 Objektorientierte Techniken

2. Implementieren Sie eine Methode *sort*, die den Stack per Quicksort-Verfahren sortiert³. Verwenden Sie dabei den zuvor implementierten Vergleichsoperator („ \leq “ bzw. „ \geq “) für *T*.
3. Ergänzen Sie die Student-Klasse um eine Member-Variable für das Geburtsdatum, die bereits bei der Instanziierung im Konstruktor gesetzt werden kann.
4. Erweitern Sie den Quellcode um einen Ausgabe-Stream-Operator „`<<`“, damit man die Daten eines Studenten direkt auf der Standardausgabe ausgeben kann (`cout << student1;`).
5. Erstellen Sie zum Testen der Funktionen einen Stack-Speicher der Größe 7 und füllen Sie diesen mit verschiedenen Instanzen der Klasse Student. Sortieren Sie den Stack und geben Sie anschließend die sortierten Daten der Studenten aus.

6.4.2 STL

Beim Lösen der Aufgabe unter 6.4.1 fallen folgende Punkte auf:

- Die Größe vom Stackspeicher muss immer festgelegt werden. Mehr Daten können nicht aufgenommen werden.
- Man muss viele Funktionen selbst implementieren, obwohl die C++-Standardbibliothek effizientere Implementierungen zur Verfügung stellt.

Machen Sie sich mit der STL vertraut und erweitern Sie die Codevorlagen um die folgenden Implementierungen.

1. anstatt der Klasse Stackspeicher in der Aufgabe unter 6.4.1 sollen Sie hier den Container *vector* verwenden.
2. Sie sollen die Methode *sort* nicht selbst implementieren. Verwenden Sie einfach die Algorithmen aus der STL.
3. Sie sollen einen Studenten laut der Matrikelnummer suchen und die Information von ihm ausgeben.
4. Löschen Sie diesen Studenten aus dem Container und geben Sie anschließend die restlichen Elemente wieder aus.

³Referenzimplementierung aus Grundgebiete Informatik 1 im Anhang

7 Vererbung und Polymorphie

Da das Thema äußerst komplex ist, kann es im Rahmen dieses Praktikums nur angerissen werden. Weiterführende Literatur und Quellen finden Sie im Literaturverzeichnis: [1, 2, 3, 5]

7.1 Vererbung — Hierarchien im Klassenkonzept

Bisher haben Sie Klassen nur als Gruppierung bzw. Klassifizierung von Objekten verwendet. In der OOP 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 – Unterklass Beziehung, die durch die sogenannte *Vererbung* realisiert wird.

Sehen Sie sich dazu einmal das in Abbildung 7.1 dargestellte Diagramm an. Die **Oberklasse** (auch **Basisklasse** genannt) *AllgemeineKlasse* besitzt ein Attribut und eine Methode. Die **Unterklasse** *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 Unterklasse 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]

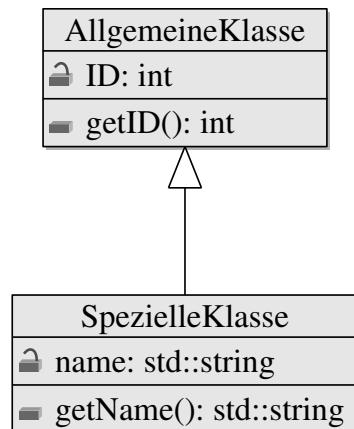


Abbildung 7.1: UML-Diagramm zur Beziehung zwischen Ober- und Unterklasse

Wir betrachten nun ein Programm, das mit den Angestellten eines Unternehmens arbeitet. Hierzu definieren wir eine Klasse *Employee*, die allgemein einen Angestellten des Unternehmens repräsentiert.

Zu jedem Angestellten des Unternehmens soll sowohl der Vorname als auch der Nachname

7 Vererbung und Polymorphie

und das Datum der Einstellung¹ gespeichert werden. Folgender Codeausschnitt zeigt die Definition der Klasse *Employee*:

```
class Employee
{
public:
    Employee(std::string, std::string, Date&);
    void print() const;

private:
    std::string firstName;
    std::string familyName;
    Date hiringDate;
};

Employee::print() const
{
    std::cout << firstName << " "
        << familyName
        << " has been hired on "
        << hiringDate.tostr()
        << std::endl;
}
```

Listing 7.1: 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 Basisklassen) erbt. Durch die Zugriffsspezifizierer kann festgelegt werden, ob die Zugriffsrechte der geerbten Elemente unverändert übernommen (*public*) oder nach außen beschnitten (*protected* und *private*) werden sollen. Die wichtigste und häufigste Form der Vererbung ist die *public*-Vererbung von einer Basisklasse. Wir 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 7.2 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*Date* ist eine erfundene Klasse, die ein Datum speichern kann. Sie wird hier nicht implementiert.

```

class Manager : public Employee
{
public:
    Manager(std::string, std::string, Date&);
    void addWorker(Worker*);
    void setLevel(int);

private:
    std::list<Worker*> group;
    int level;
};

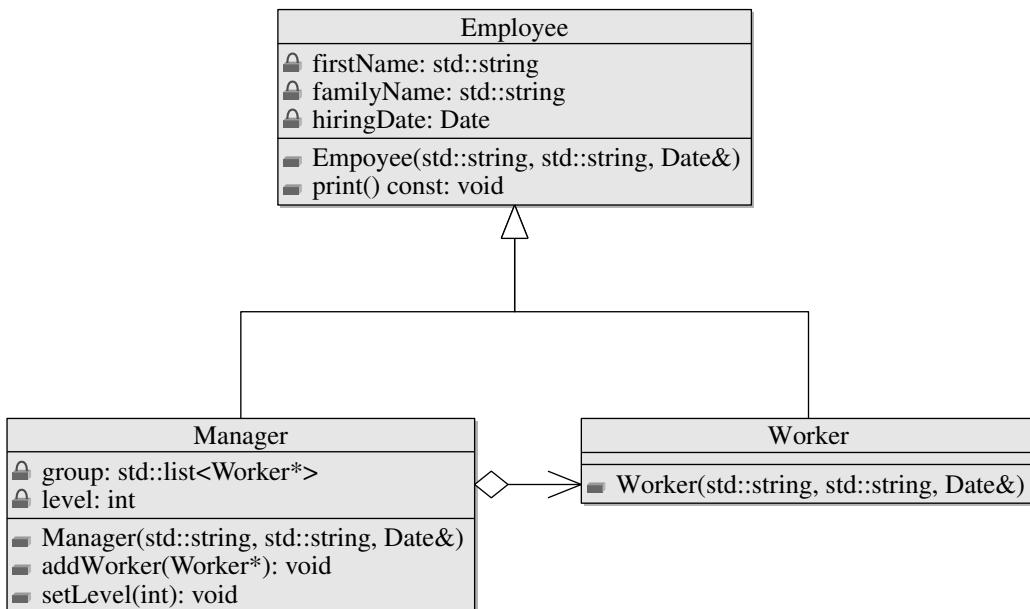
```

Listing 7.2: Die Klasse *Manager*

```

class Worker : public Employee
{
public:
    Worker(std::string, std::string, Date&);
};

```

Listing 7.3: Die Klasse *Worker*Abbildung 7.2: UML-Diagramm zu *Employee*, *Manager* und *Worker*.

