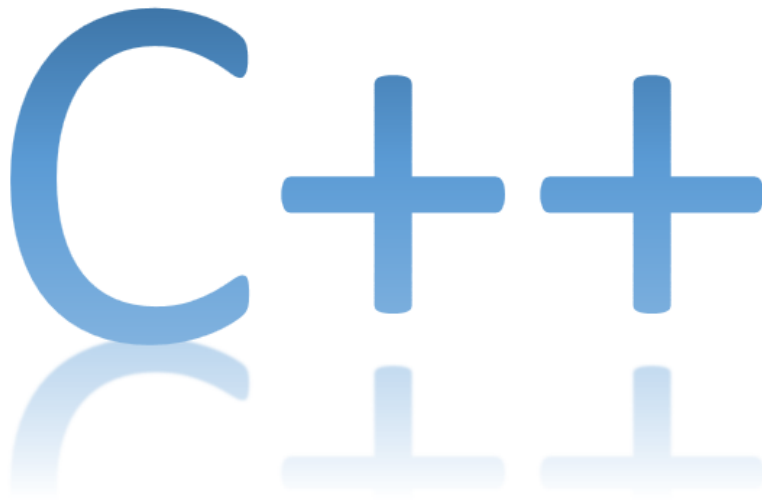


Version 0.7

Objektorientiert Programmieren



Ulrich Schrader

Nur zur Verwendung in der Vorlesung - Nicht weitergeben

Inhaltsverzeichnis

1	Einleitung	5
1.1	Die Arbeitsumgebung	5
1.2	Endlich beginnen	6
1.2.1	Elemente der objektorientierten Programmierung	6
1.3	Hello World!	7
1.4	Ein bisschen Interaktivität	10
1.5	Challenges	11
1.5.1	Freundliches Programm	11
1.5.2	Kalender	11
2	Umgang mit grundlegenden Datentypen	12
2.1	Datentypen	12
2.2	Variablen	13
2.3	Ausdrücke, Zuweisungen, Operatoren	16
2.3.1	Ausdrücke und Zuweisungen	16
2.3.2	Operatoren	17
2.4	Automatische Variablen mit <code>auto</code>	18
2.5	Präprozessor	20
2.6	Challenges	22
2.6.1	Formatierungsoptionen von <code>cout</code>	22
2.7	Konstante	23
2.8	Arrays	24
2.9	Strings	26
2.10	Typkonvertierung - <i>type casting</i>	28
2.11	Challenges	31
2.11.1	Primzahlen	31
2.11.2	Zinsentwicklung	31
2.11.3	ASCII-Tabelle	31
3	Strukturen, Klassen und Pointer	32
3.1	Strukturen mit <code>struct</code>	32
3.2	Klassen	34
3.3	Challenge	39
3.3.1	Klasse <code>Student</code>	39
3.4	Compiler und Linker	39
3.5	Programm- und Headerdateien	41

3.6	Pointer	45
3.7	Challenge	48
3.7.1	Klasse <code>student</code> reloaded	48
3.8	Referenzen	48
3.9	Parameterübergabe an Funktionen	50
3.9.1	Parameterübergabe per <i>value</i>	50
3.9.2	Parameterübergabe per Pointer	51
3.9.3	Referenzen: Nützlich für viele Zwecke	51
3.10	C++ Standard Template Library STL - Vector	53
3.11	Challenges	56
3.11.1	Header-Datei für Klasse <i>Vorlesung</i>	56
3.11.2	Implementationsdatei für Klasse <i>Vorlesung</i>	56
3.11.3	Vorlesungsübersicht	57
4	Ablaufkontrolle	58
4.1	<code>if ... else ...</code> sowie <code>while</code> und <code>do ... while</code>	58
4.2	<code>for ...</code> Schleifen	58
5	Klassendiagramme der Unified Modelling Language UML	61
5.1	UML Klassennotation	61
5.2	Sichtbarkeit	62
5.3	Richtungsabhängigkeit	63
5.4	Blickweisen auf Klassendiagramme	63
5.5	Beziehungen zwischen Klassen	65
5.5.1	Vererbung oder Generalisierung	65
5.5.2	Assoziation	67
5.5.3	Kardinalität	67
5.5.4	Aggregation	67
5.5.5	Komposition	68
5.5.6	Abhängigkeit (Dependency)	68
5.5.7	Realisation	68
5.5.8	Klassendiagramm für ein Bestellsystem	69
5.6	Challenges	69
5.6.1	Erstellen eines Klassendiagramms	69
5.6.2	Lokalisieren des Programmcodes	73
5.6.3	Modularisierung des Programmcodes	74
6	Paradigma Objektorientierte Programmierung OOP	75
6.1	Programmierparadigmen	75
6.1.1	Funktionale und logische Paradigmen	75
6.1.2	Objektorientiertes Paradigma	75
6.2	OOP-Grundkonzept	76
6.3	Warum Objekte	76

Inhaltsverzeichnis

6.4	Challenges	77
6.4.1	Modularisieren	77

1 Einleitung

1.1 Die Arbeitsumgebung

C++ ist eine der älteren Programmiersprachen, aber auch eine der mächtigsten und weit verbreiteten Programmiersprachen, die heute noch verwendet werden. Die meisten Teile von Betriebssystemen, Datenbanken, Webserver, etc. sind in C++ geschrieben. Diese Vorlesung wird Sie mit den Grundlagen der objektorientierten Programmierung in dieser Sprache vertraut machen und Ihnen die grundlegenden Elemente näher bringen. Es ist keine komplette Einführung in die Sprache C++, da Kenntnisse, die Sie bereits in der C-Programmierung erworben haben, vorausgesetzt werden. Nichts desto trotz wird es gerade am Anfang eine Reihe von Überlappungen geben, wenn es darum geht die Unterschiede aufzuzeigen. Obwohl es nicht erforderlich ist eine integrierte Entwicklungsumgebung (Integrated Development Environment IDE) zu nutzen, macht eine solche Umgebung die Arbeit einfacher, da in ihr der Editor, der Compiler und Linker integriert sind. Ich persönlich nutze

- Visual Studio Community
- CodeBlocks

Beide stehen für nicht-kommerzielle Arbeiten kostenfrei zur Verfügung. Auf den Rechnern der Hochschule ist Eclipse installiert, welches ebenfalls ein kostenfreies IDE anbietet. Hinweise zur Installation unter Windows finden Sie in nachfolgendem Link:

https://www3.ntu.edu.sg/home/ehchua/programming/howto/EclipseCpp_HowTo.html

Für Unix-basierte System ist es noch einfacher. Meine Empfehlung ist daher, diese IDE auch auf Ihren Rechnern zu installieren, damit Sie dann in einer vertrauten Umgebung - gerade auch bei der Klausur - arbeiten können. Abschließend benötigen Sie noch eine Referenz für die C++ Sprache. Ich empfehle hierfür cppreference.com. Hierbei handelt es sich um ein frei zugängliches Wiki, dass als Referenz für C und C++ gilt. Dieses Skript übernimmt viele Inhalte aus dem sehr guten C++ Kurs von Eduardo Corpeño auf LinkedIn-Learning, sowie Inhalte aus den folgenden Quellen:

- Kaiser U.; Guddat M.: C/C++ Das umfassende Lehrbuch. Galileo Computing, 2014.
- Dmitrovic S.; Modern C++ for Absolute Beginners. Apress, 2020.
- Briggs W.; C++20 for Lazy Programmers. Apress, 2021.

- Ekellem, F; Einführung in das Konzept der objektorientierten Anwendungen zu wissenschaftlichen Rechnens mit C++ und Matlab. Präsentation, FH Köln, 2013. http://www.gm.fh-koeln.de/~afomusoe/SS2013/Mathe_CPP_Vorlesung04.pdf.

1.2 Endlich beginnen

1.2.1 Elemente der objektorientierten Programmierung

Obwohl der Name C++ es nahelegt, ist C++ keine einfache Erweiterung der Programmiersprache C, sondern eine eigenständige Sprache mit einigen deutlichen Unterschieden, so dass nicht einfach C in C++ enthalten ist. Der wesentliche Unterschied ist, dass C++ eine objektorientierte Programmiersprache ist. Was bedeutet das? Zunächst versucht C++ die Realität zu *modellieren*. Dabei ist ein Modell als eine Repräsentation der Wirklichkeit zu verstehen. Modellierung ist ein ganz wesentliches Konzept der objektorientierten Programmierung. Dabei kann alles mehr oder weniger gut modelliert werden: eine Kuh, ein Auto, eine Person, eine Studentin, ... Der trickreiche Teil ist dabei, welche Elemente und Aspekte in dem Modell zu berücksichtigen sind, damit es der Aufgabe des Programms gerecht werden kann.

Nehmen wir an, wir wollen ein Auto modellieren, dann könnte man Eigenschaften aufnehmen, die jeden Aspekt des Autos, den man sich denken kann, berücksichtigen: Hersteller, Gewicht, Eigentümer, Farbe, Alter, Kilometerstand, Diese Liste könnte nahezu unendlich fortgesetzt werden, und wir würden nie mit dem Modellieren fertig werden. Daher besteht die Herausforderung des Modellierens darin, zu entscheiden, welche Eigenschaften relevant sind. So mögen der Preis, Hersteller, Modell, Anzahl der Sitze und Leistungsdaten für einen Händler relevant sein. Für einen Reisenden ist es eher ein Vehikel um von A nach B zu gelangen, daher mag ihn interessieren, wie viele Passagiere und wie viel Gepäck er mitnehmen kann. Der Betreiber eines Parkhauses ist eher an Daten interessiert, die Auskunft über die Größe des Autos geben.

Ein weiterer wesentliche Aspekt der objektorientierten Programmierung ist *Kapselung*. Kapselung bedeutet dabei, dass alle Daten und Operationen innerhalb des Modells gehalten werden. Gut definierte Modelle enthalten daher nur Informationen über sich selber. Die modellbezogenen Operationen sind knapp und präzise. Wenn eine Funktion A und B machen soll, dann macht sie auch nichts anderes. Daher sollte das Modell eines Autos etwa nicht die Kosten zum Füllen des Tanks beinhalten, da dieses eher als Bestandteil des Modells einer Tankstelle zu verstehen ist.

Als weiteres gibt es sogenannte *Klassen*, die in C++ als Konstrukte für Modelle anzusehen sind. Klassen beinhalten zwei Typen von Informationen: Daten und Funktionen, die oft auch als Methoden der Klasse bezeichnet werden. In unserem Beispiel mag unsere Autoklasse ein Datenelement zum Hersteller des Autos beinhalten, zum Modell sowie eine Funktion zum Ändern der Farbe des Autos. Eine spezielle Instanz der Klasse wird als *Objekt* bezeichnet. Wenn wir also drei verschiedenen Variablen der Klasse Auto haben, dann

haben wir damit drei Objekt vom Typ `Auto`, die alle über die *Eigenschaften (Attribute)* Hersteller, Modell und Farbe verfügen und sich darin unterscheiden können.

Darüber hinaus können Elemente der Klasse *public* (öffentlich) oder *private* (privat) sein. Öffentliche Elemente sind aus jeder Stelle des Programms ansprechbar, wohingegen private Elemente nur innerhalb der Klasse selber verwendet werden können. Aber es gibt noch eine dritte Möglichkeit den Zugriff zu regeln, *protected* (geschützte) Elemente einer Klasse können in sogenannten vererbten Klassen verwendet werden. Dieses führt auf die Konzepte der *inheritance* (Vererbung) und der *Polymorphismen*, die in C++ umgesetzt sind. So kann eine *subclass* (Unterklasse) Element der *superclass* erben. Zum Beispiel mag die Klasse `Tier` ein Datenelement für die Anzahl der Beine enthalten. Dieses wird von der Unterklasse `Hund` geerbt. Unterklassen können weitere Elemente mit aufnehmen, So könnte die Unterklasse `Vogel` ein Element für Flügel ergänzen. Eine weiterer interessanter Aspekt im Zusammenhang mit Vererbung ist der Polymorphismus. Dabei definiert die Überklasse eine Funktion, aber die vererbte Funktion wird in einigen Unterklassen anders implementiert, und macht diese Unterklassen damit polymorph. Diese Konzepte mögen Ihnen von anderen Programmiersprachen wie Java oder Python schon vertraut sein. C++ setzt all dieses um, und noch viel mehr.

1.3 Hello World!

Um Ihnen ganz einfach C++ schon einmal nahezubringen, setzen wir wieder das schon bekannte Programm `hello world` um, dass nichts anderes macht, als eine Nachricht auf den Bildschirm der Konsole zu senden. Dieses einfache traditionelle Programm dient dazu, einen ersten Einblick in die Struktur und die Syntax zu bekommen.

Listing 1.1: Hello World

```

1 // HelloWorld1 - ohne Namespace
2 //
3
4 #include <iostream>
5
6 int main()
7 {
8     std::cout << "Hello_World!" << std::endl;
9 }
```

Wie Sie sehen können werden nur wenige Zeilen benötigt. Zuerst wollen wir die Bibliotheken, die wir verwenden wollen, einfügen. Dazu verwenden wir die Präprozessoranweisung `#include`, auf die wir später noch eingehen werden. Wir verwenden eine Bibliothek mit dem Namen `iostream`, die Teil der C++ Standardbibliotheken ist. Sie enthält u.a. Objekte und Funktionen, um Text auf dem Bildschirm darzustellen oder Text von der Tastatur zu erhalten. Präprozessoranweisungen enden nie mit einem Semicolon.

Als nächstes haben wir mit der von C bekannten Hauptfunktion `main()` den Einstiegs-

1 Einleitung

punkt des Programms. Jedes C++ Programm beginnt mit der `main()` Funktion und gibt einen Integerwert zurück. In diesem Fall bekommt sie keine Argumente übergeben, ansonsten würde sie ein Array von Strings übergeben bekommen, die beim Aufruf des Programms mit angegeben wurden.

Der Rumpf jeder Funktion ist ein Anweisungsblock, der in geschweiften Klammern eingeschlossen ist, die den Anfang und das Ende des Blocks angeben. Wie in C muss jetzt entschieden werden, wie wir die geschweiften Klammern setzen wollen. Dafür gibt es viele Möglichkeiten. Ich bevorzuge die öffnende Klammer am Ende des Kontrollblocks und die schließende Klammer in einer eigenen Zeile mit derselben Einrückung wie die öffnende Zeile. Sie werden diese Formatierung bei allen längeren Programmen wiederfinden. Diese Form wird von vielen anderen ebenfalls verwendet.

C++ Programme würde nahezu unleserlich, wenn diese nicht konsistent durch Einrückungen strukturiert werden. Einrückungen sind zwar für den Compiler nicht erforderlich, aber helfen dem Menschen den Überblick zu behalten. Die meisten Entwicklungsumgebungen unterstützen automatisch bei einer konsistenten Einrückung. Andere Programmiersprachen wie etwa Python verlangen Einrückungen, damit das Programm korrekt interpretiert werden kann. Gewöhnen Sie sich daher an eine konsistente Einrückung entweder immer um eine feste Anzahl Leerzeichen oder einen Tabulator.

Aufgabe: Recherchieren Sie mittels Google, welche weiteren Formen der Einrückung es gibt, indem Sie nach "Indentation Styles" suchen, und entscheiden Sie sich für Ihren Stil.

Zurück zum Programm, die erste Zeile der `main()` Funktion ist die Anweisung, die unsere Nachricht ausgibt. Dazu verwenden wir ein Objekt der `iostream` Bibliothek, das Bestandteil der Standardbibliothek von C++ ist. Diese Zugehörigkeit wird durch den Scope-Resolution-Operator `::` ausgedrückt. `std::` bedeutet also, das nachfolgende gehört zum Namensraum der Standardbibliothek von C++. Diese ist eine wichtige Eigenschaft, da es gerade bei umfangreichen Anwendungen bei denen viele beispielsweise Funktionen entwickeln, oder auch extern entwickelte Funktionen eingesetzt werden, es zu Namensüberschneidungen kommen kann. Diese können durch Namensräume aufgelöst werden.

Das hier verwendete Objekt ist `cout`, und steht für Character Out, also Zeichenausgabe. Es handelt sich dabei um eine Instanz der Klasse `ostream`, die generell für Output-Streams entwickelt wurde. Mit dem `<<` Operator können Variablen und Konstante zur Ausgabe an das `cout` Objekt gesandt werden. Dieses kann auch verkettet werden, indem nacheinander weitere Inhalte an des Objekt gesandt werden, wie in diesem Beispiel gezeigt, indem zunächst "Hello World" und dann das Zeilenendezeichen geschickt wird, dieses könnte das bekannte `"\n"` oder die in `iostream` definierte Konstante `endl` sein, die im Namensraum `std` definiert ist. Wie in C wird jede Anweisung mit einem Semikolon beendet.

Zum Schluss, da `main()` eine Funktion, die einen Integerwert zurückgibt, ist, geben wir noch 0 als Rückgabewert an. Dabei kann der Wert in Klammern eingeschlossen werden, dieses muss aber nicht der Fall sein. Es wird 0 zurückgegeben, da dieses traditionell für einen korrekten Ablauf des Programms steht. Ein von 0 verschiedener Wert bedeutet meist einen Fehlercode, den Sie natürlich in der Dokumentation des Programms näher erläutert haben. Damit ist das Programm abgeschlossen.

Um sich häufige Angaben des jeweiligen Namensraumes zu erleichtern, kann man diesen auch einmal festlegen. Dieses werde ich in vielen der Programmbeispiele hier verwenden, da damit die Zeilen etwas kürzer werden und nicht dauernd umgebrochen werden müssen. Unser Programm lautet dann:

Listing 1.2: Hello World unter Verwendung des Namensraumes `std`

```

1 // HelloWorld2.cpp - Verwendung von namespaces
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "Hello_World!" << endl;
11 }
```

Da `cout` und `endl` Elemente des `std` Namensraumes sind, wird mit

```
using namespace std;
```

erreicht, dass nicht mehr angegeben muss, dass diese Elemente aus dem Namensraum `std` kommen, und man spart sich die Angabe von `std::`. Von manchen wird diese Lösung als Stützräder für Anfänger bezeichnet, und Sie können sich entscheiden auf diese Zeile zu verzichten, da dieses häufig als schlechter Programmierstil angesehen wird, und es sich eine Reihe von Problemen daraus ergeben können.

Aufgabe: Recherchieren Sie mittels Google, welche Vor- und Nachteile sich aus der obigen Verwendung von `using namespace` ergeben, und entscheiden Sie, ob und wann Sie es verwenden wollen.

Mein Grund für die Verwendung in den Beispielen dieses Skripts ist, dass die Zeilen dann etwas kürzer werden, wenn der Namensraum nicht immer angegeben werden muss, und die Zeilen dann nicht so oft umgebrochen werden müssen. Ansonsten würde ich aber darauf verzichten.

Das Objekt `cout` bietet eine sehr einfache Möglichkeit der Ausgabe in dem Konsolfenster. Dabei werden unterschiedliche Datentypen von dem Objekt sinnvoll behandelt. Das nachfolgende Beispiel zeigt das auf:

1 Einleitung

Listing 1.3: Verwendung von cout

```
1 // CoutExample.cpp : Umgang mit dem cout Objekt
2 //
3
4 #include <iostream>
5
6 int main()
7 {
8     int i = 42;
9     float a = 3.141;
10    std::string s = "Hi there!";
11    std::cout << "Variable i equals " << i << std::endl;
12    std::cout << "Variable a equals " << a << std::endl;
13    std::cout << "Variable s equals " << s << std::endl;
14    return(0);
15 }
```

Das Objekt `cout` hat die unterschiedlichen Datentypen der Variablen automatisch so in auszugebene Zeichenketten gewandelt, dass sinnvolle Ausgaben erzeugt werden, wie in dem Listing 1.3 gezeigt wird. Damit bietet `cout` eine sehr einfache und komfortable Form der Ausgabe in dem Konsolfenster an. Übrigens ist mit dem Objekt `cerr` dasselbe für Fehlermeldungen möglich, die standardmäßig im Konsolfenster angezeigt werden, aber auch leicht in etwa Dateien zur Protoklierung der Fehlermeldungen umgelenkt werden können.

1.4 Ein bisschen Interaktivität

Die meisten Programme leben davon, dass der Anwender etwas eingibt, worauf das Programm dann in einer vordefinierten Art und Weise reagiert. Da es ein `cout` Objekt zur Konsolenausgabe gibt, gibt es auch ein `cin` Objekt zur Eingabe über die Konsole. Das folgende Programm zeigt die Verwendung des `cin` Objekts.

Listing 1.4: Verwendung von cin

```
1 // UsingCin.cpp : Holen von Worten von der Tastatur
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 int main()
9 {
10    string str;
11    cin >> str;
12    cout << str << endl;
13    return(0);
14 }
```

14 }

In diesem Beispiel wird ein String vom Anwender eingegeben und gleich wieder ausgegeben. Auch hier werden die `cin` und `cout` Objekte verwendet, die aus den C++ Standardbibliotheken kommen. Dafür müssen wir durch die `include <iostream>` Anweisung wieder die benötigten Deklarationen einfügen. In Zeile 10 wird dann eine Stringvariable namens `str` deklariert. Beachten Sie, dass C++ den Datentyp `string` kennt. Wir werden darauf in dem nächsten Abschnitt eingehen. In dieser Variablen wird die Eingabe des Nutzers gespeichert. In der Zeile 11 haben wir jetzt das `cin` Objekt, dass in umgekehrter Richtung `>>` wie `cout` verwendet wird und auf die Zielvariable weist. Nach dem Ausführen dieser Zeile ist die Eingabe in der Variablen `str` gespeichert und kann in der folgenden Zeile über das Objekt `cout` ausgegeben werden.

