Karlsruher Institut für Technologie
Institut für Technik der Informationsverarbeitung

C++ Kompendium

Dipl.-Ing. Adnene Gharbi, Dipl.-Ing. Christoph Roth
Dipl.-Ing.(FH) Tobias Schwalb, Dipl.-Ing. Michael Tansella
cand. B.Sc. Martin Pfeiffer, cand. B.Sc. Marcus Müller
2. Auflage überarbeitet von
cand. B.Sc. Andreas Kleff, cand. B.Sc. Felix Mauch
4. Auflage überarbeitet von
Dipl.-Ing. Stephan Werner, Dipl.-Inform. Timo Sandmann

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

KIT - Universität des Landes Baden Württemberg und nationales Forschungszentrum in der Helmholtz-Gemeinschaft

Vorwort

Das vorliegende Skript ist eine Einführung in die Programmiersprache C++. Ziel des Skripts ist es, Ihnen zusammen mit der Einführungsphase den nötigen Stoff komprimiert zu vermitteln, den Sie benötigen, um das IT-Projektpraktikum erfolgreich zu absolvieren. Hierbei wird an vielen Stellen auf eine vollständige Darstellung aller Möglichkeiten von C++ verzichtet und stattdessen auf eine knappe aber einfache und verständliche DarstellungWert gelegt. Es gibt eine große Menge weiterführender Lehrbücher, in denen Sie alle weiteren Details ausführlich dargestellt finden. Das Skript gliedert sich in 9 Kapitel, welche jeweils aus einem möglichst kurzem Theorieteil und mehreren ausführlich erklärten Beispielen bestehen.
Erst durch viel Praxiserfahrung lernt man programmieren, deshalb probieren Sie aus, so viel Sie können. Spielen Sie, schreiben Sie Ihre eigenen Programme und kopieren Sie nicht bloß die Beispiele.

Bevor Sie mit dem Programmieren loslegen, lesen Sie das entsprechende Kapitel in den Programmierrichtlinien. Auf diese Weise müssen Sie sich später nicht mehr umgewöhnen, um sich an die Programmierrichtlinien zu halten.

!!! Besonders wichtige Abschnitte sind durch drei Ausrufezeichen am Rand gekennzeichnet. Hier gibt es immer etwas wichtiges zu beachten.

Verbesserungsvorschläge, sowie jede Art von Kritik zu diesem Dokument, sind sehr erwünscht (harald.bucher@kit.edu, stephan.werner@kit.edu).
An dieser Stelle möchten wir den Herren Martin Pfeiffer und Marcus Müller für ihr aktives Mitwirken bei der Erstellung des vorliegenden Dokuments ganz herzlich danken.
Für die Überarbeitung und Erstellung der 2. Auflage geht unser Dank an Andreas Kleff und Felix Mauch.

Dieses Skript kann nicht alle Themengebiete in voller Tiefe erläutern. Weiterführende Informationen finden Sie im „C++ Primer“, der in 4. Auflage im Addison-Wesley-Verlag erschienen ist. Die ISBN lautet 978-0201721485. Verweise auf den C++ Primer sind wie folgt gekennzeichnet:

Zur weiteren Lektüre	Wie man ein erstes kleines Programm schreibt, finden Sie im C++ Primer Kapitel 1, Seiten 21-25.

Um Ihnen Übungsaufgaben zu bieten, bezieht sich das vorliegende Kompendium auf das Buch „C++ Lernen und professionell anwenden“ von Ulla Kirch-Prinz und Peter Prinz, 4. Auflage, erschienen bei mitp, ISBN 978-3-8266-1764-5. Dieses Buch sollten Sie in hinreichender Stückzahl in der Universitätsbibliothek finden. Beispiel:

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1*, Seite 135

4. Auflage

Inhaltsverzeichnis

1. Erste Schritte & `Hello World!` 5
1.1. Das erste Programm - Hello World! . . . 5
1.2. Erklärung des Programms - Hello World! . . . 5

2. Variablen, Zeiger und Arrays 7
2.1. Grundlegendes zu Variablen . . . 7
2.2. Deklaration und Datentypen . . . 7
2.3. Basisdatentypen . . . 8
2.4. Zuweisung . . . 9
2.5. Zuweisung per Tastatureingabe . . . 9
2.6. Literale . . . 9
2.7. Operatoren . . . 10
2.8. Konstante Variablen . . . 10
2.9. Wert- und Adressvariablen . . . 11
2.10. Referenzen . . . 11
2.11. Arrays . . . 12
2.11.1. Zeiger und Arrays . . . 13
2.11.2. C-Strings . . . 13
2.12. Casting: Umwandeln zwischen Datentypen . . . 14

3. Kontrollstrukturen 15
3.1. Bedingungen und Wahrheitsausdrücke . . . 15
3.2. `if`-Anweisung: Programmverzweigung . . . 15
3.2.1. `if - else if`-Anweisungen . . . 16
3.3. `switch/case`: Mehrfachentscheidungen . . . 17
3.4. Schleifen . . . 18
3.4.1. `while`-Schleife . . . 18
3.4.2. `do...while`-Schleife . . . 19
3.4.3. `for`-Schleife . . . 19

4. Funktionen 20
4.1. Aufruf einer Funktion . . . 20
4.2. Deklaration einer Funktion . . . 20
4.3. Definition einer Funktion . . . 21
4.4. Prototypen, Header-Dateien und Linking . . . 22
4.5. Arten der Parameter-Übergabe . . . 23
4.5.1. Call by Value . . . 23
4.5.2. Call by Reference . . . 23
4.5.3. Call by Pointer . . . 24

5. Erweiterte Datentypen 26
5.1. Enumeration . . . 26
5.2. `struct`: Zusammenfassender Datentyp . . . 26
5.3. Klassen . . . 27
5.3.1. Beispiel für eine Klasse: `string` . . . 29
5.4. Strings . . . 30
5.4.1. Einen String erstellen . . . 30
5.4.2. Einzelne Zeichen in Strings . . . 30

5.4.3. `substr`: Einen Teilstring erstellen . . . 30
5.4.4. zusammenfügen . . . 31
5.4.5. Durchsuchen . . . 31
5.4.6. Ersetzen, löschen, einfügen . . . 32
5.4.7. `c_str`: String in einen C-String umwandeln . . . 32
5.4.8. Strings in Zahlen umwandeln . . . 33
5.4.9. `size`: Länge einer Zeichenkette . . . 34

6. Gültigkeitsbereiche 35
6.1. Globaler Gültigkeitsbereich . . . 35
6.2. Lokaler Gültigkeitsbereich . . . 35
6.3. Gültigkeit in Klassen . . . 37

7. Dynamische Speicherverwaltung 38
7.1. Motivation . . . 38
7.2. `new` und `delete` . . . 38
7.3. Gültigkeitsbereich dynamischer Variablen . . . 39

8. Objektorientierung 41
8.1. Kapselung . . . 41
8.2. Konstruktoren und Destruktoren . . . 42
8.3. Vererbung und Mehrfachvererbung . . . 43
8.3.1. Mehrfachvererbung . . . 44
8.4. Polymorphie und virtuelle Funktionen . . . 44
8.5. `this`-Zeiger . . . 46
8.6. Dateienstruktur . . . 46

9. Streams und Dateiverarbeitung 47
9.1. Streams: Ein- und Ausgabe . . . 47
9.1.1. Der `ostream` . . . 47
9.1.2. `ofstream`: Ausgabe in Datei . . . 47
9.1.3. `istream`: Einlesen von Daten . . . 48
9.1.4. `ifstream`: Einlesen aus Dateien . . . 48
9.1.5. `stringstream`: Die Lösung für Stringumwandlungen . . . 50

10. Standard Template Library 51
10.1. Iteratoren . . . 52
10.2. sequenzielle Container . . . 52
10.2.1. vector . . . 52
10.2.2. deque . . . 53
10.2.3. list . . . 54
10.3. sequenzielle Containeradapter . . . 54
10.3.1. stack . . . 55
10.3.2. queue . . . 55
10.3.3. priority_queue . . . 55
10.4. assoziative Container . . . 55
10.4.1. map . . . 55
10.4.2. set . . . 57
10.4.3. multimap und multiset . . . 57

A. Operatoren 58

B. Escape-Sequenzen 60

C. Dinge von denen nicht die Rede war... 61

Kapitel 1

Erste Schritte & Hello World!

In diesem Kapitel werden Sie ein erstes einfaches C++ Programm erstellen.

1.1. Das erste Programm - Hello World!

Bevor sie Ihr erstes einfaches C++ Programm erstellen sollten Sie einen C++ Editor installiert haben. Näheres dazu finden Sie im Dokument „Erste Schritte“.
Nachdem ein Editor installiert ist, starten Sie Ihren C++ Editor. Nun muss ein neues Projekt oder eine neue Datei angelegt werden. Dies hängt von Ihrer Wahl für einen Editor ab. Befolgen Sie dazu die Hinweise, die Sie ebenfalls im Dokument „Erste Schritte“ nachlesen können.
Geben Sie nun den folgenden Quellcode ein.

Listing 1.1: Erstes C++ Programm

```
1  // Mein erstes C++ Programm
2
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7      /* Die ganze Funktionalität dieses Programms */
8      cout << "Hello World" << endl;
9      return 0;
10 }
```

Nun muss aus dem Code ein ausführbares Programm erstellt werden, dazu muss der Code compiliert werden. Wie dies geht ist ebenfalls im Dokument „Erste Schritte“ nachzulesen. Noch das Programm ausführen und `Hello World` wird im Fenster erscheinen.

1.2. Erklärung des Programms - Hello World!

Zeile 1 `Mein erstes C++ Programm` ist ein Kommentar. Kommentare sind in C++ durch `//` gekennzeichnet. Steht `//` in einer Zeile, wird der folgende Text in dieser Zeile vom Compiler ignoriert. Kommentare dienen dem Programmierer zur Kommentierung und Erklärung seines Programmcodes.

Zeile 3 `#include <iostream>` veranlasst den Präprozessor dazu, die Funktionalität, welche in der Datei *iostream* definiert ist, auch in dieser Datei zur Verfügung zu stellen. Durch Inkludieren der Datei *iostream* kann später das Objekt `cout` verwendet werden. Anweisung, die mit `'#'` beginnen, werden vom Präprozessor verarbeitet.

Zeile 4 `using namespace std;` definiert den Namenbereich `std` als bekannt, somit müssen Objekte und Funktionen aus diesem Namensbereich (z.B. `cout`) nicht mehr mit einem `std::` beginnen.

Zeile 6 und 10 In diesen Zeilen steht das Gerüst der `main`-Funktion. Die `main`-Funktion findet sich in jedem C++ Programm wieder, denn sie ist der Startpunkt, an dem das Programm beginnt. `int main()` sagt dem Compiler, dass der zwischen den folgenden geschwungenen Klammern (Zeile 6 und Zeile 10) stehende Programmcode zur `main`-Funktion gehört.

Zeile 7 Text (auch über mehrere Zeilen), welcher zwischen `/*` und `*/` steht, wird von C++ ebenfalls als Kommentar interpretiert, hier ist `Die ganze Funktionalität dieses Programms` ein Kommentar, der sich auf die folgende Zeile bezieht.

Zeile 8 `cout << "Hello World" << endl;` gibt den Text in Anführungszeichen (also: `Hello World`) gefolgt von einem Zeilenumbruch (`endl`, steht für *End of Line*) auf der Kommandozeile aus. `cout` ist der Ausgabekanal und steht für *character output*. Das Objekt `cout` ist

in der Datei *iostream* (für *input output stream*) definiert und bekommt den Ausgabetext durch `<<` übergeben. Denkt man sich `cout` als virtuelle Kommandozeile, deuten die Pfeile `<<` in Richtung dieser virtuellen Kommandozeile, also Richtung Ausgabe.

Zeile 9 `return 0;` ist immer der letzte Befehl einer `main`-Funktion, denn er gibt eine Zahl (in diesem Fall 0) an das Betriebsystem zurück. `return` steht für die Rückgabe aus der `main`-Funktion. Der Rückgabewert 0 signalisiert dem aufrufenden Programm, dass dieses Programm erfolgreich ausgeführt wurde.

Zusammenfassend gilt:

- Eine Datei mit C++ Code endet mit *.cpp*
- Eine Anweisung endet mit `';'`
- Einen Zeilenkommentar erzeugt man mit `//`
- Text innerhalb von `/*` und `*/` wird ebenfalls als Kommentar betrachtet
- Um das Objekt `cout` zu benutzen, muss man die Datei *iostream* mit `#include` einbinden
- Jedes C++ Programm besitzt eine `main`-Funktion, die mit `int main()` beginnt
- Der Code der main-Funktion ist umrahmt von `'{'` und `'}'`
- Die letzte Anweisung in der main-Funktion ist `return 0;`
- `endl` steht für *End of Line* und bewirkt einen Zeilenumbruch
- `cout` steht für *character output* und gibt Text auf der Kommandozeile aus

Zur weiteren Lektüre	Lesen Sie hierzu im C++ Primer Kapitel 1, Seiten 21-25

Übungsaufgabe C++ Lernen und professionell anwenden	*1.*, *2.* und *3. Aufgabe*, Seite 33

Kapitel 2

Variablen, Zeiger und Arrays

In diesem Kapitel werden Sie mit Variablen, Operatoren, Literalen, Zeigern und Arrays die grundlegenden Elemente einer Programmiersprache kennen lernen.

2.1. Grundlegendes zu Variablen

Variablen sind grundlegende Elemente, die sich in fast jeder Programmiersprache wiederfinden. Da im Grunde jedes Programm eine Eingabe in eine Ausgabe verwandelt, ist eine Repräsentation der eingegeben Daten im Programmcode notwendig. Der Computer muss also praktisch eine Art Gedächtnis haben, in dem er die eingegebenen Daten lagern und eventuelle Zwischenergebnisse speichern kann.
Das ist die Aufgabe der Variablen, sie stellen somit eine Abstrahierung der Register des Prozessors dar.
Aus der Vorlesung kennen Sie den Assembler-Code einer RISC-Maschine zur Addition zweier Werte:

Register bei Assemblerprogrammierung

Listing 2.1: Assemblerprogramm

```
1 MOVE R1, a
2 ADD R1, b
3 MOVE c, R1
```

Hier werden zwei Werte `'a'` und `'b'` addiert, indem zuerst das Register `R1` mit dem Wert `'a'` geladen wird, anschließend der Wert `'b'` hinzu addiert wird und schließlich der Inhalt des Registers in `'c'` gespeichert wird. In C++ lässt sich dies auf eine Zeile reduzieren:

Variablen in C++

```
1 c = a + b;
```

An diesem Beispiel ist schon zu erkennen, dass sich Variablen in C++ nicht so verhalten wie Variablen in der Mathematik. Hier bekommt `'c'` die Summe der Werte `'a'` und `'b'` zugewiesen. Obwohl ein Gleichheitszeichen zwischen `'c'` und `a + b` steht, bedeutet dies nicht, dass beide Werte zwingend äquivalent sind, so wie es in der Mathematik der Fall ist. Weiterhin ist schon zu erkennen, dass das `'='` Zeichen in C++ eine Richtung hat, es weist nämlich immer dem linken Teil, in unserem Fall `'c'`, den Wert des rechten Teils zu, in unserem Fall `a + b`. So ist es auch möglich, einer Variablen das Doppelte ihres eigenen Werts zuzuweisen: `a = a * 2`, `'a'` hat hinterher den doppelten Wert.

`'='` als richtungsabhängiger Zuweisungsoperator

Es sollte nun klar sein, dass Variablen die Werte repräsentieren, die in ihnen gespeichert sind und somit eine einfache Möglichkeit bieten, den Speicher des Computers zu nutzen.

2.2. Deklaration und Datentypen

Bevor man einen Wert in einer Variable speichert, muss man dem Computer zuerst mitteilen, dass man gerne eine Variable in seinen Speicher schreiben würde. Man muss also die Variable bekannt machen, im Fachjargon „deklarieren“, ähnlich wie man mathematische Variablen am Anfang eines Beweises deklariert. In der Mathematik bestimmt man damit die Zugehörigkeit zu einer Zahlenmenge, zum Beispiel zu $\mathbb{N}$, $\mathbb{R}$ oder $\mathbb{C}$. In C++ bestimmt man den Namen der Variable und die Art der in ihr gespeicherten Daten (äquivalent zur Zahlenmenge in der Mathematik). Dies geht zum Beispiel mit folgendem Code:

Deklaration: *Vereinbarung* einer Variable mit Typ

```
1 int a; // Variable a des Typs Integer wird deklariert
```

Hiermit hat man eine Variable mit dem Namen `'a'` deklariert und kann diese nun im Programm verwenden. Vor dem `'a'` steht noch `int`, dies ist der so genannte *Datentyp* der Variable. Er legt fest, welche Art von Werten in der Variable `'a'` gespeichert werden können (hier sind es ganze Zahlen).
Der Datentyp legt strikt fest, welche Daten in einer Variable gespeichert werden. Versucht man, den Wert einer Variable eines anderen Typs als der eigenen Variable zuzuweisen, gibt es einen

Fehler (näheres dazu siehe *Casting: Umwandeln zwischen Datentypen*, Abschnitt 2.12). Diese so genannte *Typsicherheit* verhindert, dass dem Programmierer aus Versehen Fehler bei der Interpretation von Variablen passieren.

!!! Vor der Deklaration darf man in C++ eine Variable nicht verwenden, da der Compiler sonst nicht erkennen kann, welchen Typ sie besitzt, wie viel Speicher sie benötigt und wo sie gültig ist! Oben wurde `'a'` als Variablenname gewählt, es ist aber auch möglich, die Variable mit einem längeren Name zu versehen. Dieser darf jedoch keine Leerzeichen enthalten und bestimmte Sonderzeichen sind ebenfalls verboten. Generell sollte ein guter Programmierer seine Variablen mit aussagekräftigen Namen versehen, damit jemand, der sich in den Code einarbeitet, es einfacher hat, die Funktionsweise zu verstehen.

Programmierertugend: Lesbare Variablennamen

Listing 2.2: Gute Variablennamen, schlechte Variablennamen

```
1 int kein Variablenname;   //FEHLER: nicht zusammengeschrieben
2 int einAussagekreaftigerVariablenname;  //ein langer, ↩
    aussagekräftiger Variablenname
```

Umlaute in Variablennamen sind in C++ nicht erlaubt. Unter verschiedenen Betriebssystemen werden Umlaute häufig unterschiedlich codiert und somit nicht mehr korrekt dargestellt. Deshalb empfiehlt es sich besonders in Dateinamen auf Umlaute zu verzichten. Fehlerhaft dargestellte Umlaute in Kommentaren werden zwar vom Compiler ignoriert, erschweren aber die Lesbarkeit.
Neben `int`, als Datentyp für ganze Zahlen, gibt es jedoch noch einige Datentypen mehr, welche C++ unterstützt. Diese werden als die elementaren Datentypen bezeichnet und später werden Sie lernen, wie man sich aus ihnen seine eigenen komplexeren Datentypen baut (siehe *Erweiterte Datentypen*, Kap. 5).
Es stellt sich nun die Frage, warum man Datentypen überhaupt braucht. In der Tat gibt es Programmiersprachen, die Variablen ohne speziellen Datentyp deklarieren. Ein Datentyp ist allerdings trotzdem nötig, da er die interne Darstellung der Daten festlegt. In der Vorlesung Digitaltechnik haben Sie bereits verschiedene Arten gelernt, wie ganze Zahlen und Fließkommazahlen im Computer dargestellt werden. Diesen Unterschied findet man auch bei den elementaren Datentypen von C++ wieder. Es gibt Datentypen für ganze Zahlen (`int`, `short`, ...) und Datentypen für Fließkommazahlen (`float`, `double`), welche jeweils im Computer ganz unterschiedlich realisiert sind.