Nun können wir schon das erste Programm mit den Klassen schreiben. Das Listing 7.4 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.

```

int main(int argc, char * argv[])
{
    Date heute(1,1,1970);
    Worker fritz("Fritz", "Outgoer", heute);
    Manager klaus("Klaus", "Leiter", heute);
    klaus.setLevel(2);
    klaus.addWorker(&fritz);

    klaus.print();
    fritz.print();
}

```

Listing 7.4: Aufruf von vererbten Methoden

7.2 Polymorphismus

In diesem Abschnitt wird ein weiteres Konzept der OOP, die *Polymorphie*, eingeführt. Polymorphie bedeutet im Kontext der Objektorientierung, dass sich verschiedene Objekte beim Aufruf derselben Methode unterschiedlich verhalten können.

Betrachten wir 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 dort nicht bekannt sind. Das folgende Beispiel zeigt, wie die Ausgabefunktion in der Unterklassse *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.

```

class Manager : Employee
{
public:
    /* ... */
    void print() const

private:
    std::list<Worker*> group;
    int level;
};

void Manager::print() const
{
    Employee::print();
    std::cout << "--> Manager at level " << level
        << " with " << group.size() << " Worker(s)"
        << std::endl;
}

```

Listing 7.5: 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 haben wir einen *Polymorphismus* erzeugt, da sich zwei Objekte beim Aufruf derselben Funktion unterschiedlich verhalten.

7.2.1 Späte Bindung

Häufig ist zur Übersetzungszeit eines Programms noch gar nicht klar, welchen Typ ein Objekt später haben wird.

Zum Beispiel wollen wir alle Angestellten unseres Unternehmens in einer Liste speichern. Wir wählen also eine Liste, die Zeiger auf *Employees* speichert und initialisieren sie mit den bereits bekannten Angestellten.

```
int main(int argc, char * argv[])
{
    std::list<Employee*> register;

    Date heute(1,1,1970);
    Worker *workerFritz
        = new Worker("Fritz", "Outgoer", heute);
    Manager *managerKlaus
        = new Manager("Klaus", "Leiter", heute);
    managerKlaus->setLevel(2);
    managerKlaus->addWorker(workerFritz);

    register.push_back(workerFritz);
    register.push_back(managerKlaus);

    return 0;
}
```

Listing 7.6: In diesem Programm wird eine Liste von Angestellten erstellt.

Wir haben also in einer Liste, die Elemente von Typ *Employee** erwartet, einen *Worker** und einen *Manager** gespeichert. Es ist tatsächlich immer möglich, dass Objekte von Unterklassen dort abgelegt werden, wo Objekte einer Oberklasse erwartet werden.

Nun wollen wir die soeben erstellte Liste durchgehen und für alle Elemente die *print()*-Methode aufrufen. Unsere *main* Funktion sieht dann folgendermaßen aus:

```
int main(int argc, char * argv[])
{
    /* ... */
    std::list<Employee*>::Iterator it;

    for (it = list.begin(); it != list.end(); it++)
    {
        it->print();
    }

    return 0;
}
```

|| }

Listing 7.7: Die Liste kann in einer Schleife durchlaufen werden.

Kompilieren wir nun unser kleines Programm und führen 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. Unsere Erweiterung der *print* Methode funktioniert also nur, wenn die Variable auch mit Typ *Manager* 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 wird, 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.

Nehmen wir nun also die notwendige Änderung in der Klasse *Employee* vor und deklarieren die Methode *print* virtuell.

```
class Employee
{
    /* ... */
    virtual void print() const
    /* ... */
};

class Manager : Employee
{
    /* ... */
    virtual void print() const
    /* ... */
};
```

```

Manager::print() const
{
    Employee::print();
    std::cout << "--> Manager at level " << level
        << " with " << group.size() << " Worker(s)"
        << std::endl;
}

```

Listing 7.8: 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.

7.3 Abstrakte Klassen und Interface-Klassen

In den vorherigen Abschnitten haben Sie sich intensiv mit Vererbung und weiterführenden Techniken der OOP auseinandergesetzt.

Vererbung in der OOP hat dabei keine Verwandtschaft zur biologischen Vererbung. Die zentrale Frage in der OOP 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 wir nur bei konkreten Tierarten — wie Löwen — antreffen werden. Von diesen abstrakten Begriffen machen wir in unserer 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 OOP. Und was dieser Begriff bedeutet, haben Sie an obigem Beispiel bereits gesehen. In abstrakten Klassen werden, wie wir es bereits kennen, Gemeinsamkeiten zusammengefasst. Allerdings wird es niemals Objekte dieser Basisklassen geben können, die nicht auch Objekte einer UnterkLASSE sind.

Um dieses Konzept in Zusammenhang mit der Programmierung in C++ zu bringen, betrachten wir 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 7.3 dargestellt.[2]

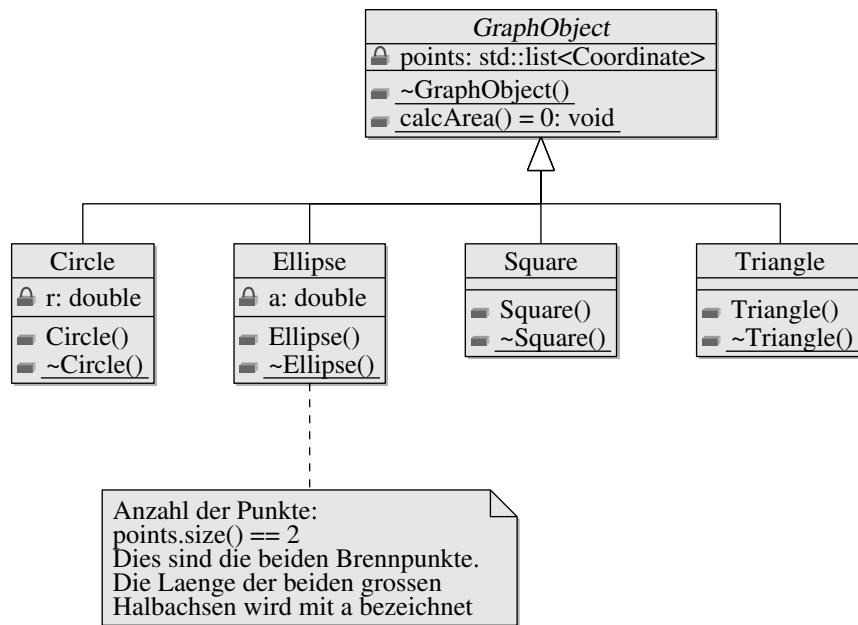


Abbildung 7.3: Ein Beispielprogramm für 2D Grafik

Das Beispiel enthält Klassen für 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 UnterkLASSE anders aussehen. Diese Klasse 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 OOP nennt man solche Methoden, die keine Implementierung haben, *abstrakte Methoden*. Man unterscheidet außerdem zwei Arten von Klassen, die abstrakte Methoden beinhalten. Eine davon haben Sie bereits kennengelernt.

Abstrakte Klassen haben mindestens eine abstrakte Methode.

Interface-Klassen haben nur abstrakte Methoden.