Beachten Sie, dass `cin` in dieser Form nur bis zum ersten Leerzeichen liest. Wenn also ganze Zeilen, die Leerzeichen enthalten eingelesen werden sollen, muss dazu eine spezielle Funktion verwendet werden.

1.5 Challenges

1.5.1 Freundliches Programm

Schreiben Sie ein Programm, welches zunächst den Anwender nach seinem Namen fragt und ihn dann unter Verwendung des Namens freundlich begrüßt. Verwenden Sie dabei die Objekte `cin` und `cout`.

1.5.2 Kalender

Schreiben Sie ein Programm, dass den aktuellen Monat in dem untenstehenden Format ausgibt:

Oktober 2021						
Mo	Di	Mi	Do	Fr	Sa	So
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Verwenden Sie dabei das Objekt `cout`. Suchen Sie nach Möglichkeiten, die `cout` bietet, die Ausgabe zu formatieren.

2 Umgang mit grundlegenden Datentypen

Wie in jeder Programmiersprache ist es immer hilfreich die unmittelbar unterstützten Datentypen kennenzulernen. Einiges davon werden Sie bereits von C her kennen. Wie C unterstützt C++ nur einige wenige Datentypen.

2.1 Datentypen

- **int**

Integerzahlen können als **signed**, d.h. als mit Vorzeichen versehene Ganzzahlen verstanden werden. Ebenso ist es aber auch möglich sie als vorzeichenlos (**unsigned**) zu deklarieren. Sie repräsentieren dann nur positive Zahlen oder den Wert 0. Es muss beachtet werden, dass der Datentyp **int** implementationsabhängig ist. Meist hat er einen Umfang von 4 Bytes, allerdings ist es auch möglich, dass er nur 2 Bytes lang ist. Der Datentyp **long** ist mindestens 4 Byte groß. Darüber hinaus gibt es noch den Datentyp **long long** der mindestens 8 Byte groß sein muss und den Datentyp **short** mit mindestens 2 Byte.

- **char**

Der **char** Datentyp ist immer 8 Bit lang und wurde entwickelt, um den ASCII-Zeichensatz abzubilden. Man kann ihn aber auch wie einen 1-Byte langen Integer typ verwenden, ebenfalls in den Ausprägungen **signed** und **unsigned**. Will man portable Programme schreiben und genau die Bytelänge spezifizieren, dann bietet **stdint.h** die Deklarationen für eine sehr hilfreiche Bibliothek mit genau spezifizierten Datentypen, die genau die Vorzeichenbehandlung und die Länge eines Integerdatentypen spezifizieren. So ist etwa der Datentyp **uint32_t** ein 32 bit langer Integer mit der Eigenschaft **unsigned** und **int8_t** eine 8 bit lange Integerzahl mit Vorzeichen.

- **float**

Wie C unterstützt auch C++ Gleitkommazahlen. Damit können reelle Zahlen wie, pi, 1.33 oder -0.5 mit unterschiedlicher Genauigkeit gemäß dem IEEE 754 Binary Floating Point Standard abgebildet werden. C++ unterscheidet drei verschiedene Genauigkeiten **float**, **double** oder **long double**. Der Unterschied besteht in unterschiedlichen Genauigkeiten und Zahlenumfang, wobei allerdings **float** deutlich performanter ist als **double**. Für viele Anwendungen ist der **float** Datentyp

ausreichend.

- **bool**

Neu ist der Datentyp **bool**, der die Werte **true** und **false** repräsentiert. Dabei sind die Keywords **true** und **false** Bestandteile der Sprache C++, so dass diese direkt einer **bool** Variablen zugewiesen werden können. Ebenso werden weiterhin alle von 0 verschiedenen Werte als **true** und der Wert 0 als **false** interpretiert.

- **string**

Ebenfalls neu gibt es den Datentyp **string**, der zwar nicht direkt in C++ definiert ist, sondern als Klasse in der Standardbibliothek implementiert wurde.. Wie in C kann mit Strings als mit dem Wert 0 terminiertes Zeichen-Array gearbeitet werden. Die **string** Headerdatei deklariert nicht nur die Klasse, sondern auch eine Vielzahl von Funktionen zur Bearbeitung von Strings.

- **Pointer**

Natürlich bietet C++ auch die Möglichkeit wie in C mit Pointern zu arbeiten. Dieser Datentyp wird für Speicheradressen verwendet und kann als Referenz auf bestehende Variablen verwendet werden. Dabei kann ein und dieselbe Pointervariable nacheinander auf verschieden Variablen verweisen.

Dieses war sicherlich keine umfassende Beschreibung der grundlegenden Datentypen. Auch hier kann wieder nur auf cppreference.com verwiesen werden. Die detaillierte Beschreibung der Datentypen finden Sie unter **basic concept** und dann **fundamental types**. Sie werden dort noch weitere Varianten vorfinden.

2.2 Variablen

Es gibt eine Reihe wichtiger Aspekte beim Umgang mit Variablen in C++. Variablen sind temporäre Speicherbereiche. Sie müssen deklariert werden, bevor darauf zugegriffen wird. Dabei muss, wie in C, die Deklaration den Datentyp und den Variablennamen festlegen. Eine Deklaration kann zusätzlich noch die Initialisierung der Variablen mit einem bestimmten konstanten Wert umfassen. Im weiteren Verlauf des Programms können dann den Variablen (neue) Werte zugewiesen werden. In dem Listing 2.1 erfolgt die Zuweisung für einige Variablen mit unterschiedlichen Datentypen.

Für den Datentyp **integer** gibt es einige spezielle Formate, die beachtet werden müssen:

- 123 oder -3

Das Dezimalformat ist das Standardformat und entspricht den Werten, so wie wir sie lesen würden.

- 010

Beginnt eine Zahl mit einer führenden 0, dann wird diese als Oktalzahl interpretiert. Diese Zahl entspricht also dem Dezimalwert 8.

2 Umgang mit grundlegenden Datentypen

- 0x10
Beginnt eine Zahl mit einem führenden 0x, dann handelt es sich um eine Hexadezimalzahl. In dem Beispiel also den Dezimalwert 16.
- 0b10
Beginnt die Zahl mit einem führenden 0b, dann handelt es sich hier um eine Binärzahl. In dem Beispiel also hat sie den Dezimalwert 2.

Wird die Zahl mit einem kleinen oder großen U beendet, dann wird diese als eine Zahl mit der Eigenschaft **unsigned** interpretiert.

Gleitkommazahlen werden mit dem Dezimalpunkt und mindestens einer Ziffer rechts davon angegeben. Auch, wenn es sich um den Wert einer ganzen Zahl handelt, muss die Ziffer rechts des Dezimalpunkts angegeben werden. Soll es sich um einen **float** Wert handeln, dann muss die Zahl mit einem f wie in 123.5f abgeschlossen werden. Wird das f weggelassen, dann wird die Zahl als **double** interpretiert. Doubles sind damit der Standarddatentyp für Gleitkommazahlen.

Variablen vom Typ **char** können einerseits Ganzzahlen, solange sie sich mit einem Byte darstellen lassen, zugewiesen werden. Zeichen können aber auch in bekannter Weise wie 'a', 'A' oder mit Backslash '\n' oder '\0' zugewiesen werden.

Die Zuordnung zu Strings haben wir bereits gesehen, indem wir das Stringliteral "Hello World" zugewiesen haben.

Beispiele für die Zuweisung von Werten zu Variablen sehen sie in dem Listing 2.1. Gehen wir das Beispiel einmal Zeile für Zeile durch.

Listing 2.1: Umgang mit Variablen

```
1 // Variablen.cpp : Zuweisung zu Variablen.
2 //
3
4
5 #include <iostream>
6
7 using namespace std;
8
9 int a, b = 5; // single line comment
10
11 /* Multi
12 * line
13 * comment */
14
15 int main() {
16     bool my_flag;
17     a = 7;
18     my_flag = false;
19     cout << "a_=" << a << endl;
20     cout << "b_=" << b << endl;
```

```

21     cout << "flag_=" << my_flag << endl;
22     my_flag = true;
23     cout << "flag_=" << my_flag << endl;
24     cout << "a+b_=" << a + b << endl;
25     cout << "b-a_=" << b - a << endl;
26     unsigned int positive;
27     positive = b - a;
28     cout << "b-a_(unsigned)_=" << positive << endl;
29     return (0);
30 }

```

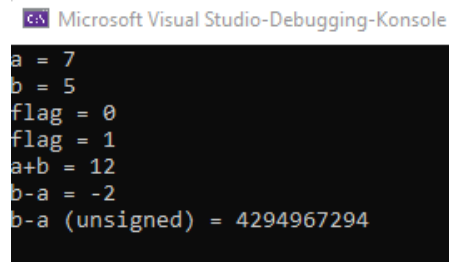
Zunächst sehen wir den Umgang mit Kommentaren. Wie in C gibt es zwei Möglichkeiten alles nach einem `//` wird als einzelzeiliger Kommentar verstanden, wie in Zeile 1,2 und Zeile 9. Alles zwischen der Zeichenkombination `/*` und `*/` wird ebenfalls als Kommentar angesehen. Dieser kann sich durchaus über mehrere Zeilen erstrecken.

In Zeile 9 werden dann zwei Integervariablen `a` und `b` deklariert. `b` wird dabei mit dem Wert 5 initialisiert. Nicht immer ist es guter Programmierstil mehrere Variablen in einer Zeile zu deklarieren. Da diese Variablen außerhalb der `main()` Funktion deklariert werden, handelt es sich hier um globale Variable, auf die von jeder Stelle des Programms zugegriffen werden kann. Daher werden diese auch statisch durch den Compiler im Datenssegment des Programms angelegt, dass während der gesamten Programmlaufzeit für die Variablen zur Verfügung steht. Erst wenn das Programm beendet wird, wird dieser Speicherplatz wieder freigegeben.

Meist sollten Variablen mit einem begrenzten Geltungsbereich angelegt werden. Damit ist gemeint, dass diese Variablen innerhalb von Funktionen, Anweisungsblöcken oder innerhalb von Schleifen deklariert werden. Diese Variablen sind nur innerhalb ihres Geltungsbereiches sichtbar und zugänglich. Wenn das Programm den Geltungsbereich verlässt, werden die Variablen gelöscht und der Speicherplatz freigegeben. Diese werden durch den Compiler verwaltet und in der Regel temporär im sogenannten Stackbereich des Speichers angelegt.

So deklarieren wir innerhalb des Geltungsbereichs der `main()` Funktion in Zeile 17 eine Bool-Variable namens `my_flag`. Für die Benennung der Variablen gibt es Regeln, natürlich dürfen Variablennamen nicht den Schlüsselworten der C++ Sprache entsprechen. So darf es etwa keine Variable mit dem Namen `return` geben. Kurz gesagt, es muss jeder Variablenname mit einer Nicht-Ziffer beginnen, und darf dann aus Buchstaben, Ziffern und Symbolen, wie etwa Binde- oder Unterstrichen, bestehen. Genaues können Sie unter cppreference.com nachlesen.

In Zeile 17 wird der Variablen `a` in der `main()` Funktion jetzt der Wert 7 zugewiesen. Der Zuweisungsoperator `=` arbeitet dabei immer von rechts nach links. Ebenso wird in Zeile 18 der boolschen Variable `my_flag` der Wert `false` zugewiesen. In den folgenden Zeilen 19-21 wird dann der Wert der Variablen ausgegeben. Wie in Abb, 2.1 gezeigt wird für den Wert des Flags 0, was aber `false` entspricht, ausgegeben.



```
Microsoft Visual Studio-Debugging-Konsole
a = 7
b = 5
flag = 0
flag = 1
a+b = 12
b-a = -2
b-a (unsigned) = 4294967294
```

Abbildung 2.1: Ergebnis der Ausgabe

In Zeile 22 wird dann `my_flag` der Wert `true` zugewiesen. In den folgenden Zeilen 23-25 wird dann wiederum das Flag, diesmal mit dem Wert 1 (`true`) und die Summe und Differenz aus `a` und `b` ausgegeben.

Zum Schluss überprüfen wir noch, was passiert, wenn wir einen negativen Wert einer Variablen vom Typ `unsigned` zuweisen. Dazu wird in Zeile 26 eine Variable `positive` vom Typ `unsigned int` deklariert. In C++ kann an nahezu jeder Stelle des Programmcodes eine Variable deklariert werden. Dieses Eigenschaft wurde erst später von C übernommen.

Dieser Variablen `positive` wird dann die Differenz `b-a`, die den negativen Wert -2 hat, in Zeile 27 zugewiesen. Der Wert von `positive` wird dann in Zeile 28 ausgegeben.

Falls das Ergebnis in der Abbildung 2.1 überrascht, dann kann man nachrechnen, dass es sich um den Wert $2^{32} - 2$, und damit dem 2er Komplement von -2 in Binärdarstellung entspricht. Die Binärdarstellung ist identisch, nur die Interpretation ist verschieden. Aus diesem Grund muss man vorsichtig sein, wenn sich entscheidet Variablen vom Typ `unsigned` oder `signed` zu deklarieren und in Ausdrücken gemischt zu verwenden.

2.3 Ausdrücke, Zuweisungen, Operatoren

2.3.1 Ausdrücke und Zuweisungen

Ausdrücke (expressions) sind in der Regel die symbolische Repräsentation einer Berechnung, genauso wie Sie es etwa von algebraischen Ausdrücken kennen. Bestandteile eines Ausdrucks können dabei Variablen, Konstanten und Operatoren sein. Wichtig ist dabei, dass jeder Ausdruck einen bestimmten Wert hat. Eine Zuweisung ist dann nichts anderes als eine Programmanweisung, die einer Variablen den Wert eines Ausdrucks zuweist, dabei steht die Variable, der der Wert zugewiesen wird immer auf der linken Seite, des Zuweisungsoperators `=` – der den Wert bestimmende Ausdruck immer auf der rechten Seite. Damit die linke Seite den Wert auch zugewiesen bekommen kann, müssen die Datentypen der linken und rechten Seite übereinstimmen. Wichtig ist dabei, dass dieses nicht durch den Compiler überprüft wird, wie wir in Listing 2.1 gesehen haben, wo einer `unsigned int` Variablen der negative Wert einer `int` Variable zugewiesen wurde. Daher liegt es in der Verantwortung des Programmierenden, darauf zu achten, dass der

Programmcode konsistent ist.

2.3.2 Operatoren

Hier noch einmal zur Wiederholung, da sich die Operatoren kaum von C unterscheiden. So haben wir arithmetische Operatoren, die bis auf Modulo für Gleitkomma- und Ganzzahlen erlaubt sind:

Tabelle 2.1: Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo – Rest einer Division (nur Integer)

Es hängt von der Art der Operanden ab, ob eine Gleitkomma- oder ganzzahlige Operation ausgeführt wird.

Eine weitere Form der Operatoren sind bitweise boolsche Operatoren:

Tabelle 2.2: bit-bezogene Operatoren

&	bitwise and
	bitwise or
~	bitwise not
^	bitwise xor
>>	bitshift rechts
<<	bitshift links

Dabei ist zu beachten, dass diese Operationen bit für bit des Wertes einer oder mehrerer Variablen durchgeführt werden. Es sind keine logischen Operationen zwischen Variablen.

Logische Operationen sind solche Operationen, wie sie beispielweise in `if` Entscheidungen oder als Bedingungen in Schleifen verwendet werden, so wie Sie das von C bereits her kennen. Entscheidend ist, dass diese Operationen immer nur die Werte `true` oder `false` liefern. Erinnern Sie sich, dass `false` immer dem Wert 0 entspricht und `true` allen anderen.

Tabelle 2.3: Logische Operatoren

&&	logisch and
	logisch or
!	logisch not

2 Umgang mit grundlegenden Datentypen

Darüber hinaus gibt es noch relationale Operatoren, um damit Werte zu vergleichen.

Tabelle 2.4: Relationale Operatoren

==	ist gleich
!=	ist ungleich
<	ist kleiner
>	ist größer
<=	ist kleiner oder gleich
>=	ist größer oder gleich

Vorsicht: Ein häufiger Fehler ist beim Vergleich auf Gleichheit nur ein einfaches `=` zu verwenden. Das Problem ist, dass dieses als Zuweisung verstanden wird, und der zugewiesene Wert dann als `true` oder `false` verstanden wird. Der Compiler kann nicht entscheiden, ob hier ein Fehler vorliegt, oder ob eine solche Anweisung gewollt ist.

Hier muss noch der sogenannte Spaceship-Operator `<=>` mit aufgenommen werden, der mit C++ 20 neu aufgenommen wurde und die relationalen Operatoren ergänzt. Dieser dient ebenfalls dem Größenvergleich zweier Objekte, resultiert aber in einen Wert kleiner 0, wenn eine `<` Beziehung erfüllt ist, dem Wert 0 bei Gleichheit und einem Wert größer 0, wenn eine `>` Beziehung gilt.

Abschließend gibt es noch die drei von C bekannten Pointer-Operatoren.

Tabelle 2.5: Pointer Operatoren

Präfix <code>*</code>	Indirektion. Hierdurch wird ein Pointer dereferenziert und erlaubt den Zugriff auf den Inhalt der Variablen, auf die er zeigt
Präfix <code>&</code>	Gibt die Adresse des Speicherplatzes einer Variablen zurück
<code>-></code>	Lässt auf Elemente einer Struktur oder Klasse zugreifen

Aufgabe: Da Ausdrücke ein ganz wesentliches Konzept der Sprache C++ sind, macht es Sinn ein wenig mehr darüber zu lernen. Googlen Sie beispielsweise `cpp reference operator precedence`, um zu sehen, in welcher Priorität komplexe Ausdrücke ausgewertet werden.

2.4 Automatische Variablen mit `auto`

Mit dem C++ 11 Standard wurden sogenannte automatische Variablen eingeführt. Hierbei handelt es sich um eine sehr bequeme Möglichkeit Variablen zu deklarieren, die an den Datentyp angepasst sind, der ihnen zugewiesen werden soll. Allerdings geht dieses nur

für die Basisdatentypen. Damit der Compiler entscheiden kann, um was für eine Variable es sich handeln soll, muss die Variable bei der Deklaration bereits initialisiert werden, das heißt, ihr muss ein Wert zugeordnet werden. Das Listing 2.2 zeigt die Verwendung automatischer Variablen.

Listing 2.2: Automatische Variablen mit `auto`

```

1 // AutoType.cpp : Beispiele für Auto Variablen
2 //
3
4 #include <iostream>
5 #include <typeinfo>
6
7 int main() {
8     auto a = 8;
9     auto b = 12345678901;
10    auto c = 3.14f;
11    auto d = 3.14;
12    auto e = true;
13    auto f = 'd';
14
15    std::cout << typeid(a).name() << std::endl;
16    std::cout << typeid(b).name() << std::endl;
17    std::cout << typeid(c).name() << std::endl;
18    std::cout << typeid(d).name() << std::endl;
19    std::cout << typeid(e).name() << std::endl;
20    std::cout << typeid(f).name() << std::endl;
21
22    return (0);
23 }
```

Das Schlüsselwort `auto` ersetzt dabei die Angabe des Datentyps. In dem Programm werden in Zeile 10 bis 15 sechs Variablen deklariert und mit Werten jeweils unterschiedlichen Datentyps initialisiert. So wird der Variablen `a` ein Integer zugeordnet. Die Variable `b` wird mit einer sehr großen Ganzzahl initialisiert, die sich nicht mehr in einer normalen `int`-Variablen ausdrücken lässt. Die Variable `c` bekommt einen `float`-Wert zugewiesen und `d` einen `double`-Wert. Die Variable `e` den boolschen Wert `true`. Zu Schluss wird noch der Variablen `f` das Zeichen `'d'` zugewiesen.

Im folgenden wird dann jeweils der Datentyp, den jede Variable automatisch erhalten hat, in einer eigenen Zeile ausgegeben. Hierzu wird der `typeid()`-Operator, der in der `typeinfo`-Headerdatei der Standardbibliothek definiert ist, verwendet. Dieser gibt ein `type_info`-Objekt zurück. Dessen `name()` Funktion wird verwendet, um dann den Namen des Datentyps der Variablen auszugeben. Es kann sein, dass bei einigen Implementationen lediglich ein Kürzel für den Datentyp ausgegeben wird. Daher wundern Sie sich nicht, wenn Sie andere Ergebnisse bekommen.

2.5 Präprozessor

C++ ist eine kompilierte Sprache, das bedeutet, dass der gesamte Programmcode nacheinander wie auf einem Fließband - einer Pipeline - eine Folge von Werkzeugen durchläuft, die versuchen die semantischen Elemente des Programms zu extrahieren. Im Gegensatz dazu stehen interpretative Sprachen, bei denen jede Anweisung für sich übersetzt und sofort ausgeführt wird. Dabei ist der Präprozessor einer der ersten Werkzeuge, die den gesamten Sourcecode verarbeiten. Man kann ihn sich wie ein automatisiertes Textverarbeitungsprogramm vorstellen, das nichts anderes macht, als bestimmte Textteile zu ersetzen, Texte aus anderen Dateien zu ergänzen oder, wenn bestimmte Bedingungen erfüllt sind, ganze Textteile zu löschen. Der Präprozessor wandelt den ursprünglichen Sourcecode, den Programmtext, in einen veränderten Sourcecode um. Dieser vorverarbeitete Programmcode wird dann an den eigentlichen Compiler weitergereicht.