2.3. Basisdatentypen

C++ kennt, neben Datentypen für ganze und Fließkommazahlen, auch noch einen Datentyp für boolsche Wahrheitswerte (`bool`) und einen Datentyp, der sowohl für (kleine) Ganzzahlen als auch für ASCII-Zeichen verwendet werden kann (`char`). All diese Datentypen unterscheiden sich sowohl durch die Daten, die sie darstellen können, als auch durch den Speicher, den sie dafür verbrauchen. Je mehr Speicher eine Variable verbraucht, desto größer ist der Zahlenbereich, den sie darstellen kann. Folgende Tabelle gibt eine Übersicht über die elementaren Datentypen in C++[1]:

`Typ`	typischer Wertebereich	typischer Speicherbedarf
`char`	ASCII-Zeichen, Ganzzahlen $0 \leq x \leq 255$	1 Byte $\widehat{=}$ 8 Bit
`short`	Ganzzahlen $-32768 \leq x \leq 32767$	2 Byte
`int`	Ganzzahlen $-2147483648 \leq x \leq 2147483647$	meist 32 Bit $\widehat{=}$ 4 Byte
`long`	Ganzzahlen mindestens `int`	mindestens `int`
`bool`	`true` oder `false`	1 Bit $\leq$ x $\leq$ Speicheradressiereinheit
`float`	Gleitkommazahlen einfacher Genauigkeit	4 Byte
`double`	Gleitkommazahlen doppelter Genauigkeit	8 bis 12 Byte
`string`	Zeichenkette, ASCII-Zeichen, `string` siehe Kap. 5.4	nicht bekannt

!!! Es sei darauf hingewiesen, dass bis auf `char` die elementaren Datentypen keine explizit spezifizierte Speichergröße besitzen. Es ist vielmehr nur garantiert, dass `short` mindestens so groß ist wie `char`, höchstens wie `int`, und dass `long` mindestens das Fassungsvermögen von `int` hat. Der Grund hierfür besteht in den unterschiedlichen Hardware-Architekturen, die von C++-Compilern unterstützt werden. So hat der Datentyp `int` auf einem 16-Bit-System eine Größe von 2 Byte, während er auf einem 32-Bit-System eine Größe von 4 Byte hat.

[1] `string` ist kein Basisdatentyp, wird aber inzwischen in der Regel für Zeichenketten verwendet.

Die ganzzahligen Datentypen `char`, `short`, `int` und `long` kann man mit der Ergänzung `unsigned` versehen. Die `unsigned`-Typen brauchen den selben Speicherplatz, die in ihnen enthaltenen Bits werden aber so interpretiert, dass diese Variablen dann nur den positiven Wertebereich abdecken. `int` ist standardmäßig `signed`[2], deckt so den Wertebereich $-2^{31} \leq$ `int x` $\leq 2^{31} - 1$ ab, während `unsigned int` den Umfang $0 \leq$ `unsigned int y` $\leq 2^{32} - 1$ besitzt.

`unsigned`

Zur weiteren Lektüre	Siehe C++ Primer, Kapitel 2.1, Seiten 58ff

Übungsaufgabe	*Aufgabe 1*, Seite 54
C++ Lernen und professionell anwenden	

2.4. Zuweisung

Wenn nun die Variable deklariert ist, existiert sie für den Compiler und hat einen Namen im Programm. Der logische nächste Schritt ist die Speicherung von Daten in der Variable. Um einer Variablen einen Wert zuzuweisen, verwendet man das `'='` Zeichen.

Zuweisung

Listing 2.3: Grundlegende Verwendungsweise des `'='`

```
1 int a = 1; // a wird bei der Deklaration der Wert 1 zugewiesen
2 a = a + 234; // a wird der Wert 'a + 234' zugewiesen
```

Mit diesem Code wird eine `int`-Variable `'a'` im Speicher angelegt und direkt der Wert `1` in ihr gespeichert. In der zweiten Zeile wird der Wert `1 + 234`, also `235`, in `'a'` abgelegt. Auf der linken Seite des `'='` Zeichens steht immer die Variable, die den Wert der rechten Seite speichert. Hierbei ist es möglich, dass auf der rechten Seite feste Werte (im Beispiel `234`) und/oder andere Variablen (im Beispiel `'a'` selbst) stehen, die untereinander verknüpft sind (im Beispiel mit `'+'`).

2.5. Zuweisung per Tastatureingabe

Möchte man den Inhalt einer Variable mit der Tastatur eingeben, so macht man das mit dem Befehl `cin`. Wie bei `cout` muss man auch hierfür die beiden Zeilen

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

am Beginn der Datei hinzufügen. Nun kann man Variablen folgendermaßen mit `cin` zuweisen:

Listing 2.4: Grundlegende Verwendung von `cin`

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    int zahl;
6
7    cout << "Bitte eine Zahl eingeben: ";
8    cin >> zahl;
9    cout << "Die eingegebene Zahl lautet " << zahl << endl;
10 }
```

Die Ausgabe wäre beispielsweise:

```
Bitte eine Zahl eingeben: 5
Die eingegebene Zahl lautet 5
```

Beachten Sie, dass die >> Pfeile bei `cin` anders herum sind als bei `cout`.

2.6. Literale

Diese festen Werte, welche unveränderlich im Programmcode geschrieben sind (im Fachjargon: *hard coded*), nennt man Literale. Es können Zahlen sein, in dezimaler oder hexadezimaler Schreib-

Literale im Gegensatz zu Eingabedaten

[2] `signed` macht das Gegenteil von `unsigned` und ist für `int` und `long` der Standardzustand, während es dem Compiler überlassen bleibt, `char` und `short` ohne weiteren Vermerk als vorzeichenbehaftet oder -frei zu betrachten.

weise, aber auch, wie im ersten Kapitel, Zeichenketten. Ein weiteres Beispiel zur Verdeutlichung von Literalen:

Listing 2.5: Literale

```
1 cout << "Ich lerne C++" << endl;  // gibt "Ich lerne C++" aus, "Ich lerne C++" ist das Literal
2 int zehnAlsDezimalZahl = 10;      // speichert das Literal 10 in der Variable
3 int zehnAlsHexadezimalZahl = 0xA; // speichert das Literal 0xA in der Variable, 0xA ist 10 in hexadezimaler Schreibweise
```

In Zeichenketten können nicht alle Zeichen verwendet werden, da einige spezielle Bedeutungen aufweisen, siehe hierzu Anhang B.

Zur weiteren Lektüre	Literale sind Gegenstand des C++ Primers, Kapitel 2.2, Seiten 62ff

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 2*, Seite 54

2.7. Operatoren

Operatoren: Verknüpfungen von Variablen

Werden nun Variablen und Literale miteinander verknüpft, kommen Operatoren ins Spiel. Sie kennen bereits '=' und '+', beide sind Operatoren, deren Bedeutung inzwischen klar ist. C++ unterstützt noch eine ganze Menge anderer Operatoren, zum Beispiel zum Vergleichen zweier Werte, welche hier nicht explizit besprochen werden (eine vollständige Liste finden Sie im Anhang A). Wichtig ist, dass Operatoren anhand ihrer Priorität abgearbeitet werden: Zunächst die Operatoren mit der höchsten Priorität, dann absteigend. Bei mehreren Operatoren gleicher Priorität wird grundsätzlich von Links nach Rechts vorgegangen, wobei dieses Verhalten nicht ausdrücklich festgelegt ist.

Priorität

!!! Als generelle Empfehlung gilt: wenn der Ausdruck, den Sie programmieren, zu komplex aussieht, als dass man zweifelsfrei auf den ersten Blick die Abarbeitungsreihenfolge erkennen kann, **verwenden Sie Klammern**. Klammern haben die höchste Priorität, der geklammerte Ausdruck wird immer zuerst abgearbeitet.

Zur weiteren Lektüre	Operatoren werden im C++ Primer unter Ausdrücke, Kapitel 5, speziell Seiten 187-221, behandelt

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1-4*, Seite 112

2.8. Konstante Variablen

Wird bei der Variablendeklaration vor dem Variablentyp das Schlüsselwort `const` gesetzt, so ist diese Variable eine Konstante, das heißt, sie kann nach der ersten Wertzuweisung nicht mehr verändert werden.

Listing 2.6: Benutzen von konstanten Variablen

```
1 const double pi = 3.141592654; // pi ist eine Konstante
2 double radius;
3 double umfang;
4 radius = 4.2;
5 umfang = 2 * pi * radius; //OK
6 radius = umfang / 2 / pi; //OK
7 pi = umfang / 2 / radius; //FEHLER; pi ist konstant und darf nicht neu zugewiesen werden.
```

Da man konstante Variablen später nicht mehr ändern kann, müssen sie direkt bei der Initialisierung einen Wert zugewiesen bekommen.

2.9. Wert- und Adressvariablen

Im Gegensatz zu den bisher bekannten Variablen existieren in C++ noch die so genannten Zeiger- oder Adressvariablen (der englische Begriff *Pointer* findet häufig Verwendung). Diesen werden keine Werte, sondern die Adressen von Speicherzellen bzw. Registern zugewiesen.

Listing 2.7: Wert- und Adressvariablen in der Gegenüberstellung

```
1 float wertVariable;     // Variable
2 float* adressVariable = NULL; // Zeigervariable
```

Die obere Zeile deklariert eine `float`-Variable, in der eine Fließkommazahl gespeichert werden kann. Die untere Zeile dagegen deklariert eine Variable, in der die Speicheradresse einer `float`-Variablen abgelegt werden kann. Somit gibt es zu jedem Datentyp (nicht nur zu den elementaren, auch zu den selbst definierten) einen Datentyp für die jeweiligen Adressvariablen. Dieser Datentyp muss nicht extra definiert werden, sondern ergibt sich automatisch durch Anhängen eines `'*'` an den schon bekannten Datentyp. Die Definition einer Adressvariable erfolgt genauso wie bei einer Wertvariablen mit dem `'='` Operator. Es wird aber eine Adresse gespeichert, welche auch ein Zahlenwert ist.

```
1 int* adressVariable = 0xA1BCF6; // Zuweisung einer beliebigen ↩
    Adresse, dortige Daten unbekannt
```

In der Praxis sollte man einer Adressvariablen als Wert jedoch nie ein Literal zuweisen (wie `0xA1BCF6`
im obigen Beispiel), da die interessanten Speicheradressen normalerweise nicht exakt bekannt sind bzw. von der Speicherverwaltung des Betriebssystems verschleiert werden. Eine Ausnahme bildet etwa die Treiber-Programmierung, bei der durch die direkte Zuweisung von Adressen auf bestimmte Ressourcen der Hardware zugegriffen werden kann. Üblicherweise weist man einem Zeiger aber die Adresse einer Wertvariable zu:

Zeiger als Speicher für Adressen anderer Variablen

Listing 2.8: Speichern einer Adresse in einem Zeiger

```
1 int wertVariable = 0;
2 int* adressVariable = &wertVariable; // adressVariable wird die↩
    Adresse von wertVariable zugewiesen
```

Um die Adresse einer Variable zu bekommen, muss man den Operator `'&'` vor diese Variable setzen. Dann wird (wie im obigen Beispiel) die Adresse der Variable rechts des `'&'` in der Adressvariable gespeichert. Um die Wertvariable zu erhalten, auf welche ein Zeiger verweist, verwendet man den `'*'` Operator (*Dereferenzierung*):

Dereferenzierung

Listing 2.9: Dereferenzierung

```
1 int wertVariable = 1000; // Wert
2 int* adressVariable = &wertVariable; // Adresse
3 int wiederWertVariable = *adressVariable; // Wert
```

Setzt man also ein `'&'` vor eine Wertvariable, kann man diese Kombination als eine Adresse interpretieren. Steht ein `'*'` vor einer Adressvariable, kann man dies zusammen als einen Wert interpretieren.

2.10. Referenzen

Referenzen ermöglichen es, den gleichen Wert durch zwei Namen zu repräsentieren, was vor allem bei Funktionsparametern (siehe Funktionen, Kap. 4) nützlich ist. Bei der Deklaration einer Referenz wird im Gegensatz zur Deklaration einer Wert- oder Adressvariable kein weiterer Speicher belegt. Eine Referenz ist nur ein anderer Name für einen Wert oder eine Adresse, die bereits in einer Variable gespeichert ist.

Referenzen: Aliase für Variablen

Ähnlich wie Zeiger durch ein `'*'` nach dem Datentyp gekennzeichnet werden, wird bei Referenzen ein `'&'` an den Datentyp gehängt. Die Definition der Referenz geschieht ebenfalls mit dem `'='` Operator. Dabei ist zu beachten, dass man Referenzen beim Deklarieren gleich auf eine Variable verweisen muss. Auch können Literale nicht zum Deklarieren einer Referenz verwendet werden, da sie selbst keine Variable darstellen.

Listing 2.10: Erstellen von Referenzen

```
1 int wert = 10; // Wert
2 int& verweis = wert; // verweis ist Referenz für Variable wert
```

Listing 2.11: Fehler beim Erstellen von Referenzen

```
1 int& referenzImmerAufEineVariable; // Fehler! nicht zugewiesen
2 int& eineReferenzReferenziertKeineLiterale = 10; // Zuweisung ↩
    einer Zahl nicht möglich
```

Im oberen Beispiel (Listing 2.10) wird zunächst eine „reguläre“ Wertvariable `wert` deklariert und ihr im Anschluss (aber in der selben Zeile) der Wert `10` zugewiesen. Daraufhin wird eine Referenz `verweis` auf diese Variable erstellt.

Listing 2.12: Verwendung von Referenzen

```
1 wert = wert + 2;
2 verweis = 2 * wert;
```

Da `verweis` und `wert` letztendlich nur zwei unterschiedliche Namen für die gleiche Variable sind, würde eine Ausgabe von den beiden jeweils nach Listing 2.12 `24` liefern.
Abschließend ein Überblick über die verschiedenen Einsatzmöglichkeiten von `'&'` und `'*'`:

`int* zeiger;`	Vereinbart eine Adressvariable (Zeiger, Pointer) `zeiger` für Adressen von Variablen vom Typ `int`
`int& ref = variable;`	Vereinbart `ref` als Referenz (Alias) auf `variable`
`&variable`	Adresse von `variable`
`*zeiger`	Variable, auf die `zeiger` zeigt
`a = b * c;`	`'a'` bekommt das Produkt von `'b'` und `'c'` zugewiesen

In C++ ist die Position des Leerzeichens unerheblich, man kann also beispielsweise genauso gut `int *zeiger;` verwenden. `int* zeiger` macht aber deutlicher, dass es sich bei `zeiger` um eine Adressvariable mit der Adresse einer `int`-Variable handelt.

2.11. Arrays

Arrays (auch Felder genannt) bieten die Möglichkeit, mit einer Zeile gleich eine Vielzahl von Variablen eines Datentyps anzulegen, welche sich über einen Index unterscheiden lassen.

```
1 int zahlenliste[3]; // drei Variablen des Typs Integer
```

Mit dieser Zeile werden gleich 3 `int`-Variablen im Speicher reserviert, in Speicherbereichen direkt hintereinander. Die Zahl in den eckigen Klammern gibt die Anzahl der zu erzeugenden Variablen an und kann auch eine ganzzahlige Variable sein.

Zugriff über Indizes

Um nun einen Wert in einer der drei Variablen zu speichern, muss man einen Index angeben, wobei dieser immer bei 0 anfängt. Im Beispiel hat also die dritte Komponente den Index 2:

```
2 zahlenliste[2] = 234567; // dritte Integer-Variable = 234567
```

Mit dieser Zeile wird `234567` in der „dritten Komponente“ des Arrays abgelegt, `zahlenliste[2]` ist somit der Name einer Variablen. Der Index in den eckigen Klammern kann auch durch eine Variable mit ganzen Zahlen dargestellt werden:

Listing 2.13: Variablen für Indizes bei Arrays

```
1 short index = 0;
2 int zahlenliste[3];
3 zahlenliste[index] = 1234; // erste Variable = 1234, Index = 0
4 index = 2;
5 zahlenliste[index] = 1234; // dritte Variable = 1234, Index = 2
6 // zweite Variable undefiniert, Wert unbekannt
```

Mit diesem Code wird ein Array mit 3 `int`-Variablen angelegt und unter Verwendung einer Index-Variable `index` in der ersten und dritten Variable des Arrays der gleiche Wert `1234` gespeichert.

Übungsaufgabe *Aufgabe 1*, Seite 355
C++ Lernen und professionell anwenden

2.11.1. Zeiger und Arrays

Arrayname: Zeiger auf das erste Element

Zeiger und Arrays sind in C++ sehr eng miteinander verwandt, wie man im folgenden Beispiel sieht:

Listing 2.14: Zusammenhang zwischen Arrays und Zeigern