Von beiden Klassentypen gibt es keine direkten Exemplare. Alle Objekte müssen hier zugleich Objekte einer UnterkLASSE sein. Für keine dieser Klassentypen gibt es in C++ ein Schlüsselwort, weswegen sie nur eine Definition für bestimmte Klasseneigenschaften darstellen (s.u.).

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:

```

class GraphObject
{

```

```

public:
    /* ... */
    virtual void calcArea() = 0;
    /* ... */
};

```

Listing 7.9: 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.

Interface-Klassen nennt man in C++ alle Klassen, die nur rein virtuelle Methoden besitzen. Außerdem besitzen Interface-Klassen in dieser Sprache keine objektbezogenen Datenelemente.

7.4 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.

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 bitte 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]

7.5 Aufgaben

Sie bekommen den Auftrag, einen simplen Taschenrechner zu programmieren. Ihr Auftraggeber legt allerdings viel Wert darauf, dass er Ihren Code auch in eigenen Entwicklungen wiederverwenden kann. Aus diesem Grund möchte er, dass Sie den Taschenrechner objektorientiert implementieren.

Die mathematischen Berechnungen sollen dabei auf Ausdrücken basieren. Diese bestehen aus Operanden in Form von konstanten Zahlen und Operatoren für die Grundrechenarten +, -, * und /. Es soll eine möglichst große Flexibilität erreicht werden. Jede Operation soll mit jeder anderen Operation verknüpft werden können. Hierzu soll das Konzept der abstrakten Datentypen benutzt werden. Jede Operatorklasse arbeitet dabei mit Operanden, die nur durch eine abstrakte Basisklasse *Expression* gegeben sind.

Gegeben ist eine Klasse *Expression* mit zwei virtuellen Methoden. Die Funktion *evaluate()* berechnet den Ausdruck und gibt den berechneten Wert als double zurück. Die Methode *print()* erstellt eine für den Menschen lesbare Form des Ausdrucks, der anschließend mithilfe von *cout* ausgegeben werden kann.

7.5.1 Ein einfacher Taschenrechner

1. Damit Sie und Ihr Auftraggeber dieselben Vorstellungen von der Klassenhierarchie haben, sollen Sie zunächst ein Klassendiagramm zeichnen, das, neben der schon vorhanden *Expression*-Klasse, auch je eine Klasse für die vier Grundrechenarten (*Add*, *Sub*, *Mul*, *Div*) enthält. Zeichnen Sie auch schon zwei weitere Klassen ein, die eine einfache Zahl und ein Ergebnis repräsentieren (*Const*, *Result*). All diese Klassen sollen von *Expression* erben und die virtuellen Funktionen der *Expression*-Klasse implementieren.
Neben diesen Funktionen sollen die Klassen auch jeweils eine Referenz auf die Operanden speichern, mit denen sie arbeiten sollen. Diese Zeiger sollen beim Erstellen des Objektes übergeben werden. *Const* speichert, anders als die übrigen Operatoren, eine Zahl als *double*, während *Result* einen Zeiger auf den gesamten Ausdruck speichert.
2. Legen Sie ein neues Projekt an und importieren Sie die im Ordner Versuch07 aus dem Vorlagenverzeichnis zur Verfügung gestellten Dateien. Die *Expression*-Klasse ist hier schon implementiert und es existiert eine *main*-Funktion, die Sie zum Testen Ihres Programmes nutzen können.
3. Damit Ihr Programm immer getestet werden kann, implementieren Sie zuerst nur die Klassen *Add* und *Const* aus Ihrem Klassendiagramm. Legen Sie für jede Klasse eine neue *cpp*- und eine *h*-Datei an. Mithilfe des Codes in der Funktion *testAddConst()* können Sie testen, ob Ihre Implementierung die gewünschte Funktionalität hat.
4. Beachten Sie auch die Ausgabe. Sowohl der Konstruktor als auch der Destruktor der *Expression*-Klasse zeigt jedes Mal an, wenn er aufgerufen wurde. Sind alle Objekte, die beim Durchlauf des Programmes erstellt wurden, auch wieder erfolgreich gelöscht worden?
Überlegen Sie sich, welche Objekte gelöscht wurden und welche nicht. Denken Sie auch darüber nach, wie Sie dafür sorgen können, dass alle Objekte gelöscht werden, sobald das Programm durchgelaufen ist.
5. Implementieren Sie die Klasse *Result*, die das Endergebnis verwaltet und somit die Schnittstelle zu einem komplizierten Ausdruck bildet.
Aktualisieren Sie die *testAddConst()* Funktion und testen Sie Ihre neue Klasse.
6. Implementieren Sie nun die fehlenden Klassen und testen Sie diese mithilfe der Funktion *finalTest()*

8 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 graphische Benutzeroberfläche, auch GUI 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 graphischen Benutzeroberfläche bereitstellt. Des Weiteren werden Sie in diesem Versuch eigenständig ein großes Softwareprojekt erstellen und dabei die meisten Konzepte aus den vorhergehenden Versuchen erneut anwenden.

- Grundlagen der GUI-Programmierung
- Grundlagen von *Qt*
- Erstellung eines größeren Softwareprojekts

8.1 Qt Referenz

Bei *Qt*¹ handelt es sich um eine *C++*-Klassenbibliothek, mit der plattformübergreifend graphische 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 graphischer Benutzeroberflächen. *Qt* wird unter einer sehr liberalen Open Source Lizenz veröffentlicht: Der Quellcode ist einsehbar und man darf *Qt* nahezu beliebig in eigenen Projekten nutzen. *Qt* unterstützt alle aktuellen Betriebssysteme sowohl im Desktop- als auch im Mobile-Bereich und es gibt für viele weiterverbreitete Programmiersprachen eine API, sodass man *Qt* nicht nur in Verbindung mit *C++* nutzen kann.

Im Folgenden sollen einige Klassen vorgestellt werden, die verschiedene Teile der Benutzeroberfläche realisieren. Dabei werden nur auf Grundlagen adressiert. Für eine detailliertere Beschreibung wird auf die gut lesbare Dokumentation des *Qt*-Projekts [9] verwiesen.

In den Beispielen zu einzelnen GUI-Elementen wird meist relativ umfangreicher Beispielcode präsentiert. Eine *Qt*-Installation umfasst ein Hilfsprogramm zur Erstellung des Oberflächen-Entwurfs, den *Qt Designer*. Dieser kann im praktischen Einsatz einen großen Teil des Rahmen-codes automatisch generieren. Wir werden dies hier kaum nutzen, um mit der „handkodierten“ Version detailliert zu verstehen, was im Hintergrund passiert.

¹ausgesprochen wie das englische Wort *cute*

8.1.1 Beispielprogramm

Qt bietet die Möglichkeit, mit wenigen Befehlen eine funktionsfähige graphische Oberfläche zu implementieren. Dies soll durch das folgende Codebeispiel verdeutlicht werden:

```

1 #include <QApplication>
2 #include <QPushButton>
3
4 int main( int argc, char* argv[] )
5 {
6     QApplication app(argc, argv);
7
8     QPushButton hello("Hello world!");
9     hello.resize(100, 30);
10
11    hello.show();
12
13    return app.exec();
14 }
```

Die Header-Datei *QApplication.h* definiert die Klasse *QApplication*, die für die allgemeine Verwaltung der Applikation von der Initialisierung bis zum Beenden des Programms sorgt und in jeder *Qt*-Anwendung genau einmal instanziert werden muss. Die Klasse *QApplication* enthält als Parameter dieselben Argumente, die auch der main-Funktion übergeben wurden. Als elementares graphisches Element, bietet *QPushButton.h* Funktionalitäten zur Verwaltung einer einfachen Schaltfläche. Eine detaillierte Beschreibung der Funktionen finden Sie im weiteren Verlauf dieses Kapitels.