Alle Präprozessoranweisungen starten mit dem Beginn der Zeile mit dem # Zeichen. Eine Präprozessoranweisung haben Sie bereits kennengelernt.

```
#include <...>
```

Die `#include`-Anweisung dient dazu den gesamten Inhalt der angegebenen Datei in das Programm anstelle der Anweisung einzufügen.

Listing 2.3: Einfache Präprozessoranweisungen

```
1 // PreprocessorExample.cpp : Example for simple preprocessor instructions
2 //
3 #include <iostream>
4 #include <string>
5 #include <cstdlib>
6
7 #define CAPACITY 5000
8 #define DEBUG
9
10 using namespace std;
11
12 int main() {
13     int32_t large = CAPACITY;
14     uint8_t small = 37;
15 #ifdef DEBUG
16     cout << "[About to perform the addition]" << endl;
17 #endif
18     large += small; // shorthand for large = large + small
19     cout << "The large integer is " << large << endl;
20     return (0);
21 }
```

So wird in Zeile 3 in dem Listing 2.3 der gesamte Inhalt der Datei `iostream`, gefolgt von dem Inhalt der Dateien `string` und `cstdlib` (Zeile 4 und 5) eingefügt.

Beachten Sie, dass der Dateiname in `<` und `>` eingeschlossen ist. Das hat zur Folge, dass nach diesen Dateien an einer vordefinierten Stelle gesucht wird. Es braucht also kein Pfad zu der Datei angegeben werden. Dieser wird meist durch die Entwicklungsumgebung vorgegeben. Auf diese Weise können alle Header-Dateien der Standardbibliotheken eingefügt werden. Diese Dateien benötigen auch nicht die Dateierweiterung `.h` oder `.hpp`.

Die in Zeile 5 eingefügte Datei `cstdint` ist eine Header-Datei der Standard-C-Bibliothek, und stellt Integer Datentypen mit fest definierten Längen zur Verfügung, so dass man von der Abhängigkeit von der jeweiligen Implementierung des `int` Datentyps befreit ist. Diese Header-Datei beinhaltet C Code. Daher beginnt der Dateiname mit einem vorangestellten kleinen `c`. Hierbei handelt es sich um eine für C++ aufbereitete Variante der C-Header-Datei `stdint.h`.

Die nächste Präprozessoranweisung ist `#define`. Diese Anweisung definiert Symbole, die jeweils das rechts davon stehende bedeuten. Diese Symbole werden auch als Makros bezeichnet. Aber eigentlich macht die Anweisung nichts anderes, als ein bestimmter Text in dem Sourcecode gefunden und ersetzt wird. So wird in Zeile 7 ein Symbol namens `CAPACITY` mit der Bedeutung 5000 definiert.

Üblich ist es für Makros ausschließlich Großbuchstaben zu verwenden, obwohl das nicht zwingend erforderlich ist. Aber es hat sich als guter Programmierstil etabliert.

Diese Anweisung bewirkt, dass an jeder Stelle im Sourcecode, wo der Präprozessor das Symbol `CAPACITY` findet, dieses durch 5000 ersetzt wird. Beachten Sie, dass Präprozessoranweisungen nicht mit einem Semikolon abgeschlossen werden.

In dem Hauptprogramm werden zwei Integer-Variablen unterschiedlicher Länge definiert. Die erste ist eine signed integer Variable der Länge 32 bit, so wie sie in `stdint` festgelegt ist. Würde der Datentyp mit einem `u` beginnen, dann wäre es eine unsigned Variable. Diese Variable `large` wird nun mit `CAPACITY` initialisiert. In Zeile 14 wird dann eine 8 bit lange unsigned Variable `small` deklariert und mit dem Wert 37 initialisiert. Diese wird dann in Zeile 18 zu dem Wert der Variablen `large` hinzu addiert, und das Ergebnis in Zeile 19 ausgegeben.

Der dritte Typ einer Präprozessoranweisung erlaubt das bedingte Einfügen von Programmcode. Dabei sind es einfache if-else Bedingungen, die in der Regel auf das Vorhandensein einer `#define`-Anweisung testen. Dabei handelt es sich natürlich nicht um bedingte Verzweigungen im Programmcode selber, sondern sie sind eher als Anweisungen zu sehen, die Programmcode unter bestimmten Bedingungen im Sourcecode stehen lassen, oder, die ihn entfernen, bevor der durch den Präprozessor vorverarbeitete Sourcecode compiliert wird. In dem Listing 2.3 gibt es in Zeile 16 eine Anweisung einen Text auszugeben, damit man weiß, wo das Programm sich gerade in der Verarbeitung befindet. Diese Ausgabe mag während der Entwicklungs- und Testphase sehr hilfreich sein, ist aber bei dem fertigen Programm eher überflüssig. Da man oft nicht weiß, ob die Ausgabe bei der Suche nach später auftretenden Problemen vielleicht noch einmal hilfreich sein kann, möchte man die Anweisung nicht einfach löschen oder auskommentieren. Insbesondere bei umfangreichen Programmen hat man oft eine Vielzahl solcher

2 Umgang mit grundlegenden Datentypen

die Fehlersuche unterstützenden Anweisungen. Diese im Fehlerfalle alle nach und nach wieder einzufügen oder auszukommentieren, ist einfach nur lästig. Das heißt, ich möchte den Code in Abhängigkeit von der Entwicklungsphase einfügen oder auch nicht. Dieses geschieht in Zeile 15-17. Die fragliche Ausgabeanweisung wird in `#ifdef DEBUG` und `#endif` eingeschlossen. Immer wenn das Makro `DEBUG` definiert ist, wie in Zeile 8, dann bleibt die Ausgabeanweisung in Sourcecode stehen. Gibt es aber die Definition des Makros `DEBUG` nicht, dann wird die Programmzeile vom Präprozessor entfernt. Dabei muss das Makro bei der Definition keinen Wert zugeordnet bekommen.

Weitere Einsatzbereiche dieser bedingten Präprozessoranweisung sind Programmteile, die je nach Betriebssystem unterschiedlich ausfallen. Hierbei wird ausgenutzt, dass je nach Betriebssystem unterschiedliche Makros bereits vordefiniert sind oder über entsprechende Header-Dateien eingefügt werden müssen (siehe Tabelle 2.6).

Tabelle 2.6: Makros, die das Betriebssystem identifizieren

Windows 32 bit + 64 bit	<code>_WIN32</code>	alle Windows OS
Windows 64 bit	<code>_WIN64</code>	nur 64 bit Windows
Apple	<code>__APPLE__</code>	all Apple OS
MacOS	<code>TARGET_OS_MAC</code>	include TargetConditionals.h
Android	<code>__ANDROID__</code>	subset of linux
Unix based OS	<code>__unix__</code>	
Linux	<code>__linux__</code>	subset of unix

2.6 Challenges

2.6.1 Formatierungsoptionen von `cout`

Erstellen Sie ein C++ Programm, welches eine Handelskalkulation nach dem folgenden Beispiel durchführt. Die Bildschirmeingabe und -ausgabe soll dabei wie in Tabelle 2.7 aussehen. Die Ausgabe soll mit 2 Nachkommastellen rechtsbündig formatiert werden. Außerdem soll die Bezeichnung EUR hinter den Beträgen stehen.

Die in Tabelle 2.8 Folgende Variablen sollen zur Eingabe und für die Berechnungen verwendet werden:

Hinweis: Entwickeln Sie das Programm schrittweise. Testen Sie erst die Eingabe, dann schrittweise die Ausgabe mit den Berechnungen. Formatierungen am besten erst am Ende, wenn die Logik stimmt. Verwenden Sie die Manipulatoren aus der Bibliothek `iomanip.h`.

Tabelle 2.7: Handelskalkulation: Ein- und Ausgabe

Eingabe	Ausgabe
Listeneinkaufspreis: 250	Listeneinkaufspreis: 250,00 EUR
Rabatt (in %): 10	Rabatt: 25,00 EUR
Skonto(in %): 2	Zieleinkaufspreis: 225,00 EUR
Bezugskosten: 30	Skonto: 4,50 EUR
Handlungskostensatz (in %): 30	Bareinkaufspreis: 220,50 EUR
Gewinnzuschlag (in %): 5	Bezugskosten: 30,00 EUR
	Bezugspreis: 250,50 EUR
	Handlungskosten: 75,15 EUR
	Selbstkostenpreis: 325,65 EUR
	Gewinn: 16,28 EUR
	Barverkaufspreis: 341,93 EUR
	Mehrwertsteuer (19%): 64,97 EUR
	Bruttoverkaufspreis: 406,90 EUR

2.7 Konstante

An dieser Stelle ist es sinnvoll etwas mehr auf Konstanten einzugehen. Konstante sind Variable, deren Wert während der Ausführung des Programms sich nicht ändern. Meist haben sie die Bedeutung von Parametern in dem Programm. Beispielsweise könnte es die verwendete Bildschirmauflösung sein, oder es könnte die Länge eines Speicherbereichs sein. Konstante können natürlich als `#define` Anweisungen definiert werden, wie wir in Abschnitt 2.5 kennengelernt haben. Eine andere Möglichkeit wäre es, sie einfach als normale Variablen zu deklarieren und selber sicherzustellen, dass sich der Wert nicht ändert.

Wir können also auf der einen Seite Konstante als Makros definieren, die letztendlich nichts anderes als Suchen-und-Ersetzen Anweisungen für den Präprozessor sind, und alle Vorkommnisse des Makronamens durch seinen Wert ersetzt werden. Allerdings wird die Verwendung von Makros manchmal als schlechter Programmierstil in C++ angesehen. Warum? Nun die Antwort ist einfach, Makros haben keinen Kontext, der Compiler kann sie beispielsweise nicht auf grundlegende Eigenschaften wie Datentyp und syntaktische Korrektheit überprüfen. So wäre eine Makroanweisung wie

```
#define NUMBER_OF_ROOMS A4
```

durchaus korrekt, aber sinnlos. Andererseits gibt es eine gute Alternative in C++ in dem `const` Schlüsselwort. Dieses kann in jeder regulären Variablendeklaration verwendet werden.

```
const int NUMBER_OF_ROOMS = 4;
```

Da in jeder Deklaration der Datentyp festgelegt wird, kann der Compiler mögliche Probleme, die sich aus dem Datentyp ergeben, entdecken. Darüber hinaus kann bei Variablen

Tabelle 2.8: Handelskalkulation: Variablen

Variable	Bedeutung	Verwendung	Typ
listenp	Listenpreis	Eingabe	Float
rabatt	Rabattsatz	Eingabe	Float
skonto	Skontosatz	Eingabe	Float
bk	Bezugskosten	Eingabe	Float
hks	Handlungskostensatz	Eingabe	Float
gkz	Gewinnzuschlag	Eingabe	Float
zek	Zieleinkaufspreis	Berechnung	Float
bek	Bareinkaufspreis	Berechnung	Float
bp	Bezugspreis	Berechnung	Float
hk	Handlungskosten	Berechnung	Float
sk	Selbstkostenpreis	Berechnung	Float
gw	Gewinn	Berechnung	Float
nvk	Barverkaufspreis	Berechnung	Float
mw	Mehrwertsteuer	Berechnung	Float
bvk	Bruttoverkaufspreis	Berechnung	Float

immer der Gültigkeitsbereich eingeschränkt werden - etwa nur auf eine Schleife, oder Funktion, oder gar global verfügbar.

Obwohl es das **const** Schlüsselwort in C und C++ gibt, verhalten diese sich während des Compilierens unterschiedlich: In C wird die Variable in einem Speicherbereich außerhalb des eigentlichen Programms erzeugt und ist damit während der Kompilation nicht sichtbar. Da es in C keine Klassen gibt, hat dieses Schlüsselwort natürlich auch keine Bedeutung für die klassenbezogenen Methoden. In C++ werden Konstante während des Kompilierens in der Regel nicht außerhalb des Programms angelegt und sind damit für weitere Überprüfungen und Optimierungen während des Kompilierens sichtbar. Klassen-eigene Methoden können als **const** deklariert werden und diese können durch andere ebenfalls als **const** deklarierte Funktionen verwendet werden. Darüber bestehen alle Möglichkeiten der Kapselung von Variablen und die Einschränkung auf den jeweiligen Gültigkeitsbereich auch für konstante Variable.

2.8 Arrays

Arrays sind bereits aus C bekannt. Es hat sich in C++ daran nicht wesentlich etwas geändert. Sie stellen eine Sammlung von Daten eines einheitlichen Datentyps dar. Dabei ist jedes Element eines Arrays über einen Index direkt erreichbar. Die Anzahl der Elemente eines Arrays wird bei der Deklaration festgelegt und kann nicht direkt geändert werden. Dabei sind alle Elemente eines Arrays zusammenhängend im Speicher angelegt.

Nehmen wir Listing 2.4 als Beispiel. Dabei wurde auf den Einsatz von Kontrollstrukturen

verzichtet, da wir später erst darauf eingehen werden.

Listing 2.4: Beispiel für die Verwendung von Arrays

```

1 // ArrayExample.cpp : Beispiel für Arrays mit und ohne #define
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 // #define AGE_LENGTH 4
9
10 int main() {
11     const int AGE_LENGTH = 4;
12
13     // explicit declaration of array length
14     int age[AGE_LENGTH];
15
16     // implicit declaration of array length by initialisation
17     // warning: double in float conversion
18     float temperature[] = { 31.5, 32.7, 38.9 };
19
20     age[0] = 25;
21     age[1] = 20;
22     age[2] = 19;
23     age[3] = 19;
24
25     cout << "The Age array has " << AGE_LENGTH << " elements" << endl;
26     cout << "Age[0] = " << age[0] << endl;
27     cout << "Age[1] = " << age[1] << endl;
28     cout << "Age[2] = " << age[2] << endl;
29     cout << "Age[3] = " << age[3] << endl;
30     cout << endl;
31     cout << "Temp[0] = " << temperature[0] << endl;
32     cout << "Temp[1] = " << temperature[1] << endl;
33     cout << "Temp[2] = " << temperature[2] << endl;
34
35     return (0);
36 }
```

In dem Listing 2.4 wird ein Integer-Array deklariert, um das Alter von vier Personen speichern zu können. Dieses geschieht in der Form:

```
int age[AGE_LENGTH];
```

Dabei wird die Größe des Arrays explizit in eckigen Klammern angegeben. Dieses kann auf verschiedene Weise geschehen. Zum einen könnte direkt die Zahl eingetragen werden. Sollte im weiteren Programm diese Zahl weiterhin verwendet werden, etwa als Abbruchbedingung in Schleifen, dann müssten alle diese Stellen bei einer eventuell notwendigen

2 Umgang mit grundlegenden Datentypen

Größenänderung des Arrays angepasst werden. Das kann zu einer Reihe von Fehlern führen. Eine bessere Form wäre daher diesen Wert über ein Präprozessor-Makro zu definieren. Diese Möglichkeit ist in Zeile 8 auskommentiert. Wie in Abschnitt 2.7 beschrieben, wird die Deklaration einer konstanten Variable (Zeile 14) hier bevorzugt, da damit weitere Kontrollmöglichkeiten auf Korrektheit des Programms während des Kompilierens gegeben sind.

Eine andere Form der Deklaration eines Arrays ist in Zeile 18 zu sehen, in der ein Float-Array deklariert und initialisiert wird. Die Werte mit denen das Array initialisiert werden soll, werden dabei in geschweiften Klammern angegeben. Durch die Initialisierung kann der Compiler implizit ableiten, dass dieses Array die Länge 3 haben muss, daher muss die Länge des Arrays nicht in den eckigen Klammern angegeben werden.

Beachten Sie bitte, dass in diesem Beispiel die Initialisierung mit Werten vom Typ `double` erfolgt. In diesem Fall gibt der Compiler eine Warnung aus und nimmt selbstständig eine Konvertierung der `double`-Werte in `float`-Werte vor. Um diese Konversion zu umgehen, sollten alle Werte mit einem `f` enden, damit der Compiler erkennt, dass hier Gleitkommawerte vom Typ `float` gemeint sind.

Aufgabe: Wenn Sie mehr darüber lesen wollen, nutzen Sie cppreference.com und suchen Sie nach `implicit conversion`.

In den Zeilen 20-23 können Sie sehen, dass C++ analog zu C die Elemente eines Arrays immer mit 0 beginnend indiziert. Der lesende und schreibende Zugriff auf die Elemente eines Arrays erfolgt indem der Index des betreffenden Elementes in eckigen Klammern nach dem Namen des Arrays angegeben wird. Dieses ist auch in den folgenden Zeilen 32-33 gezeigt.

2.9 Strings

In C waren Strings als ein Array aus Elements des Typs `char` definiert, die mit einem 0-Wert enden. Auch in C++ sind Strings nicht direkter Bestandteil der Sprache, sondern werden als Klasse der Standardbibliothek definiert. Dahinter verbirgt sich eine ganz ähnliche Implementierung als Folge von Zeichen, die mit einem Nullzeichen `'\0'` enden. Allerdings umfasst die String-Klasse eine Vielzahl nützlicher Funktionen, die den Umgang mit Strings verglichen mit dem in der Sprache C deutlich vereinfachen. Das Listing 2.5 soll einen ersten Eindruck davon geben.

Listing 2.5: Arbeiten mit Strings

```
1 // StringExample.cpp : Beispiele für Strings.
2 //
3
4 #include <iostream>
```

```

5 #include <string>
6
7 using namespace std;
8
9 int main()
10 {
11     string str1 = "Hi_Guys_";
12     string str2;
13
14     str2 = "-_How_are_you?";
15
16     cout << "string1_has_length:_ " << str1.size() << "_->_" << str1 << endl;
17     cout << "string2_has_length:_ " << str2.size() << "_->_" << str2 << endl;
18     cout << str1 + str2 << endl;
19
20     str1 = str1 + str2;
21     cout << "string1_has_length:_ " << str1.size() << "_->_" << str1 << endl;
22
23     str1 = "Peter";
24     str2 = "Paul";
25
26     auto result = str1 <=> str2;
27
28     if (result < 0)
29         cout << "Peter<_Paul" << endl;
30     else
31         if (result == 0)
32             cout << "Peter=_Paul" << endl;
33         else
34             cout << "Peter>_Paul" << endl;
35
36     return (0);
37 }

```

Man kann zwar auch weiterhin mit Strings wie in C umgehen, aber dieses Beispiel soll Ihnen zeigen, um wieviel einfacher der Umgang mit der String-Klasse von C++ ist.

Um mit den Stringfunktionen der Standardbibliothek arbeiten zu können, muss in Zeile 5 die Header-Datei `string` inkludiert werden. In Zeile 11 wird dann eine Variable `str1` vom Typ `string` deklariert und initialisiert. Zeile 12 deklariert ebenfalls eine Variable `str2` vom Typ `string`. Allerdings muss hier keine Länge angegeben werden, wie das in C der Fall wäre. In Zeile 14 weisen wir dann dieser Variable einen konstanten String zu. Die beiden Strings und ihre jeweilige Länge werden in Zeile 16 und 17 ausgegeben. In Zeile 18 werden dann zusätzlich noch die beiden jetzt aneinander gefügten Strings ausgegeben. Dazu kann hier einfach intuitiv der `+` Operator verwendet werden.

Noch etwas weiter geht die Zuweisung von `str1 + str2` an wiederum `str1` in Zeile 20. Hier wird der Speicherplatz von `str1` automatisch ausreichend groß neu alloziert, da-

mit das Ergebnis komplett gespeichert werden kann, ohne dass man sich um mögliche Größenbeschränkungen der Strings kümmern muss. Dieses ist bereits vorweggenommen ein gutes Beispiel für den in C++ möglichen Polymorphismus, der es erlaubt, dass Operatoren in unterschiedlichen Kontexten andere Bedeutungen erlangen können (engl. operator overloading). Hier erhält der `+` Operator, der üblicherweise Zahlen addiert, die Bedeutung der Stringverbindung, wenn die beiden Operanden vom Datentyp `string` sind.

Auch der lexikalische Vergleich zweier Strings lässt sich mittels der Stringklasse einfach realisieren. Dazu wird im Beispiel der Spaceship-Operator herangezogen (Zeile 26). Das Ergebnis des Vergleichs wird der in derselben Zeile deklarierten Autovariablen `result` zugeordnet. Diese einfache Variable muss in Folge dann nur noch ausgewertet werden, um das Ergebnis des Vergleichs der beiden Strings auszugeben (Zeile 28-34).

Aufgabe: Diese kurze Darstellung ist bei weitem keine umfassende Übersicht der Möglichkeiten, welche die String-Klasse bietet. Suchen Sie beispielsweise auf cppreference.com nach weiteren Möglichkeiten und verschaffen Sie sich eine Übersicht der Funktionen und Operatoren.

2.10 Typkonvertierung - *type casting*

Mit einer der wertvollsten Aspekte von C++ ist die Möglichkeit der Typkonvertierung, die eine nahezu unbegrenzte Kontrolle über die vorliegenden Daten gibt. Durch die Typkonvertierung kann gesteuert werden, wie die Daten interpretiert werden sollen. So kann beispielsweise der Wert einer 32-bit Integervariablen in einen Wert vom Datentyp `float` gewandelt werden. Die Syntax ist dabei ganz einfach. Es muss einfach der gewünschte Datentyp in runden Klammern der Variablen oder dem Ausdruck vorangestellt werden.