```
1 int liste[3] = { 1, 2, 3 };
2 int* arrayZeiger = &liste[0]; //alternativ:  = liste;
3
4 cout << *( arrayZeiger + 2 ) << " ist gleich mit "
5      << liste[2] << endl;
```

Dieser Code gibt `3 ist gleich mit 3` aus, denn `liste[2]` ist nur eine praktischere Schreibweise für `*( arrayZeiger + 2 )`. Es gilt also: der Arrayname ist ein Zeiger auf das erste Element des Arrays! Der Index für den Zugriff auf ein bestimmtes Element eines Arrays kann als Offset zu dieser Startadresse interpretiert werden.
Bei letzterem wird der Zeiger auf das erste Element des Arrays um 2 Elemente weiter gezählt, also zeigt `( arrayZeiger + 2 )` letztendlich auf das dritte Element des Arrays.

Zeigerarithmetik

`arrayZeiger` ist letztlich nur eine Adressvariable mit der Adresse einer `int`-Variable, doch C++ nimmt die Zugehörigkeit der `int`-Variable zu einem Array an. Genau genommen *geht* C++ *einfach davon aus*, dass ein Zeiger, den der Programmierer erhöht, auf ein Element eines Arrays zeigt. Eine Garantie, dass Sie beim Hochzählen nicht hinter das Ende Ihres Arrays geraten, gibt es nicht, dann entsteht entweder eine *Speicherschutzverletzung* oder Sie lesen ungültige Daten ein. Achten Sie also unbedingt darauf, dass Sie, egal ob Sie zum Zugriff `[ ]` oder Zeigerarithmetik verwenden, niemals ein Element zu benutzen versuchen, das außerhalb des Arrays liegt! Somit wird `arrayZeiger + 2` als die Adresse des Nachnachfolgerelements interpretiert. Diese Eigenschaft von Arrays wird besonders in Verbindung mit Funktionen interessant, welche Sie in einem späteren Kapitel kennen lernen.

Übungsaufgabe *Aufgabe 1-2, 4*, Seite 389
C++ Lernen und professionell anwenden

2.11.2. C-Strings

Sie kennen bereits `char` als Datentyp, der sich zur Darstellung eines Buchstaben oder sonstigen Zeichens benutzen lässt. Dementsprechend erscheint es logisch, eine *Zeichenkette* (engl. *String*, „Kette“) als ein Array von `char`s zu realisieren.
Alle Zeichenkettenliterale (`char* zeichenkette = "abc";`) sind so genannte *C-Strings*[3]. Dabei handelt es sich um nichts anderes als Arrays von `char`-Variablen, wobei das letzte Zeichen der Zeichenkette **unbedingt** von einem `char` mit dem Wert `0` gefolgt werden muss. Dies liegt daran, dass in C++ die Länge eines Arrays nicht bekannt ist; die Funktionen, die mit C-Strings hantieren, sind daher darauf angewiesen, dass ihnen durch ein 0-Byte (siehe *Escape-Sequenzen*, Anhang B) das Ende der Zeichenkette angezeigt wird.
Es ist daher notwendig, wenn man selbst `char`-Arrays anlegt, darauf zu achten, dass die Länge des Arrays um mindestens ein `char` größer ist, als die längste darin zu speichernde Zeichenkette. Passiert dies nicht, so versucht das Programm etwa bei der Ausgabe über das Ende der Zeichenkette hinaus zu lesen und gibt dann den folgenden Speicherinhalt aus, bis es auf ein Nullbyte trifft. Umgekehrt ist es auch möglich, dass durch das Einlesen einer Zeichenkette in ein Array, das hierfür zu klein ist, der Speicherinhalt, der „hinter“ dem Array liegt, von der Zeichenkette überschrieben wird. Diesen Vorgang nennt man dann einen *Buffer Overflow*, und stellt ein ernst zu nehmendes Sicherheitsrisiko dar, da so ein Benutzer des Programms interne Variablen, Sprunganweisungen oder sogar Programmcode verändern kann.

Buffer Overflow

[3] Der Name rührt daher, dass die C-Strings in C, der Grundlage von C++, in dieser Form eingeführt wurden

2.12. Casting: Umwandeln zwischen Datentypen

Möchte man den Wert einer Variable in einer anderen Variable eines unterschiedlichen Typs speichern, wird die Variable dem *Casting* unterzogen. Das kann implizit geschehen, wenn es sich bei den Variablen um Basisdatentypen handelt und bei der Umwandlung keine Genauigkeit verloren geht, oder explizit.
Beim expliziten Casting wird der Zielvariablentyp in Klammern vor die umzuwandelnde Variable geschrieben. Explizites Casting setzt voraus, dass definiert ist, wie der Typ umgewandelt wird. Dies ist aber für die meisten Basisdatentypen der Fall. Für eigene Datentypen (Abschnitt 5.3) können mittels Operatorenüberladung (siehe C) eigene Methoden definiert werden, die sich um die Umwandlung kümmern.

Listing 2.15: Casting

```
1 int ganz = 10;
2 double* zeiger;
3 double fliesskomma;
4 fliesskomma = ganz; // implizites Casting, da double eine ↩
    höhere Genauigkeit als int hat
5 ganz = fliesskomma; // WARNUNG, es geht Genauigkeit verloren
6 zeiger = fliesskomma; // FEHLER, int* und int sind nicht ↩
    kompatibel
7 ganz = ( int ) fliesskomma; // OK, explizites Casting
```

Zur weiteren Lektüre	Mehr Informationen erhalten Sie im C++ Primer, Kapitel 2

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1, 2, 4*, Seite 169

Kapitel 3

Kontrollstrukturen

Ein Programm muss unter bestimmten Umständen Entscheidungen treffen und dementsprechend Anweisungen ausführen oder nicht. Oft sind auch Anweisungen mehrfach hintereinander auszuführen, zum Beispiel solange eine Bedingung erfüllt ist. In diesem Kapitel lernen Sie die Kontrollstrukturen zur Formulierung solcher Probleme kennen.

Zur weiteren Lektüre	Eine ausführlichere Behandlung findet im C++ Primer statt, Kapitel 6 ab Seite 237

3.1. Bedingungen und Wahrheitsausdrücke

Kontrollstrukturen sind immer mit einer Bedingung verbunden und C++ bietet Operatoren um diese zu formulieren.
Die Bedingung *„Ist a mindestens so groß wie b?“* lautet in C++ `a >= b`
Der Wert einer solchen Bedingung ist ein Wahrheitswert, der in einer Variable des Typs `bool` (siehe 2.3) gespeichert werden kann. `>=` ist der Vergleichsoperator (*„größer gleich“*) der Bedingung, welcher seinen linken Operanden `'a'` mit dem rechten Operanden `'b'` vergleicht. Entsprechen die Werte der Operanden dem Vergleichskriterium, liefert der Operator `true`, andernfalls `false` als Wahrheitswert.
Wahrheitsausdrücke können mithilfe der logischen Operatoren UND (`&&`), ODER (`||`) und NICHT (`!`) beliebig verkettet werden. Hierbei ist das Setzen von Klammern zur Verbesserung der Lesbarkeit und Veränderung der Operatorenreihenfolge ratsam.

```
1 ( a > 0 && b > 0 ) || ( a < 0 && b < 0 )
```

Diese Bedingung ist eine Abfrage, ob beide Zahlen `'a'` und `'b'` das gleiche Vorzeichen haben. Die Klammern verbessern die Lesbarkeit, sind jedoch nicht zwingend notwendig, da in C++ der `&&`-Operator (logisches UND) eine höhere Priorität hat als der `||`-Operator (logisches ODER).

Äquivalenzoperator `==`

Eine große Stolperfalle stellt immer wieder der Äquivalenzoperator `==` dar, der den bitweisen Vergleich von Werten erlaubt. Häufig wird statt diesem nämlich der Zuweisungsoperator `=` verwendet, der allerdings, sofern die Zuweisung des rechten Wertes zum linken Operanden möglich ist, den zugewiesenen Wert zurück liefert.
Oft wird statt dem Vergleichsoperator `a == b` der Zuweisungsoperator `a = b` eingesetzt. Dies führt zu Fehlern im Programmablaufs und stellt eine häufige Fehlerquelle dar.

!!!

Zahlen als Wahrheitswerte

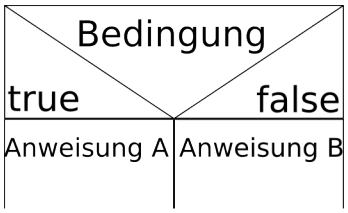
C++ interpretiert auch Zahlenwerte als Wahrheitswerte, wobei 0 für `false` steht und alle Zahlen $\neq 0$ als `true` interpretiert werden.

```
1 ( 0 == true ) || a
```

Dieser Wahrheitsausdruck ist `true`, wenn `'a'` ungleich 0 ist. `( 0 == true )` ist immer `false` und der logische ODER-Operator ergibt nur `true`, wenn einer der beiden Operanden `true` ist.

3.2. `if`-Anweisung: Programmverzweigung

Bedingungen können im Rahmen einer Programmverzweigung genutzt werden, wo sie festlegen, welche Anweisungen als nächstes ausgeführt werden. In Abb. 3.1 ist eine Programmverzweigung sowohl grafisch, als auch in C++-Syntax abgebildet.
Ist der Wahrheitswert von `Bedingung` `true` wird `AnweisungA` ausgeführt, in jedem anderen Fall `AnweisungB`. Die Verwendung des `else`-Blocks ist optional, findet der Compiler nach `AnweisungA` kein `else`, so wird im `false`-Fall einfach fortgefahren.



Nassi-Shneiderman-Diagramm: Verzweigung

```
1 if( Bedingung ) {
2     AnweisungA;
3 } else {
4     AnweisungB;
5 }
```

C++ Syntax: `if`-Anweisung

Abbildung 3.1.: Nassi-Shneiderman und C++ im Vergleich

Listing 3.1: `if`: Beispiel

```
1  if( a < b ) { // wenn a kleiner b
2    c = b - a;
3    a -= 1;
4  } else { // wenn a größer gleich b
5    c = a - b;
6  }
7
8  if( c == 2 ) { // wenn c gleich 2 ist
9    c = 0;
10 }
11 c = c / 2; // wird immer ausgeführt
```

Dieser Code wird wie folgt ausgeführt:

1. Es wird überprüft, ob `a < b` zutrifft. Ist dies der Fall, so wird `c = b - a` (Zeile 2) gesetzt und 1 von `'a'` abgezogen, andernfalls wird `c = a - b` in Zeile 5 ausgeführt.
2. Danach wird überprüft, ob `'c'` den Wert 2 hat (Zeile 8). Ist dies der Fall, so wird es auf 0 (Zeile 9) gesetzt. Trifft dies nicht zu, wird direkt fortgefahren in Zeile 11.
3. Auf jeden Fall wird im letzten Schritt `'c'` halbiert (Zeile 11).

Sollen mehrere Anweisungen unter einer Bedingung ausgeführt werden, so werden wie in Zeile 1 und 4 die Operatoren `'{'` und `'}'` um den Anweisungsblock gesetzt. Es ist aus Gründen der Lesbarkeit üblich, die geschweiften Klammern auch dann zu schreiben, wenn tatsächlich nur eine einzelne Anweisung ausgeführt wird.

3.2.1. `if - else if`-Anweisungen

Da `if` selbst eine Anweisung ist, kann es auch den Platz der einzigen Anweisung einnehmen, die im `else`-Zweig steht. Damit können im Falle, dass die erste Überprüfung nicht zutraf, weitere Fälle getestet werden.

Listing 3.2: `if - else if`: Beispiel

```
1  string aufgabe;
2  int wassertemperatur, windstaerke;
3  // ...
4  if( aufgabe == "arbeiten" ) {
5    calculateLikeCrazy(); // diese Anweisung tut einfach ↩
       irgendetwas, siehe Funktionen
6  } else if( wassertemperatur > 14 && windstaerke <= 4 ) {
7    badehoseEinpacken();
8    rausNachWannsee(); // Freizeit!
9  } else if( windstaerke > 4 && windstaerke < 8 ) { // Wenn ↩
     Windstärke zwischen 4 und 8 ist und keiner der vorherigen ↩
     Fälle zutrifft
10   drachenSteigenLassen();
```

```
11 } else { // Wenn keine vorherige Bedingung erfüllt ist
12   cplusplusKompendiumNehmen();
13   lesen();
14 }
```

Dieser Code wird wie folgt ausgeführt:

1. Überprüfung, ob die Zeichenkette `aufgabe` gleich „`arbeiten`“ ist (näheres zu *Strings*, siehe 5.4). Ist dies der Fall, so wird die Anweisung `calculateLikeCrazy()` ausgeführt.
2. Andernfalls wird überprüft, ob Badebedingungen gegeben sind: ist die `wassertemperatur` höher als 14 und der Wind nicht zu stark, wird `badehoseEinpacken()` und danach `rausNachWannsee()` ausgeführt.
3. Stimmen die Bedingungen zum Badengehen ebenfalls nicht, wird schließlich geprüft, ob Drachensteigen möglich ist.
4. Wenn keine andere Beschäftigung möglich ist, wird im letzten `else`-Block das C++ Kompendium zur Hand genommen.

In einer `if - else if`-Anweisung werden also der Reihe nach Bedingungen überprüft. Trifft eine der Bedingungen zu, werden die nachfolgenden Bedingungen nicht mehr beachtet.

3.3. `switch/case`: Mehrfachentscheidungen

Es sind viele Fälle vorstellbar, in denen eine einzelne Variable gegen eine Reihe fester Werte verglichen werden und anhand ihres Wertes eine bestimmte Anweisung (bzw. ein Anweisungsblock) ausgeführt werden soll. Um sich viele kaskadierte `if - else if`-Blöcke zu ersparen, kann der C++ Programmierer in solchen Fällen auf das `switch`-Konstrukt zurückgreifen. Die folgende Syntax überprüft, ob `variable` äquivalent zu vier konstanten Werten ist:

Listing 3.3: Beispiel zur `switch/case`-Syntax

```
1  switch( variable ) {
2    case konstanterWert1:
3      Anweisung1;
4      break;
5    case konstanterWert2:
6      Anweisung2;
7      Anweisung2a;
8    case konstanterWert3:
9    case konstanterWert4:
10     Anweisung34;
11     break;
12   default:
13     AnweisungDefault;
14     break;
15 }
```

Bei der Verwendung von `switch` ist folgendes zu beachten:

- Die zu überprüfende Variable muss von einem ganzzahligen Variablentyp sein.
- `case` akzeptiert nur Literale (siehe 2.6), die Variable kann also nur gegen fest vorgegebene Werte verglichen werden, nicht jedoch gegen andere Variablen.
- Die verschiedenen `case`-Fälle werden der Reihe nach überprüft. Die Ausführung endet erst, wenn ein `break` als Anweisung auftritt. Im obigen Syntaxbeispiel wird die Ausführung nach `Anweisung1` und `Anweisung34` abgebrochen, nicht jedoch nach `Anweisung2a`.
- Soll für mehrere Werte die gleiche Anweisung ausgeführt werden, ergibt sich eine Syntax wie im Beispiel bei `konstanterWert3` und `konstanterWert4`.
- `default` wird in dem Fall ausgeführt, dass *keiner* der `case`s zutrifft. Allerdings kann die Ausführung aus dem vorherigen `case` in `default` laufen, wenn kein `break` dies verhindert.

Listing 3.4: Beispiel für `switch/case`

```
1  switch( machineState ) {
2   case 1:   // Soll-Zustand > weiter arbeiten
3      continueWorking();
4      break;
5   case 0:   // Stillstand > starten und kalibrieren
6      startWorking();
7   case 2:   // unkalibriert > kalibrieren
8      calibrate();
9      break;
10  case -1:  // Not-Aus! > Stop und Service
11     stopWorking();
12  default:  // immer sonst > Service
13     callService();
14     break;
15 }
```

Im Beispiel 3.4 werden abhängig vom Wert der Variablen `machineState` verschiedene Anweisungen ausgeführt:

- Ist der Zustand 1, so wird `continueWorking()` aufgerufen und dann durch die `break`-Anweisung der `switch`-Block verlassen (Sprung in Zeile 15).
- Ist der Zustand 0 (Stillstand) wird `startWorking()` aufgerufen. Dann wird auch `calibrate()` ausgeführt, bis schließlich mit `break` die `switch`-Anweisung verlassen wird.
- Läuft die Maschine schon, ist jedoch nicht kalibriert (`machineState == 2`) springt `switch()` direkt in Zeile 8.
- Bei einem Not-Aus (`machineState == -1`) wird die Maschine gestoppt (Zeile 11), dann der Service angerufen (Zeile 13).
- Der `default`-Fall tritt bei allen Zuständen auf, die nicht vorher durch einen `case` abgefangen werden und ist hier ein Fehlerfall.

3.4. Schleifen

Schleifen wiederholen eine oder mehrere Anweisungen, solange eine Bedingung gilt oder eine vorgegebene Anzahl von Wiederholungen noch nicht erreicht ist.

3.4.1. `while`-Schleife

Die `while`-Schleife (Syntax 3.5) wiederholt ihren Anweisungsinhalt solange, wie die ihr gegebene Bedingung wahr ist. Die Überprüfung findet jedes mal **vor** dem potentiellen Ausführen des Anweisungsblocks statt. Ist die Bedingung nicht (mehr) erfüllt, so wird die Ausführung sofort hinter der `while`-Schleife die Programmausführung fortgesetzt.

Listing 3.5: Syntax der `while`-Schleife

```
1 while( Bedingung ) {
2   Anweisungsblock;
3 }
```

Listing 3.6: Beispiel zur Verwendung der `while`-Schleife

```
1 // Berechnung der Fibonacci-Zahlen kleiner 1000
2 #include <iostream>
3 using namespace std;
4 int main() {
5   int fibonacci1 = 1, fibonacci2 = 1;
6   while( fibonacci1 < 1000 ) {
7     // "alte" Fibonacci-Zahl ausgeben
8     cout << fibonacci2 << ", ";
```

```
9       int tmp = fibonacci2;
10      fibonacci2 = fibonacci1;
11      fibonacci1 = fibonacci1 + tmp;
12    }
13    return 0;
14  }
```

Die Schleife in diesem Beispiel läuft solange, bis die Variable `fibonacci1` nicht mehr `< 1000` ist. Bei jedem Schleifendurchlauf wird zuerst die alte Zahl ausgegeben und dann die beiden neuen Zahlen gebildet.
Die Ausgabe lautet `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,` .

3.4.2. `do...while`-Schleife

Die `do...while`-Schleife funktioniert analog zur `while`-Schleife mit dem einzigen Unterschied, dass stets die Überprüfung der Bedingung **nach** dem Schleifendurchlauf geschieht. Somit wird die `do...while`-Schleife auf jeden Fall einmal ausgeführt, auch wenn ihre Bedingung schon zum Anfangszeitpunkt `false` war.

Listing 3.7: Syntax der `do...while`-Schleife

```
1 do {
2   Anweisungsblock;
3 } while( Bedingung );
```

3.4.3. `for`-Schleife

Die `for`-Schleife unterscheidet sich von der `while`-Schleife insofern, dass in ihrem Kopf neben einer Schleifenbedingung auch eine Initialisierung und eine Iterationsvorschrift angegeben wird.

Listing 3.8: Syntax der `for`-Schleife

```
1 for( Initialisierung; Bedingung; Iterationsvorschrift ) {
2   Anweisungsblock;
3 }
```

Die `for`-Schleife wird daher gerne für Zählvorgänge genutzt, indem als Iterationsvorschrift eine einfache Variablenerhöhung mittels Prä- oder Postfix `++` (siehe Formelsammlung) vorgenommen wird. Mit Hilfe des Kommaoperators ist es möglich mehrere Initialisierungen oder Iterationsvorschriften in einem Schleifenrumpf durchzuführen.

Listing 3.9: Beispiel zur `for`-Schleife