Während die ersten Befehle der Erstellung der Oberfläche dienen, startet mit der Methode *exec()* die sog. *Ereignis-Hauptschleife*. Diese Schleife wartet auf Ereignisse, z.B. in Form von Benutzereingaben, um diese dann z.B. an die graphischen Elemente 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.

8.1.2 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 graphischen Element oder auch vom eigenen Programm gesendet wird.

Slot Ein Slot ist grundsätzlich eine *normale* Funktion, die mit einem Signal verknüpft 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.

Diese Sender-Empfänger-Beziehung zwischen Signalen und Slots wird in *Qt* mit Hilfe der Funktion *QObject::connect* hergestellt. Diese benötigt folgende Parameter:

1. Referenz auf das Objekt, welches das Signal ausgibt.
2. Das Signal, das verbunden werden soll.
3. Referenz auf das Objekt, das den Slot enthält.
4. Der Slot, der verbunden werden soll.

Soll zum Beispiel bei Klick auf den Button das Programm schließen, könnte man dies wie folgt realisieren:

```

1 #include <QApplication>
2 #include <QPushButton>
3
4 int main( int argc, char* argv[] )
5 {
6     QApplication app(argc, argv);
7
8     QPushButton hello("Hello world!");
9     hello.resize(100, 30);
10    QObject::connect(&hello, SIGNAL(clicked()),
11                      &app, SLOT(quit()));
12
13    hello.show();
14
15    return app.exec();
16 }
```

Hier wird also das Signal *clicked* des Buttons *hello* mit dem (von *Qt* intern bereitgestellten) *quit*-Slot unserer *QApplication* verbunden.

8.1.3 QString

Die Klasse *QString* ist dem *std::string* sehr ähnlich. Die grundlegenden Funktionen werden alle auch von *QString* unterstützt und um zusätzliche Funktionen ergänzt.

Die wichtigsten Funktionen sind *QString::fromStdString(const std::string &str)* und *toStdString()* welche eine Konvertierung zwischen den beiden Stringtypen ermöglichen.

8.1.4 QMessageBox

Die Klasse *QMessageBox* öffnet einen modalen Dialog, mit dem Nutzer informiert oder befragt werden können. Eine *QMessageBox* hat standardmäßig nur eine Schaltfläche, die mit Ok beschriftet ist. Mit 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 Ganzzahlwert zurückgibt, der anzeigt, 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 <QApplication>
2 #include < QMessageBox >
3
4 int main (int argc, char* argv[])
5 {
6     QApplication app(argc, argv);
7
8     QMessageBox msgBox;
9     msgBox.setText("The document has been modified.");
10    msgBox.setInformativeText("Do you want to save your changes?
11        ");
12    msgBox.setStandardButtons(QMessageBox::Save |
13                             QMessageBox::Discard |
14                             QMessageBox::Cancel);
15    msgBox.setDefaultButton(QMessageBox::Save);
16    int ret = msgBox.exec();
17
18    QMessageBox result;
19    switch (ret)
20    {
21        case QMessageBox::Save:
22            result.setText("Save was clicked.");
23            app.quit();
24            break;
25        case QMessageBox::Discard:
26            result.setText("Don't save was clicked.");
27            app.quit();
28            break;
29        case QMessageBox::Cancel:
30            result.setText("Cancel was clicked.");
31            app.quit();
32            break;
33        default:
34            result.setText("This should be impossible");
35            app.quit();
36            break;
37    }
38    result.exec();
39
40    return app.exec();
}

```

8.1.5 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 Konstruk-

tor 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`. Ein Beispiel-Programm ist im Abschnitt 8.1.2 zu finden.

8.1.6 QRadioButton

Ein `QRadioButton` ist ebenfalls eine Schaltfläche und hat damit ähnliche Eigenschaften wie ein `QPushButton`. Allerdings kann ein `QRadioButton` zwei Zustände haben, nämlich `checked` oder eben nicht. Mithilfe der Funktion `isChecked()` kann der aktuelle Wert ausgelesen werden. Radio-Buttons nutzt man in der Regel, wenn man aus einer Gruppe von Alternativen nur eine einzige auswählen darf. Im folgenden Beispiel werden zwei `QRadioButton` und ein `QPushButton` erzeugt und angezeigt. Je nach ausgewähltem `QRadioButton` wird eine `QMessageBox`, ein kleines Nachrichtenfenster, mit einer entsprechenden Botschaft angezeigt.

```

1 #include <QPushButton>
2 #include <QRadioButton>
3 #include <QVBoxLayout>
4 #include <QWidget>
5 #include <QMessageBox>
6
7 class RadioButtonExample : public QWidget
8 {
9     /*
10      * Dieses Macro enthält zusätzliche Anweisungen für den
11      * Compiler, die Qt genau dann benötigt, wenn eine
12      * Klasse von QObject erbt.
13      */
14     Q_OBJECT
15
16 public:
17     RadioButtonExample();
18
19 private slots: // Hier werden eigene Slots definiert
20     void button_clicked();
21
22 private:
23     QPushButton* button;
24     QRadioButton* yesRadioButton;
25     QRadioButton* noRadioButton;
26 };

```

radiobuttonexample.h

```

1 #include "radiobuttonexample.h"
2
3 RadioButtonExample::RadioButtonExample()
4 {
    // Erstellen der Elemente

```

```

6     button = new QPushButton("Wählen Sie aus");
7     yesRadioButton = new QRadioButton("Ja");
8     noRadioButton = new QRadioButton("Nein");
9
10    /*
11     * Verknüpfung des clicked()-Signals des Buttons
12     * mit unserer Funktion.
13     */
14    connect(button, SIGNAL(clicked()),
15            this, SLOT(button_clicked()));
16
17    /*
18     * Ein Layout wird benötigt, um mehrere Elemente
19     * darzustellen, in diesem Fall werden die
20     * Elemente einfach untereinander angeordnet
21     */
22    QVBoxLayout* layout = new QVBoxLayout();
23    layout->addWidget(button);
24    layout->addWidget(yesRadioButton);
25    layout->addWidget(noRadioButton);
26    setLayout(layout);
27}
28
29 void RadioButtonExample::button_clicked()
30{
31    QMessageBox msgBox; // Hier wird eine MessageBox erstellt
32    // Überprüfen, welcher RadioButten ausgewählt wurde
33    if (yesRadioButton->isChecked() == true)
34    {
35        msgBox.setText("Sie haben Ja ausgewählt");
36    }
37    else if (noRadioButton->isChecked() == true)
38    {
39        msgBox.setText("Sie haben Nein ausgewählt");
40    }
41    else
42    {
43        msgBox.setText("Bitte wählen Sie.");
44    }
45    msgBox.exec(); // Anzeigen der MessageBox
46}

```

radiobuttonexample.cpp