Listing 2.6: Verschiedene Formen der Typkonvertierung

```
1 // TypeConversion.cpp : type conversions and type castings
2
3 #include <iostream>
4 #include <cstdlib>
5
6 #define EXAMPLE1
7 #define EXAMPLE2
8 #define EXAMPLE3
9 #define EXAMPLE4
10
11 using namespace std;
12
13 int main() {
14
```

```

15 // EXAMPLE 1: implicit conversion of different numerical types
16 #ifdef EXAMPLE1
17     float flt = -7.44f;
18     int32_t sgn;
19     uint32_t unsgn;
20
21     sgn = flt;
22     unsgn = sgn;
23
24     cout << "_float:_ " << flt << endl;
25     cout << "_int32:_ " << sgn << endl;
26     cout << "_uint32:_ " << unsgn << endl;
27 #endif
28
29 //declaration for EXAMPLE2 and EXAMPLE3
30 int fahrenheit = 100;
31 int celsius;
32
33 // EXAMPLE 2: Error thru conversion
34 #ifdef EXAMPLE2
35     celsius = (5 / 9) * (fahrenheit - 32);
36     cout << endl;
37     cout << "Fahrenheit:_ " << fahrenheit << endl;
38     cout << "Celsius_:_ " << celsius << endl;
39 #endif
40
41 // EXAMPLE 3: Correct conversion
42 #ifdef EXAMPLE3
43     celsius = ((float)5 / 9.0) * (fahrenheit - 32);
44
45     cout << endl;
46     cout << "Fahrenheit:_ " << fahrenheit << endl;
47     cout << "Celsius_:_ " << celsius << endl;
48 #endif
49
50 // EXAMPLE 4: Separating integer and fractional part of a float
51 #ifdef EXAMPLE4
52     float weight = 10.99;
53
54     cout << endl;
55     cout << "Float_:_ " << weight << endl;
56     cout << "Integer_part_:_ " << (int)weight << endl;
57     cout << "Fractional_part:_ " << (int)((weight - (int)weight)) * 10000 << endl;
58 #endif
59
60     return (0);
61 }

```

Sehen wir uns einige Beispiele für die Typkonvertierung in C++ Listing 2.6 an. Im ersten

2 Umgang mit grundlegenden Datentypen

Beispiel (Zeile 17-26) haben wir drei Variablen, eine `float` Variable `flt`, die mit `-7.44` initialisiert wird, eine `int` Variable `sgn` und eine `unsigned int` Variable `unsgn`. Zunächst wird `flt` der Variablen `sgn` zugewiesen. Hierbei wird der Datentyp `float` in den Datentyp `int` konvertiert. Dieses geschieht implizit, ohne das wir es spezifiziert haben. Es wird nur der ganzzahlige Anteil übernommen, und `sgn` hat damit den Wert `-7`.

Als nächstes wird dieser Wert der Variablen `unsgn` zugewiesen. Wir weisen also einen negativen Wert einer `unsigned int` Variablen zu. Wie wir schon in Listing 2.1 gesehen haben, wird hierbei einfach das bit-Muster der `int` Variablen übernommen. Die Variable `unsgn` hat jetzt den Wert $2^{32} - 7$. Alle diese Konvertierungen erfolgen implizit. In der Regeln warnt der Compiler aber, dass eine Konvertierung vorgenommen wurde bei der ein möglicher Datenverlust erfolgen kann. In den folgenden Zeilen werden dann die Werte ausgegeben.

In dem nächsten Beispiel in Zeile 33-38 soll ein Temperaturwert von Fahrenheit in Celsius konvertiert werden. Da nur ganzzahlige Temperaturwerte interessieren werden auch Werte vom Type `int` dafür verwendet. Der Wert für die Temperatur in Fahrenheit wird mit `100` initialisiert. Führt man die Berechnung in Zeile 35 durch und gibt das Ergebnis in Zeile 38 aus, so wird als Ergebnis `0` Grad Celsius ausgegeben, was offensichtlich falsch ist. Was ist passiert?

Nun die Division `5/9` wird als Integerdivision interpretiert und hat damit das Ergebnis `0`. Damit ist das Gesamtergebnis der Berechnung ebenfalls `0`. Damit das Ergebnis korrekt wird, muss erreicht werden, dass die Berechnung mit dem Datentyp `float` erfolgt. Hierzu muss mindestens ein Operand zu einem Gleitkommawert werden. Beispiel 3 in Zeile 43-47 zeigt die korrekte Berechnung. Durch die Wandlung von `9` in `9.0` wird nun die Berechnung automatisch als Gleitkommadivision durchgeführt. Dasselbe wird durch die explizite Konversion von `5` in einen `float` Datentyp durch die Anweisung `(float)5` erreicht. Durch das voranstellen des Datentyps in runden Klammern vor eine Variable oder Ausdruck, wird dieser in einen Wert gemäß dem vorangestellten Datentyp gewandelt.

In dem vierten und letzten Beispiel in Zeile 52-57 soll die Gleitkommazahl `weight` mit dem Wert `10.99` in den ganzzahligen Vorkommaanteil und den Nachkommaanteil zerlegt werden. Dabei erhält man den ganzzahligen Anteil durch die expliziten Konversion (type cast) der Gleitkommazahl in eine Ganzzahl `(int)weight`. Der Nachkommaanteil entspricht dann der Differenz dieser beiden Werte. In Zeile 57 wird dieser Anteil mit `10000` multipliziert, um den Wert sichtbar zu machen. Eigentlich sollte dieser Anteil genau `0.9900` sein. Wir sehen aber, dass dieses mit `0.9899` nicht ganz der Fall ist. Dieses liegt an der Ungenauigkeit der binären Darstellung einer Gleitkommazahl mit den nur 4 Bytes eines `float` Wertes. Dieses ist kein C++ eigenes Problem, sondern ein Problem der binären Zahlendarstellung, wie sie auch von anderen Programmiersprachen verwendet wird. Würden wir stattdessen eine `double` Darstellung der Zahlen verwenden, wäre die Ungenauigkeit deutlich geringer. Versuchen sie einfach mal das Beispiel mit einem `double` Variablen.

2.11 Challenges

2.11.1 Primzahlen

Schreiben Sie ein Programm, welches alle Primzahlen < 1000 ausgibt. Es sollen jedoch nur jeweils 25 Zahlen angezeigt werden. Danach soll zu einer beliebigen Eingabe aufgefordert werden, um die nächsten 25 Zahlen anzuzeigen. Spoiler: Verwenden Sie den Modulo-Operator (%).

2.11.2 Zinsentwicklung

Schreiben Sie ein Programm ZINS, welches die Entwicklung eines Kapitals einschließlich Zinsen für einen beliebigen Zeitraum berechnet. Eingabe: Kapital, Zinssatz (in %), Laufzeit (in Jahren) Ausgabe: Kapital: 1000, Zinssatz(in %): 5, Laufzeit(Jahre): 10

Jahr	Kapital
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	u.s.w.

2.11.3 ASCII-Tabelle

Schreiben Sie ein Tool, welches eine ASCII-Tabelle der ASCII-Zeichen 32 bis 255 am Bildschirm in formatierter Ausgabe angezeigt.

3 Strukturen, Klassen und Pointer

3.1 Strukturen mit struct

Strukturen sind Container für Daten meist unterschiedlichen Datentyps, die aber inhaltlich zusammengehören. Es könnte beispielsweise eine Struktur für eine Person geben, die sich aus dem Namen, dem Wohnort, dem Alter und dem Beruf zusammensetzt. In einer Struktur kann man also Integer, Strings oder sogar Objekte gruppieren. Strukturen sind zur Modellierung einfacher Modelle nützlich, die nur Datenelemente benötigen, aber keine Methoden. Es gibt Strukturen auch in C allerdings sind die Syntaxregeln in Teilen etwas unterschiedlich. Listing 3.1 gibt ein einfaches Beispiel für den Umgang mit Strukturen.

Listing 3.1: Beispiel für Strukturen

```
1 // Structures.cpp : Example for struct
2 //
3
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
8
9 enum dog_purpose { cuddle, therapeutic, hunting, protection, searching };
10
11 struct dog {
12     string name;
13     int age;
14     unsigned char purpose;
15 };
16
17 int main() {
18     dog my_dog;
19     my_dog.age = 5;
20     my_dog.name = "Betsy";
21     my_dog.purpose = therapeutic;
22     cout << my_dog.name << "is_a_type-"
23          << (int)my_dog.purpose << "dog." << endl;
24     cout << my_dog.name << "is_"
25          << my_dog.age << "years_old." << endl;
26     return (0);
27 }
```


Das Listing 3.1 definiert in den Zeilen 11 - 15 die Struktur eines Hundes. Diese setzt sich zusammen aus dem Namen, einer `string` Variablen, dem Alter, einer `int` Variablen und einer Kennung für den Zweck des Hundes, einer `unsigned char` Variablen. Jede Definition einer Struktur beginnt mit dem `struct` Schlüsselwort und deklariert die enthaltenen Variablen zwischen geschweiften Klammern. Jede Definition wird mit einem Semikolon abgeschlossen.

```
struct dog {
    string name;
    int age;
    unsigned char purpose;
};
```

Jeder Verwendungszweck eines Hundes kann durch einen eindeutigen Wert beschrieben werden. Das heißt, ein bestimmter Wert repräsentiert einen bestimmten Zweck. So könnte etwa der Wert 0 dem Zweck *Schmusehund* oder der Wert 1 dem Zweck *Therapiehund* entsprechen. Jetzt könnte man Konstanten für jeden dieser Zwecke definieren, aber C++ bietet mit der Enumeration eine bessere Möglichkeit. Eine Enumeration stellt eine Menge von Symbolen dar, die als Konstanten agieren, denen aber aufeinander folgende Werte zugeordnet sind. Deshalb werden sie als Enumerationen bezeichnet. Eine Enumeration wird in C++ einfach in der folgenden Form erstellt:

```
enum dog_purpose {cuddle, therapeutic, hunting, protection, searching};
```

Beginnend mit dem Schlüsselwort `enum` folgt der Name der Enumeration und dann in geschweiften Klammern eine Liste der Symbole, denen bei Null beginnend einfach aufsteigend Nummern zugeordnet werden.

So, wie wird nun die Struktur verwendet? In der `main()` Funktion wird zunächst in Zeile 18 die Variable `my_dog` als Struktur `dog` deklariert. In den darauf folgenden Zeilen werden dann den Elementen der Struktur Werte zugewiesen. Dabei wird `my_dog.purpose` mit dem Symbol `therapeutic` der Wert 1 zugewiesen.

Die Zeilen 22-23 geben dann die Werte von `my_dog` wieder aus.

Ein weiteres Beispiel für eine Struktur wäre die Abbildung eines Kalenderdatums.

```
struct datum {
    unsigned int day;
    unsigned int month;
    unsigned int year;
}
```

Allerdings werden hier jeweils 4 Byte (32 Bit) für Tag, Monat und Jahr reserviert. Das sind insgesamt 12 Bytes für ein Datum. Eigentlich wären für eine Tageszahl (1-31) aber lediglich 5 Bit notwendig und für einen Monat reichen sogar 4 Bit, während für ein Jahr 11 bit ausreichend wären. Insgesamt wären also 20 Bit ausreichend für die Darstellung eines Kalenderdatums. Man könnte diese Information also leicht in einer 4-Byte Integerzahl

ablegen. Nun kann man aber dem Compiler bei einer Struktur leicht mitteilen, wie viele Bits wirklich benötigt werden und so die Datenstruktur optimieren.

```
struct datum {  
    unsigned int day : 5;  
    unsigned int month : 4;  
    unsigned int year : 11;  
    unsigned int leapyear : 1;  
}
```

Auf diese Weise können sogar noch weitere Informationen in der 4-Byte langen Datenstruktur unterbringen, wie etwa ein 1 Bit großes Element, das Aufschluss darüber gibt, ob es sich um ein Schaltjahr handelt.

3.2 Klassen

Klassen sind die wesentlichen Bausteine der objektorientierten Programmierung. Dabei werden durch Klassen Modelle, die Aspekte der Realität abbilden, erstellt. Klassen sind aus zwei wesentlichen Bestandteilen aufgebaut: Daten und Funktionen. Dabei unterscheiden sie sich nicht von dem Klassenbegriff in anderen Programmiersprachen wie Java oder Python. Lassen Sie mich mit dem einfachen Beispiel eines Hundes beginnen, den wir in Listing 3.1 als Struktur modelliert hatten.

Um Ihnen die Verwandtschaft zwischen Klassen und Strukturen zu zeigen, ersetzen wir einfach das Schlüsselwort `struct` durch `class`. Wenn wir jetzt versuchen, das Programm zu übersetzen, bekommen wir aber einige Fehler. Die Definition unserer Klasse `dog` hat dabei keine Probleme gemacht. Diese treten erst auf, wenn wir auf die Elemente der Klasse `dog` zugreifen wollen, um diese in der `main()` Funktion auszugeben. Der Zugriff ist nicht möglich. Das liegt daran, dass die Elemente von Klassen standardmäßig als `private` deklariert werden, und damit sind sie außerhalb der Klasse nicht sichtbar. D. h. Klassen erledigen also genau dass, was sie eigentlich sollen, nämlich ihre Elemente nach außen hin zu verbergen, so dass sie wie eine *Black Box* wirken, auf die nur über definierte Schnittstellen zugegriffen werden kann.

Diese Fehler können wir aber leicht ändern, indem wir die Elemente einfach als `public` deklarieren.

Listing 3.2: Die erste Klasse

```
1 // ClassExample.cpp : Example for first class  
2 // ... does not work as data are private by default  
3 //  
4  
5 #include <iostream>  
6 #include <string>  
7  
8 using namespace std;
```

```

9
10 enum dog_purpose { cuddle, therapeutic, hunting, protection, searching };
11
12 class dog {
13 public:
14     string name;
15     int age;
16     unsigned char purpose;
17 };
18
19 int main() {
20     dog my_dog;
21     my_dog.age = 5;
22     my_dog.name = "Snafu";
23     my_dog.purpose = therapeutic;
24     cout << my_dog.name << "is_a_type-"
25           << (int)my_dog.purpose << "dog." << endl;
26     cout << my_dog.name << "is_"
27           << my_dog.age << "years_old." << endl;
28     return (0);
29 }

```

Wenn wir Listing 3.2 übersetzen und laufen lassen, sehen wir, dass wir damit die Struktur in eine Klasse umgewandelt haben, und die Klasse verhält sich wie die Struktur in dem Listing 3.1. Damit haben wir die Struktur in eine Klasse umgewandelt. Genauer müsste man allerdings sagen, wir haben die Klasse gezwungen, sich wie eine Struktur zu verhalten.

Da Klassen alle Konzepte der objektorientierten Programmierung umsetzen, sollten wir auch diese nutzen. So wollen wir zunächst die Kapselung der Klasse realisieren, durch die die Elemente der Klasse außerhalb der Klasse versteckt werden. Dazu werden alle Datenelemente als **private** und nicht mehr als **public** deklariert. In dem Beispiel greifen wir aber sowohl lesend als auch schreibend darauf zu. Wenn wir Werte abspeichern, greifen wir schreibend darauf zu, und wenn wir diese ausgeben wollen, so greifen wir lesend auf die Werte zu. Es wird bei der objektorientierten Programmierung immer empfohlen, die Daten soweit möglich als **private** zu deklarieren. Aber wie können wir jetzt auf diese zugreifen?. Dazu muss man sich erinnern, das Klassen sowohl aus Daten als auch aus Funktionen bestehen, und genau diese verwenden wir jetzt, um den Zugriff auf die Daten zu regeln, wenn wir diese explizit als **private** deklarieren. Jetzt müssen wir unser Programm, wie in Listing 3.3 gezeigt anpassen, um darauf zuzugreifen.

Listing 3.3: Die erste richtige Klasse mit Kapselung

```

1 // ClassExample2.cpp : Class with setter and getter functions
2 //
3 //
4
5 #include <iostream>

```

3 Strukturen, Klassen und Pointer

```
6 #include <string>
7
8 using namespace std;
9
10 enum dog_purpose { cuddle, therapeutic, hunting, protection, searching };
11
12 class dog {
13 private:
14     string name;
15     int age;
16     unsigned char purpose;
17 public:
18     // getter functions
19     string getName() {
20         return name;
21     }
22     int getAge() {
23         return age;
24     }
25     int getPurpose() {
26         return purpose;
27     }
28     // setter functions
29     void setName(string strName) {
30         name = strName;
31     }
32     void setAge(int iAge) {
33         age = iAge;
34     }
35     void setPurpose(unsigned char cPurpose) {
36         purpose = cPurpose;
37     }
38 };
39
40 int main() {
41     dog my_dog;
42
43     my_dog.setAge(5);
44     my_dog.setName("Snafu");
45     my_dog.setPurpose(therapeutic);
46     cout << my_dog.getName() << "is_a_type-"
47          << (int)my_dog.getPurpose() << "dog." << endl;
48     cout << my_dog.getName() << "is"
49          << my_dog.getAge() << "years_old." << endl;
50     return (0);
51 }
```

In diesem Listing sieht man sofort, dass es zwei Arten von Funktionen gibt: *Getter-Funktionen*, um auf Datenelemente zuzugreifen und *Setter-Funktionen*, um Werte in

Datenelemente zu schreiben. Damit diese von außerhalb der Klasse aufgerufen werden können, müssen diese Funktionen jetzt als `public` deklariert werden. Wir haben jetzt also in diesem Beispiel für jedes Datenelement eine Setter-Funktion und eine Getter-Funktion, um damit Werte in die Datenelemente zu schreiben, beziehungsweise diese zu lesen.

Die Setter-Funktionen verändern Datenelemente eines bereits existierenden Objekts einer Klasse, wie sieht es aber aus, wenn wir ein neues Objekt einer Klasse deklarieren und auch gleich initialisieren wollen? Dazu können wir eine spezielle `public` Funktion verwenden, einen sogenannten Konstruktor. Ein Konstruktor wird immer aufgerufen, wenn ein neues Objekt einer Klasse geschaffen wird. Jede Klasse hat immer einen impliziten Konstruktor, der nichts anderes macht, als den Speicherplatz für die Datenelemente der Klasse zu allozieren. Allerdings kann man weitere Konstruktoren schaffen, die sich jeweils spezielle Parameter verwenden. Dieses wird als Überladen oder *Overloading* einer Funktion bezeichnet. Darunter versteht man, dass man eine Funktion desselben Namens definiert, die sich lediglich in den übergebenen Parametern unterscheidet. Ein klassischer Konstruktor übergibt dann Werte für alle Datenelemente des Objekts. Einen solcher Konstruktor ist in Listing 3.4 realisiert.

Listing 3.4: Erzeugen eines Objekts mit dem Konstruktor

```

1 // ClassExample3.cpp : added constructor
2 //
3 //
4 //
5
6 #include <iostream>
7 #include <string>
8
9 using namespace std;
10
11 enum dog_purpose { cuddle, therapeutic, hunting, protection, searching };
12
13 class dog {
14 private:
15     string name;
16     int age;
17     unsigned char purpose;
18 public:
19     // constructor
20     dog(string iName, int iAge, unsigned char cPurpose) {
21         name = iName;
22         age = iAge;
23         purpose = cPurpose;
24     }
25     // getter functions
26     string getName() {
27         return name;
28     }

```

3 Strukturen, Klassen und Pointer

```
29     int getAge() {
30         return age;
31     }
32     int getPurpose() {
33         return purpose;
34     }
35     // setter functions
36     void setName(string strName) {
37         name = strName;
38     }
39     void setAge(int iAge) {
40         age = iAge;
41     }
42     void setPurpose(unsigned char cPurpose) {
43         purpose = cPurpose;
44     }
45 };
46
47
48 int main() {
49     dog my_dog("Snafu", 5, therapeutic);
50
51     cout << my_dog.getName() << "is a type-"
52          << (int)my_dog.getPurpose() << "dog." << endl;
53     cout << my_dog.getName() << "is"
54          << my_dog.getAge() << "years old." << endl;
55     return (0);
56 }
```

Ein Konstruktor unterscheidet sich von anderen Funktionen, weil er keinen Rückgabewert spezifiziert. Der Name des Konstruktors muss derselbe sein, den auch die Klasse selber hat. In unserem Fall wird der Konstruktor verwendet, um alle Datenelemente des Objekts zu initialisieren. Der Konstruktor selber übernimmt die Aufgabe den benötigten Speicherplatz für das Objekt zu allozieren.

Auf den ersten Blick sieht es so aus, als würden die Setter- und Getter-Funktionen den Vorteil, den wir durch die privaten Datenelemente haben zunichte machen. Allerdings ist es nicht der Zweck, diese Datenelemente komplett un erreichbar zu machen, sondern eher den Zugriff über eine definierte Schnittstelle sicherzustellen. So kann beispielsweise eine Setter-Funktion gewährleisten, dass sich die übergebenen Werte immer in einem erlaubten Bereich bewegen. Weiterhin ist es beispielsweise möglich andere Datenelemente, die in Abhängigkeit zu dem zu ändernden Wert stehen, ebenfalls anzupassen, so dass alle Datenelemente der Objekts untereinander immer in einem sinnvollen Bezug stehen.