```
1  for( int i = 0, int sum = 0; i < 10; i++ ) {
2    cout << i << ", ";
3    sum += i;
4  }
5  cout << endl;
6
7  int zaehler = 0;
8  int sum = 0;
9  while( zaehler < 10 ) {
10   cout << zaehler << ", ";
11   sum += zaehler;
12   zaehler++;
13 }
```

Im Beispiel 3.9 wird beide Male `0, 1, 2, 3, 4, 5, 6, 7, 8, 9,` ausgegeben und die Summe der Zahlen berechnet, jedoch einmal formuliert als `for-` und einmal als `while`-Schleife.

Übungsaufgabe *Aufgabe 2, 3*, Seite 135
C++ Lernen und professionell anwenden

Kapitel 4

Funktionen

Paradigma

Dem *prozeduralen Programmierparadigma* folgend, werden Problemstellungen bei der Softwareentwicklung in kleinere Funktionseinheiten zerlegt. Viele Algorithmen basieren darauf, dass der gleiche Programmablauf mit anderen Daten mehrmals wiederholt wird. Hierbei orientiert man sich am Konzept der mathematischen Funktion: eine wohldefinierte Anweisungsfolge erzeugt aus den übergebenen *Parametern* oder *Argumenten* einen einzelnen, eindeutigen Rückgabewert. Auch C++ unterstützt, so wie viele Programmiersprachen, die so genannten Funktionen, welche Sie in diesem Kapitel kennenlernen werden.

Zur weiteren Lektüre	Funktionen sind Objekt eines C++ Primer Kapitels, Kapitel 7 ab Seite 277

4.1. Aufruf einer Funktion

Funktionen können entweder explizit als Anweisung aufgerufen werden (Listing 4.1), oder sie können wie mathematische Funktionen anstelle des jeweiligen Funktionswerts Verwendung finden (Listing 4.2). In letzterem Fall steht der Funktionsaufruf für seinen Rückgabewert: `quadrat( 12 )` im Listing 4.2 steht also für den Wert 144, wobei `sqrt( i )` für die Wurzel von `'i'` (die = 12 ist) steht.

Listing 4.1: Verwendung einer Funktion als explizite eigenständige Anweisung

```
1 gebeAus( "Vogonische Kampfpoesie" );
```

Listing 4.2: Verwendung einer Funktion (`quadrat()`,`sqrt()`) als Funktionswert

```
1 int i = quadrat( 12 ); // quadrat ist eine eigene Funktion
2 cout << i << " ist das Quadrat von " << sqrt(( double ) i );
```

Beim Aufruf einer Funktion mit Argumenten (siehe 4.2) ist auf den Datentyp der Argumente zu achten, welchen die Funktion in ihrer Definition festlegt. Es müssen daher kompatible Typen übergeben werden, da der Compiler ansonsten eine Fehlermeldung ausgibt.
Oftmals werden Funktionen verwendet, die zu einer Bibliothek gehören und erst durch inkludieren einer Datei eingebunden werden.

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1*, Seite 73

4.2. Deklaration einer Funktion

Um eine eigene Funktion zu schreiben, muss diese wie eine Variable erst deklariert und dann definiert werden. Bei der Deklaration (Bekanntmachung) der Funktion werden der Name der Funktion, der Datentyp des Rückgabewertes und die Datentypen der Argumente festgelegt. Diese Informationen bilden zusammen den so genannten Prototypen der Funktion, welcher folgende Syntax hat:

Listing 4.3: Grundsyntax eines C++-Funktionsprototypen

```
rueckgabeTyp funktionName( parameterTyp1 parameter1, ↩
   parameterTyp2 parameter2,... );
```

Im konkreten Beispiel wird nun eine Funktion `multiplizieren()` deklariert, welche einen Wert vom Typ `double` zurück gibt. Diese Funktion bekommt außerdem zwei Parameter vom Typ `int` und `float` übergeben mit den Namen `faktor` und `multiplikant`:

Parameter

```
double multiplizieren( int faktor, float multiplikant );
```

Eine Funktion muss keine Argumente haben und muss auch keinen Rückgabewert haben. In letzterem Fall wird als Datentyp für den Rückgabewert `void` gewählt. Außerdem existiert keine Beschränkung für die Anzahl an Argumenten die einer Funktion übergeben werden können.
Eine Funktion ohne Argumente und Rückgabewert:

```
void herunterfahren();
```

Hier unterscheidet sich die Deklaration einer Funktion nur durch den Typ des Rückgabewertes (`void`) vom eigentlichen Aufruf der Funktion, der `herunterfahren();` lauten würde.

4.3. Definition einer Funktion

Sobald in einer Deklaration der Name, die Parameter und der Rückgabetyp einer Funktion festgelegt wurden, muss noch die eigentliche Funktionalität beschrieben werden. Dies geschieht in der Definition einer Funktion:

Listing 4.4: Syntax der Definition

```
1 rueckgabeTyp funktionName( argumentTyp1 argument1, ... ) {
2   Anweisungen;
3   // ...
4   return rueckgabeWert;
5 }
```

Die Definition einer Funktion besteht aus der Deklaration der Funktion gefolgt von einem in geschwungenen Klammern gefassten Anweisungsblock. In diesem Anweisungsblock ist die Funktionalität der Funktion beschrieben und Ihnen sollte auffallen, dass die `main()`-Funktion auch nur eine Funktion ist. Wichtig innerhalb der Anweisungen einer Funktion ist das Kommando `return`, welches die Rückgabe eines Wertes und das Verlassen der Funktion bewirkt. Sobald die Anweisungsabfolge einmal ein `return` findet, wird die Funktion sofort verlassen und alle Anweisungen nach `return` nicht mehr ausgeführt. Nur in Funktionen mit dem Rückgabetyp `void` muss kein `return` stehen und hier wird die Funktion bis zum Erreichen von `'}'` durchlaufen.

```
1 double multiplizieren( int faktor, float multiplikant ) {
2   double ergebnis = faktor * multiplikant;
3   cout << "Das Produkt aus " << faktor << " und "
4        << multiplikant << " ist " << ergebnis << "." << endl;
5   return ergebnis;
6 }
```

Hier wird die schon bekannte Funktion `multiplizieren` mit den beiden Argumenten `faktor` und `multiplikant` definiert. Wird sie aufgerufen, so multipliziert sie ihre Parameter (Zeile 2), gibt eine entsprechende Meldung aus (Zeile 3) und gibt das Produkt zurück an den Aufrufer (Zeile 4).
Die für die Argumente gesetzten Variablen sind nur innerhalb der geschweiften Klammern bekannt und stehen für die der Funktion übergebenen Werte. Insbesondere müssen die Namen der Parameter nicht mit denen der Variablen übereinstimmen, die man für den Funktionsaufruf verwendet:

```
1 int a = 10;
2 double produkt = multiplizieren( a, 123.3 );
```

Hier werden die Werte 10 aus der Variablen `'a'` und 123.3 in die Variablen `int faktor` und `float multiplikant` kopiert. Die Letzteren werden erst beim Funktionsaufruf im Speicher des Computers erzeugt.
Ist eine Funktion vor ihrer Definition noch nicht deklariert, findet eine *implizite Deklaration* statt, jede definierte Funktion ist also automatisch deklariert. Deshalb wird die explizite Deklaration mit Funktionsprototypen oft nicht durchgeführt, sondern nur die Funktionsdeklaration geschrieben.
Da es aber notwendig ist, eine Funktion vor Verwendung mindestens einmal zu deklarieren, und es die Übersichtlichkeit erhöht, werden Funktionen am Anfang einer Quelltextdatei vorab deklariert.

4.4. Prototypen, Header-Dateien und Linking

Um den Quellcode in logische Module aufzuteilen und nicht allen Code in eine unübersichtliche Datei zu schreiben, wird der Code normalerweise auf mehrere Quelldateien aufgeteilt. Ein weiterer Grund, neben der Übersichtlichkeit und besseren Lesbarkeit, ist die Wiederverwendbarkeit von Code. So kann der gleiche Code an mehreren Stellen im Programm benutzt werden, was unter anderem den Wartungsaufwand senkt. Um auf in anderen Dateien definierte Funktionen zugreifen zu können, werden die Funktionsprototypen in Header-Dateien geschrieben. Der Compiler übersetzt die einzelnen Quelldateien in Objektdateien, welche der Linker untereinander verbindet. Die Header-Dateien werden in die Quelldateien inkludiert und bilden jeweils die *Schnittstelle* eines Moduls.
Das folgende Beispiel zeigt, wie eine sinnvolle Aufteilung des Codes in mehrere Dateien aussehen kann:
In Anlehnung an die obigen Beispiele denken wir uns ein Programm aus, dessen Hauptprogramm in der Datei **main.cpp** liegt. In einer weiteren Datei (**funktionen.cpp**) liegt unsere Funktion `multiplizieren`, die im Hauptprogramm verwendet wird. Bei **funktionen.h** handelt es sich um eine sogenannte *Header-Datei*, welche nur die Funktionsprototypen beinhaltet.
`#include` bewirkt, dass bevor die Quelldatei **main.cpp** an den Compiler übergeben wird, der Text aus der Header-Datei einfach in **main.cpp** kopiert wird. So sind dem Compiler nun die Prototypen der verwendeten Funktionen bekannt.[1]

Listing 4.5: main.cpp

```
1  #include <iostream>
2  #include "funktionen.h"
3  using namespace std;
4
5  int main() {
6    double unten, oben, schrittw;
7    cout << "Bitte untere Grenze der Multiplikationstabelle ↩
       eingeben: ";
8    cin >> unten;
9    cout << endl << "Bitte obere Grenze der ↩
       Multiplikationstabelle eingeben: ";
10   cin >> oben;
11   cout << endl << "Bitte Schrittweite eingeben: ";
12   cin >> schrittw;
13   for( double vZaehler = unten; vZaehler <= oben; vZaehler += ↩
       schrittw ) { // v = Vertikal
14     for( double hZaehler = unten; hZaehler <= oben; hZaehler +=↩
           schrittw ) { // h = Horizontal
15       cout << multiplizieren( vZaehler, hZaehler ) << ' ';
16     }
17     cout << endl;
18   }
19   return 0;
20 }
```

Listing 4.6: funktionen.h

```
1  #ifndef FUNKTIONEN_H
2  #define FUNKTIONEN_H
3
4  double multiplizieren( double faktor1, double faktor2 );
5
6  #endif  //FUNKTIONEN_H
```

Um Probleme beim Linken zu vermeiden, sollten Header-Dateien immer den im Beispiel gezeigten Präprozessor-Anweisungen `#ifndef`, `#define` und `#endif` geschützt werden. Dies verhindert eine

[1] **iostream** wird per `#include <...>` eingebunden, da es sich hierbei um einen Header der Standardbibliotheken (siehe Kapitel 9) handelt. Eigene Headerdateien werden per `#include "..."` eingebunden.

mehrfache Einbindung in das Programm (siehe Seite 14 in den Programmier-Richtlinien/Coding-Guidelines).

Listing 4.7: funktionen.cpp

```
1 #include "funktionen.h"
2 double multiplizieren( double faktor1, double faktor2 ) {
3   return faktor1 * faktor2;
4 }
```

Nachdem die beiden .cpp-Dateien in Maschinencode übersetzt worden sind, setzt der Linker sie zusammen.

Zur weiteren Lektüre	Zu Header-Dateien siehe auch „Eigene Headerdateien schreiben“, C++ Primer Kapitel 2.9, Seiten 95ff

4.5. Arten der Parameter-Übergabe

Werden an eine Funktion *Variablen* (Wert- oder Adressvariablen) als Argumente übergeben, so arbeitet die Funktion mit einer *Kopie* der übergebenen Variable. Beim Funktionsaufruf werden die Werte der übergebenen Variablen intern in neue Variablen kopiert, welche innerhalb der Funktion verwendet werden. Die Änderungen an einer Argumentsvariable innerhalb der Funktion betreffen also die beim Aufruf verwendete Variable nicht. Dieses Verhalten bezeichnet man als *Call by Value*, es wird der *Wert* der Variable und nicht die Variable selbst übergeben. Dies hat aber den Nachteil, dass zunächst die Variable in eine neue kopiert werden muss, was Speicher und Rechenzeit kostet. Außerdem kann die Funktion dann nicht auf die übergebenen Variablen selbst verändernd zugreifen, da sie ja nur Kopien erhält.

Kopie vs. Referenz

Ist ein direkter Zugriff auf die übergebene Variable nötig oder das interne Kopieren zu aufwendig oder nicht sinnvoll, sollte statt *Call by Value* *Call by Reference* eingesetzt werden. Hierbei wird statt einer Variable eine Referenz an die Funktion übergeben, so dass die Variable bei der Übergabe lediglich umbenannt wird. Wird innerhalb der Funktion nun der Wert der Argumentvariable geändert, hat nach dem Verlassen der Funktion auch die übergebene Variable einen anderen Wert. Es gibt auch das *Call by Pointer* Verfahren, welches in Abschnitt 4.5.3 erklärt wird.

4.5.1. Call by Value

Wird eine Funktion mit den herkömmlichen Variablentypen deklariert, so findet automatisch *Call by Value* statt:

Call by Value

Listing 4.8: *Call by Value*

```
1  double quadrieren( double zahl ) {
2    // verändert "basis" nicht, da "zahl" eine Kopie ist
3    zahl = zahl * zahl;
4    return zahl;
5  }
6
7  int main() {
8    double basis = 10;
9    double quadrat = quadrieren( basis );
10   cout << basis << " zum Quadrat ist " << quadrat;
11   return 0;
12 }
```

Dieses Programm (Listing 4.8) gibt wie erwartet `10 zum Quadrat ist 100` aus. Beim Aufruf von `quadrieren()` in der `main()`-Funktion wird der Wert 10 von `basis` in eine neu erstellte Variable `zahl` kopiert. Diese neue Variable existiert nur solange im Speicher, wie die Programmausführung sich in der Funktion `quadrieren()` befindet, danach wird `zahl` gelöscht. Daher ist auch der Wert von `basis` nach dem Aufruf von `quadrieren()` immer noch 10.

4.5.2. Call by Reference

Für *Call by Reference* muss im Funktionsprototyp eine Referenz als Datentyp des Arguments

Call by Reference

gewählt werden: aus `double` wird `double&`.

Listing 4.9: *Call by Reference*

```
1  void quadrierMich( double& zahl ) {
2    // verändert "basis", da "zahl" eine Referenz auf "basis" ist
3    zahl = zahl * zahl;
4  }
5
6  int main() {
7    double basis = 10;
8    quadrierMich( basis );
9    cout << "Das Quadrat lautet " << basis;
10   return 0;
11 }
```

Beim Aufruf von `quadrierMich()` wird die übergebene Variable `basis` in `zahl` umbenannt. Die Änderung des Wertes von `zahl` ist also gleichzeitig auch eine Änderung des Wertes, der in der `main()`-Funktion `basis` heißt. Dieses Programm (Listing 4.8) gibt also
`Das Quadrat lautet 100` aus.

Zur weiteren Lektüre	Der C++ Primer schreibt hierüber als Wertparameter (*Call by Value*) bzw. als Referenzparameter (*Call by Reference*) in den Abschnitten 7.2.1 und 7.2.2 ab Seite 282

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1, 2*, Seite 257

4.5.3. Call by Pointer

Pointerübergabe als Alternative zu Call by Reference

Häufig ist es sinnvoll, mit Zeigern statt mit Wertvariablen zu arbeiten. Auch dies bietet die Möglichkeit, auf die übergebene Variable selbst zuzugreifen. Die Pointerübergabe kann aber auch dazu verwendet werden, den Pointer auf ein Array zu übergeben, und so der Funktion den Zugriff auf dieses zu ermöglichen. Bei der Übergabe eines Pointers wird eine Kopie des übergebenen Zeigers angefertigt, nicht aber des Objekts. Trotzdem verbraucht das Erstellen bzw. Kopieren eines Zeigers Speicherplatz und Rechenzeit.

Listing 4.10: Übergabe eines Zeigers

```
1  double quadrieren( double* zeig ) {
2    // zeig ist ein Zeiger
3    *zeig = ( *zeig ) * ( *zeig );
4    // lies die obige Zeile: In der Variable, auf die zeig zeigt,↩
         speichere das Produkt aus der Variable, auf die zeig ↩
        zeigt, mit sich selbst
5    return *zeig;
6  }
7
8  int main() {
9    double basis = 10;
10   double quadrat = quadrieren( &basis );
11   cout << basis << " zum Quadrat ist " << quadrat;
12   return 0;
13 }
```

Beim Aufruf von `quadrieren()` in der `main()`-Funktion wird die Adresse von `basis` in einen neu erstellten Zeiger `zeig` kopiert. Nun wird der Wert auf den `zeig` verweist verändert und das ist der Wert von `basis`. Somit ergibt sich die Ausgabe `100 zum Quadrat ist 100`. Da C++ aber Referenzen besitzt, sollte sich der Programmierer jedesmal überlegen, ob er lieber eine Referenz übergeben möchte, oder ob es sinnvoller ist, die Adresse der Variable in Form eines Zeigers zu übergeben, der dann beispielsweise gespeichert oder andersweitig weiterverwendet werden kann.

Übungsaufgabe	*Aufgabe 3, 4*, Seite 257
C++ Lernen und professionell anwenden	

Kapitel 5

Erweiterte Datentypen

Sie kennen bereits die elementaren Datentypen (`int`, `double`, `bool`, ...).
Darüber hinaus unterstützt C++ erweiterte Datentypen. Diese werden aus den elementaren Datentypen zusammengesetzt und ermöglichen eine deutlich elegantere Lösung vieler Problemstellungen.
Stellen Sie sich zum Beispiel vor, Sie wollen die Daten Ihrer 100 Autos speichern. Das Programm hat also intern mehrere Arrays, eins für Baujahr, eins für die PS, eins für die Marke, ..., jeweils mit 100 Einträgen. Es muss darauf geachtet werden, dass die Daten eines speziellen Wagens in allen Arrays den gleichen Index haben. Ansonsten vermischt das Programm die Daten und Ihre Datenbank ist nutzlos. Wäre es nicht elegant, einen Datentyp „Auto“ zu haben, so dass intern nur ein Array mit Einträgen vom Typ „Auto“ existiert?
Dieser Datentyp würde alle interessanten Daten zusammenfassen, als Unterpunkte verfügbar machen und somit eine Vermischung verhindern.
Weiterhin stellt sich die Frage, wie die Marke gespeichert wird. Es wäre wünschenswert einen Datentyp „Marke“ zu haben, welcher als Wert nur die Marken Ihrer Autos annehmen kann: Mercedes, Audi, VW, ...
Diese beiden Probleme sind Einsatzgebiete für die erweiterten Datentypen.

5.1. Enumeration

Im obigen Beispiel ist ein Datentyp `Marke` nötig, dieser lässt sich als sogenannte *Enumeration* (Aufzählung) umsetzten. Eine Enumeration ist eine Liste von benannten ganzzahligen Konstanten, welche mit dem Schlüsselwort `enum` definiert wird:

```
1 enum typname { Konstanten };
```

Oder praktisch:

```
1 enum marke { Mercedes, Audi, VW, BMW };
```

Die obige Zeile definiert den neuen Datentyp `Marke`, welcher die 4 Werte (`Mercedes`, `Audi`, `VW` und `BMW`) annehmen kann. Intern gibt C++ jedem Namen eine Nummer (`Mercedes = 0`, `Audi = 1`...) und arbeitet mit diesen statt den wirklichen Namen. Nachdem einmal im aktuellen Namensbereich (siehe *Gültigkeitsbereiche*, Kap. 6) der Datentyp deklariert wurde, kann er wie gewohnt verwendet werden:

Listing 5.1: Beispiel für `enum`