```

1 #include <QApplication>
2 #include "radiobuttonexample.h"
3
4 int main (int argc, char* argv[])
5 {
6     QApplication app(argc, argv); // Erstellen der QApplication

```

```

7  /*
8   * Erstellen eines Widgets von unserer
9    * selbstdefinierten Klasse
10  */
11 RadioButtonItemExample widget;
12 widget.show();                      // Anzeigen des Widgets
13 return app.exec();                  // Ausführen des Programms
14 }
```

main.cpp

8.1.7 QLabel

Ein *QLabel* ist ein Element zur Anzeige von Text. Interessant sind die Eigenschaften *size* und *text*. Im folgenden Beispiel zeigt ein Label den Text *Hello World!*:

```

1 #include < QApplication >
2 #include < QLabel >
3
4 int main (int argc, char* argv[])
{
5     QApplication app(argc, argv);    // Erstellen der QApplication
6
7     QLabel label;
8     label.setText("Hello World!");  // Text Setzen
9     label.resize(100, 30);          // Größe Setzen
10
11    label.show();
12    return app.exec();             // Ausführen des Programms
13 }
14 }
```

8.1.8 QLineEdit

Ein *QLineEdit* ist dem *QLabel* sehr ähnlich, allerdings kann hier auch der Nutzer Text eingeben. Den vom Benutzer eingegebenen Text liest man aus mit der Funktion *text()*, 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 #ifndef LINEEDITEXAMPLE_H
2 #define LINEEDITEXAMPLE_H
3
4 #include < QWidget >
5 #include < QPushButton >
6 #include < QLineEdit >
7 #include < QVBoxLayout >
8 #include < QMessageBox >
9
10 class LineEditExample : public QWidget
```

```

11 | {
12 |     Q_OBJECT
13 |
14 | public:
15 |     LineEditExample();
16 |
17 | private slots:
18 |     void button_clicked();
19 |
20 | private:
21 |     QPushButton *button;
22 |     QLineEdit *lineEdit;
23 | };
24 |
25 | #endif // LINEEDITEXAMPLE_H

```

lineeditexample.h

```

1 #include "lineeditexample.h"
2
3 LineEditExample::LineEditExample()
4 {
5     lineEdit = new QLineEdit();
6     button = new QPushButton("Anzeigen");
7
8     connect(button, SIGNAL(clicked()),
9             this, SLOT(button_clicked()));
10
11    QVBoxLayout* layout = new QVBoxLayout();
12    layout->addWidget(lineEdit);
13    layout->addWidget(button);
14
15    setLayout(layout);
16 }
17
18 void LineEditExample::button_clicked()
19 {
20     QMessageBox msgBox;
21     /*
22      * Kopiert den Text in die MessageBox
23      */
24     msgBox.setText(lineEdit->text());
25     msgBox.exec();
26 }

```

lineeditexample.cpp

```

1 #include <QApplication>
2 #include "lineeditexample.h"
3

```

```

4 | int main (int argc, char* argv[])
5 | {
6 |     QApplication app(argc, argv); // Erstellen der QApplication
7 |
8 |     LineEditExample widget;
9 |     widget.show();
10 |
11 |     return app.exec();           // Ausführen des Programms
12 |

```

main.cpp

8.1.9 QMenu

Ein Menü² besteht in *Qt* aus mehreren Elementen. Die Klasse *QMenuBar* bietet ein Grundgerüst, in dem die verschiedenen Drop-Down-Menüs angelegt werden können. Zu dieser Menüleiste können nun einzelne Objekte der Klasse *QMenu* hinzugefügt werden, die die verschiedenen Menüs mit weiteren Menüeinträgen repräsentieren. Die Menüeinträge sind vom Typ *QAction*. Wichtiges Attribut eines *QMenu* ist *text*, welches den Anzeigetext in der Menüleiste bestimmt. Die *QAction* Klasse bietet neben dem *text*-Attribut auch ein Signal mit dem Namen *triggered()* an, das mit einem Klick auf den entsprechenden Eintrag emitiert wird.

Beim Konstruktor der *QAction*-Klasse gibt es eine kleine *Qt* spezifische Besonderheit. Man übergibt nämlich nicht nur den Text, der im Menü angezeigt werden soll, sondern auch ein so genanntes *parent*-Objekt. Dieser Mechanismus stellt sicher, dass mit einem übergeordneten *parent*-Objekt ebenfalls alle untergeordneten Objekte automatisch gelöscht werden. *Qt* kümmert sich um die Verwaltung der Objekthierarchie und nimmt dem Programmierer an dieser Stelle die Arbeit ab, den Menüeintrag mit *delete* zu löschen, wenn das übergeordnete Fenster gelöscht werden soll.

Zu einem Menü können einzelne Einträge mit dem Befehl *addAction(QAction *action)* hinzugefügt werden. Alternativ kann man mit dem Befehl *addAction(const QString &title)* ebenfalls einen Menüeintrag hinzufügen. In diesem Fall erhält man einen Zeiger auf den erstellten Menüeintrag zurück.

Einen Separator, also ein Trennelement, das Gruppen von Menüeinträgen graphisch voneinander abhebt, erstellt man durch *addSeparator()*.

Der Menüleiste kann man mit dem Befehl *addMenu(QMenu *menu)* ein Menü hinzufügen, und auch hier kann man alternativ einen *QString* übergeben, um einen Zeiger auf das erstellte Menü zurückzubekommen.

Im folgenden Beispiel wird in die Menüleiste ein einzelnes Menü eingefügt. Dieses Menü enthält drei Einträge. Zwei öffnen einfach nur eine *QMessageBox* und der dritte Menüeintrag schließt das Programm.

```

1 | #include < QMainWindow >
2 | #include < QMenuBar >
3 | #include < QMessageBox >
4 |
5 | /*
6 |  * Wir nutzen hier ein QMainWindow, da dieses bereits

```

²Gemeint ist hier eine Menüleiste am oberen Rand des Fensters.

```

7  * eine Instanz von QMenuBar enthält
8  */
9 class MenuExample : public QMainWindow
10 {
11     Q_OBJECT
12
13 public:
14     MenuExample();
15
16 private slots:
17     void firstActionClicked();
18     void secondActionClicked();
19
20 private:
21     QMenu* fileMenu;
22     QAction* closeAction;
23     QAction* firstAction;
24     QAction* secondAction;
25 };

```

menuexample.h

```

1 #include "menuexample.h"
2
3 MenuExample::MenuExample()
4 {
5     /*
6      * Hier wird eine neue QAction erstellt. "Close" ist dabei
7      * das, was die Action anzeigt, und der this Zeiger wird
8      * übergeben, um der QAction mitzuteilen, zu welchem
9      * Fenster sie gehört. Das ist zum Beispiel nötig, um
10     * dafür zu sorgen, dass auch die QAction gelöscht wird,
11     * wenn das Hauptfenster gelöscht wird.
12     */
13     closeAction = new QAction("Close", this);
14     /*
15      * Wir verbinden das Triggered-Signal unserer QAction mit
16      * dem close-Slot unseres QMainWindow. Damit wird das
17      * Programm beendet, wenn auf diesen Eintrag geklickt wird.
18     */
19     connect(closeAction, SIGNAL(triggered()),
20             this, SLOT(close()));
21     firstAction = new QAction("First Action", this);
22     connect(firstAction, SIGNAL(triggered()),
23             this, SLOT(firstActionClicked()));
24     secondAction = new QAction("Second Action", this);
25     connect(secondAction, SIGNAL(triggered()),
26             this, SLOT(secondActionClicked()));
27
28     /*