3.3 Challenge

3.3.1 Klasse Student

Definieren Sie eine Klasse `student`, die aus den folgenden Datenelementen besteht:

- `studentID` (int)
- `firstname` (string)
- `lastname` (string)
- `birthdate` (int day, int month, int year)
- `studiengang` (string)

Definieren Sie zugehörigen Setter- und Getter-Funktionen und einen Konstruktor, der alle Datenelemente füllt. Erweitern Sie die Klasse um eine Funktion, die einen Kurznamen in der Form `U. Schrader` zurückgibt, wenn der Vorname beispielsweise `Ulrich` lautet. Testen Sie Ihre Klasse, indem Sie in der `main()` Funktion entsprechende Objekte mit Ihrem Konstruktor instantiieren oder mit dem Default-Konstruktor und den Setter-Funktionen initialisieren. Geben Sie Ihre Objekte aus und kontrollieren Sie, ob alles geklappt hat.

3.4 Compiler und Linker

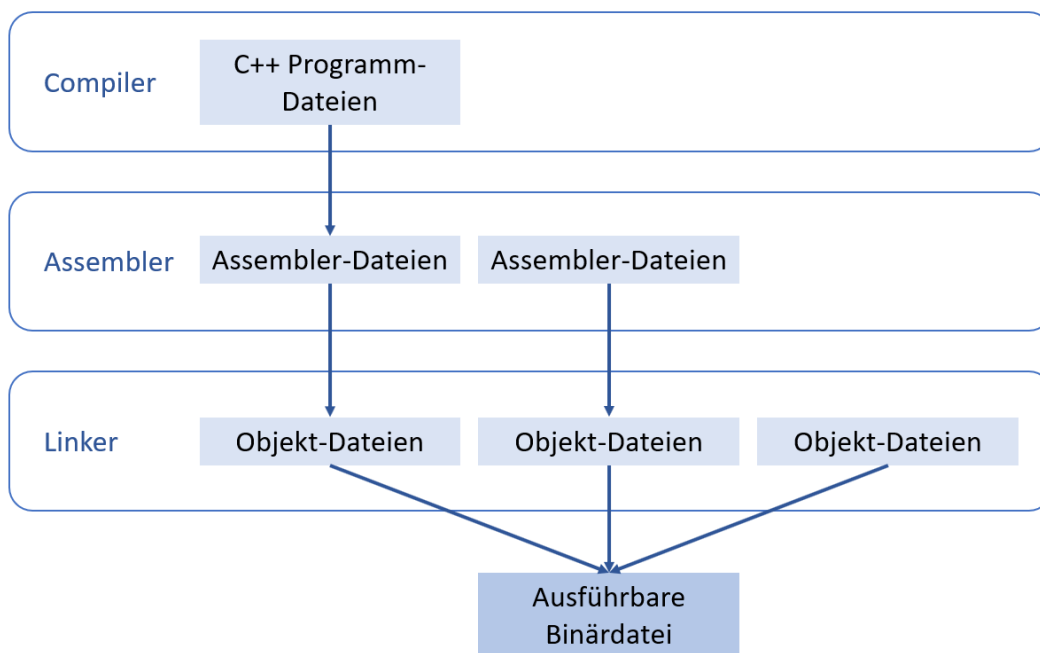


Abbildung 3.1: Hauptschritte zum Erzeugen eines ausführbaren Programms

3 Strukturen, Klassen und Pointer

An dieser Stelle macht es Sinn etwas näher auf den Prozess zum Erzeugen einer vom Rechner ausführbaren Datei einzugehen. Dieses geschieht in der Regel durch eine Reihe wie in einer Pipeline hintereinander geschalteter Werkzeuge. Dabei gibt es drei wesentliche Fälle, die unterschieden werden müssen. Sie machen in der Regel die Bestandteile eines komplexen Software-Entwicklungsprojekts aus.

Der erste Fall liegt vor, wenn C++ Programmcode oder Sourcecode geschrieben wird. Wenn diese Dateien dem Compiler zugeführt werden, dann wird der C++ Code in Assembler übersetzt, was einer lesbare Version der nativen binären Sprache des Zielprozessors entspricht. Es wird also für jede C++ Datei eine neue Datei in Assembler Code erzeugt. Diese Assembler-Dateien werden jetzt jeweils dem Assembler zugefügt, der daraus fast ausführbare sogenannte Objekt-Dateien in Maschinencode erzeugt, die bereits wesentliche Teile des ausführbaren Programms enthalten. Diese Objektdateien haben aber in der Regeln nichts mit den Objekten der objektorientierten Programmierung zu tun, sondern es ist einfach ein historisch bedingter Name für diesen Dateityp.

Der zweite Fall liegt vor, wenn man seinen eigenen Assembler-Code schreiben muss. Obwohl das etwas etwas aufwändig sein kann, gibt es Fälle bei denen einem nichts anders übrig bleibt als *low-level-code* zu schreiben. Dies ist oft der Fall, wenn etwas Treiber für Peripheriegeräte entwickelt werden müssen. In diesem Fall werden dann die selbst geschriebenen Assembler-Dateien wiederum dem Assembler zugefügt und erzeugen weitere Objektdateien. Das Assembler-Programme nicht immer gut lesbar sind, zeigt das Listing 3.5. Das Programm ist zwar noch von Menschen lesbar, allerdings kann es mühsam werden, umfangreiche Programme auf diese Weise zu entwickeln.

Listing 3.5: Hello World in Assembler

```
1 #-----
2 # Writes "Hello, World" to the console using only system
3 # calls. Runs on 64-bit Linux only.
4 # To assemble and run:
5 #     gcc -c hello.s && ld hello.o && ./a.out
6 # or
7 #     gcc -nostdlib hello.s && ./a.out
8 # -----
9
10     .global _start
11
12     .text
13 _start:
14     # write(1, message, 13)
15     mov     $1, %rax           # system call 1 is write
16     mov     $1, %rdi           # file handle 1 is stdout
17     mov     $message, %rsi     # address of string to
18                                # output
19
20     mov     $13, %rdx          # number of bytes
21     syscall                    # invoke operating system to
```



```

22                                     # do the write
23
24     # exit(0)
25     mov     $60, %rax               # system call 60 is exit
26     xor     %rdi, %rdi             # we want return code 0
27     syscall                        # invoke operating system
28                                     # to exit
29 message:
30     .ascii  "Hello, \world\n"

```

Im dritten Fall werden etwa Bibliotheken (*libraries* benötigt, bei denen man keinen Zugriff auf den Sourcecode hat. Dabei werden diese oft als Objektdateien oder in kompatiblen Formaten angeboten. Insbesondere, wenn die Bibliotheken gekauft sind, hat man in der Regel keinen Zugang zu dem Sourcecode. Diese Objektdateien oder Bibliotheken können aber hinzugefügt werden, wie wir das etwa mit der Stringbibliothek oder der I/O-Stream-Bibliothek in unseren Beispielen gemacht haben.

In diesem Stadium haben wir also eine Handvoll von Objektdateien, die alles wichtige Teile des Gesamtprogramms ausmachen. Diese Dateien werden nun an den Linker gesandt, der sie alle zu der ausführbaren binären Datei zusammenfügt. Das ist Ihr Programm, dass Sie jetzt endlich lauffähig fertiggestellt haben. Programme wie `gcc` oder `g++` umfassen alle drei Arbeitsschritte.

Wenn eine integrierte Entwicklungsumgebung IDE wie Eclipse, Visual Studio o. ä. verwendet wird, ist es besonders einfach, da sich diese um den korrekten Ablauf der Übersetzung Ihres Programmes in eine ausführbare Version selber kümmert. Jedoch gibt es manchmal die Notwendigkeit den Compiler über die Kommandozeile mit entsprechenden Parametern aufzurufen. Dann kann es etwas kompliziert werden. All die notwendigen Arbeitsschritte werden üblicherweise in einer Textdatei, die als Make-Datei bezeichnet wird festgehalten. IDEs übernehmen diese Aufgabe für Sie und erstellen die Make-Datei üblicherweise in Ihrem Projektverzeichnis. Natürlich können Sie eine solche Make-Datei auch selber erstellen, sie ist eine ganz normale Textdatei. Allerdings übernimmt in der Regel die IDE diese Aufgabe für Sie.

3.5 Programm- und Headerdateien

Mit diesem Wissen können wir zur eigentlichen C++ Programmierung zurückkehren und über Programmdateien, auch als Sourcedateien bezeichnet, und Headerdateien sprechen.

Üblicherweise werden C++ Programme in Paaren von Header und Implementationsdateien geschrieben, die jeweils eine oder mehrere zusammenhängende Klassen umfassen. Damit kann jede Klasse für sich von anderen Programmen leicht verwendet werden. In ganz gleicher Weise, wie wir etwa die `string` Klasse in unsere Anwendungen eingebunden haben. Die Header-Datei hat dabei die `.h` Endung und die Source-Datei die gewohnte

.cpp Endung. Die Header-Dateien beinhalten dabei die Klassen- und Funktionsdeklarationen sowie benötigte Makros, aber *keinen* ausführbaren Programmcode. Source- und Header-Dateien haben dabei üblicherweise denselben Namen. Betrachten wir also alles, was zu der Header-Datei der `dog` Klasse gehört in Listing 3.6

Listing 3.6: Header-Datei für die `dog` Klasse

```
1 #pragma once
2
3 #ifndef DOG_H
4 #define DOG_H
5
6 #include <string>
7
8 enum dog_purpose { cuddle, therapeutic, hunting, protection, searching };
9
10 class dog {
11 private:
12     std::string name;
13     int age;
14     unsigned char purpose;
15 public:
16     // constructor
17     dog(std::string iName, int iAge, unsigned char cPurpose);
18
19     // getter functions
20     std::string getName();
21     int getAge();
22     int getPurpose();
23
24     // setter functions
25     void setName(std::string strName);
26     void setAge(int iAge);
27     void setPurpose(unsigned char cPurpose);
28 };
29
30 #endif // DOG_H
```

Die Implementations- oder Source-Dateien umfassen dann die Implementation aller Funktionen die in der Header-Datei deklariert worden sind. Dieses ist der ausführbare Programmcode. Die Implementationsdateien müssen natürlich die zugehörigen Header-Dateien einfügen, da der Compiler die Klassen- und Funktionsdeklarationen sowie eventuelle Makros kennen muss, damit er den Programmcode in den Implementationsdateien korrekt übersetzen kann. Ebenso muss natürlich der externe Programmcode, der die Klasse nutzen soll, auch die Header-Datei inkludieren. Schlussendlich müssen ja alle Programmdateien korrekt übersetzt werden.

In diesem Beispiel wollen wir unsere Klasse `dog` einem anderen Programm zur Nutzung anbieten. Dazu benötigen wir die Header-Datei `dog.h` aus Listing 3.6 und die Implemen-

tationsdatei `dog.cpp` aus Listing 3.7. Sehen wir zunächst die Header-Datei etwas näher an.

Beachten Sie, dass die Header-Datei mit der Direktive `#pragma once` beginnt. Dieses teilt dem Präprozessor mit, dass die Datei nur einmal eingefügt werden soll. Es kommt oft vor, dass dieselbe Header-Datei in mehreren Source-Dateien eingefügt wird, und das ist nicht notwendigerweise eine schlechte Programmierpraxis. Durch diese Direktive wird verhindert, dass der Compiler Deklarationen vorfindet, die er bereits kennt. Allerdings muss beachtet werden, dass `#pragma once` nicht von allen Compilern erkannt wird. Daher gibt es noch einen alternativen *Include-Wächter*. Hierbei wird die `#ifndef` Präprozessoranweisung ausgenutzt. Hier wird getestet, ob das Makro `DOG_H` definiert ist. Bei der Benennung dieses Makros nimmt man meist etwas, das dem Dateinamen der Header-Datei nahe kommt. Wenn jetzt wie in diesem Fall `DOG_H` noch nicht deklariert ist, dann werden alle nachfolgenden Deklarationen eingefügt und das Makro `DOG_H` definiert. Damit kann dieser Inhalt kein zweites Mal eingefügt werden, da `DOG_H` jetzt ja bekannt ist. Diese Lösung funktioniert mit allen Compilern.

In diese Header-Datei gehört jetzt die Deklaration der Klasse `dog` mit ihren Datenelementen und die Deklaration der zugehörigen Funktionen. Ebenso wird hier die Definition der Enumeration `dog_purpose` aufgenommen. Da wir die Klasse `string` verwenden, muss auch hier die zugehörige Header-Datei inkludiert werden. Da diese Header-Dateien in einer Vielzahl von zukünftigen Programmen verwendet werden sollen, ist es schlechte Programmierpraxis in den Header-Dateien einen `namespace` festzulegen, da dieses beispielsweise bereits in der inkludierenden Source-Datei geschehen sein kann. Daher müssen wir an den entsprechenden Stellen noch `std::` in die Header-Datei einfügen. Würde man hier `using namespace std` einfügen, so würde man diesen Namespace für das gesamte Projekt festlegen. Meist ist das aber nicht gewollt. Zu guter Letzt muss sichergestellt werden, dass die Klasse `dog` nur noch die Deklarationen der zugehörigen Funktionen enthält, nicht aber die komplette Definition mit den ausführbaren Anweisungen. Diese gehören in die Implementationsdatei `dog.cpp` wie in 3.7 gezeigt.

Listing 3.7: Implementationsdatei für die `dog` Klasse

```

1 #include "dog.h"
2
3 // constructor
4 dog::dog(std::string iName, int iAge, unsigned char cPurpose) {
5     name = iName;
6     age = iAge;
7     purpose = cPurpose;
8 }
9 // getter functions
10 std::string dog::getName() {
11     return name;
12 }
13 int dog::getAge() {
14     return age;
15 }

```

3 Strukturen, Klassen und Pointer

```
16 int dog::getPurpose() {
17     return purpose;
18 }
19 // setter functions
20 void dog::setName(std::string strName) {
21     name = strName;
22 }
23 void dog::setAge(int iAge) {
24     age = iAge;
25 }
26 void dog::setPurpose(unsigned char cPurpose) {
27     purpose = cPurpose;
28 }
```

Wie sie sehen können, muss die Implementationsdatei jetzt auch `dog.h` inkludieren, damit die Deklaration der Klasse bekannt und ihre Funktionen und Datenelemente bekannt sind. Ferner müssen die Funktionen als der Klasse zugehörig gekennzeichnet werden. Dieses geschieht durch ein vorangesetztes `dog::` vor den Funktionsnamen.

Nach diesen Vorarbeiten können wir jetzt einfach die Klasse `dog` verwenden. Dieses ist beispielhaft in Listing 3.8 aufgezeigt.

Listing 3.8: Verwendung der `dog` Klasse

```
1 // ClassExample4.cpp : Class has its own file + header file
2 // ClassExample3.cpp : added constructor
3 //
4 //
5 //
6
7 #include <iostream>
8 #include <string>
9 #include "dog.h"
10
11
12 int main() {
13     dog my_dog("Snafu", 5, therapeutic);
14
15     std::cout << my_dog.getName() << "is_a_type-"
16         << (int)my_dog.getPurpose() << "dog." << std::endl;
17     std::cout << my_dog.getName() << "is_"
18         << my_dog.getAge() << "years_old." << std::endl;
19     return (0);
20 }
```

Dieses einfache Programm verwendet die Klasse `dog`. Wir haben damit die "Innereien" der Klasse gut gekapselt und erlauben den Zugriff nur über definierte Schnittstellen (Funktionen). Ist die Klasse ausgiebig getestet worden, so steht sie jetzt allen weiteren Programmen zur Verfügung.

3.6 Pointer

Pointer sind ebenso wie in C ein wesentlicher Bestandteil der Programmiersprache C++. Dabei sind Pointer ganz normale Variable, deren Wert aber immer die Adresse eines Speicherplatzes ist. Um dieses noch einmal kurz zu demonstrieren, zeigt Listing 3.9 noch einmal die wesentlichen Aspekte beim Umgang mit Pointern.

Listing 3.9: Einfaches Beispiel für Pointer

```

1 // Pointer1.cpp : Simple example for the use of pointers
2
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 int a = 37;
9 int* ptr;
10
11 int main() {
12     ptr = &a;
13
14     cout << "The content of a is " << a << endl;
15     cout << "ptr is pointing to address " << ptr << endl;
16     cout << "The address of a is " << &a << endl;
17     cout << "Where ptr is pointing, we have " << *ptr << endl;
18     cout << "The address of ptr is " << &ptr << endl;
19     cout << endl;
20
21     return (0);
22 }
```

In diesem Listing werden die wesentlichen Operatoren noch einmal in ihrer Verwendung gezeigt. Zunächst wird in Zeile 8 eine, in diesem Falle globale, Integer-Variable deklariert und mit dem Wert 37 initialisiert. Die darauf folgende Zeile deklariert einen Pointer `ptr`, der auf jede beliebige Integer-Variable zeigen kann. In der `main()` Funktion in Zeile 12 wird dann dieser Pointer-Variablen die Adresse der Integer-Variablen `&a` zugewiesen. Sie zeigt damit direkt auf `a`. Zeile 14 gibt dann direkt den Wert der Integer-Variablen `a` aus. In Zeile 15 und 16 wird dann gezeigt, dass die Pointer-Variable `ptr` als Inhalt die Adresse von `a` gespeichert hat. Zeile 17 gibt dann den Wert aus, der an der Stelle gespeichert ist, auf die `ptr` zeigt. Zeile 18 zeigt dann nur noch die Adresse auf, an der die Inhalte von `ptr` selber gespeichert sind. Wichtig sind dabei der `*` Operator, der eine Pointer-Variable dereferenziert, so dass auf den Inhalt zugreifen kann, der an der Stelle gespeichert ist, auf die die Pointer-Variable zeigt. Ebenso ist der Adressoperator `&` wichtig, der zu jeder Variablen ihre Adresse zurückgibt.

In dem Beispiel in Listing 3.9 haben wir gesehen, wie mit Pointern gearbeitet werden kann, allerdings ist es meist nicht notwendig mit Pointern auf bestehende Variablen zu

3 Strukturen, Klassen und Pointer

arbeiten. Viel interessanter wird es, wenn mit Pointern gearbeitet wird, weil zur Laufzeit Speicher dynamisch alloziert wird und wieder freigegeben wird. Dazu sind Pointer unerlässlich.

Zur Erinnerung, immer wenn eine Variable deklariert wird, so wird statisch Speicher reserviert. Der Compiler weiß bereits im Vorfeld, dass dieser Speicherplatz benötigt wird. Allerdings benötigen die meisten Anwendungen zusätzlichen Speicherplatz zur Laufzeit des Programms. In C wurde dieser Speicherplatz dann durch die Familie der `malloc()` Funktionen (kurz für memory allocation) dynamisch nach Bedarf alloziert. Um diesen Speicherplatz nutzen zu können, sind Pointer unerlässlich. Als Entwickler ist man dann allerdings dafür zuständig, sicherzustellen, dass dieser dynamisch allozierte Speicherplatz auch wieder freigegeben wird, wenn er nicht mehr benötigt wird. Geschieht dieses nicht, kann ein Programm zur Laufzeit immer mehr Speicher allokieren, bis schließlich keiner mehr zur Verfügung steht. Man spricht dann von sogenannten *memory leaks*. Verglichen mit C macht einem C++ hier vieles einfacher.

Auch in C++ wird die Verwendung von Pointern unterstützt, um dynamisch Speicher für neue Objekte zu allokieren. Dieses geschieht, wenn ein Konstruktor der Klasse aufgerufen wird, gleichermaßen, wenn der Speicher nicht mehr benötigt wird. Listing 3.10 erläutert das Vorgehen.

Listing 3.10: Pointer auf Objekte

```
1 // Pointer2Class.cpp : Instantiate a new object of a class.
2 #include <iostream>
3 #include <string>
4 #include "D:\...\put_your_path_here...\dog.h"
5
6 using namespace std;
7
8 int a = 37;
9 int* ptr;
10 dog* my_dog;
11
12 int main() {
13     // dynamically create a new object of the dog class
14     my_dog = new dog("Suki", 3, cuddle);
15     cout << my_dog->getName()
16          << "is_a_type_"
17          << (int)my_dog->getPurpose()
18          << "dog."
19          << endl;
20     cout << my_dog->getName()
21          << "is_"
22          << my_dog->getAge()
23          << "years_old."
24          << endl;
25
26     // free the memory used by my_dog
```

```

27     delete my_dog;
28     return (0);
29 }

```

Anstelle jetzt den Speicher für ein Objekt der Klasse `dog` statisch bereitzustellen, wird in Zeile 10 eine Pointer-Variable `my_dog`, die auf ein solches Objekt zeigt deklariert. Damit ist noch kein Speicher für das Objekt alloziert worden. In Zeile 14 wird jetzt der Speicherplatz für das neu instantiierte Objekt dynamisch alloziert und der Pointer-Variablen `my_dog` zugewiesen. Dieses geschieht durch das Schlüsselwort `new`. Der Konstruktor stellt sicher, dass dieses neue Objekt mit den entsprechenden Anfangswerten initialisiert wird. Damit sind wir fast fertig. Allerdings kann jetzt auf die Datenelemente und die Funktionen nicht mehr mit dem Punkt-Operator zugegriffen werden, da `my_dog` ein Pointer ist und kein Objekt. Daher muss zunächst der Pointer dereferenziert werden, um auf die Datenelemente zuzugreifen. Dazu könnte der folgende Ausdruck verwendet werden.

```
(*my_dog).getName();
```

Der Ausdruck `(*my_dog]` muss dabei in Klammern gesetzt werden, da ja zunächst der Pointer dereferenziert werden muss, bevor auf die Datenelemente oder Funktionen zugegriffen werden kann. Dieses ist syntaktisch korrekt, erfordert aber die Verwendung vieler Klammern, wodurch die Konstrukte nicht gut lesbar sind. Glücklicherweise gibt es dafür einen verkürzenden Operator in Form der `->` Operators. Der direkt veranschaulicht, dass der Pointer irgendwo hin zeigt. Er verbindet den Pointer auf ein Objekt und das zugehörige Datenelement oder die zugehörige Funktion. Klammern werden dabei nicht benötigt. In dem Listing 3.10 werden auf diese Weise die Getter-Funktionen aufgerufen und die Inhalte der Datenelemente des `dog` Objekts ausgegeben.