```
2 marke meineAutoMarke = Audi;
3 if( meineAutoMarke == Audi ) {
4     cout << "Ich würde gerne Audi fahren" << endl;
5 }
```

Zur weiteren Lektüre	Siehe hierzu auch den C++ Primer, Kapitel 2.7, Seite 90

5.2. `struct`: Zusammenfassender Datentyp

Oft ist es nützlich, verschiedene Daten zusammenzufassen, im einleitenden Beispiel waren es die Kennwerte eines Autos. Aus diesem Grund bietet C++ *Structs* (engl. kurz für *Strukturen*), welche mehrere schon bekannte Datentypen zu einem Neuen zusammenfassen.
Eine Struktur muss ebenso wie eine Enumeration vor ihrer Verwendung definiert werden:

```
1 struct Typname { Datentypen };
```

Oder praktisch:

Listing 5.2: `struct`: Beispiel

```
1 struct Auto {
2     int psZahl;
3     Marke marke;
4     short baujahr;
5 };
```

Hier wird eine Struktur `Auto` definiert, welche drei Variablen von verschiedenen Datentypen zu einem Neuen vereinigt. Wenn also eine Variable vom Typ `Auto` angelegt wird, entstehen im Speicher eigentlich 3 Variablen. Auch dieser neue Datentyp wird mit `struct` vorangestellt wie gewohnt verwendet, doch um den Unterdatentypen Werte zuweisen zu können, benötigt man den `'.'` Operator. Mit Strukturen können auch Arrays gebildet werden und sie können verschachtelt werden.

Typdefinition: Anweisung

Da es sich bei der Definition eines `struct`-Datentyps um eine Anweisung handelt, muss auch unbedingt nach der abschließenden geschweiften Klammer auf das `';'` geachtet werden (Zeile 5).

Listing 5.3: Beispiel für die Verwendung von `struct`

```
1 Auto meinAuto;
2 meinAuto.psZahl = 200;
3 meinAuto.baujahr = 1986;
4 meinAuto.marke = VW;
5
6 Auto anderesAuto;
7 anderesAuto = meinAuto;
```

Hier wird als erstes eine Variable `meinAuto` vom Typ Auto erzeugt, deren Untervariablen danach Werte zugewiesen bekommen. `meinAuto.psZahl` ist eine reine `int`-Variable, welche jedoch zu einer Struktur gehört. Dann wird eine andere Variable vom Typ `Auto` erzeugt, welche dann die Werte von `meinAuto` zugewiesen bekommt. Letztlich sind jeweils `meinAuto.psZahl` und `anderesAuto.psZahl`, sowie `meinAuto.baujahr` und `anderesAuto.baujahr` ... äquivalent. Im Beispiel war `meinAuto` eine Wertevariable vom Typ `Auto` und auch die Verwendung eines Zeigers ist mit dem Pfeil-Operator (`->`) äquivalent möglich:

Listing 5.4: Einsatz des Pfeil-Operators (`->`)

```
1 Auto meinAuto;
2 Auto* meinAutoZeiger = &meinAuto;
3 meinAutoZeiger->psZahl = 200;
```

-> Operator: Zugriff auf Untervariablen von mit Zeigern referenzierten Objekten

Hier wird mit dem Operator `->` über einen Zeiger auf die Struktur auf ein Unterelement zugegriffen. Obwohl `meinAutoZeiger` eine Adressvariable ist, ist wiederum `meinAutoZeiger->psZahl` eine reine `int`-Wertvariable.

Der `->` Operator ist eine handliche Abkürzung. `meinAutoZeiger->psZahl = 200;` ist äquivalent zu `(*meinAutoZeiger).psZahl = 200;` (siehe Dereferenzierung, Abschnitt 2.9).

Genau genommen sind Strukturen in C++ ein Spezialfall von Klassen (siehe 5.3). Gegenüber der standardmäßigen Sichtbarkeit bei Klassen sind alle Elemente von Strukturen jedoch öffentlich sichtbar (public), wenn nichts anderes angegeben wurde. Genau wie Klassen können auch Strukturen neben Attributen noch Methoden umfassen.

5.3. Klassen

Erweitert man die einfache Zusammenfassung von Daten um die Konzepte der Objektorientierung (Vererbung, Kapselung, ...) spricht man von Klassen. Oft fassen Klassen nicht nur Variablen zu einem neuen Datentyp zusammen, sondern auch die zugehörigen Funktionen (siehe Kap. 4, Seite 20). Im Zusammenhang von Objektorientierung und Klassen ist das die Kapselung von Attributen und Methoden. Attribute sind die zur Klasse gehörenden Variablen, während mit Methoden die zur Klasse gehörenden Funktionen bezeichnet werden. Mehr über Objektorientierung erfahren Sie im Kapitel Objektorientierung. Die Definition einer Klasse erfolgt analog zu Enumeration und Struct:

Klassen als Vereinigung von Datenstruktur und Funktionalität

```
1 class Typname { Attribute und Methoden };
```

Oder praktisch:

Listing 5.5: Klasse: Beispiel

```
1  class Auto {
2  private:
3    int psZahl;
4    Marke marke;
5    short baujahr;
6    int geschwindigkeit;
7
8  public:
9    Auto();
10   ~Auto();
11
12   void beschleunigen( int betrag );
13   int bremsen();
14 };
```

Im obigen Beispiel wird eine Klasse `Auto` deklariert, welche 4 Methoden besitzt: `Auto()`,`~Auto()`, `beschleunigen()` und `bremsen()`. Außerdem besitzt die Klasse vier Attribute, nämlich `psZahl`, `geschwindigkeit`, `marke` und `baujahr`. Die Methoden werden als Funktionsprototypen in die Klassendeklaration geschrieben, wobei jede Klasse zwei besondere Methoden besitzt: den Konstruktor und den Destruktor. Im obigen Beispiel ist `Auto()` der Konstruktor und `~Auto()` ist der Destruktor. Der Konstruktor und der Destruktor müssen genauso heißen, wie die Klasse, zu der sie gehören. Der Destruktor hat im Unterschied zum Konstruktor die vorgestellte Tilde und weiterhin haben beide Methoden keinen Rückgabewert. Der Konstruktor wird aufgerufen, sobald ein Objekt der Klasse erzeugt wird, in ihm setzt man zum Beispiel die Attribute auf ihre Startwerte. Im Gegenzug wird der Destruktor aufgerufen, wenn das Objekt gelöscht wird und in ihm steht Code um zum Beispiel Speicher wieder freizugeben.
Neu ist außerdem das Schlüsselwort `public`, dessen Bedeutung im Kapitel über Objektorientierung besprochen wird. Bis zu diesem Punkt sollten Sie es einfach immer in Ihrer Klassendeklaration schreiben.
Bisher wurde die Klasse nur deklariert, es fehlt noch die Definition der Methoden. Die Definition einer Methode erfolgt analog zur Definition einer Funktion (Kapitel 4), allerdings muss mit dem `::` Operator die Klassenzugehörigkeit kenntlich gemacht werden. Die geschieht nach folgender Syntax:

Listing 5.6: Definition einer Klassenmethode

```
1 rueckgabeWert KlassenName::methodenName() {
2   // methodenkorpus
3 }
```

Im Beispiel:

Listing 5.7: Klasse mit Methoden

```
1  Auto::Auto() {
2     psZahl = 100;
3     marke = VW;
4     baujahr = 2003;
5     geschwindigkeit = 0;
6  }
7
8  Auto::~Auto() {
9  }
10
11 void Auto::beschleunigen( int betrag ) {
12     geschwindigkeit += betrag;
```

```
13 }
14
15 int Auto::bremsen() {
16     geschwindigkeit -= 10;
17     return geschwindigkeit;
18 }
```

Hier wird als erstes der Konstruktor definiert, welcher den Klassenattribute die Anfangswerte zuweist. Es folgt ein leerer Destruktor, welcher keine Befehle durchführt. Als letztes kommen die beiden Methoden `beschleunigen()` und `bremsen()`, welche die aktuelle Geschwindigkeit des Autos verändern.
Ist die Klasse deklariert und ihre Methoden definiert, kann sie als Datentyp analog zu Strukturen verwendet werden:

Listing 5.8: Zugriff auf Klassenelemente

```
19 Auto meinAuto;
20 meinAuto.psZahl = 10000;
21 meinAuto.beschleunigen( 10 );
22 meinAuto.bremsen();
23
24 Auto* meinAutoZeiger = &meinAuto;
25 meinAutoZeiger->baujahr = 2002;
26 meinAutoZeiger->beschleunigen( 16 );
```

Hier wird eine Variable der Klasse Auto erstellt, deren `psZahl`-Attribut auf `10000` gesetzt wird. Dann wird das Auto beschleunigt und anschließend wieder abgebremst, sowie ein Zeiger auf die Klasse verwendet. Es ist ersichtlich, dass die Verwendung von Klassen mit den Operatoren `'.'` und `->` analog zu Strukturen funktioniert.

Zur weiteren Lektüre	Zum Konzept von Klassen als Datentypen: C++ Primer , Kapitel 2.8, Seite 91

5.3.1. Beispiel für eine Klasse: `string`

C++ spezifiziert eine Reihe von Standardbibliotheken, die zu einer „vollständigen“ Compilersuite dazugehören. Eine dieser Bibliotheken nennt sich *string*:

Listing 5.9: Klassenbeispiel: `string`

```
1  // Bibliothek einbinden
2  #include <string>
3  // für Ein- und Ausgabe
4  #include <iostream>
5  using namespace std;
6  int main() {
7    string meinString; // ein Objekt der Klasse string wird ↩
       erzeugt
8    string zweiterString( "abc" );
9    string gehtAuchSo = "def";
10   cout << "Die ersten drei Buchstaben des Alphabets lauten: "
11        << zweiterString << endl
12        << "Die ersten beiden Buchstaben lauten: "
13        << zweiterString.substr( 0, 2 ) << ", danach kommt ↩
            übrigens "
14        << gehtAuchSo << endl;
15   zweiterString += gehtAuchSo;
16   cout << "Der string lautet jetzt insgesamt " << zweiterString↩
         << endl;
17   return 0;
18 }
```

Die Ausgabe des Programms lautet:

```
Die ersten drei Buchstaben des Alphabets lauten: abc
Die ersten beiden Buchstaben lauten: ab, danach kommt übrigens def
Der String lautet jetzt insgesamt abcdef
```

Wie man sieht, verhält sich `string` von der Handhabung her wie ein einfacher Datentyp. Zeile 8 zeigt aber, dass er offensichtlich einen *Konstruktor* besitzt, also eine Funktion, mit der man das Objekt erstellen kann. Zeile 9 zeigt, dass es möglich ist, einen klassischen C-String (siehe 2.11.2) in ein Objekt der Klasse `string` zu wandeln (*casting*).
In Zeile 11 zeigt sich, dass in der Klasse `string` die Methode `substr( int anfang, int laenge )` existiert, die den Unterstring (engl. *substring*) ab Position `anfang` bis `anfang + laenge` extrahiert und ein `string`-Objekt mit diesem Inhalt zurückliefert.

Zur weiteren Lektüre	Die `string`-Klasse kann noch viel mehr. Um mehr über die Funktionalitäten dieser Klasse (und anderer Standard-Bibliotheks-Klassen) zu erfahren, empfehlen sich Websites wie `http://www.cplusplus.com`

Übungsaufgabe C++ Lernen und professionell anwenden	*Übungsaufgabe* auf Seite 281, *1. Aufgabe* auf Seite 307 (ignorieren Sie gegebenfalls den Destruktor), *2. Aufgabe* auf Seite 331

5.4. Strings

Mittels `#include <string>` lässt sich die *strings library* der STL (Kapitel 10, Seite 51) einbinden. Sie ermöglicht dann die Verwendung der Klasse `string`.

5.4.1. Einen String erstellen

Um einen String zu erstellen, muss man einfach eine Variable vom Typ `string` deklarieren. Der Konstruktor der Klasse `string` erlaubt es auch, einen neuen STL-String aus einem C-String (siehe 2.11.2) zu erstellen, oder einem anderen String zu erstellen.

Listing 5.10: Erstellen von Strings

```
1 string s1;  // leerer String
2 string s2( "abc" ); // Initialisierung mit "abc"
3 string s3 = "def"; // leerem String wird Inhalt "def" ↩
    zugewiesen
4 string s4( 5, 'a' ); // String füllen mit aaaaa
5 s1 = s3; // String s1 "def" zuweisen
```

5.4.2. Einzelne Zeichen in Strings

In einem String kann man wie in einem Array auf jedes Zeichen einzeln zugreifen. Das erste Zeichen ist, wie bei Arrays auch, mit dem Index '0' erreichbar. Der Zugriff kann über zwei Wege erfolgen:

Listing 5.11: Zugriff auf einzelne Zeichen in Strings

```
1 string str = "hallo!";
2 char buchstabe = str[1]; // Wert von 'buchstabe': 'a'
3 buchstabe = str.at(4); // Wert von 'buchstabe': 'o'
```

5.4.3. `substr`: Einen Teilstring erstellen

`substr` liest aus dem String einen Teilstring aus (engl. *substring*), und gibt diesen zurück. Der Original-String wird nicht verändert.

```
stringobjekt.substr( anfang, laenge = npos );
```

Dabei nimmt die Funktion ab dem `anfang` Zeichen die nächsten `laenge` Zeichen, erstellt aus diesen einen neuen String und gibt diesen zurück. Würde man so über das Ende des Strings hinauslesen, werden nur die noch vorhandenen Zeichen ausgegeben. Die Zählung fängt bei 0 an!

Wird `laenge` nicht angegeben, wird der Methode automatisch die Konstante `npos` übergeben und es wird bis zum Ende der Zeichenkette gelesen.

Listing 5.12: Beispiel für `substr`

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

int main () {
  string kette = "Nichts gibt es, was Rhabarber nicht besser machen würde";
  string sub;
  sub = kette.substr( 43, 6 ); // "machen"
  cout << sub << endl;
  return 0;
}
```

gibt aus: `machen`

5.4.4. Strings zusammenfügen

Die Klasse `string` überlädt den '+' Operator, sodass man zum zusammenfügen zweier Strings nur diese „addieren“ muss.

Listing 5.13: Beispiel zur Stringverkettung

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

int main () {
  string a = "du noch viel lernen musst";
  string b = "Junger Padawan!";
  string c = a + ", " + b;
  cout << c << endl;
  return 0;
}
```

gibt aus: `du noch viel lernen musst, Junger Padawan!`

5.4.5. `find`: Einen String nach einer Zeichenkette durchsuchen

Mit `find( suchstring, startposition )` lässt sich in einem String nach einem anderen String suchen. Wird *startposition* nicht angegeben, startet die Suche am Anfang des Strings. Ansonsten startet sie ab dem Zeichen mit dem Index *startposition*.
Außer `find` gibt es noch `rfind`, welches fast gleich wie `find` funktioniert. Allerdings beginnt `rfind` am Ende und sucht rückwärts.

Listing 5.14: Beispiel für `find`

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

int main () {
    string heuhaufen = "Hier ist irgendwo eine Nadel versteckt. Ja, das ist sie!";
    int posNadel = heuhaufen.find( "Nadel" );
    int posIst1 = heuhaufen.find( "ist" );
    int posIst2 = heuhaufen.find( "ist", 10 );
    int posIst3 = heuhaufen.rfind( "ist" );
    cout << "Nadel: "       << posNadel << endl
         << "erstes ist: "  << posIst1  << endl
```

```
13          << "erstes ist nach Stelle 10: " << posIst2 << endl
14          << "letztes ist: " << posIst3  << endl;
15 }
```

gibt aus:

```
Nadel: 23
erstes ist: 5
erstes ist nach Stelle 10: 48
letztes ist: 48
```

Zeile 7 sucht nach der Zeichenkette "Nadel". Dieses Wort beginnt an Position 23 des Strings.

Zeile 8 sucht nach „ist“. Dies wird an 5. Stelle gefunden, die Suche wird danach abgebrochen.

Zeile 9 sucht nach „ist“, fängt allerdings erst ab dem 10. Zeichen an zu suchen, daher wird das „ist“ an Stelle 5 nicht gefunden. Das nächste „ist“ kommt an Stelle 48.

Zeile 10 sucht nach dem ersten „ist“, fängt aber am Ende des Strings an zu suchen und sucht dann rückwärts.

5.4.6. Abschnitte ersetzen, löschen oder neue Abschnitte einfügen

Mit `erase` lässt sich ein Teilstring löschen. Mit `replace` kann man einen Teilstring mit einer Zeichenkette ersetzen. `insert` fügt eine Zeichenkette an einem bestimmten Punkt im String ein. Die Verwendung dieser drei Methoden ist ähnlich zu den bisher vorgestellten.