```

```

29 * Wir fügen ein Menü mit dem Namen "File" unserer
30 * Menüleiste hinzu
31 */
32 fileMenu = menuBar()->addMenu("File");
33 // Hinzufügen der verschiedenen Menüeinträge
34 fileMenu->addAction(firstAction);
35 fileMenu->addAction(secondAction);
36 /*
37 * Ein Separator trennt die einzelnen Menüeinträge
38 * optisch voneinander ab.
39 */
40 fileMenu->addSeparator();
41 fileMenu->addAction(closeAction);
42 }
43
44 void MenuExample::firstActionClicked()
45 {
46     QMessageBox msg;
47     msg.setText("You clicked on the first QAction");
48     msg.exec();
49 }
50
51 void MenuExample::secondActionClicked()
52 {
53     QMessageBox msg;
54     msg.setText("You clicked on the second QAction");
55     msg.exec();
56 }
```

menuexample.cpp

```

1 #include <QApplication>
2 #include "menuexample.h"
3
4 int main (int argc, char* argv[])
5 {
6     QApplication app(argc, argv); // Erstellen der QApplication
7
8     MenuExample window;
9     window.show();
10
11    return app.exec();           // Ausführen des Programms
12 }
```

main.cpp

8.1.10 QGraphicsView

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 zum Beispiel Funktionen zur Verfügung, die Ellipsen, Rechtecke oder auch Linien zeichnen. Die Syntax lautet wie folgt:

- `addRect(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())`
- `addEllipse(qreal x, qreal y, qreal w, qreal h, const QPen &pen = QPen(), const QBrush &brush = QBrush())`
- `addLine(qreal x1, qreal y1, qreal x2, qreal 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. Der Text kann im Konstruktor übergeben (z.B. `QGraphicsTextItem("Hallo Welt!")`) oder mithilfe der Funktion `setPlainText(const QString &text)` direkt gesetzt werden. Die Position wird mit der Funktion `setPos(qreal x, qreal y)` gesetzt.

Im folgenden Beispiel wird in einen Zeichenbereich eine gelbe Linie gezeichnet und auch ein roter Punkt gesetzt, der mit einem Text beschriftet wird.

```

1 #include < QApplication >
2 #include < QGraphicsView >
3 #include < QGraphicsTextItem >
4
5 int main (int argc, char* argv[])
6 {
7     QApplication app(argc, argv);
8
9     QGraphicsView view; // Erstellen der QGraphicsView.
10    QGraphicsScene scene; // Erstellen der QGraphicsScene.
11
12    /*
13     * Man kann keine einzelne Punkte erstellen,
14     * sondern nur eine Elipse.
15     */

```

```

16     scene.addEllipse(100, 100, 4, 4, QPen(Qt::red),
17                         QBrush(Qt::red, Qt::SolidPattern));
18     QPen pen;
19     pen.setWidth(5);
20     pen.setColor(Qt::yellow);
21     scene.addLine(10, 10, 190, 10, pen);
22     // Eine Klasse, um Text anzuzeigen
23     QGraphicsTextItem *text = new QGraphicsTextItem;
24     text->setPos(80, 70);           //Position des Textes
25     text->setPlainText("Hallo Welt!"); // Text
26     scene.addItem(text);
27
28     /*
29      * Wir übergeben unsere GraphicsScene der
30      * GraphicsView, um sie anzuzeigen.
31      */
32     view.setScene(&scene);
33     // Die GraphicsView wird in der Application angezeigt.
34     view.show();
35
36     return app.exec();
37 }
```

8.1.11 QMainWindow

Die *QMainWindow*-Klasse bietet ein Hauptfenster für die Applikation. Darin wird die programmspezifische grafische Benutzeroberfläche eingebettet. Ein *QMainWindow* besteht aus einer Menüleiste (*QMenuBar*), einer horizontalen Leiste auf der Statusinformationen angezeigt werden (*QStatusBar*), einem Panel, auf dem Kontrollelemente angeordnet werden (*QToolBar*) und einem Bereich, in dem die Hauptkomponenten der Benutzeroberfläche liegen (*Central Widget*). Zusätzlich können Objekte vom Typ *QDockWidget* hinzugefügt werden, um zum Beispiel eine Seitenleiste zu realisieren.

8.1.12 QDialog

Dialoge sind Fenster, die eine Interaktion mit dem Benutzer ermöglichen. Eigene Dialoge werden von *QDialog* abgeleitet. Dieses Vorgehen erlaubt es, das Aussehen und die Funktion von Dialogen mit wenig Aufwand wie gewünscht zu gestalten, die „Routineaufgaben“ übernimmt nach wie vor die Basisklasse *QDialog*.

Alle Dialoge lassen sich mit der Funktion *exec()* anzeigen. Die Handhabung wurde mit der Klasse *QMessageBox*, die von *QDialog* erbt, in vorhergehenden Beispielen gezeigt.

8.2 Aufgaben

In diesem Versuch erstellen Sie ein Programm zur Darstellung einer einfachen Karte. Es wird dem Nutzer die Möglichkeit bieten Städte, sowie fiktive 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 i.W. 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. Zudem finden Sie im L²P eine funktionierende Demoversion des Programms.

8.2.1 Eine neue Applikation

Legen Sie ein neues Projekt vom Typ *Qt-Widgets-Anwendung* mit dem *Qt Creator* in Ihrem Benutzeroberverzeichnis an. Geben Sie dem Projekt den Namen *StreetPlanner*.

Die Vorlage für *Widgets*-Anwendungen beinhaltet drei Dateien (jeweils eine Header-, Quell- und Formulardatei), die ein leeres Fenster beschreiben. Mit dem grünen Start-Knopf lässt sich das (noch funktionslose) Programm bereits starten, das Hauptfenster erscheint.

Einfache GUI-Elemente

Fügen Sie nun einige einfache GUI-Elemente zum Hauptfenster hinzu.

1. Klicken Sie doppelt auf die Formulardatei, um diese mit einigen Elementen zu füllen. Es öffnet sich daraufhin der *Qt Designer*. Hier sehen Sie im Hauptbereich 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 zuerst ein *Grid Layout* auf das Fenster. Dieses Element hilft dabei, GUI-Elemente tabellenartig anzurichten und deren Größe festzulegen, nachdem diese dem Layout hinzugefügt wurden.
3. Ziehen Sie weitere Elemente in das Layout und ordnen Sie diese anhand folgender Tabelle an. Um das *Graphics View* über mehrere Tabellenzellen anzurichten, ziehen Sie es über den entsprechenden Rand.