So zum Schluss muss noch der Speicherplatz für das Objekt wieder freigegeben werden. Diese geschieht in Zeile 27 mit der Anweisung:

```
delete my_dog;
```

Das Schlüsselwort `delete` macht dabei das Gegenteil des `new` Schlüsselworts.

In diesem Fall endet das Programm zwar unmittelbar darauf und der Speicherplatz würde dennoch freigegeben werden. Aber es ist in jedem Fall gute Programmierpraxis den Speicherplatz für dynamisch allokierte Objekte auch wieder freizugeben. Dabei ruft der `delete` Operator eine spezielle Funktion der Klassen auf, den sogenannten Destruktor, dessen Aufgabe darin besteht den dynamisch allokierten Speicher freizugeben. Wie den Konstruktor kann man den vorgegebenen implizierten Destruktor auch durch eine eigene Version überladen. So ist es beispielsweise möglich, die Inhalte eines Objekts zu speichern bevor der Speicher dann freigegeben wird.

```

class X {
public:
    // Constructor for class X
    X();

```

```
// Destructor for class X
~X();
};
```

Wenn **X** eine Klasse ist, dann übernimmt der Konstruktor einfach den Klassennamen. Für den Destruktor wird dem Klassennamen einfach eine `~` vorangestellt, so dass dann `~X()` die Funktion des Destruktors ist.

3.7 Challenge

3.7.1 Klasse **student** reloaded

Nehmen Sie die in Challenge 3.3 definierte Klasse **student** und wandeln diese Klasse in die eigenständigen Klassendateien **student.h** und **student.cpp**, so dass Sie eine Header-Datei erhalten, die Sie in weitere Projekte einbinden können.

- Schreiben Sie ein Programm, welches **student.h** inkludiert, um die Klasse zu nutzen. Legen Sie ein Objekt der Klasse **student** in einer Variablen an, und testen Sie, dass damit korrekt gearbeitet werden kann.
- Schreiben Sie ein Programm, welches **student.h** inkludiert, um die Klasse zu nutzen. Erzeugen Sie dynamisch ein Objekt der Klasse **student** und überprüfen Sie, dass es korrekt funktioniert. Entfernen Sie das Objekt dann wieder.

3.8 Referenzen

Bei den sogenannten Referenzen handelt es sich um einen Kategorie von Datentyp, die bisher noch nicht erwähnt wurden. Man kann Referenzen als Alias zu bestehenden Variablen ansehen. Damit ist eine Referenz so etwas wie ein alternativer Name für eine Variable. Um eine Referenz zu deklarieren, wird an den Datentyp bei der Deklaration einfach ein `&` angefügt.

```
int& my_ref = my_int;
```

In dem obigen Statement wird eine Referenz namens **my_ref** deklariert, die eine Referenz auf die bestehende Integer-Variable **my_int** darstellt. Sobald diese deklariert ist, kann damit in derselben Syntax auf die Variable **my_int** zugegriffen werden, wie mit **my_int** selber. Dabei muss **my_ref** nicht etwa dereferenziert werden, wie das bei Pointern der Fall wäre.

Drei wesentliche Aspekte müssen dabei im Blick behalten werden:

1. Referenzen sind keine Pointer.
2. Referenzen sind keine Objekte. Sie können aber sehr wohl Objekte referenzieren.
3. Referenzen haben keine Adresse.

Hieraus ergeben sich einige wesentliche Konsequenzen. Zum einen kann es kein Array von Referenzen geben. Um ein Element eines Arrays anzusprechen, benötigt man eine Adresse. Zum anderen gibt es keine Pointer auf Referenzen aus demselben Grund. Und zusätzlich kann es keine Referenzen auf Referenzen geben. Wenn eine Referenz als Referenz auf eine andere Referenz initialisiert wird, dann sind beide Aliase für dieselbe Variable. Daher macht es keinen Sinn, Referenzen auf Referenzen zu bilden.

Da Pointer und Referenzen beide indirekt auf eine Variable zeigen, ist ein wichtiger Unterschied zu beachten. Wenn Pointer deklariert werden, ist eine gleichzeitige Initialisierung optional, das heißt, nicht erforderlich. Hingegen müssen Referenzen bei der Deklaration gleich initialisiert werden. Damit wird garantiert, dass eine Referenz immer auf die Daten einer Variablen zeigen, und so auf die Daten der Variablen lesend und schreibend zugegriffen werden können. Bei nicht initialisierten Pointern ist das nicht der Fall. Zusätzlich können Pointer im Laufe des Programms auf unterschiedliche Variablen oder Datenelemente zeigen. Eine Referenz kann immer nur auf die Variable zeigen, mit der sie bei der Deklaration initialisiert wurde.

Pointer haben einige verwandte Konzepte. Zum Beispiel kann ein Pointer mit dem Datentyp `void` deklariert werden, was einfach bedeutet, dass er auf Variablen unterschiedlichen Typs oder sogar auf Objekte zeigen kann. Pointern kann darüber hinaus auch der Wert `NULL` zugewiesen werden, was wiederum bedeutet, dass er auf nicht valide Daten zeigt.

Referenzen dagegen haben immer einen festen Datentyp und ein festes Objekt oder eine feste Variable auf die sie sich beziehen. Pointer können natürlich auch mehrere Ebenen der Indirektion aufweisen. Das heißt, dass ein Pointer auf einen Pointer verweisen kann, der dann auf die Daten zeigt. In dem Beispiel wären es zwei Ebenen der Indirektion. Referenzen wiederum haben immer nur eine Ebene der Indirektion. Sie zeigen immer direkt auf die Daten.

Der ganz wesentliche Unterschied besteht aber darin, dass Pointer immer dereferenziert werden müssen, um auf die Daten zuzugreifen. Bei einer Referenz ist das nicht notwendig, da es sich hier einfach um ein Alias handelt, also einen anderen Namen für das Objekt oder die Variable.

Basierend auf diesen Unterschieden könnte man meinen, dass Pointer mächtiger als Referenzen sind. Allerdings Referenzen können sicherer und weniger fehlerträchtig als Pointer sein.

Soweit sieht es so aus, als wären Referenzen nicht besonders nützlich. Lediglich einen anderen Namen für eine Variable zu haben, löst nur selten ein Problem. Allerdings können Referenzen eine wesentliche Rolle bei der Übergabe von Argumenten an eine Funktion haben.

3.9 Parameterübergabe an Funktionen

Parameter lassen sich in C++ an Funktionen wie in C per `textitvalue` oder per `textitpointer` übergeben. Allerdings ist es in C++ zusätzlich möglich, sie per *reference* zu übergeben. Damit stellt sich die Frage, wann ist welche Übergabe am besten? Obwohl es möglich ist, Funktionen wie

```
void abort();
```

zu verwenden, die weder Parameter noch Rückgabewert haben, werden die meisten Funktionen doch einen oder mehrere Parameter aufweisen müssen, um etwas Sinnvolles tun zu können. Zum Beispiel könnte die Funktion

```
int sign( int i);
```

dazu dienen, das Vorzeichen der als Parameter übergebenen Zahl zu ermitteln.

3.9.1 Parameterübergabe per *value*

Bei der Übergabe per *value* wird dieser kopiert. D.h, sieht man sich das folgende Beispiel an, dann wird der Funktionsparameter `i` (bzw. die beim Aufruf aktuell übergebene Variable) per *value* (als Wert) an die Funktion übergeben. Das heisst, dass während des Ablaufs der Funktion `sign()` eine Kopie der Variablen auf dem Stack angelegt wird. Innerhalb der Funktion `sign()` kann deshalb mit dem Funktionsparameter `i` - was auch immer angestellt werden - nach der Beendigung von `sign(a)` kann der Programmierer sich darauf verlassen, dass `a` auf jeden Fall den gleichen Wert wie vorher hat:

Listing 3.11: Übergabe als Wert

```
1 // PassingByValue.cpp : Example for passing by value
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 int sign(int i) {
9     if (i > 0) i = 1;
10    if (i < 0) i = -1;
11    return i;
12 }
13
14 int main()
15 {
16     int a = -5;
17     cout << sign(a) << endl;
18     cout << a << endl;           // a is still -5!
19 }
```

3.9.2 Parameterübergabe per Pointer

Für alle Funktionen gilt, es kann nur einen (Rückgabewert) geben! Dieses - im vorangegangenen Fall durchaus erwünschte - Verhalten führt dann zu Problemen, wenn eine Funktion mehr als einen Wert an das aufrufende Programm zurückgeben soll; es kann schließlich nur einen einzigen Rückgabewert geben! Was ist also zu tun, wenn zum Beispiel eine Funktion `swap()` geschrieben werden soll, die zwei Integerwerte miteinander vertauscht? Die für typische C-Lösung, die auch für C++ gilt, ist, dass nicht die Integerwerte an sich, sondern nur deren Adressen an die Funktion übergeben werden. Die Parameterübergabe erfolgt dann per Pointer:

Listing 3.12: Übergabe per Pointer

```

1 // PassingByPointer.cpp : Example for passing by address
2 //
3
4 #include <iostream>
5
6 void swap(int* i1, int* i2) {
7     int t;
8     t = *i1;
9     *i1 = *i2;
10    *i2 = t;
11 }
12
13
14 int main()
15 {
16     int a, b;
17     a = 1;
18     b = 2;
19
20     std::cout << "Before:_" << a << "_-" << b << std::endl;
21     swap(&a, &b);
22     std::cout << "After:_" << a << "_-" << b << std::endl;
23 }
```

Dieses funktioniert zwar bestens, ist aber wahrscheinlich mit ein Grund dafür, dass sich manch ein lernwilliger C-Anfänger mit Grausen von dieser Sprache abgewendet hat.

3.9.3 Referenzen: Nützlich für viele Zwecke

C++ hat für diese Zwecke nun eine weitere Form der Parameterübergabe, die Übergabe per Referenz eingeführt. Diese wird dadurch kenntlich gemacht, dass dem Funktionsparameter ein `&` vorangestellt wird, welches in diesem Kontext bitte nicht mit dem Adressoperator `&` verwechselt werden darf! Obiges `swap()` Beispiel würde unter Verwendung von Referenzen wie folgt aufgeschrieben werden können:

Listing 3.13: Übergabe per Referenz

```

1 // PassingByReference.cpp : Example for passing by reference
2 //
3
4 #include <iostream>
5
6 void swap(int& i1, int& i2) {
7     int t;
8     t = i1;
9     i1 = i2;
10    i2 = t;
11 }
12
13 int main()
14 {
15     int a, b;
16     a = 1; b = 2;
17
18     std::cout << "Before:_" << a << "_-" << b << std::endl;
19     swap(a, b);
20     std::cout << "After:_" << a << "_-" << b << std::endl;
21
22 }

```

Die ganze Passage ist wesentlich besser lesbar; die Parameterübergabe per Referenz ist bei der Verwendung von C++ in jedem Fall der per Pointer vorzuziehen!

Das in den bisherigen Beispielen vorgestellte trifft auch dann zu, wenn es sich bei den Übergabeparametern statt um einfache Typen (wie `int`, `float` oder `double`) um komplexe Typen wie Objekte handelt:

```

class A {
// ...
};

int count(A a);

```

Hierbei wird ein Objekt vom Typ `A` per Value an die Funktion `count()` übergeben. Im Abschnitt 3.9.1 wurde gesagt, dass bei dieser Form der Übergabe eine Kopie des Parameters auf dem Stack angelegt wird. Während bei einfachen Typen wie `int` der Prozessor weiss, wie er eine Kopie anlegen muss, gestaltet sich das Kopieren bei Objekten wesentlich komplizierter. Hier wird nämlich der Copy-Konstruktor der Klasse `A` aufgerufen!. Ein fehlender oder falsch programmierter Copy-Konstruktor kann in dieser Situation richtig Ärger verursachen. Wenn kein Copy-Konstruktor explizit definiert wurde, dann erzeugt der C++-Compiler zwar selbst einen solchen - dass dieser dann aber immer das Richtige tut, ist nicht selbstverständlich.

Darüber hinaus gilt: Kopieren kostet Zeit! Neben der Notwendigkeit des Vorhandenseins

eines korrekten Copy-Konstruktors kommt bei der Übergabe von Objekten per Value noch ein zweiter Aspekt zum Tragen: Unter Umständen kann ein Objekt sehr viele Daten enthalten - man denke nur an einen Textstring, der dieses Kapitel hier speichert. Diese Daten müssen jedesmal bei einer Wertübergabe kopiert werden - das kostet wertvolle Zeit! Daher bietet sich auch in diesem Falle die Übergabe per Referenz an. Der Tatsache, dass nun ja die Funktion die per Referenz übergebene Variable auch im aufrufenden Kontext ändern könnte, lässt sich durch zusätzliche Verwendung des Schlüsselwortes `const` ein Riegel verschieben (siehe dazu den Abschnitt 2.7, der sich mit Konstanten befasst). Eine Funktionsdeklaration der Form

```
int count( const string& a);
```

spart also das Kopieren des Strings `a` bei jedem Funktionsaufruf, stellt jedoch sicher, dass `a` innerhalb von der Funktion `count()` nicht verändert werden kann.

Dieses Vorgehen lässt sich analog auf die Verwendung von Pointern als Funktionsparameter übertragen, die meisten Stringfunktionen aus der C Standardbibliothek profitieren davon. So wie zum Beispiel die Funktion

```
int strcmp(const char *cs, const char *ct);
```

zum Vergleich zweier Zeichenketten ein (sinnloses) Kopieren von `cs` und `ct` vermeidet, durch die Angabe des Schlüsselwortes `const` jedoch dem Anwender garantiert, dass seine per Pointer übergebenen Variablen durch `strcmp()` nicht angetastet werden.

3.10 C++ Standard Template Library STL - Vector

Bisher hatten wir Arrays kennengelernt, um viele Daten mit immer demselben Datentyp zu verwalten. Allerdings haben Arrays den Nachteil, dass immer entsprechend ihrer festgelegten Größe Speicherplatz statisch alloziert wird. Die Vorteile einer dynamischen Speicherverwaltung gehen damit verloren. Ist also die Größe eines Arrays von vornherein bei der Programmierung schlecht festzulegen, dann sollte man sich überlegen, ob nicht das Vector Template eine gute Alternative darstellt. Vectors sind Datencontainer, deren Größe dynamisch dem Bedarf angepasst wird. Dabei erfolgt die Verwaltung des benötigten Speichers automatisch. Das bedeutet, dass man sich nicht selber darum kümmern muss, Speicher zu allozieren oder wieder freizugeben. Eine Ausnahme gibt es allerdings, falls die Objekte, die in dem Vector gespeichert werden, selber manuell Speicher allozieren, dann ist man natürlich auch selber dafür verantwortlich, diesen freizugeben.

Vector ist dabei eine generische Containerklasse, das heißt, sie kann beliebige Datenelemente desselben Typs speichern, also nicht nur Integerwerte, Gleitkommazahlen oder Strings, sondern auch selber definierte Klassen. Die Vector-Klasse bringt dabei eine Vielzahl von Methoden mit, die es erlauben, den Vector zu manipulieren. Die Vector-Klasse ist Bestandteil der C++ Standard Template Library. Diese umfasst weitere Container-Klassen, Funktionen und Algorithmen, die alle generisch definiert sind, und damit für

3 Strukturen, Klassen und Pointer

Ihre Anwendungen zur Verfügung stehen. Lassen Sie uns mit einem einfachen Beispiel beginnen. In dem Listing 3.14 wird die Vector-Klasse verwendet, um Primzahlen zu speichern. Hier haben wir eine feste Anzahl von Primzahlen, die der Vector aufnehmen soll, aber stellen Sie sich vor, sie wollen alle Primzahlen speichern, die kleiner als 100000 sind. In der Regel wissen Sie nicht, wie viele es sind. Eine ideale Aufgabe für die Vector-Klasse.

Listing 3.14: Beispiel für einen Integer-Vector

```
1 // Vector1.cpp : Simple example of a vector of integers
2 //
3
4
5 #include <vector>
6 #include <iostream>
7 #include <string>
8
9 using namespace std;
10
11 int main() {
12     vector<int> primes;
13     // first the vector is empty
14     cout << "The vector has " << primes.size() << " elements." << endl;
15     primes.push_back(2);
16     primes.push_back(3);
17     primes.push_back(5);
18     primes.push_back(7);
19     primes.push_back(11);
20
21     // now it should have some elements in it
22     cout << "The vector has " << primes.size() << " elements." << endl;
23
24     // accessing the elements by index
25     cout << "The element at index 2 is " << primes[2] << endl;
26     primes[2] = 13;
27     cout << "The element at index 2 is " << primes[2] << endl;
28     cout << endl;
29
30     return (0);
31 }
```

Als erstes müssen wir in Zeile 5 den Header der Vector-Klasse einfügen. In Zeile 12 wird dann ein Vector für Integerwerte deklariert. Erinnern Sie sich daran, dass die Vector-Klasse eine generische Klasse, ein Template, ist. Wir müssen ihr also den benötigten Datentyp mitteilen. Hier sind es Integerwerte. Dieses geschieht zwischen den `<...>` Zeichen. Diesen Vector ordnen wir der Variablen `primes` zu.

```
vector<int> primes;
```

Zunächst ist der Vector `primes` leer, daher fügen wir jetzt die ersten Primzahlen dem

Container hinzu (Zeile 15-19) und überprüfen wieder die Anzahl der Elemente des Vectors. Dazu nutzen wir die Vector-eigene Funktion `size()`, welche die Anzahl der Elemente eines Vectors zurückgibt. Jetzt sollte also die Zahl 5 zurückgegeben werden.

Vectors haben zwei Enden, den Anfang und das Ende. Allerdings kann man immer nur Datenelemente am Ende anfügen. Das ist darin begründet, dass Vectors garantieren, dass die Elemente hintereinander im Speicher abgelegt werden. Ein Einfügen würde dann jedesmal dazu führen, das Speicher neu alloziert oder aber verschoben werden muss. Deswegen kann immer nur am Ende angefügt werden. Daher wird die `push_back()` Funktion verwendet, um an Ende Datenelemente anzufügen.

Wie wir es bereits von Arrays her kennen, können die einzelnen Datenelemente des Vectors auch direkt über einen Index angesprochen werden. So wird beispielsweise in Zeile 25, der dritte Wert des Vectors ausgegeben und in Zeile 26 dann verändert. Die Indizes beginnen bei einem Vector wie bei Arrays mit 0.

In dem Beispiel haben wir gesehen, dass ein Vector gut als Container für Basisdatentypen wie Integer dienen kann. Wie sieht es aber mit komplexen Datentypen wie Objekten aus? In Listing 3.15 deklarieren wir jetzt einen Vector für Objekte der Klasse `dog`, den wir in Zeile 12 `doggies` nennen.

Listing 3.15: Nutzung eines Vectors als Container für Objekte

```

1 // Vector2.cpp : A vector of objects of a new class
2 //
3 #include <vector>
4 #include <iostream>
5 #include <string>
6 // #include "... your path .../dog.h"
7 #include "..\ClassExample4\dog.h"
8
9 using namespace std;
10
11 int main() {
12     vector<dog> doggies;
13     doggies.push_back(dog("Suki", 3, cuddle));
14     doggies.push_back(dog("Hannah", 5, hunting));
15     doggies.push_back(dog("Oscar", 6, protection));
16     doggies.push_back(dog("Jaaber", 7, therapeutic));
17
18     cout << "The first dog is " << doggies.begin()->getName() << endl;
19     cout << "At index 1 we have " << doggies[1].getName() << endl;
20     cout << "Next to last is " << prev(doggies.end(), 2)->getName() << endl;
21     cout << "The last dog is " << (doggies.end() - 1)->getName() << endl;
22     return (0);
23 }
```

Um jetzt den Vector mit einzelnen Objekten zu füllen, nutzen wir den Konstruktor, den wir für die Klasse geschrieben haben. In Zeile 13.-16 werden dann vier Objekte vom Typ

`dog` hinzugefügt. Anschließend sehen wir uns die Namen der hinzugefügten Hunde an. Dazu verwenden wir unterschiedliche Methoden, um darauf zuzugreifen.

Vectors unterstützen Iteratoren, das sind spezielle Objekte, mit denen man durch einen Container navigieren kann. Sie funktionieren ähnlich wie Pointer und folgen einer ähnlichen Semantik. Tatsächlich sind Pointer eine Art von Iterator. In Zeile 18 wird die `begin()` Funktion verwendet, die einen Zeiger auf das erste Element des Vectors zurückgibt. Daher muss hier auch der `->` Operator verwendet werden, um den Pointer auf das erste Objekt zu dereferenzieren und auf die Funktion `getName()` zugreifen zu können.