Listing 5.15: Syntax von `erase, replace, insert`

```
str.erase( startindex, länge );
str.replace( startindex, länge, ersatzstring );
str.insert( startindex, string );
```

Listing 5.16: Beispiel für `erase, replace, insert`

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main () {
6    string heuhaufen = "Hier ist irgendwo eine Nadel versteckt. ↩
       Ja, das ist sie!";
7    heuhaufen.erase( 39, 17 ); // " Ja, das ist sie!" entfernen
8    cout << heuhaufen << endl;
9    heuhaufen.replace( 23, 5, "Milchkanne" ); // "Nadel" ersetzen↩
         durch "Milchkanne"
10   cout << heuhaufen << endl;
11   heuhaufen.insert( 23, "blaue " ); // vor "Milchkanne" "blaue"↩
         einfügen
12   cout << heuhaufen << endl;
13 }
```

gibt aus:

```
Hier ist irgendwo eine Nadel versteckt.
Hier ist irgendwo eine Milchkanne versteckt.
Hier ist irgendwo eine blaue Milchkanne versteckt.
```

5.4.7. `c_str`: String in einen C-String umwandeln

Oftmals (z.B. s. 5.4.8) ist es nötig, einen String in einen C-String (siehe 2.11.2) umzuwandeln. Hierfür stellt die String-Klasse eine Methode bereit:

Listing 5.17: Syntax von `c_string`

```
1    string.c_str();
```

5.4.8. Strings in Zahlen umwandeln

Will man einen String in eine Zahl umwandeln, mit der C++ rechnen kann, gibt es verschiedene Möglichkeiten:

!!!

Hinweis	Die folgenden Funktionen gehören nicht zur String Klasse, sondern zur Bibliothek `cstdlib`. Um sie zu verwenden, müssen Sie also `#include <cstdlib>` zu Beginn der Datei einbinden. Alle drei hier vorgestellten Methoden zur Umwandlung eines Strings in Zahlen erwarten einen C-String als Argument. Sie müssen also mit dem Argument `string.c_str()` aufgerufen werden.

atoi Wandelt einen String so lange in eine Ganzzahl um, bis ein Zeichen kommt, das keiner Ganzzahl entspricht. `123.23` wird also zu `123`

atof Wandelt einen String so lange in eine Fließkommazahl um, bis ein Zeichen kommt, das keiner Fließkommazahl entspricht. `123.23abc` wird also zu `123.23`

strtod Wandelt ebenfalls in eine Fließkommazahl um, benötigt jedoch zusätzlich einen Zeiger auf einen C-String, in dem der Reststring gespeichert wird. Diese Methode ist dann besonders nützlich, wenn ein String mehrere Zahlen hintereinander enthält.

Listing 5.18: Wandlung von Strings in Zahlen

```
1  #include <iostream>
2  #include <string>
3  #include <cstdlib>
4  #include <sstream>
5  using namespace std;
6
7  int main () {
8      string zahlen = "12.37 1.568";
9      int ganzzahl = atoi( zahlen.c_str() );
10     float kommaAtof = atof( zahlen.c_str() );
11     double kommaStrtod1, kommaStrtod2;
12     char* reststring; // Hier wird der Rest gespeichert
13     kommaStrtod1 = strtod( zahlen.c_str(), &reststring );
14     kommaStrtod2 = strtod ( reststring, NULL );
15
16     cout << "Ganzzahl: "     << ganzzahl     << endl
17          << "Kommazahl 1: " << kommaAtof    << endl
18          << "Kommazahl 2: " << kommaStrtod1 << endl
19          << "Kommazahl 3: " << kommaStrtod2 << endl;
20 }
```

gibt aus:

```
Ganzzahl: 12
Kommazahl 1: 12.37
Kommazahl 2: 12.37
Kommazahl 3: 1.568
```

Zeile 9 wandelt den String in eine Integer-Zahl um. Da an Stelle 3 ein '.' steht, wird hier abgebrochen.

Zeile 10 macht das Gleiche wie Zeile 9, allerdings wird in eine Float-Zahl umgewandelt, die mit „.37“ durchaus etwas anfangen kann. Das Leerzeichen gehört allerdings auch nicht in eine Float-Variable, darum wird nach „12.37“ die Umwandlung beendet.

Zeile 13 verfährt zuerst analog zu Zeile 10. Allerdings wird beim Beenden der String nach dem (ungültigen) Leerzeichen („1.568“) in `reststring` gespeichert.

Zeile 14 setzt nun die `strtod`-Funktion nochmals auf den Reststring an.

5.4.9. `size`: Länge einer Zeichenkette

Wie der Name vermuten lässt, gibt `string.size()` die Länge der Zeichenkette zurück.

Zur weiteren Lektüre	`http://www.cplusplus.com/reference/string/string/`. Außerdem finden Sie ausführliche Erläuterungen im C++ Primer in Kapitel 3.2.

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1, 2, 3*, Seite 187

Kapitel 6

Gültigkeitsbereiche

Als Gültigkeitsbreich (oder Scope) bezeichnet man den Bereich, in dem ein deklariertes Objekt im Speicher existiert und somit verwendet werden kann. Der Gültigkeitsbereich eines Objekts ist abhängig vom Ort und der Art der Deklaration.

6.1. Globaler Gültigkeitsbereich

Im einfachsten Fall wird ein Objekt beim Programmstart erzeugt. Es belegt während der ganzen Ausführungszeit Speicher und wird erst bei Programmende wieder gelöscht. Dieses Objekt kann folglich überall im Quellcode genutzt werden, daher spricht man von einem *globalen* oder auch *global deklarierten* Objekt.
Trotz der einfachen Benutzung überwiegen die Nachteile globaler Variablen: Sie belegen ständig Speicher, was gerade bei großen Datenmengen störend ist. Außerdem verliert der Programmierer schnell die Übersicht über die verwendeten Variablen und deren Namen. Dies kann zu ungewollten Seiteneffekten führen, d.h. ein Programmabschnitt verändert ungewollt bzw. unkontrolliert eine Variable, die in einem anderen Teil des Programms benutzt wird. Daher sollten globale Variablen vermieden werden. !!!
Eine Variable ist dann global deklariert, wenn sie nicht innerhalb von geschwungenen Klammern {} steht, also bei einfachen Programmen außerhalb von Funktionen.

6.2. Lokaler Gültigkeitsbereich

Eine nicht globale Variable wird als *lokal* bezeichnet und wird in einem vom *Blockoperator* {} umrandeten Bereich deklariert. Dieser Bereich kann sich innerhalb einer Funktionsdefinition, innerhalb einer Schleife (oder einer anderen Kontrollstruktur) oder innerhalb einer Klasse befinden. Die lokale Variable wird erst im Speicher angelegt, wenn der Programmablauf ihre Deklaration erreicht wird. Sie lebt so lange, wie sich der Programmablauf innerhalb der {}-Klammern befindet. Außerhalb ihres Gültigkeitsbereichs ist die Variable unbekannt und kann nicht verwendet werden.
Ist eine Variable mit einem Namen deklariert, der in einem umgebenden Block schon verwendet wird, so wird die „alte“ Variable durch die neu deklarierte *verdeckt*. Der Compiler geht davon aus, dass der Programmierer in diesem kleineren Block seine frisch deklarierte Variable meint (und nicht die verdeckte). Die verdeckte Variable ist weiterhin *gültig*, jedoch *nicht sichtbar*. Verdeckung

Listing 6.1: Globale-/lokale Variablen

```
1  #include <iostream>
2  using namespace std;
3
4  int zahl = 11; // globale Variable
5
6  void machWas( int zahl ); // Funktionsdeklaration
7  void machWasNeues();
8
9  int main() {
10   int zahl = 25; // lokale Variable
11   machWas( zahl ); // Übergabe der lokalen Variable
12   machWasNeues();
13   return 0;
14 }
15
16 void machWas( int zahl ) { // Funktionsdefinition, lokale ↩
       Variable zahl
```

```
17 cout << "uebergebene Variable: " << zahl << "; globale Variable↩
       : " << ::zahl << endl;
18 }
19
20 void machWasNeues() { // Funktionsdefinition
21 cout << "globale Variable: " << zahl << "; globale Variable: " ↩
       << ::zahl << endl;
22 }
```

Die Ausgabe dieses Programms ist:

```
uebergebene Variable: 25; globale Variable: 11
globale Variable: 11; globale Variable: 11
```

Die Variable `zahl` wird zuerst ist global deklariert, denn sie steht nicht innerhalb von {}-Klammern. Die Variable ist sowohl in der `main()`-Funktion, als auch in den anderen Funktionen bekannt.
In der Funktion `machWas( int zahl )` wird die Globale Variable verdeckt. Auf sie kann jedoch mit dem Scope-Operator `::` zugegriffen werden (Zeile 14). Ist eine globale Variable nicht verdeckt, kann auf sie auch ohne den Scope-Operator `::` zugegriffen werden, wie in der Funktion `machWasNeues()` (Zeile 18).

Lokale Variablen sind nur solange gültig, wie man sich in einem Funktionsblock aufhält. In Funktionen, die aus diesem Funktionsblock heraus aufgerufen werden, sind sie nicht bekannt.

Listing 6.2: Lokale Variablen

```
1 void machWas() { // Funktionsdeklaration und Definition
2   cout << "globale Variable: " << zahl << endl; // FEHLER
3 }
4
5 int main() {
6   int zahl = 25; // lokale Variable
7   machWas();
8   return 0;
9 }
```

Der Compiler wird bei der Übersetzung dieses Quellcodes einen Fehler melden, denn innerhalb des Gültigkeitsbereichs der Funktion `machWas()` ist die Variable `zahl` nicht bekannt. `zahl` wird innerhalb der `main()`-Funktion erzeugt und ist nur in diesem Bereich bekannt. Um die Variable oder deren Wert auch in der Funktion `machWas()` bereit zu stellen, müsste sie übergeben werden. (Kapitel 4.5, Seite 23)

Listing 6.3: Verdeckung

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5   for( int i = 0; i <= 5; i++ ){
6     short i = 1;
7     i += 2;
8     cout << i << ",";
9   }
10  return 0;
11 }
```

Dieses Programm hat als Ausgabe `3,3,3,3,3,`. Man könnte eine Endlosschleife erwarten oder `0,1,2,3,4,` als Ausgabe, doch die beiden Variablen `'i'` haben nicht den gleichen Gültigkeitsbereich. Erreicht der Programmablauf den Kopf der `for`-Schleife, wird eine Variable `'i'` erzeugt und auf 0 gesetzt. Innerhalb dieses Schleifenkopfs ist die Variable `'i'`, welche später innerhalb der {}-Klammern erzeugt wird, nicht bekannt. Diese überdeckt nur innerhalb der geschwungenen

Klammern die Schleifenvariable `'i'`, wird aber nach jedem Durchlauf wieder gelöscht, wenn der Ablauf `'}'` in Zeile 9 erreicht.
Dies funktioniert jedoch nicht bei alle Compilern. Z.B. in Visual Studio lässt sich das gezeigte Beispiel nicht kompilieren, aber die Verdeckung globaler Variablen, wie in Listing 6.1 ist ohne Probleme möglich.
In Visual Studio kann man in Funktionen und Klassen keinen Bezeichner doppelt verwenden! Bei Klassen können die Klassenattribute jedoch wie globale Variablen durch lokale Variablen verdeckt werden (siehe 6.3).

Übungsaufgabe *Aufgabe 1*, Seite 233
C++ Lernen und professionell anwenden

6.3. Gültigkeit in Klassen

Klassenattribute (siehe 5.3) besitzen Gültigkeit in der gesamten Klasse, das heißt auch in allen Klassenmethoden. Gleiches gilt für die Methoden einer Klasse. Attribute
Wird innerhalb einer Methode eine gleichnamige Variable deklariert, so wird das Klassenattribut verdeckt. Über den `this`-Zeiger oder den Scope-Operator `::` ist dieses jedoch trotzdem nutzbar:

Listing 6.4: Gültigkeit in Klassen

```
1  class Auto {
2  private:
3    int psZahl;    // PS
4    int maxSpeed; // km/h
5  public:
6    Auto() {
7      this->psZahl = 40; // this-Zeiger
8      maxSpeed = 140;
9    }
10   // lokales maxSpeed verdeckt das gleichnamige Klassenattribut
11   void setMaxMeterPerSecond( int maxSpeed ) {
12     // mit dem :: Operator kann auf das Klassenattribut ↩
         zugegriffen werden, auch this->maxSpeed
13     Auto::maxSpeed = ( int ) ( maxSpeed * 1.2 );
14   }
15 };
```

`Auto::` sorgt dafür, dass `maxSpeed` nicht im aktuellen Gültigkeitsbereich gesucht wird. Stattdessen wird `maxSpeed` als Element des Klassen-Scopes interpretiert und damit wird das `maxSpeed`-Attribut der Klasse gefunden.

Kapitel 7

Dynamische Speicherverwaltung

7.1. Motivation

Im letzten Kapitel haben Sie erfahren, dass Objekte einen Gültigkeitsbereich haben. Sie werden beim Eintritt in diesen Bereich erzeugt und beim Austritt wieder gelöscht. Der Speicherbereich, in dem diese so genannten statischen Variablen existieren heißt *Stack*. Neue erzeugte Variablen werden auf diesen Stack (=*Stapel*) gelegt, überdecken eventuell unter ihnen liegende Variablen und werden schließlich wieder vom Stapel genommen. Die Größe dieses Stacks ist während des Programmablaufs konstant und wird durch den Compiler aus der Anzahl der statischen Variablen errechnet.

Selbstverwalteter Speicher

Dies führt bei einer großen Anzahl von statischen Variablen zu Problemen, wenn die Variablen vielleicht nur punktuell einmal benötigt werden, jedoch für einen ständig überdimensionalen Stack sorgen. Außerdem ist es oft nötig die Lebensdauer von Variablen selbst zu bestimmen und sich nicht den automatischen Gültigkeitsbereichen zu unterwerfen.
Aus diesen Gründen existiert aus Sicht des Programms neben dem automatisch verwalteten Stack mit fester Größe noch der nicht verwaltete, sehr viel größere Heap. Dieser Speicher benötigt einiges mehr an Verwaltungsaufwand und ist auch langsamer als der Stack, unterliegt jedoch keinen Einschränkungen. Dynamische Speicherverwaltung bezeichnet die Arbeit mit dem Heap als Speicherform und wird das Thema dieses Kapitels sein.

7.2. `new` und `delete`

Speicher reservieren

Um Speicher auf dem Heap anzulegen, benutzt man den `new`-Operator mit folgender Syntax:

```
Typ* ZeigerAufTyp = new Typ;
```

Speicher freigeben

Den so angelegten Speicher gibt man mit dem `delete`-Operator wieder frei:

```
delete ZeigerAufTyp;
```

Oder als praktisches Beispiel:

```
1 int* dynamischeVariable = new int;
2 // ... ein bisschen Code
3 delete dynamischeVariable;
```

Dieser Code reserviert mit dem `new`-Operator den Speicherplatz für eine `int`-Variable auf dem Heap und speichert die Adresse dieses Bereichs im Zeiger `dynamischeVariable`. In der folgenden Zeile wird die so erzeugte `int`-Variable mit dem `delete`-Operator wieder gelöscht.
Besonders praktisch ist die dynamische Speicherverwaltung im Zusammenhang mit Arrays, bei denen so die Größe dynamisch gewählt werden kann:

Listing 7.1: new und delete mit Arrays

```
1  int main() {
2    int arrayGroesse;
3    cout << "Wie groß soll Ihr Array werden?" << endl;
4    cin >> arrayGroesse;
5
6    int* zeigerAufArray = new int[arrayGroesse];
7    // ...
8    delete[] zeigerAufArray;
9    // ...
10 }
```

In diesem Beispiel wird ein Array mit vom Benutzer vorgegebener Größe zur Laufzeit erzeugt. Der `new`-Operator liefert nur einen Zeiger auf das erste Element des Arrays, so dass hier die Zeigerarithmetik Anwendung findet. Wichtig ist außerdem, dass zum Löschen eines dynamischen Arrays `[]` hinter den `delete`-Operator gesetzt werden muss. Ansonsten wird nur das erste Element des dynamischen Arrays gelöscht, nicht jedoch alle Elemente.

7.3. Gültigkeitsbereich dynamischer Variablen

Wie bereits erwähnt unterliegen dynamisch erzeugte Variablen nicht den üblichen Gültigkeitsbereichen. Eine dynamisch erzeugte Variable lebt von dem Zeitpunkt ihrer Erzeugung mit dem `new`-Operator, bis sie mit dem `delete`-Operator gelöscht wird. Nützlich ist dies in Fällen, wie dem folgendem:

Listing 7.2: Problem mit dem Gültigkeitsbereich

```
1  int* gibZeigerAufSumme( int zahl1, int zahl2 ) {
2    int summe = zahl1 + zahl2; // addieren
3    int* pointer = &summe; // Zeiger auf Summe
4    return pointer; // Zeiger zurückgeben
5    // hier wird summe automatisch zerstört
6  }
7
8  int main() {
9    int* meinPointer =  gibZeigerAufSumme( 10, 4 );
10   // meinPointer zeigt jetzt auf ein zerstörtes Objekt
11   // ...
12   int meineSumme = *meinPointer; // meinPointer wird ↩
         dereferenziert
13   cout << meineSumme;
14   return 0;
15 }
```

Das Programm (7.2) gibt nicht wie erwartet den Wert `14` aus - tatsächlich ist es zufällig, was das Programm ausgibt. Die Rückgabe eines Zeigers auf eine lokale Variable stellt das Problem dar. Letztere wird nämlich beim Verlassen automatisch gelöscht und somit zeigt der Zeiger ins Leere.
Erzeugt man die Variable stattdessen dynamisch, verhält sich das Programm wie erwartet:

```
1  int* gibPointerAufSumme( int zahl1, int zahl2 ) {
2    int* summe = new int; // summe dynamisch erzeugen
3    *summe = zahl1 + zahl2; // addieren
4    return summe; // summe zurückgeben
5    // hier wird summe automatisch zerstört,
6    // nicht jedoch der Wert auf den summe zeigt
7  }
8
9  int main() {
10   int* meinPointer =  gibPointerAufSumme( 10, 4 );
11   // meinPointer zeigt jetzt auf ein existierendes Objekt
12   ...
13   int meineSumme = *meinPointer; // meinPointer wird ↩
         dereferenziert
14   cout << meineSumme;
15   delete meinPointer; // Speicher freigeben
16   return 0;
17 }
```

Im Gegensatz zum vorhergehenden Beispiel ist die Ausgabe dieses Programms `14`. Hier wird in `gibPointerAufSumme()` die `int`-Variable `summe` dynamisch erzeugt und lebt somit auch nach Ende der Funktion weiter. Somit kann ihr Wert in der `main()`-Funktion ausgegeben werden und sie muss vor Programmende gelöscht werden.

Listing 7.3: dynamic2d.cpp

```
1  int main(void) {
2    // 2-dimensionales Array als Pointer auf andere Pointer ↩
       deklarieren
3    int **matrix;
4
5    ...
6
7    // matrix als Array von int-Pointern definieren
8    matrix = new int*[3];
9    // matrix[.] als Pointer auf ein Array mit zwei Integer-↩
       Variablen definieren
10   for(int i = 0; i < 3; ++i)
11     matrix[i] = new int[2];
12
13   ...
14
15   // Speicher in umgekehrter Reihenfolge wieder freigeben
16   for(int i = 0; i < 3; ++i)
17     delete [] matrix[i];
18   delete [] matrix;
19
20   return 0;
21 }
```

Im obigen Beispiel wird gezeigt wie man mit Hilfe der dynamischen Speicherverwaltung ein 2-dimensionales Array erzeugen kann. Ein 2-dimensionales Array ist im Grunde ein Pointer, der auf ein Array von Pointern verweist, die dann auf ein "normales" Array von Variablen zeigen. Besonders wichtig ist, dass die Freigabe der Pointer sozusagen von Innen nach Außen erfolgen muss, d.h. die Dimensionen müssen in umgekehrter Reihenfolge mit `#delete[]` freigegeben werden wie sie allokiert wurden.

!!!

Hinweis zur dynamischen Speicherverwaltung

Trotz ihrer Vorteile ist die dynamische Speicherverwaltung nicht problemlos anwendbar. Wenn eine dynamische Variable erzeugt wird, muss diese auch wieder gelöscht werden. Dies stellt vor allem bei komplexen Programmabläufen ein Problem dar. Wird eine dynamische Variable nicht gelöscht, obwohl ihre Adresse verloren geht, entsteht ein sogenanntes *memory leak*. So nennt man das Phänomen, dass sich mit der Laufzeit des Programms ein immer größerer Speicherbedarf ausbildet, der jedoch nicht mit wirklichen Anforderungen zusammenhängt, sondern durch das nicht vollständige Wiederfreigeben von Speicher verursacht wird.

memory leak

Auch das Gegenteil zum memory leak ist ein Problem, nämlich wenn noch benötigte Werte gelöscht werden. Das Programm versucht dann über die Adresse auf Speicher zuzugreifen, in dem nichts mehr gespeichert ist und stürzt folglich ab oder liefert fehlerhaftes Verhalten.