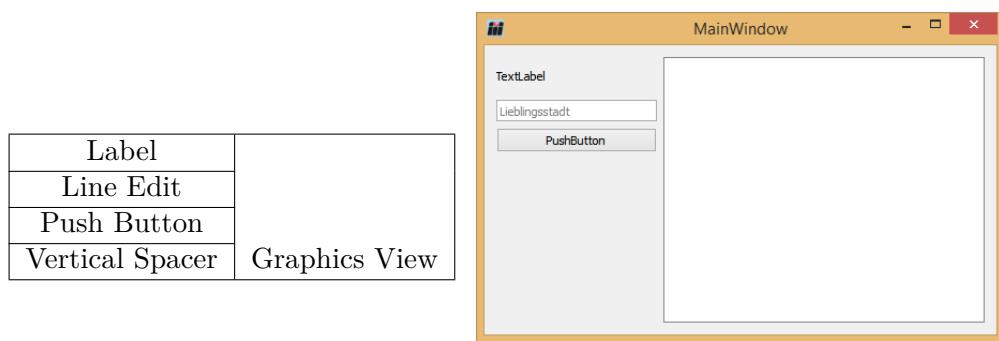


Abbildung 8.1: Anordnung erster Elemente

4. Auf der rechten Seite werden die Einstellungen der GUI-Elemente angezeigt. Nutzen Sie die Eigenschaft *text*, um die Beschriftung des *Push Button* und des *Label* zu ändern. Definieren Sie auch die Eigenschaft *placeholderText* des Elements *Line Edit*.
5. Führen Sie einen Rechtsklick auf den Bereich aus, in dem das *MainWindow* noch nicht durch ein anderes GUI-Element verdeckt wird, und wählen Sie *Layout → Objekte waagerecht anordnen*, damit das Layout über das gesamte Fenster gestreckt wird.

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 Menüpunkt *Slot anzeigen* aus. Die erscheinende Liste zeigt alle Signale, die dieser Knopf aussenden kann. Wählen Sie das Signal *clicked()*: Dadurch wurde ein Slot 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.
 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*.
- Hinweis:** Auf die GUI-Elemente wird über den Zeiger *ui* zugegriffen. Zum Beispiel ist *ui->lineEdit* ein Zeiger auf das GUI-Element *Line Edit*.
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*.

Die GUI zeichnet

1. Legen Sie in der Klasse *MainWindow* ein Attribut vom Typ *QGraphicsScene* an.
2. Übergeben Sie diese im Konstruktor der Klasse *MainWindow* an das GUI-Element *Graphics View*.
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 *qrand()*9/RAND_MAX*.

8.2.2 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.

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.

2. Implementieren Sie eine Funktion `draw(QGraphicScene& 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 und erstellen Sie einen Slot, der bei jedem Klicken des Knopfs aufgerufen wird.
4. Erstellen Sie in diesem Slot zwei Städte, die „sich selbst“ in dem *Graphics View* zeichnen.
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.

Hinweise:

Nutzen Sie für den Konstruktor die Signatur `City(QString name, int x, int y)`.

Nutzen Sie im *Qt Designer* auch die Eigenschaft `objectName`. Hiermit definieren Sie Namen, über die GUI-Elemente im Code angesprochen werden.

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 abstrakten *Interface*-Klasse ableiten, die im L²P 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 hält.
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`, die das Zeichnen aller Städte veranlasst.
5. 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. Legen Sie die `Map` als ein privates Attribut in der Klasse `MainWindow` an.
6. Kopieren Sie mit dem Windows-Explorer die Datei `abstractmap.h` in das Projektverzeichnis. Wählen Sie im *Qt Creator* im Kontextmenü des Projektverzeichnisses den Punkt `Existierende Datei Hinzufügen` und fügen Sie diese Datei 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 das Interface entspricht. Gehen Sie sicher, dass die Testfunktion noch einwandfrei funktioniert.

Hinweis: Zu dem Versuch sind einige Vorschläge für Testfunktionen beigelegt.

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. Bauen Sie auch die Klasse `Map` soweit aus, 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 Karte 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.
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.

Hinweis: In der Klasse `AbstractMap` sind Typ-Definitionen für Stadtlisten und Straßenlisten vorhanden, die Sie verwenden können.

8.2.3 Karte bearbeiten

In dieser Teilaufgabe widmen Sie sich erneut der GUI. Sie werden dem Benutzer eine Möglichkeit geben Städte einzutragen. Hierzu werden Sie einen Abfragedialog 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, wie bei einer einfachen Schaltfläche (`QPushButton`), den Slot `clicked()`. Setzen Sie dort die Eigenschaft `visible` aller Test-Elemente.

Hinweis: Der Status der `Check Box` 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 → Dateien und Klassen → Qt → Qt-Designer-Formularklasse → Dialog with Buttons`. Fügen Sie diesem GUI-Elemente hinzu, sodass der Nutzer die notwendigen Informationen eingeben kann.
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 [9] nach, wie der Rückgabewert der Funktion definiert ist. Testen Sie mit Hilfe einer Debug-Ausgabe wie der Rückgabewert sich 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.

8.2.4 Karte einlesen

In dieser Teilaufgabe werden Sie das Programm soweit 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 soweit vorbereiten, dass verschiedene Datenquellen genutzt werden können. Dem Versuch ist eine Interface-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 Dummy-Implementierung bei, die eine fest vorgegebene Karte erzeugt.

1. Fügen Sie die Dateien *mapio.h*, *mapiodummy.h* und *mapiodummy.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 *MapIo-Dummy* 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.

8.2.5 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 das Interface *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 ein.
3. Gehen Sie sicher, dass sie den Test durchführen können. Dieser muss komplett durchlaufen werden und sollte aktuell noch auf Fehler in der Implementierung hinweisen.
4. Implementieren Sie nun schrittweise die Interface-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 Implementierungszustand 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 nun die Straßen, die zu dem Weg gehören, in rot als breite Linie. Ergänzen Sie dazu die Funktion `drawRed(QGraphicScene& scene)` in der Klasse *Street*.
3. Fügen Sie GUI-Elemente dem Hauptfenster hinzu, die für eine freie Wegesuche notwendig sind und implementieren Sie die Suche.

8.2.6 Wahlpflichtaufgaben

Entwickeln Sie das Programm weiter, indem Sie mindestens zwei der folgenden Aufgaben bearbeiten.

Benutzer 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.

Straßen einfügen

Implementieren Sie einen Dialog, der es dem Benutzer ermöglicht, auch Straßen einzufügen.

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.

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. Ersetzten 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.

³Dieser Algorithmus ist in der Lage, in einem Graphen einen kürzesten Weg zwischen zwei Knoten zu bestimmen. Siehe Anhang

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 *mapiodummy.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.