Umgekehrt, um auf das letzte Element zuzugreifen, wird in Zeile 21 die `end()` Funktion verwendet, die allerdings einen Iterator auf der erste freie Objekt nach dem letzten zurückgibt, daher müssen wir um 1 zurückgehen. Also wird einfach eins von dem zurückgegebenen Pointer abgezogen. Eine andere Möglichkeit wäre die Funktion `prev()` (previous) zu verwenden, die es erlaubt von einer Position einige Schritte zurückzugehen. In diesem Fall nutzen wir sie, um das vorletzte Objekt anzusprechen. Diese Funktion benötigt zwei Argumente, zum einen die Position auf die sie sich bezieht und zum anderen die Anzahl der Schritte, um die sie zurückgehen soll. Um das vorletzte Objekt anzusprechen müssen wir also von `end()` zwei Schritte zurückgehen. Abschließend zeigen wir noch den Namen des zweiten Hundes direkt über seinen Index. Damit sprechen wir das Objekt direkt an und können mittels des Punktoperators `.` direkt auf die Funktion `getName()` zugreifen.

Diese beiden Beispiele zeigen, wie einfach es ist, anstelle von statischen Arrays dynamisch eine unbekannte Anzahl von beliebiger gleichartiger Datentypen zu verwalten und mit ihnen zu arbeiten. Da der Overhead relativ gering ist, der durch die generische Vector-Klasse entsteht, empfiehlt es sich immer gut zu überlegen, ob ein Array wirklich die sinnvollste Lösung für ein Problem darstellt.

3.11 Challenges

3.11.1 Header-Datei für Klasse Vorlesung

Erstellen Sie die Header-Datei für eine Klasse **Vorlesung**, die aus den Datenelementen `id`, `name`, `dozent`, `ect` und `sws` besteht. Deklarieren Sie die Setter- und Getter-Funktionen sowie einen Konstruktor, der alle Felder befüllt.

3.11.2 Implementationsdatei für Klasse Vorlesung

Erstellen Sie die Implementationsdatei, in der die in der vorangegangenen Aufgabe deklarierten Klassenfunktionen der Klasse **Vorlesung** definiert werden.

3.11.3 Vorlesungsübersicht

Schreiben Sie ein Programm, welches die Klasse **Vorlesung** nutzt, um nach und nach Ihre Vorlesungen abzufragen und diese in einem Vector-Container speichert, um dann eine gesammelte Übersicht formatiert auf dem Bildschirm auszugeben.

4 Ablaufkontrolle

4.1 if ... else ... sowie while und do ... while

Die Kontrollstrukturen `if ... else ...` sowie `while` und `do ... while` dienen dazu den Programmablauf in Abhängigkeit von Werten zu verändern. Im wesentlichen funktionieren die in der Programmiersprache C bereits kennengelernten Kontrollstrukturen in C++ genauso, wie wir es gewohnt sind.

```
if (... logischer Ausdruck) {
    Anweisungsblock
}
else {
    Anweisungsblock
}
```

Mit `if ... else` Anweisungen kann man innerhalb des Programms verzweigen, dieses erfolgt in Abhängigkeit von der Auswertung eines logischen Ausdrucks. Gleiches gilt auch für die verschiedenen `while` Schleifen. Auch diese verhalten sich genauso, wie man es von der Programmiersprache C gewohnt ist.

```
do {
    Anweisungsblock
} while (... logischer Ausdruck ...);
```

oder auch

```
while (... logischer Ausdruck ...) {
    Anweisungsblock
}
```

4.2 for ... Schleifen

Die `for` Schleifen wurden ein wenig erweitert, insbesondere was die Nutzung von Container-Klassen wie `vector` betrifft. Eine `for` Schleife kann zwar immer auch als `while` Schleife realisiert werden, ist aber wesentlich kompakter und einfacher zu lesen. Da etwa einzelnen die Datenelemente der `vector` Klasse über den Index angesprochen werden, ist es möglich hiermit über den Inhalte des Vectors zu navigieren, wie in der ersten Schleife in

dem Listing 4.1 dargestellt (Zeile 15f). In diesem Beispiel wird das arithmetische Mittel der Werte des Vectors bestimmt.

Listing 4.1: Beispiele für for Schleifen

```

1 // ForExample.cpp : Example for regular for loop and
2 //                      for loop with range
3
4 #include <iostream>
5 #include <vector>
6
7 using namespace std;
8
9 vector<int> numbers = { 12,25,31,47,58 };
10 float average;
11
12 int main() {
13     cout << "For_loop_with_index" << endl;
14     average = 0.0f;
15     for (int i = 0; i < numbers.size(); i++)
16         average += numbers[i];
17     average /= numbers.size();
18     cout << "Average:" << average << endl << endl;
19
20     cout << "For_loop_with_iterator" << endl;
21     average = 0.0f;
22     for (auto x : numbers)
23         average += x;
24     average /= numbers.size();
25     cout << "Average:" << average << endl;
26
27     cout << "while_loop_with_iterator" << endl;
28     return (0);
29 }
```

Interessanterweise kann ein Vector genauso wie ein Array initialisiert werden, wie in Zeile 9 zu sehen.

Die bekannte Syntax der for Schleife hat drei Teile, die jeweils mit einem Semikolon getrennt werden. Der erste Teil ist das Initialisierungsanweisung, welches zu Beginn der Schleife einmal aufgerufen wird. Dieses wird von der Bedingung gefolgt, die solange sie den Wert `true` hat, weiter durchlaufen wird. Als drittes folgt dann die Inkrementanweisung. In diesem Fall also zuerst die Deklaration und die Initialisierung des iterierenden Laufindex `i`. Im zweiten Teil wird dann festgelegt, dass der Laufindex in der Schleife solange laufen soll, bis von ersten zum letzten Element des Vectors alle ($i = 0 \dots \text{numbers.size()} - 1$) erreicht wurden. Dazu wird der Laufindex im dritten Teil bei jedem Schleifendurchlauf um den Wert 1 erhöht. Der Anweisungsblock der Schleife ist ein Einzeiler. Es wird einfach der Wert des aktuellen Vector-Elements zu der Variablen `average` hinzuaddiert, so dass dieser am Ende der Schleife die Summe aller Vector-

Elemente als Wert hat. Das arithmetische Mittel ergibt sich dann durch Division mit der Anzahl der Vector-Elemente. Dieses wird dann ausgegeben.

Seit der Version C++ 11 lässt der Standard eine weitere Form der `for` Schleife zu. Dabei wird ausgenutzt, dass die Container-Klassen, zu denen auch die Vector-Klasse gehört, eigene Iteratoren besitzen. In diesem Beispiel in Zeile 22f wird zunächst die Variable angegeben, über die iteriert werden soll, hier also die Integerwerte des Vectors. Diese wird hier mit `x` bezeichnet, und damit man sich nicht um den Datentyp kümmern muss, wird das `auto` Schlüsselwort davor gesetzt. Nach einem Doppelpunkt folgt dann mit `numbers` die Reihe der Datenelemente über die gelaufen werden soll. Hier also unser Vector. Man könnte diese Anweisung also lesen als ... **für jedes `x` aus `numbers`.**

Im Anweisungsblock der Schleife findet dann dasselbe wie in der ersten Schleife statt. Es werden alle Werte aufaddiert. Als Ergebnis bekommen wir natürlich in beiden Schleifen dasselbe Ergebnis 34.6.

Vorsicht: Wenn eine Variable desselben Typs wie die Elemente des Vectors genutzt wird, dann wird der Wert jedes Elements im Laufe der Schleifendurchläufe in die Variable kopiert. In diesem Fall ist das problemlos, da es ja nur Integerwerte sind. Sollte man aber einen Vector von Objekten haben, so kann ein solches Vorgehen sehr aufwändig werden und man sollte hier eher eine Referenz auf das Objekt verwenden. Beispiele dafür sehen wir später.

5 Klassendiagramme der Unified Modelling Language UML

Objektorientiertes Programmieren bedeutet das ein Aspekt der Realität in Form von Klassen modelliert wird. Dabei helfen graphische Darstellung oft in der frühen Entwicklungsphase und später auch bei der Dokumentation. Die Klassendiagramme der Unified Modelling Language UML bieten dabei eine weit verbreitete Notation, um objektorientierte Systeme zu konstruieren und zu visualisieren. Dieses Kapitel folgt dabei der sehr gelungenen Beschreibung auf der Seite von Visual Paradigm (<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial>).

Das Klassendiagramm ist dabei ein statisches Strukturdiagramm, das die Struktur eines Modells beschreibt, in dem ihre

- Klassen,
- ihre Attribute (Datenelemente),
- Methoden (Funktionen),
- und die Beziehungen zwischen Objekten

formal beschreiben werden.

5.1 UML Klassennotation

Eine Klasse repräsentiert ein Konzept, dass den Status (Attribute) und das Verhalten (Methoden) kapselt. Dabei hat jedes Attribute einen Datentyp. Nur der Name der Klasse ist verpflichtend. Die weiteren Angaben sind optional, aber oft für das Verständnis notwendig.

Jede Klasse in UML besteht aus einem Rechteck das in drei Abschnitte unterteilt ist (Abb. 5.1). Soll die Beschreibung ausführlich sein, dann werden auch die Datentypen und die Signaturen der Methoden mit spezifiziert.

1. Klassenname
2. Attribute
 - Attribute werden im 2. Abschnitt aufgeführt.
 - Der Datentyp wird nach dem Doppelpunkt genannt.



Abbildung 5.1: Die drei Bereiche der Klasse

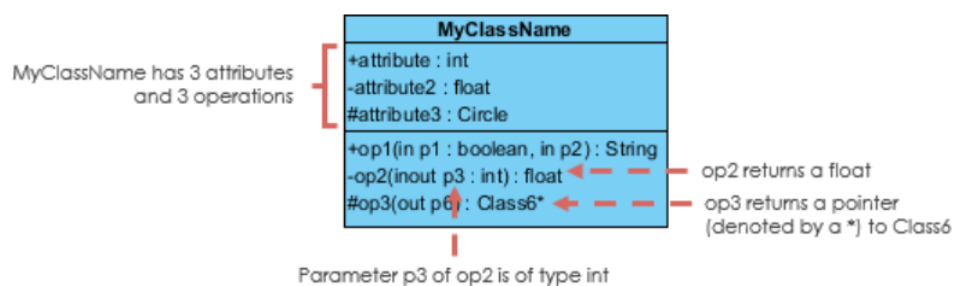


Abbildung 5.2: Angaben der Datentypen der Attribute und der Parameter

- Attribute entsprechen den Namen der Variablen im Programmcode

3. Methoden

- Methoden werden im 3. Abschnitt aufgeführt.
- Der Datentyp des Rückgabewerts der Methode wird nach dem Doppelpunkt genannt.
- Der Datentyp des Parameter der Methode wird nach einen Doppelpunkt auf den Parameternamen genannt.

5.2 Sichtbarkeit

Um die Sichtbarkeit der Attribute und Methoden zu kennzeichnen, können die Zeichen +, - und # verwendet werden, wie in Abb. 5.3 gezeigt.

- + kennzeichnet **public** Attribute oder Methoden
- - kennzeichnet **private** Attribute oder Methoden

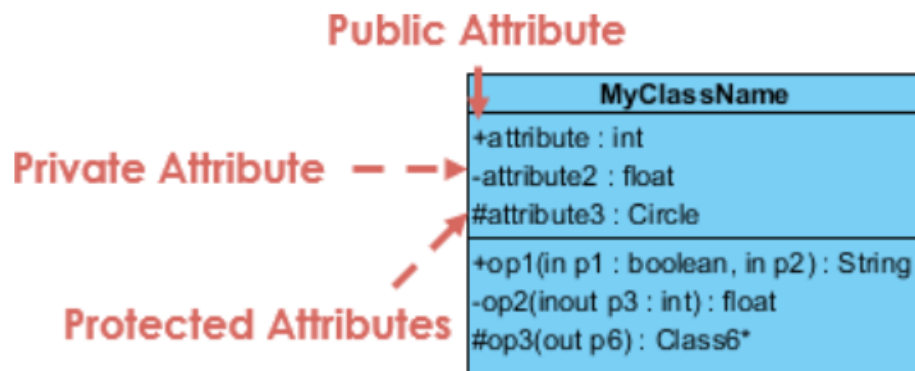


Abbildung 5.3: Angaben zur Sichtbarkeit von Attributen und Methoden

- # kennzeichnet **protected** Attribute oder Methoden

5.3 Richtungsabhängigkeit

In der Signatur der Methoden kann für die einzelnen Parameter angegeben werden, ob diese nur in die aufgerufene Methode gehen - **in**, oder lediglich an die Aufrufer gegeben werden - **out** oder aber in beide Richtungen fließen und gegebenenfalls durch die aufgerufene Methode verändert werden - **inout**. Dieses ist in Abb. 5.4 hervorgehoben.

5.4 Blickweisen auf Klassendiagramme

Je nachdem, wie weit man in dem Entwicklungsprozess sich befindet, kann die Blickweise sehr abstrakt oder auch sehr detailliert sein. Während zu Beginn eher konzeptionell und abstrahiert gearbeitet wird, wird in späteren Phasen dann mehr Wert auf die Spezifikation gelegt bis zum Ende dann spezifische Information für die Implementierung des Klassenmodells in den Vordergrund rücken. Es gibt also im wesentlichen drei unterschiedliche Blickweisen auf Klassendiagramme (Abb. 5.5:

- *Konzeptionell*: Alle wesentlichen Konzepte werden repräsentiert
- *Spezifikation*: Schwerpunkt auf dem Interface der Klassen als abstrakte Datentypen
- *Implementation*: Beschreibt, wie die Interfaces der Klassen implementiert werden

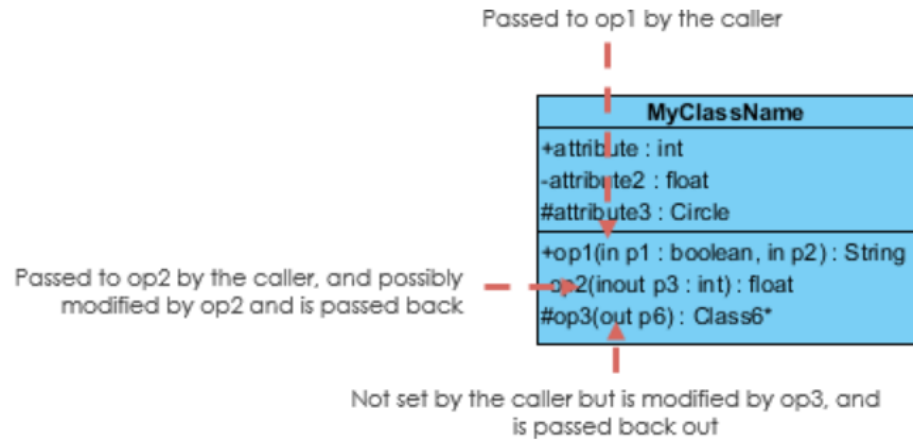


Abbildung 5.4: Angaben zur Richtungsabhängigkeit der Parameter

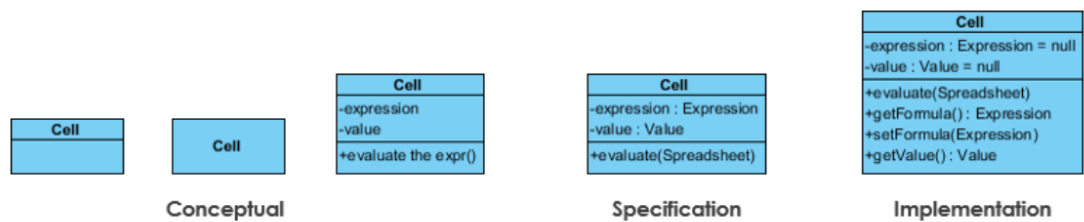


Abbildung 5.5: Blickweisen auf Klassendiagramme

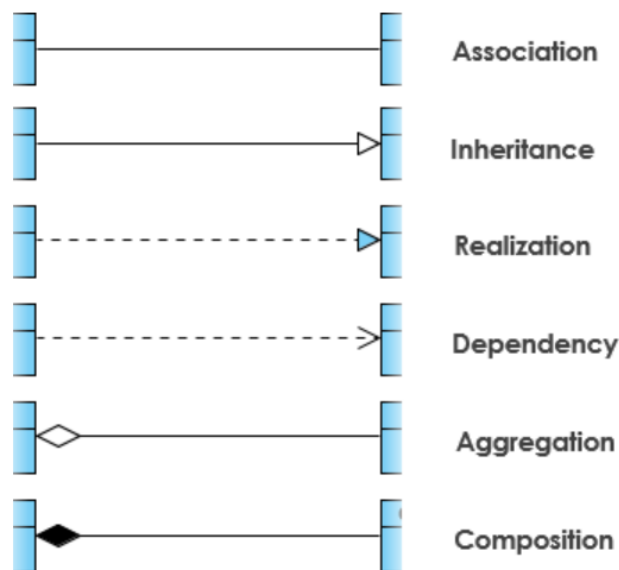


Abbildung 5.6: Beziehungen zwischen Klassen

5.5 Beziehungen zwischen Klassen

UML hat nicht zum Ziel am Ende schöne Bilder zu haben. Wenn es korrekt verwendet wird, können daraus direkt Anleitungen zur Implementierung der Klassen abgeleitet werden. Klassen stehen allerdings nicht alleine für sich da, sondern sind meist in Bezug zueinander zu sehen. Dabei kann eine Klasse durchaus mehrere Beziehungen zu anderen Klassen haben. Die in Abb. 5.6 dargestellten Beziehungen werden dabei unterschieden. Lassen Sie uns näher darauf eingehen.

5.5.1 Vererbung oder Generalisierung

Eine Generalisierung basiert meistens auf einer taxonomischen Beziehung zwischen einer allgemeineren und einer spezifischeren Klasse. Dabei erbt die spezifischere Klasse die Eigenschaften der generalisierten Klasse.

- Wird durch eine *is-a* Beziehung beschrieben.
- Ein abstrakter Klassenname wird oft kursiv dargestellt
- SubClass1 und SubClass2 sind Spezialisierungen der SuperClass

In dem Beispiel in Abb. 5.7 werden zwei Möglichkeiten zur Darstellung von Vererbungshierarchien vorgestellt. Beide sind semantisch äquivalent. Die Beziehung wird durch eine durchgezogene Linie mit einer nicht gefüllten Pfeilspitze dargestellt, die von der Kindklasse zur Elternklasse zeigt.

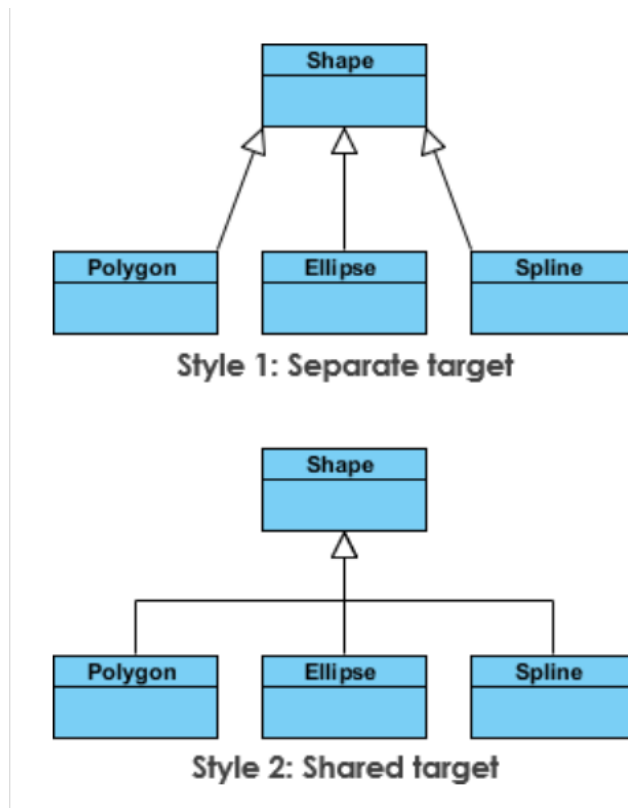


Abbildung 5.7: Vererbung

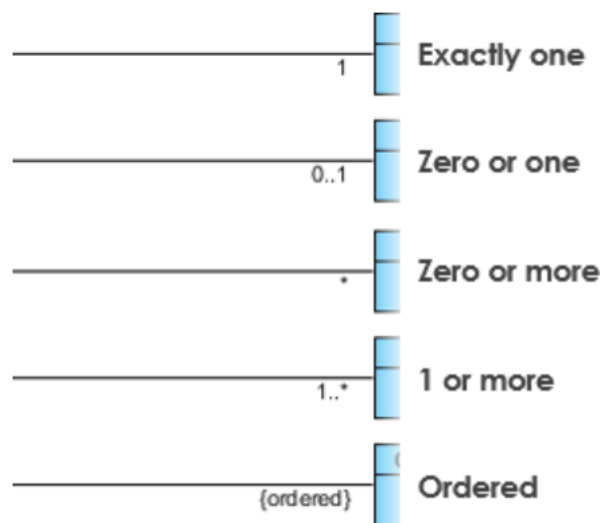


Abbildung 5.8: Darstellung der Kardinalität einer Beziehung

5.5.2 Assoziation

Assoziationen sind Beziehungen zwischen Klassen in einem UML Klassendiagramm. Sie werden durch eine durchgezogene Linie zwischen den Klassen dargestellt. Sie werden dabei häufig mit einem Verb benannt, welches aus der Problemdomäne der realen Welt stammt. Ein Beispiel wäre ist Beziehung *is-married-to*.