Zur weiteren Lektüre	Eine weit über den Bedarf dieses Kompendiums oder der Vorlesung Informationstechnik hinausgehende Einführung in die Speicherverwaltung in C++ bietet der C++ Primer in Kapitel 18.1 ab Seite 873.

Übungsaufgabe C++ Lernen und professionell anwenden	Die Übungsaufgaben aus *C++ Lernen und professionell anwenden* auf Seite 489 gehen auch über den Horizont dieser Erläuterungen hinaus.

Kapitel 8
Objektorientierung

Programmierparadigma

Das Programmierparadigma der *Objektorientierung* beruht darauf, reale Objekte oder dem Menschen naheliegende Gedankenkonstrukte in Programmstrukturen möglichst sinngemäß abzubilden. Ein *Objekt* soll, wie in der Realität, auch in der Programmierung etwas sein, das gehandhabt wird, Eigenschaften und Fähigkeiten hat und mit anderen Objekten in Beziehung steht.
Klassen stellen in modernen Softwareprojekten oft die Grundeinheit der Codegliederung dar. Der Grund sind nicht nur die Verwendung als erweiterter Datentyp, sondern vor allem die Möglichkeiten hinsichtlich der objektorientierten Programmierung. In diesem Kapitel lernen Sie die praktische Umsetzung der bereits aus der Vorlesung bekannten Konzepte der Objektorientierung kennen.

Zur weiteren Lektüre	Objektorientierung ist auch ein auf dem Bereich der theoretischen Informatik ergiebiges Feld. Als Einstieg für den Interessierten sei hier durchaus der C++ Primer, kompletter IV. Teil (Kapitel 15-16) empfohlen, der jedoch die theoretischen Aspekte nur streift, dafür einen guten Überblick über die verwendbaren Techniken gibt.

8.1. Kapselung

Ein wichtiges Prinzip der Objektorientierung ist die Kapselung von Daten und Methoden in ein Objekt. Mit der Kapselung einher geht immer das Verstecken von Daten und die Bereitstellung einer Schnittstelle. Eine Klasse präsentiert nicht alle ihre Methoden und Attribute nach außen hin (wie stark vereinfachend im Kapitel 5 *Erweiterte Datentypen* gezeigt), sondern erlaubt nur Zugriff auf die für den Nutzer relevanten Bestandteile. In C++ regeln die drei Schlüsselwörter `public`, `private` und `protected` die Zugriffsrechte auf Elemente einer Klasse.

`public`, `private`, `protected`

- `public` kennen Sie bereits aus dem Kapitel *Erweiterte Datentypen*, es bewirkt freien Zugriff auf alle `public` deklarierten Elemente der Klasse, von außer- und innerhalb der Klasse.
- `private`-Klassenelemente sind nur innerhalb der Klasse sichtbar, zu der sie gehören. Für alle anderen Nutzer der Klasse, auch für abgeleitete Klassen, sind sie unsichtbar und es kann nicht auf sie zugegriffen werden.
- Die als `protected` deklarierten Attribute oder Methoden sind von außerhalb nicht sichtbar, innerhalb der Klasse und jeder abgeleiteten Klasse sind sie sichtbar (seihe 8.3).

Getter und Setter

Grundsätzlich sollte ein Programmierer so wenig Klassenelemente wie möglich nach außen sichtbar machen. Aus diesem Grund ist es gängige Praxis, dass alle Attribute einer Klasse als `private` deklariert werden und ihre Werte über *get*- und *set*-Methoden verfügbar sind. *get*- und *set*-Methoden sind `public` Methoden, welche entweder den Wert eines Klassenattributes zurückliefern (*get*) oder diesen ändern (*set*).

Listing 8.1: Beispiel zur Kapselung

```
1  class Auto {
2    private: // geschützte Attribute & Methoden
3      int psZahl;
4      Marke marke;
5      short baujahr;
6      int geschwindigkeit;
7
8    public: // ungeschützte Attribute & Methoden
9      void beschleunigen( int betrag );
10     int bremsen();
```

```
11
12      int getPsZahl();
13      void setMarke( Marke m );
14      short getBaujahr();
15      void setBaujahr( short jahr );
16  };
```

Hier ist die bereits bekannte Autoklasse nun ordentlich gekapselt dargestellt und besitzt 4 *get-set*-Methoden. Deren Definition liegt auf der Hand, denn sie sollen nur Werte setzen oder zurückliefern:

Listing 8.2: get- und set-Methoden

```
1 // Definition der Klasseneigenen Methoden
2 int Auto::getPsZahl() {
3   return psZahl;
4 }
5
6 void Auto::setMarke( Marke m ) {
7   marke = m;
8 }
9 // ...
```

Zur weiteren Lektüre	Kapselung als Konzept wird behandelt in Teil III des C++ Primer, speziell Kapitel 12.1, Seiten 512ff

8.2. Konstruktoren und Destruktoren

Ein Konstruktor ist eine Methode, die aufgerufen wird sobald ein neues Objekt erzeugt wird. Der Name des Konstruktors entspricht dem Klassennamen, für unsere Autoklasse ist der Konstruktor also `Auto()`. In der Regel wird der Konstruktor genutzt um die Attribute der Klasse zu initialisieren, es ist aber beliebiger Code erlaubt. Wird kein Konstruktor explizit definiert, wird vom Compiler automatisch der Default-Konstruktor eingebunden.

Ein Destruktor wird normalerweise genutzt um durch das Objekt belegten Speicher wieder freizugeben, auch hier ist wieder beliebiger Code erlaubt. Er wird aufgerufen, sobald das Objekt zerstört wird (durch einen Aufruf von `delete`, wenn das Objekt den Gültigkeitsbereich verlässt oder wenn das Programm beendet wird). Jede Klasse besitzt auch einen Default-Destruktor, der für den Einstieg in C++ vollkommen ausreichend ist.

Zur weiteren Lektüre	Konstruktoren und Destruktoren werden behandelt in Teil III des C++ Primer, speziell Kapitel 12.4 und 13.1 bis 13.3

Listing 8.3: Beispiel zur Verwendung von Konstruktoren

```
1  class Auto {
2    private:
3      int psZahl;
4      Marke marke;
5      short baujahr;
6      int geschwindigkeit;
7
8    public:
9      Auto() {
10       psZahl = 120;
11       marke = audi;
12       baujahr = 2010;
13       geschwindigkeit = 0;
14     }
15     Auto( short baujahr, Marke marke ) {
```

```
16        psZahl = 120;
17        this->marke = marke;  //this: siehe Kapitel 8.5
18        this->baujahr = baujahr;
19        geschwindigkeit = 0;
20      }
21      ~Auto() {}; //leerer Destruktor
22
23    // weitere Methoden wie oben
24  };
```

In diesem Beispiel gibt es zwei Konstruktoren von denen `Auto()` standardmäßig aufgerufen wird. Soll bei der Erschaffung eines neuen Autoobjekts Marke und Baujahr direkt angegeben werden, kann man den zweiten Konstruktor nutzen.

8.3. Vererbung und Mehrfachvererbung

Vererbung von Eigenschaften und Fähigkeiten

Vererbung ermöglicht nicht nur eine Reduktion des Quellcodes, sondern auch die Erstellung einer logischen Hierarchie im Code. Eine Basisklasse vererbt der abgeleiteten Klasse all ihre *public* Methoden und Attribute. Auf die *private* Methoden und Attribute der Basisklasse hat die Kindklasse aber genauso viel Zugriff wie allgemeine Nutzer der Klasse, nämlich keinen. Um nun abgeleiteten Klassen den Zugriff zu ermöglichen aber allgemeinen Benutzern der Klasse den Zugriff zu verwehren kann man den Zugriffspezifizierer *protected* nutzen. Mit folgernder Syntax kann eine Klasse von einer Basisklasse erben:

Listing 8.4: Syntax der Vererbung

```
1  class erbendeKlasse : public basisKlasse {
2    // Attribute und Methoden
3  };
```

oder am Beispiel unseres Autos:

Listing 8.5: Vererbung am Beispiel Auto

```
1  class Hybridauto : public Auto {
2    private: // geschützte Attribute & Methoden
3      int batterieLadeZustand;
4
5    protected: // Attribute & Methoden auf die auch abgeleitete ↩
       Klassen Zugriff haben sollen
6      string batterieKapazitaet;
7
8    public: // ungeschützte Attribute & Methoden
9      Hybridauto(); // Konstruktor
10     virtual ~Hybridauto(); // Destruktor
11
12     void bremsEnergieGewinnen();
13     int getLadeZustand();
14 };
15
16 Hybridauto::Hybridauto() : Auto() { // Definition des ↩
     Konstruktors
17   batterieLadeZustand = 0;
18 }
```

Hier ist die Basisklasse Auto um zwei Methoden und zwei Attribute zur Klasse `Hybridauto` erweitert worden. Wichtig bei der Vererbung ist der Konstruktor der erbenden Klasse, denn an ihn hängt man mit einem `':'` den Aufruf für den Konstruktor der Basisklasse. Auf diese Art können auch Variablen der Basisklasse initialisiert werden. Im obigen Beispiel wird erst der Konstruktor `Auto()` aufgerufen und dann der Konstruktor `Hybridauto`. Also beinhaltet ein Objekt vom Typ `Hybridauto` ein Objekt vom Typ `Auto` als Unterobjekt und es ist möglich, verschiedene abgeleitete Klassen auf ihre Basisklasse zurückzuführen (siehe Beispiel 8.7). Der Befehl *virtual* wird in Abschnitt 8.4 ausführlicher erläutert.

Zur weiteren Lektüre	Vererbung ist Thema des kompletten 15. Kapitels des C++ Primers

Übungsaufgabe C++ Lernen und professionell anwenden	*Aufgabe 1, 2, 3*, Seite 543, 545

8.3.1. Mehrfachvererbung

Eine Klasse kann auch von mehreren Basisklassen erben. Die Klasse `AmphibienFahrzeug` könnte zum Beispiel von der Klasse `Auto` und von der Klasse `Schiff` erben. Syntaktisch ergibt sich keine große Neuerung gegenüber einfacher Vererbung:

Listing 8.6: Mehrfachvererbung

```
1 class erbendeKlasse : public basisKlasse1, public basisKlasse2 ↩
    {
2   // Attribute und Methoden
3 };
```

Oder anhand des Amphibienfahrzeugs:

Listing 8.7: Beispiel zur Mehrfachvererbung

```
1  class Schiff {
2    public:
3      Schiff(); // Konstruktor
4      ~Schiff(); // Destruktor
5      void schwimmen();
6  };
7
8  class AmphibienFahrzeug : public Schiff, public Auto {
9    // ...
10 };
```

Zur weiteren Lektüre	Mehrfachvererbung ist ein mächtiges Werkzeug, kann aber auch diverse Probleme bereitstellen. Wer in einem unübersichtlichen Projekt Mehrfachvererbung einsetzen möchte, möge vorher zumindest den C++ Primer, Kapitel 17.3 (Seiten 850ff) lesen. Absolut unerlässlich ist auch das folgende Kapitel 8.4 des Kompendiums!

8.4. Polymorphie und virtuelle Funktionen

Polymorphie (ungefähr *Vielgestaltigkeit*) ist ein wesentlicher Aspekt moderner Objektorientierung. Hinter ihr steht der Gedanke, dass verschiedene abgeleitete Klassen einer Basisklasse verschiedene Aufgaben erfüllen sollen. Das Programm möchte aber einfach nur immer die Methode mit dem überall gleichen Namen aufrufen.

Jede Tochterklasse ist eine Form der Basisklasse

Es steht dabei im Zentrum, dass jede abgeleitete Klasse letztendlich auch eine Form der Basisklasse ist. Daher kann man an jeder Stelle, wo beispielsweise eine Funktion eine Variable vom Typ der Basisklasse fordert, eine Variable vom Typ einer von dieser Basisklasse abgeleiteten Klasse verwenden.

Grundsätzlich ermöglicht es C++ dem Programmierer, in seinen abgeleiteten Klassen stets die Methoden der Mutterklasse neu zu definieren. Allerdings wird standardmäßig nicht dafür gesorgt, dass ein Programm, das eine Variable vom Basistyp verwendet, überprüft, ob es sich bei der Variable nicht tatsächlich um eine Instanz einer abgeleiteten Klasse handelt. Es wird dann die Methode der Basisklasse aufgerufen, nicht die gleichnamige (und mit dem selben Prototypen, siehe 4.4, versehene) Methode der Kindklasse.

Damit dies ohne manuelle Überprüfung möglich ist, bietet C++ das `virtual`-Schlüsselwort. Eine mit `virtual` gekennzeichnete Methode wird immer automatisch aus der jeweiligen abgeleiteten Klasse verwendet.

Listing 8.8: Beispiel zur Polymorphie

```
1  class Pflanze {
2    public:
3      Pflanze() {};
4      virtual ~Pflanze() {};
5      void vergammel();
6      virtual int bringeSamen(); // als virtuell definiert
7  };
8  class Blume : public Pflanze {
9    public:
10     void vergammel();
11     int bringeSamen(); // virtuell, weil schon die gleiche ↩
           Methode der Basisklasse virtuell ist
12 };
13 class Baum : public Pflanze {
14   public:
15     int bringeSamen();
16 };
```

Wir haben drei Klassen eingeführt: `Pflanze`, unsere Basisklasse, sowie `Blume` und `Baum`, jeweils von dieser abgeleitet. Nur `bringeSamen()` ist virtuell.

Listing 8.9: Definition polymorpher Methoden

```
17 void Pflanze::vergammel() {
18   cout << "oh ich zerfalle zu Humus" << endl;
19 }
20 int Pflanze::bringeSamen() {
21   cout << "Ich bin eine grundlegende Pflanze und bringe daher ↩
         nur einen Samen" <<endl;
22   return 1;
23 }
24 int Blume::bringeSamen() {
25   cout << "Ich habe geblueht. Viele Samen." << endl;
26   return 400;
27 }
28 void Blume::vergammel() {
29   cout << "oh nein, meine Schoenheit ist dahin!" << endl;
30 }
31 int Baum::bringeSamen() {
32   cout << "ich habe Samen geworfen." << endl;
33   return 5000;
34 }
```

Wir haben nun verschiedene Methoden, die schon deklariert wurden, implementiert. Diese werden nun verwendet.

Listing 8.10: Verwendung polymorpher Klassen

```
1  int main() {
2    Pflanze* einfachePflanze = new Pflanze();
3    Baum* meinBaumBernd = new Baum();
4    Blume* narzisse = new Blume();
5    Pflanze* irgendeinePflanze = NULL;
6    irgendeinePflanze = narzisse; // Das ist richtig, da eine ↩
         Blume ja eine Pflanze ist!
7
8    cout << "Bernd sagt beim Samen werfen: ";
9    meinBaumBernd->bringeSamen(); // meinBaumBernd ist ein Baum*,↩
         also wird auch Baum::bringeSamen() aufgerufen.
10   cout << "Die Narzisse sagt beim Samen werfen: ";
```

```
11   narzisse->bringeSamen(); // Hier wird Blume::bringeSamen() ↩
         aufgerufen.
12   cout << "Irgend eine Pflanze, die in Wirklichkeit aber eine ↩
         Blume ist, sagt beim Samen werfen: ";
13   irgendeinePflanze->bringeSamen(); // Hier wird Blume::↩
         bringeSamen() aufgerufen, da Pflanze->bringeSamen() ↩
         virtual ist.
14
15   cout << "Die Narzisse verwelkt: ";
16   narzisse->vergammel(); // ist Blume*, daher Blume::vergammel↩
         ();
17   cout << "Irgend eine Pflanze, die in Wirklichkeit aber eine ↩
         Blume ist, verwelkt: ";
18   irgendeinePflanze->vergammel(); // hier wird einfach Pflanze↩
         ::vergammel() aufgerufen, weil für vergammel() nicht per "↩
         virtual" gefordert wurde, dass C++ auf Polymorphie ↩
         überprüft
19   return 0;
20 }
```

Ausgabe:

```
Bernd sagt beim Samen werfen: ich habe Samen geworfen.
Die Narzisse sagt beim Samen werfen: Ich habe geblueht. Viele Samen.
Irgend eine Pflanze, die in Wirklichkeit aber eine Blume ist, sagt beim
Samen werfen: Ich habe geblueht. Viele Samen.
Die Narzisse verwelkt: oh nein, meine Schoenheit ist dahin!
Irgend eine Pflanze, die in Wirklichkeit aber eine Blume ist, verwelkt:
oh ich zerfalle zu Humus
```

Übungsaufgabe — *Aufgabe 1*, Seite 581

C++ Lernen und professionell anwenden

8.5. this-Zeiger

Jede Klasse besitzt einen this-Zeiger, welcher automatisch erzeugt und nicht extra deklariert wird. Dieser Zeiger verweist auf das Objekt selbst, also besitzt jedes Objekt eine Adressvariable mit der eigenen Adresse. Der this-Zeiger ist nützlich um zum Beispiel auf Attribute des Objekts zuzugreifen, welche von lokalen Variablennamen überlagert werden. Oder die Klasse kann sich anhand des this-Zeigers selbst als Argument an eine Funktion übergeben. Ein Beispiel anhand der set-Methode der obigen Autoklasse:

Listing 8.11: Beispiel für den `this`-Zeiger

```
1 void Auto::setMarke( Marke marke ) {
2   this->marke = marke;
3 }
```

Hier wird der set-Methode ein Wert mit dem gleichen Namen wie das Klassenattribut `marke` übergeben. `marke = marke` würde keinen Sinn ergeben, da mit `marke` beidesmal die lokale Variable aus dem Methodenkopf gemeint ist. Mit `this->marke` wird klar, dass das Klassenattribut `marke` verwendet werden soll.

8.6. Dateienstruktur

In vielen modernen Softwareprojekten bilden Klassen die grundlegenden Codeeinheiten. Darum ist es gängige Praxis, für jede Klasse ein neues Paar von Dateien anzulegen, nämlich eine Headerdatei (*.h) und eine Quelltextdatei (*.cpp). In die Headerdatei wird nur die Klassendeklaration geschrieben, es werden keine Methoden definiert (nur deklariert), so dass sich der gesamte ausführbare Quellcode in der .cpp-Datei befindet. Dies ermöglicht einfacheres Debuggen mit vielen Compilern und legt eine klare Struktur fest. Um die Deklaration einer Klasse bekannt zu machen, muss einfach nur der entsprechende Header inkludiert werden (siehe auch 4.4).

Kapitel 9

Streams und Dateiverarbeitung

9.1. Streams: Ein- und Ausgabe

C++ verwendet bei der seriellen Kommunikation etwa mit der Konsole oder mit beim Lesen und Schreiben in Dateien das Konzept der Streams. Der Benutzer kann in Streams hineinschreiben und muss sich weder darum kümmern, wie die Daten letztendlich dort ankommen, wo sie hinsollen, noch darum, in welchen speziellen Stream er schreibt. So kann ein Programm, das normalerweise auf die Konsole ausgibt, ohne wesentliche Anpassung auch in Dateien ausgeben. Die STL (Kapitel 10, Seite 51) stellt einige Streams zur Verfügung. Diese wollen wir nun etwas genauer betrachten.