8.3 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. Teileimplementierungen 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 Programms 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

```
void quick_sort (int A[], int l, int r)
{
    int k;
    if (r <= l)
        return;
    k = partition(A, l, r);
    quick_sort(A, l, k - 1);
    quick_sort(A, k + 1, r);
}

int partition (int A[], int l, int r)
{
    int i, j, k, v;
    k = r;
    v = A[k];
    i = l;
    j = r - 1;
    while (1)
    {
        while (A[i] <= v && i < r)
            i++;
        while (A[j] >= v && j >= l)
            j--;
        if (i >= j)
            break;
        else
            exchange(A[i], A[j]);
    }
}
```

¹Beispiele und Erklärung aus dem Grundgebiete Informatik 1 Skript von Herrn Professor Leupers

```

    }
    exchange(A[i], A[k]);
    return i;
}

```

Dijkstra

Vorbemerkungen:

- Voraussetzung: Datenstruktur für Graphen (*graph*) und Mengen (*set*) verfügbar
- *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

```

void shortest_path(graph G = (V = {v0, ..., vn}, E))
{
    set GREEN = {}, YELLOW {v0}, RED = {};
    dist[v0] = 0;
    while (YELLOW != {}) /* solange noch unbearbeitete Knoten */
    {
        v = MinDist(YELLOW); /* v ∈ YELLOW mit minimalem dist-Wert */
        Insert(v, GREEN); /* GREEN = GREEN ∪ {v} */
        Delete(v, YELLOW); /* YELLOW = YELLOW \ {v} */

        for (w ∈ Succ(v)) /* für alle Nachfolger w von v */
        {
            if (!(w ∈ YELLOW ∪ GREEN)) /* neuer Knoten erreicht */
            {
                Insert({v, w}, RED);
                Insert(w, YELLOW);
                dist[w] = dist[v] + cost(v, w);
            }
            else if (w ∈ YELLOW) /* w erneut erreicht */
            {
                if (dist[v] + cost(v, w) < dist[w])
                {
                    Insert({v, w}, RED);
                }
            }
        }
    }
}

```

```
    e = PreviousEdge(w); /* vorher rote Kante zu
                           w */
    Delete(e, RED);
    dist[w] = dist[v] + cost(v, w);
}
}
}
}
```


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] Peter Damann, *Unterrichtsmaterial für das Fach Informatik*,
Bezirksregierung Düsseldorf.
<http://www.brd.nrw.de/lerntreffs/informatik/structure/material/sek2/delphi/einfuehrungdelphi/startseite.php>
- [5] Wikipedia-Artikel zu Vererbung:
[https://de.wikipedia.org/wiki/Vererbung_\(Programmierung\)](https://de.wikipedia.org/wiki/Vererbung_(Programmierung))
- [6] C/C++ Development User Guide
http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.cdt.doc.user/concepts/cdt_o_home.htm
- [7] Einführung in C++ von IBM
<http://www.ibm.com/developerworksopensource/library/os-eclipse-stlcdt/index.html>
- [8] Doug Schaefer,
<http://adobedev.adobe.acrobat.com/p20583465/>
- [9] Qt Referenz,
<http://doc.qt.io/qt-5/index.html>

Index

- Überladen, 95
 - Funktionen, 96
 - Operatoren, 96
 - Streamoperator, 98
- abstrakte Klassen, 109
- abstrakte Methoden, 110
- Anfrage, 66
- Arrays, 24
 - als Funktionsargumente, 44
 - eindimensionale, 25
 - Initialisierung, 25
 - mehrdimensionale, 25
- Auftrag, 66
- Ausführen, 11
- Basisklasse, 103
- Block, 30
- break, 51
- Call by reference, 43
- Call by value, 42
- Code Convention, 54
- Compilieren, 10
- continue, 52
- Datenobjekte, 19
- Datenstrukturen, 24
- Datentypen, 19
 - void, 42
- Debuggen, 11
 - Bedinung in Eclipse, 14
- Defaultparameter, 77
- Deklarationen, 20
- Delegationsprinzip, 65
- Destruktor, 70
- do while, 51
- Doxygen
 - Doxyfile, 57
 - Syntax, 55
- Dynamische Speichernutzung, 34
- enum, 28
- Erstellen eines Projektes, 9
- Felder, *siehe* Arrays
- for, 49
- Funktionen, 39
 - Call by reference, 43
 - Call by value, 42
 - Gültigkeitsbereich von Parametern, 41
 - Konstante Parameter, 41
- Gültigkeitsbereiche, 31
- if-else, 45
 - else if, 45
- Include-Wächter, 81
- Initialisierungsliste, 72
- Interface-Klassen, 110
- Klassen, 64
 - Konstante Funktionsparameter, 41
 - Konstanten, 21
 - Konstruktor, 68
- Liste
 - doppelt verkettet, 89
 - einfach verkettet, 88
- Member, 64
- Oberklasse, *siehe* Basisklasse
- Objektorientierte Programmierung, 63
- OOP, 63
 - abstrakte Klassen, 109
 - abstrakte Methoden, 110
 - rein virtuelle Methoden, 110
 - Call By Reference, 77
 - Interface-Klassen, 110
 - Klassen, 64
 - Anfrage, 66
 - Attribute, 65
 - Auftrag, 66
 - Destruktor, 70
 - Initialisierungsliste, 72
 - Konstruktor, 68
 - Methoden, 65
 - Standardkonstruktor, 70

Index

- This-Zeiger, 67
- Zugriffsbeschränkungen, 75
- Klassen in C++, 64
- Konstante Methode, 78
- Konstante Parameter, 77
- Member, 64
- Objekte, 63
- Polymorphie, 106
 - späte Bindung, 108
 - virtuelle Methoden, 108
- Statische Attribute, 79
- Statische Methode, 80
- Vererbung, 103
 - Basisklasse, 103
 - Oberklasse, *siehe* Basisklasse
 - Unterklasse, 103
 - Zugriffsspezifizierer, 104
 - virtuelle Destruktoren, 111
- Operatoren, 21
 - arithmetische, 21
 - Dekrement, 22
 - Inkrement, 22
 - logische, 22
 - Vergleich, 21
- Pointer, 33
- Polymorphie, 106
- private, *siehe* Zugriffsbeschränkungen
- protected, *siehe* Zugriffsbeschränkungen
- public, *siehe* Zugriffsbeschränkungen
- Qt, 113
 - Ereignis-Hauptschleife, 114
 - QApplication, 114
 - QBrush, 124
 - QDialog, 125
 - QGraphicsView, 124
 - QLabel, 119
 - QLineEdit, 119
 - QMainWindow, 125
 - QMenu, 121
 - QMMessageBox, 115
 - QPen, 124
 - QPushButton, 114, 116
 - QRadioButton, 117
 - QString, 115
 - Signal-Slot-Konzept, 114
- Queue, 90
- Referenzen, 33
- rein virtuelle Methoden, 110
- Sichtbarkeiten, 34
- späte Bindung, 108
- Speicherbereich, 34
- Stack, 90
- Standard Template Library, *siehe* STL
- Stapelspeicher, 90
- static, 79
- STL, 99
 - Vektor, 100
- Strings, 26
- Strukturen, 26
 - Deklaration, 27
 - in Feldern, 27
 - Kopieren, 27
- Strukturierte Programmierung, 63
- switch, 47
- Template, 93
 - Funktionen, 94
 - Klasse, 94
- Testgetriebene Entwicklung, 53
- Typumwandlung, 23
- Union, 29
- Unterklsse, 103
- Variablen, 19
 - Wertebereiche, 19
- Vereinbarungen, 20
- Vererbung, 103
- virtuelle Destruktoren, 111
- virtuelle Methoden, 108
- void, 42
- Vorausdeklaration, 82
- Warteschlange, 90
- while, 50
- Zeiger, 33
- Zugriffsbeschränkungen, 75
- Zugriffsspezifizierer, 104