5.5.3 Kardinalität

Die Kardinalität einer Beziehung wird meist drei grundlegende Begrifflichkeiten ausgedrückt und wie in Abb. 5.8 durch Beschriftung der Verbindungslinien dargestellt.

- one to one
- one to many
- many to many

5.5.4 Aggregation

Hierbei handelt es sich um eine spezielle Form der Assoziation mit folgenden Eigenschaften:

- Wird durch eine *is-part-of* repräsentiert
- Class2 is part of Class1
- Es kann viele Beziehungen von Class2 zu Class1 geben



Abbildung 5.9: Darstellung Abhängigkeitsbeziehung

- Objekte von Class1 und Class2 haben unterschiedliche Lebensdauern haben

Die Aggregation wird durch eine durchgezogene Linie mit einer offenen Raute auf der Seite der aggregierenden Klasse dargestellt.

5.5.5 Komposition

Die Komposition ist eine Spezialform der Aggregation, bei der alle Teile zerstört werden, wenn das aggregierende Objekt zerstört wird.

- Objekte von Class2 leben und sterben mit den zugehörigen Objekten der Klasse 1
- Objekte von Class2 können nicht unabhängig von Class1 existieren.

In der Darstellung unterscheidet sich die Komposition von der Aggregation nur dadurch, dass jetzt die Raute gefüllt ist.

5.5.6 Abhängigkeit (Dependency)

Ein Objekt einer Klasse mag ein Objekt einer anderen Klasse in einer Methode verwenden. Wenn der Bezug zu dem Objekt aber nicht gespeichert wird, dann wird das in einer Abhängigkeit modelliert.

- Eine spezielle Art der Assoziation
- Existiert zwischen zwei Klassen, wenn Veränderungen der Definition in der einen Klasse zu Veränderungen in der anderen führt (aber nicht umgekehrt).
- Class1 ist von Class2 abhängig

Eine solche Beziehung besteht beispielsweise, wenn die Personenklasse eine `hasRead` Methode mit einem Parameter `Book` besitzt, die `True` oder `False` zurückgibt, wenn die Person das Buch gelesen hat, dieses kann etwa durch Befragen einer Datenbank erfolgen, wie in Abb. 5.9 dargestellt.

5.5.7 Realisation

Die Realisation ist eine Beziehung zwischen einer abstrakten (blueprint) Klasse und weiteren Klassen, die dann diese implementieren. Zum Beispiel könnte eine Klasse `Owner`

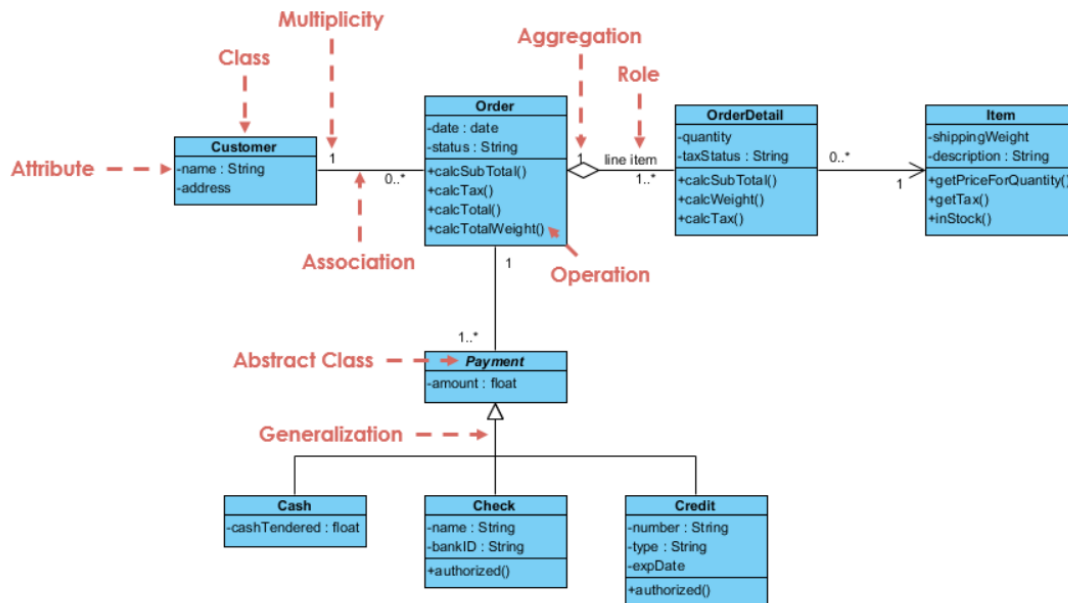


Abbildung 5.10: Klassendiagramm für ein Bestellsystem

die Methoden erwerben() und loswerden() spezifizieren. Die Klassen Person und Korporation als Realisation von Owner müssten dann die Methoden gegebenenfalls auf sehr unterschiedliche Weisen implementieren.

5.5.8 Klassendiagramm für ein Bestellsystem

Versuchen Sie mit den Erläuterungen das Klassendiagramm des Bestellsystems in Abb. 5.10 nachzuvollziehen.

5.6 Challenges

5.6.1 Erstellen eines Klassendiagramms

Analysieren Sie den Programmcode aus drei folgenden Listings 5.1, 5.2 und 5.3

Listing 5.1: records.h

```

1 #pragma once
2
3 #ifndef RECORDS_H_
4 #define RECORDS_H_
5
6 #include <string>
7

```

```

8 class Student {
9 private:
10     int id;
11     std::string fname;
12     std::string lname;
13
14 public:
15     Student(int the_id, std::string the_fname, std::string the_lname);
16     int getID();
17     std::string getName();
18     std::string getShortName();
19 };
20
21 class Course {
22 private:
23     int id;
24     std::string name;
25     unsigned char credits;
26
27 public:
28     Course(int the_id, std::string the_name, unsigned char the_credits);
29     int getID();
30     std::string getName();
31     int getCredits();
32 };
33
34 class Grade {
35 private:
36     int student_id;
37     int course_id;
38     char grade;
39
40 public:
41     Grade(int sid, int cid, char grd);
42     int getStudentID();
43     int getCourseID();
44     char getGrade();
45 };
46
47 #endif

```

Listing 5.2: records.cpp

```

1 #include "records.h"
2
3 using namespace std;
4
5 Student::Student(int the_id, string the_fname, string the_lname) {
6     id = the_id;
7     fname = the_fname;

```

```

8         lname = the_lname;
9     }
10 int Student::getID() {
11     return id;
12 }
13 string Student::getName() {
14     return fname + " " + lname;
15 }
16
17 string Student::getShortName() {
18     return lname + ", " + fname[0];
19 }
20
21 Course::Course(int the_id, string the_name, unsigned char the_credits) {
22     id = the_id;
23     name = the_name;
24     credits = the_credits;
25 }
26 int Course::getID() {
27     return id;
28 }
29 string Course::getName() {
30     return name;
31 }
32 int Course::getCredits() {
33     return credits;
34 }
35
36
37 Grade::Grade(int sid, int cid, char grd) {
38     student_id = sid;
39     course_id = cid;
40     grade = grd;
41 }
42 int Grade::getStudentID() {
43     return student_id;
44 }
45 int Grade::getCourseID() {
46     return course_id;
47 }
48 char Grade::getGrade() {
49     return grade;
50 }

```

Listing 5.3: StudentCourseExam1.cpp

```

1 // StudentCourseExam1.cpp : program with student, course, exams classes
2 //
3
4 #include <iostream>

```

5 Klassendiagramme der Unified Modelling Language UML

```
5 #include <vector>
6 #include "records.h"
7
8 using namespace std;
9
10 // initialize vectors
11
12 vector<Student> students = { Student(1,"George", "Burdell"),
13                             Student(2,"Nancy", "Rhodes") };
14
15 vector<Course> courses = { Course(1,"Algebra",5),
16                             Course(2,"Physics",4),
17                             Course(3,"English",3),
18                             Course(4,"Economics",4) };
19
20 vector<Grade> grades = { Grade(1,1,'B'),
21                           Grade(1,2,'A'),
22                           Grade(1,3,'C'),
23                           Grade(2,1,'A'),
24                           Grade(2,2,'A'),
25                           Grade(2,4,'B') };
26
27 float GPA = 0.0f;
28 int id;
29
30 int main() {
31     string studentName;      // long name of student
32     string studentShortName; // short name of student
33
34     // select a student
35     cout << "Enter a student ID: ";
36     cin >> id;
37
38     // find name of student
39     int i = 0;
40     while (i < students.size() && students[i].getID() != id)
41         i++;
42
43     if (i == students.size()) {
44         cout << "Student not found!" << endl;
45         exit(1);
46     }
47     studentName = students[i].getName();
48     studentShortName = students[i].getShortName();
49
50     cout << "Results for: " << studentName << endl;
51
52     float points = 0.0f;
53     float credits = 0.0f;
```



```

54
55     for (Grade& grd : grades)
56     if (grd.getStudentID() == id) {
57         // float for the numeric grade
58         float num_grd;
59         switch (grd.getGrade()) {
60             case 'A': num_grd = 4.0f;
61                 break;
62             case 'B': num_grd = 3.0f;
63                 break;
64             case 'C': num_grd = 2.0f;
65                 break;
66             case 'D': num_grd = 1.0f;
67                 break;
68             default: num_grd = 0.0f;
69                 break;
70         };
71         // calculate total credits and points
72         int j = 0;
73         while (j < courses.size()
74             && courses[j].getID() != grd.getCourseID())
75             j++;
76         credits += courses[j].getCredits();
77         points += num_grd * courses[j].getCredits();
78
79         cout << studentShortName << ":_ "
80             << courses[j].getName() << "_-"
81             << courses[j].getCredits() << "_-"
82             << grd.getGrade() << "_-"
83             << num_grd << endl;
84
85     }
86     GPA = points / credits;
87
88     cout << "The_GPA_for_" << studentShortName
89         << "_is_" << GPA << endl;
90     return (0);
91 }

```

5.6.2 Lokalisieren des Programmcodes

Passen Sie das Programm so an, dass anstelle der amerikanischen Noten A-F die Noten 1+, 1, 1-, ... verwendet werden.

5.6.3 Modularisierung des Programmcodes

Versuchen Sie die `main()` Funktion zu vereinfachen, indem zusammengehörige Teile des Programms in eigenständige Funktionen ausgelagert werden.

6 Paradigma Objektorientierte Programmierung OOP

6.1 Programmierparadigmen

6.1.1 Funktionale und logische Paradigmen

Mit der objektorientierten Programmierung ist ein neues Paradigma der Programmierung hinzugekommen. So gab es davor primär ein *imperatives oder prozedurales Paradigma* bei dem Funktionen im Vordergrund standen (Tue das!) und mit bedingten Anweisungen und Sprung-Anweisungen Programmteile übersprungen oder wiederholt werden. Bei den prozeduralen Programmiersprachen werden zu lösende Probleme aufgeteilt und als Funktionen (C) oder Prozeduren (Modula, PASCAL) genannt. Weiter Programmierparadigmen sind die

- *Funktionale-Paradigmen*: Diese stellen im Wesentlichen eine Menge von Funktionsdefinitionen in Verbindung mit einem Ausdruck dar. Ein typischer Vertreter ist dabei die Sprache LISP.
- *Logische-Paradigmen*: Basiert auf Fakten und Regeln. Dabei wird eine Problemlösung nicht genau spezifiziert, sondern von einem Interpreterprogramm erstellt. Ein darauf basierender Vertreter ist Prolog.

Funktionale und logische Paradigmen trennen sehr klar die **Welche Aspekte** eines Programms (Programmierer Verantwortung) von den **Wie Aspekten** (Durchführungsbeschlüsse). Dagegen enthalten imperative/prozedurale und auch objektorientierte Programme sowohl die Spezifikation und die Details der Implementierung. Sie sind untrennbar miteinander verbunden.

6.1.2 Objektorientiertes Paradigma

Bei dem objektorientierten Paradigma stehen die Daten und nicht einzelne Funktionen oder Prozeduren des Programms im Vordergrund. Dabei werden die Daten in Objekten gekapselt (Information hiding), die auch über Funktionen verfügen, um auf Daten zuzugreifen oder zu verändern. Allerdings wird hierbei nicht mehr von Funktionen sondern meist von *Methoden* gesprochen. Die Programme werden aus verschiedenen Objekten aufgebaut.

So wurde dann die Sprache C++ hybrid aus dem imperativen, prozeduralen C und objektorientierten Erweiterungen aufgebaut. Weit verbreitete Vertreter der objektorientierten Programmierung sind neben C++ auch Java, Python oder PHP.

6.2 OOP-Grundkonzept

Das Grundkonzept der objektorientierten Programmierung besteht aus den folgenden Elementen:

- Identifikation von Objekten und der anschließenden Implementierung von Objekt-Typen in Klassen
- Zuweisung von Aufgaben zu den Objekten (Methoden)
- Erzeugte Objekte von Klassen kommunizieren mit anderen Objekten durch Senden von Nachrichten über die verfügbaren Methoden der Objekte.
- Die Nachrichten werden von den Methoden des Objekts empfangen und verarbeitet.

Dabei repräsentiert ein Objekt ein Individuum, einen identifizierbaren Artikel, eine Einheit oder einen realen oder abstrakten Rechtsträger, mit einer klar definierten Rolle bei einer Problemdomäne. Beispiele sind:

- *Materielle Dinge*: Auto, Drucker, ...
- *Rollen*: Mitarbeiter, Chef, Teamleiter, ...
- *Vorfälle*: Flugvorfälle, Unfälle, Immatrikulation, ...
- *Interaktionen*: Vertrag, Verkauf, Benotung, ...
- *Technische Daten*: Farbe, Form, ...

Ein Objekt besteht also immer aus zwei Teilen: Den Daten und den Methoden. In anderen Worten: Ein Objekt hat die Verantwortung, zu *wissen* und die Verantwortung zu *tun*. Im Gegensatz dazu obliegen in der Sprache C diese Aufgaben dem Programmierer.

6.3 Warum Objekte

Nun kann man sich fragen, warum ein solches Paradigma benötigt wird. Mit der objektorientierten Programmierung will man folgendes erreichen:

- **Modularisierung** - große Software-Projekte lassen sich in kleinere abgeschlossene Stücke mit definierten Schnittstellen teilen. Diese lassen sich voneinander getrennt entwickeln und testen.
- **- Wiederverwendbarkeit** - Programme können aus bereits geschriebenen Softwarekomponenten (Klassen) zusammengesetzt werden.

- - **Erweiterbarkeit** - Neue Software-Komponenten (Klassen) können geschrieben oder aus bestehenden weiterentwickelt werden. Klassen bekommen weitere Daten oder Methoden, oder neue Klassen werden entwickelt (Vererbung).
- Objektdaten **Abstraktion** bezeichnet den Prozess, eine Darstellung der wesentlichen Attribute oder Merkmale herauszufinden und die unwesentlichen wegzulassen.
- **Datenkapselung** - Daten (Attribute) sind nicht sichtbar aber Methoden zur Manipulation der Attribute sind sichtbar. So kann nichts Unsinniges mit den Daten passieren.
- OOP erfüllt mit der Unterstützung von **Modularisierung, Wiederverwendbarkeit, Erweiterbarkeit, Abstraktion** und **Kapselung** von Objekten die zentralen Anforderung, die immer Komplexere Anwendungslandschaft beherrschbar zu machen.

6.4 Challenges

6.4.1 Modularisieren

Betrachtet man das Programmbeispiel der Challenge 5.6, dann sieht man, dass zwar schon Klassen verwendet werden, allerdings könnte noch viel intensiver modularisiert werden. Dieses erkennt man besonders gut an der Datei 5.3. Modularisieren die Anwendung noch weitergehend indem Sie die TODO Anweisungen in der Datei umsetzen.

Listing 6.1: StudentCourseExam1 mit Anmerkungen

```

1 // StudentCourseExam1.cpp : program with student, course, exams classes
2 //
3
4 #include <iostream>
5 #include <vector>
6 #include "records.h"
7
8 using namespace std;
9
10 // initialize vectors
11
12 // TODO: Neue Klasse StudentRecords bestehend aus den
13 //       vectors: students, courses, grades
14 // TODO: Neue Klasse StudentRecords sollte lokal zu main sein
15 // TODO: Die Initialisierung des Objekts SR sollte eine
16 //       eigenstaenige Funktion sein, spaeter eine Methode
17 //       von StudentRecords
18 // TODO: Die Noten sollten dem deutschen Benotungsschema 1+, 1, 1-, ... folgen
19
20
21 vector<Student> students = { Student(1,"George", "Burdell"),

```

6 Paradigma Objektorientierte Programmierung OOP

```
22         Student(2,"Nancy", "Rhodes") };
23
24 vector<Course> courses = { Course(1,"Algebra",5),
25                           Course(2,"Physics",4),
26                           Course(3,"English",3),
27                           Course(4,"Economics",4) };
28
29 vector<Grade> grades = { Grade(1,1,'B'),
30                          Grade(1,2,'A'),
31                          Grade(1,3,'C'),
32                          Grade(2,1,'A'),
33                          Grade(2,2,'A'),
34                          Grade(2,4,'B') };
35
36 // TODO: Warum global? Werden diese Variablen benoetigt?
37 float GPA = 0.0f;
38 int id;
39
40 int main() {
41     string studentName;      // long name of student
42     string studentShortName; // short name of student
43
44     // select a student
45     cout << "Enter a student ID: ";
46     cin >> id;
47
48     // TODO:
49     // find name of student, sollte eine Methode von StudentRecords
50     // sein. IN: StudentID, OUT: StudentName, or NULL, if not found
51
52     int i = 0;
53     while (i < students.size() && students[i].getID() != id)
54         i++;
55
56     if (i == students.size()) {
57         cout << "Student not found!" << endl;
58         exit(1);
59     }
60     studentName = students[i].getName();
61     studentShortName = students[i].getShortName();
62
63     // TODO:
64     // print a student record. Ausdruck aller Kurse:
65     // Name, credits, grades, numerical grades
66     // and the global point average GPA
67     // Methode von StudentRecords IN: StudentID
68
69     cout << "Results for: " << studentName << endl;
70
```

```

71     float points = 0.0f;
72     float credits = 0.0f;
73
74     // TODO:
75     // Method of StudentRecords for converting any grade into
76     // a numerical grade - float getNumGrade(IN: studentID)
77
78     for (Grade& grd : grades)
79         if (grd.getStudentID() == id) {
80             // float for the numeric grade
81             float num_grd;
82             switch (grd.getGrade()) {
83                 case 'A': num_grd = 4.0f;
84                     break;
85                 case 'B': num_grd = 3.0f;
86                     break;
87                 case 'C': num_grd = 2.0f;
88                     break;
89                 case 'D': num_grd = 1.0f;
90                     break;
91                 default: num_grd = 0.0f;
92                     break;
93             };
94
95             // TODO:
96             // method of StudentRecords to calculate Globabl point average
97             // for a student - float getGPA(in: StudentID)
98
99             int j = 0;
100             while (j < courses.size()
101                 && courses[j].getID() != grd.getCourseID())
102                 j++;
103             credits += courses[j].getCredits();
104             points += num_grd * courses[j].getCredits();
105
106             // Ausdruck Notenuuebersicht
107             cout << studentShortName << ":_ "
108                 << courses[j].getName() << "_-"
109                 << courses[j].getCredits() << "_-"
110                 << grd.getGrade() << "_-"
111                 << num_grd << endl;
112
113         }
114     GPA = points / credits;
115     // Ausdruck GPA
116     cout << "The_GPA_for_" << studentShortName
117         << "_is_" << GPA << endl;
118     return (0);
119 }

```

Listings

1.1	Hello World	7
1.2	Hello World unter Verwendung des Namensraumes <code>std</code>	9
1.3	Verwendung von <code>cout</code>	10
1.4	Verwendung von <code>cin</code>	10
2.1	Umgang mit Variablen	14
2.2	Automatische Variablen mit <code>auto</code>	19
2.3	Einfache Präprozessoranweisungen	20
2.4	Beispiel für die Verwendung von Arrays	25
2.5	Arbeiten mit Strings	26
2.6	Verschiedene Formen der Typkonvertierung	28
3.1	Beispiel für Strukturen	32
3.2	Die erste Klasse	34
3.3	Die erste richtige Klasse mit Kapselung	35
3.4	Erzeugen eines Objekts mit dem Konstruktor	37
3.5	Hello World in Assembler	40
3.6	Header-Datei für die <code>dog</code> Klasse	42
3.7	Implementationsdatei für die <code>dog</code> Klasse	43
3.8	Verwendung der <code>dog</code> Klasse	44
3.9	Einfaches Beispiel für Pointer	45
3.10	Pointer auf Objekte	46
3.11	Übergabe als Wert	50
3.12	Übergabe per Pointer	51
3.13	Übergabe per Referenz	51
3.14	Beispiel für einen Integer-Vector	54
3.15	Nutzung eines Vectors als Container für Objekte	55
4.1	Beispiele für <code>for</code> Schleifen	59
5.1	<code>records.h</code>	69
5.2	<code>records.cpp</code>	70
5.3	<code>StudentCourseExam1.cpp</code>	71
6.1	<code>StudentCourseExam1</code> mit Anmerkungen	77