9.1.1. Der `ostream`

Der *output stream* dient dem Programmierer dazu, Objekte auszugeben. Die Syntax ist dabei immer die gleiche, und insbesondere schon bekannt: `std::cout` ist ein solcher `ostream`.

```
meinostream << "text" << stringklasse;
```

Um seine Daten per ostream ausgeben zu können, muss ein Programmierer für seine Klassen nur den << Operator entsprechend überladen. Hierzu siehe Anhang C. Wie erkenntlich, lässt sich die Ausgabe auch mittels mehrerer << verketten, um so mehrere Objekte hintereinander ausgeben zu können. Längere mehrzeilige Ausgaben müssen nicht unbedingt in jeder Codezeile mit einem ; abgeschlossen werden, sondern können in der folgenden Zeile fortgesetzt werden.

Umgang mit ostreams

Wichtige Methoden, um den Status eines ostreams zu überprüfen sind `ostream.good()` und `eof()` (*End Of File*).

9.1.2. `ofstream`: Ausgabe in Datei

Ein *Output File Stream* erlaubt es dem Programmierer, Daten formattiert in eine Datei zu schreiben.

Hinweis	Für `ofstream` und `ifstream`, müssen Sie mittels `#include <fstream>` die Dateistream-Bibliothek laden.

!!!

Listing 9.1: Beispiel für `ofstream`

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main() {
6    ofstream datei( "zahlen.txt" );
7    datei << "Die Zahlen von 1 bis 20 lauten: ";
8    for( int i = 1; i <= 20; i++ ) {
9      datei << i << " ";
10   }
11   datei << "Und das wars auch schon!" << endl;
12   datei.close();
13   return 0;
14 }
```

Das Programm erstellt selbst die Textdatei `zahlen.txt`, sofern sie noch nicht existiert. Nach dem Aufruf des Programms enthält die Datei folgenden Inhalt:

```
1  Die Zahlen von 1 bis 20 lauten: 1 2 3 4 5 6 7 8 9 10 11 12 13 ↩
       14 15 16 17 18 19 20 Und das wars auch schon!
```

Zu beachten ist, dass durch die Konstruktion mit Dateiname die Datei zum einfachen Ausgeben geöffnet wird, was bedeutet, dass ehemaliger Inhalt gelöscht wird! Andere Schreibmodi wie das Anhängen lassen sich durch weitere Paramater des Konstruktors vereinbaren, siehe `http://www.cplusplus.com/reference/iostream/ofstream/ofstream/`.
Um beispielsweise jede Änderung am Ende der Datei zu schreiben, muss man Zeile 1 im Listing 9.1 folgendermaßen abändern:

```
ofstream datei( "zahlen.txt", ios::app );
```

9.1.3. `istream`: Einlesen von Daten

Analog existiert ein *Input Stream*. Aus diesem werden Daten mittels >> in Variablen gelesen. Der `istream` kümmert sich dabei selbst darum, die enthaltenen Daten in den für die angegebene Variable notwendigen Typ zu konvertieren. Ein bekanntes Beispiel ist die Standardeingabe `cin`.

9.1.4. `ifstream`: Einlesen aus Dateien

Analog zum `ofstream` funktioniert der `ifstream` zum Einlesen von Dateien. Auf jedem Fall sollte vor dem wiederholten Einlesen von Daten überprüft werden, ob der Stream sich in einem brauchbaren Zustand befindet, hierzu verwendet man `good()` und `eof()`.

Listing 9.2: Beispiel zum Einlesen von Dateien mit `ifstream`

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std;
5
6  int main() {
7    int zahl;
8    cout << "Bitte geben Sie eine ganze Zahl ein. Ihnen wird das ↩
       Doppelte berechnet" << endl;
9    cin >> zahl;
10   cout << "Das Doppelte Ihrer Eingabe ist " << zahl * 2 << endl↩
       ;
11   cout << "Wir werden nun versuchen, " << zahl * 2 << " Worte ↩
       einzulesen" << endl;
12   ifstream in( "zahlen.txt" ); // Textdatei einlesen
13   string* worte = new string[zahl*2]; // Dynamisches Array
14
15   for ( int i = 0; i < zahl * 2 && !in.eof() && in.good(); i++)↩
       {
16     in >> worte[i];
17     cout << worte[i] << " ";
18   }
19
20   cout << endl;
21
22   delete[] worte;
23
24   return 0;
25 }
```

Die Ausgabe könnte beispielsweise so aussehen:

```
Bitte geben Sie eine ganze Zahl ein. Ihnen wird das Doppelte berechnet
4
Das Doppelte Ihrer Eingabe ist 8
Wir werden nun versuchen, 8 Worte einzulesen
Die Zahlen von 1 bis 20 lauten: 1
```

Zeile 12 öffnet die Textdatei `zahlen.txt`, in Zeile 15 wird mit einer for-Schleife der `ifstream` in das Array `worte` gespeichert und das jeweils aktuelle Element des Arrays ausgegeben. Die Bedingung `!in.eof()` bewirkt, dass nicht über das Dateiende hinaus gelesen wird. Wenn also

die Datei nur 5 Worte enthält, man aber 10 Worte auslesen will, gibt `in.eof()` nach dem 5. Wort `true` zurück, somit ist die Bedingung der for-Schleife `false`.
Mit `in.good()` wird überprüft, ob die letzte Aktion, die mit dem Stream erledigt wurde, korrekt verlaufen ist. Wenn also beispielsweise ein Lesefehler auftritt, gibt `in.good()` `false` zurück, somit wird die for-Schleife abgebrochen. Anstatt `if( in.good() )` kann man auch kurz `if( in )` schreiben.

Zeilenweises Einlesen

Oftmals ist es nötig, eine ganze Zeile einer Textdatei auszulesen und diese dann mittels Stringoperationen (5.4, Seite 30) zu verarbeiten.
Dieses Vorgehen wollen wir an einem kleinen Beispiel betrachten:
Es existiert die Datei `studenten.txt` mit folgendem Inhalt:

```
# Datenbanksdatei für Teilnehmer des Praktikums "Informationstechnik"

[hertz]
matrikelnummer:1153863
name:Hertz,Heinrich Rudolf
```

Listing 9.3 soll die Datei `studenten.txt` auswerten und die Daten in ein neues Objekt der Klasse `Student` schreiben.

Listing 9.3: Beispiel zum zeilenweisen Einlesen

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

int main() {
  string kurzbezeichnung, nachname, vorname, matrikelnummer;
  string datei = "studenten.txt";
  ifstream in( datei.c_str() ); // Datei öffnen
  if( in ) { // öffnen erfolgreich?
    string zeile;
    while( !in.eof() ) { // Sind noch Daten vorhanden?
      getline( in, zeile );// Zeilenweise einlesen
      size_t raute = zeile.find( "#" );
      size_t eckeAuf = zeile.find( "[" );
      size_t doppelpunkt = zeile.find( ":" );
      if( eckeAuf != string::npos && raute == string::npos ) {
        int eckeZu = zeile.find( "]" );
        kurzbezeichnung = zeile.substr( eckeAuf + 1, eckeZu - eckeAuf - 1 );
      }
      else if( doppelpunkt != string::npos && raute == string::npos ) {
        string wort = zeile.substr( 0, doppelpunkt );

        if( wort == "matrikelnummer" ) {
          matrikelnummer = zeile.substr( doppelpunkt + 1 );
        } else if( wort == "name" ) {
          nachname = zeile.substr( doppelpunkt + 1, zeile.find( "," ) - doppelpunkt - 1 );
          vorname = zeile.substr( zeile.find( "," ) + 1 );
        }
      }
    }

```

```
34      cout << "Kurzbezeichnung: " << kurzbezeichnung << endl
35           << "Matrikelnummer: "  << matrikelnummer  << endl
36           << "Nachname: " << nachname << endl
37           << "Vorname: "  << vorname  << endl;
38    } else {
39      cout << "Fehler beim Oeffnen der Datenbank-Datei" << endl;
40    }
41  }
```

Die Ausgabe ist:

```
Kurzbezeichnung: hertz
Matrikelnummer: 1153863
Nachname: Hertz
Vorname: Heinrich Rudolf
```

Zeile 13 sorgt dafür, dass `in` für zeilenweise in einer Schleife eingelesen wird und die aktuelle Zeile im String `zeile` gespeichert wird.

Zeile 14 überprüft, ob die Zeile lesbar ist. Falls dies nicht der Fall ist, wird die Fehlermeldung in Zeile 33 ausgegeben. Dies wäre dann ein Fehler auf dem Datenträger, da die Datei zwar existiert (wurde in Zeile 12 überprüft) und geöffnet werden kann, sie aber nicht lesbar ist.

Zeile 15 - Zeile 31 verarbeiten die eingelesene Zeile.

Übungsaufgabe — *Aufgabe 1, 2, 3*, Seite 415
C++ Lernen und professionell anwenden

9.1.5. `stringstream`: Die Lösung für Stringumwandlungen

Häufig steht der Programmierer vor dem Problem, verschiedene Daten, die in Zeichenketten gespeichert sind, in andere Formate, vor allem Zahlen, umzuwandeln. Zu diesem Zweck existieren die Stringstreams. Sie können mit allen Arten von Daten gefüttert werden, für die der Stream-Ausgabeoperator << existiert, und können jede Art von Variable füllen, für die ein >> Stream-Eingabeoperator definiert ist.

Listing 9.4: Umwandlung mittels `stringstream`

```
1  #include <iostream>
2  #include <sstream>
3  using namespace std;
4
5  int main() {
6    stringstream strstr;
7    strstr << "12 25 49 99 200";
8    float wert1, wert2;
9    strstr >> wert1;
10   for( int i = 0; i < 4; i++ ) {
11     strstr >> wert2;
12     cout << "Quotient " << i << '-' << i + 1 << ": " << ( wert2↩
           / wert1 ) << endl;
13     wert1 = wert2;
14   }
15   return 0;
16 }
```

gibt aus:

```
Quotient 0-1: 2.08333
Quotient 1-2: 1.96
Quotient 2-3: 2.02041
Quotient 3-4: 2.0202
```

Kapitel 10
Standard Template Library

Die *Standard Template Library* gehört zum Lieferumfang jeder C++ Entwicklungsumgebung. Neben dem in 5.3.1 schon angesprochenen Typen String unterstützt sie den Programmierer mit diversen Containern, die es ermöglichen, viele Objekte des selben Typs dynamisch zu verwalten. Außerdem bietet die STL Streams, um Ein- und Ausgabe und auch Typwandlung zu ermöglichen.

Sequenzielle Container
Die STL stellt dem Programmierer drei unterschiedliche Container zur sequentiellen Verwaltung von Elementen zur Verfügung: `vector`, `deque` und `list`. Jeder dieser Container hat eigene Stärken (und Schwächen), und die Wahl zwischen diesen beeinflusst unter Umständen die Performance eines Programms erheblich.

vector ist im Prinzip ein Array (siehe 2.11), das nach hinten schnell wachsen und seine letzten Objekte schnell löschen kann. Dafür sind das Einfügen und das Löschen von Objekten in der Mitte des vectors eher langsam, da dann der komplette vector an eine neue Stelle kopiert wird. Ein Vektor ist genauso schnell wie ein Array beim Zugriff auf ein Element durch dessen Index.

deque ist optimiert darauf, sowohl am Anfang als auch am Ende schnell Objekte hinzufügen zu können. Dafür funktionieren sie intern nicht wie Arrays und sind daher unter Umständen etwas langsamer als vector. Dennoch ist ein Zugriff auf die einzelnen Elemente über deren Index möglich.

list ist eine doppelt verkettete Liste, wie in der Vorlesung vorgestellt. Daher sind Einfüge-, Lösch- und Verschiebeoperationen an beliebiger Stelle sehr schnell, weshalb sich **list** z.B. für Sortieralgorithmen gut eignet. Jedoch besitzen sie keine Möglichkeit, ein bestimmtes Element mittels dessen Index auszuwählen. Hierfür existiert allerdings das Iterator-Konzept (siehe 10.1).

Sequenzielle Containeradapter
Durch die Containeradapter kann einem sequenziellen Container ein bestimmtes Verhalten aufgezwungen werden.

stack lässt einen beliebigen Container funktionieren wie einen LIFO-Stack (Last-in-first-out).

queue funktioniert nicht mit **vector**. Es handelt sich um eine FIFO-Warteschlange (first-in-first-out).

priority_queue benötigt **vector** oder **deque** Container als Basis. Hier wird das Verhalten einer prioritätsgesteuerten Warteschlange aufgezwungen.

Assoziative Container
Die Aufgabe der assoziativen Container ist es, bestimmten Variablen (Werten) bestimmte Elemente zuzuordnen, etwa wie in einem Telefonbuch Telefonnummern zu Namen zugeordnet werden. Es wird effizientes Abrufen und Suchen durch einen Schlüssel unterstützt. Die wichtigsten assoziativen Containertypen sind **map** und **set**.

map enthält Paare aus Schlüssel und Wert, wobei der Schlüssel als eine Art Index fungiert. Es ist nicht möglich einen Wert zusammen mit einem schon vorhandenen Schlüssel einzutragen.

set enthält nur Schlüssel. Es wird am sinnvollsten für eine Sammlung von verschiedenen Werten genutzt. Es ist nicht möglich den gleichen Schlüssel mehrmals hinzuzufügen.

multimap wie **map**. Schlüssel dürfen jedoch mehrfach vorkommen.

multiset wie **set**. Schlüssel dürfen jedoch mehrfach vorkommen.

Zur weiteren Lektüre	Der C++ Primer widmet Containern einen ganzen Buchteil (Teil II), siehe Kapitel 9 und 10 ab Seite 369.

10.1. Iteratoren

Jede Containerklasse hat seinen eigenen Iteratortypen, der den Zugriff auf die Elemente des Containers ermöglicht. Iteratoren stellen eine Alternative zum Zugriff per Index dar, denn jede Containerklasse definiert einen Iteratortypen, auch wenn Indizes, z.B. in `list`, nicht unterstützt werden.

Ein Iterator lässt sich durch

```
1 containertyp<typ>::iterator mein_iterator;
```

erzeugen. Iteratoren unterstützen nun verschiedene Operationen um die Arbeit mit Containern zu erleichtern. In ihrer Verwendung ähneln sie sehr den Zeigern.

`*meinIterator` gibt eine Referenz auf das Element zurück auf das meinIterator zeigt.

`meinIterator->data` dereferenziert `meinIterator->data` und gibt das Datenelement mit Namen `data` aus dem Element auf das `meinIterator` zeigt, zurück.

`++meinIterator` `meinIterator` zeigt nun auf das nächste Element im Container.

`--meinIterator` `meinIterator` zeigt auf das vorherige Element im Container.

`meinIterator1 == meinIterator2` Zwei Iteratoren sind gleich wenn sie auf das selbe Element im selben Container zeigen.

`meinIterator1 != meinIterator2` Zwei Iteratoren sind ungleich wenn sie nicht auf das selbe Element im selben Container zeigen.

Die Iteratoren der Containertypen `vector` und `deque` unterstützen noch zusätzliche Operationen, da diese Container einen schnellen Zugriff auf einzelne Elemente unterstützen:

`meinIterator + n` der Iterator wird um n Elemente verschoben.

`<, <=, >=, >` Ein Iterator ist größer als ein anderer, wenn sich das Element auf das er zeigt im Container weiter hinten befindet.

10.2. sequenzielle Container

10.2.1. vector

Erschaffung

einbinden

Um den Container `vector` überhaupt nutzen zu können, muss er erst eingebunden werden:
`#include <vector>`
Wie jedem Container muss einem `vector` der Typ der Objekte übergeben werden, den er verwalten soll:

neuen `vector` erzeugen

`vector<typ> containerName;`

Zugriff auf Elemente

lieber `at()` benutzen

Zum Zugriff auf einzelne Elemente bietet `vector` die folgenden Funktionen:

`[]` funktioniert wie im Array. `at()` ist identisch, weist aber darauf hin, wenn außerhalb der tatsächlich vorhandenen Elemente gelesen wird.

`containerName.front()` gibt eine Referenz auf das erste Element zurück.

`containerName.back()` gibt eine Referenz auf das letzte Element zurück.

`containerName.begin()` und `containerName.end()` geben Iteratoren auf das erste bzw. letzte Objekt zurück.

`containerName.size()` gibt die Zahl der im Container enthaltenen Elemente zurück.

Einfügen von Elementen

`containerName.push_back( x )` fügt ein Element mit dem Wert `x` am Ende des Containers hinzu. (Schnell!)

`containerName.insert( iter, x )` fügt ein Element mit dem Wert `x` vor dem Element ein, auf das der Iterator `iter` zeigt. Gibt einen Iterator auf das neue Element zurück. (Langsam)

`containerName.insert( iter, n, x )` fügt `n` Elemente mit dem Wert `x` vor dem Element ein, auf das der Iterator `iter` zeigt. (Langsam)

Löschen von Elementen

`containerName.pop_back()` löscht das letzte Element im Container (Schnell!). Hat keinen Rückgabewert.

`containerName.erase( iter )` löscht das Element auf das der Iterator `iter` zeigt und gibt einen Iterator zurück, der auf ein Element hinter dem gelöschten zeigt.

`containerName.erase( iter_a, iter_b )` Löscht alle Elemente im Bereich `iter_a` bis `iter_b`. Gibt einen Iterator zurück, der auf ein Element hinter dem letzten gelöschten Element zeigt.

`containerName.clear()` Löscht alle Elemente im Container.

Achtung! Wird am Container eine Veränderung z.B. durch `insert()` vorgenommen, werden unter Umständen gespeicherte Iteratoren ungültig! !!!

Zur Veranschaulichung ein kurzes Beispiel:

Listing 10.1: Beispiel zur Verwendung von `vector`

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main () {
6    unsigned int i;
7    vector<unsigned int> vektorchen; // vector für unsigned int ↩
       Elemente
8
9    for( i = 1; i <= 10; i++ ) {
10     vektorchen.push_back( i );
11   }
12   vektorchen.erase( vektorchen.begin() + 5 ); // 6. Element ↩
       löschen
13   cout << "vektorchen beinhaltet:";
14
15   for( i = 0; i < vektorchen.size(); i++ ) {
16     cout << " " << vektorchen.at(i);
17   }
18
19   cout << endl;
20   return 0;
21 }
```

gibt aus: `vektorchen beinhaltet: 1 2 3 4 5 7 8 9 10`

10.2.2. deque

Deque weist die selben Methoden wie `vector` auf, unterstützt aber zusätzlich

`containerName.push_front( x )` fügt ein Element mit dem Wert `x` am Anfang des Containers hinzu.

`containerName.pop_front()` löscht das erste Element des Containers. Hat keinen Rückgabewert.

10.2.3. list

List hat die oben erwähnten Vorteile der *Linked List*. Sie unterstützt die folgenden oben bereits erwähnten Methoden:
`front()`, `back()`, `clear()`, `pop_back()`, `pop_front()`, `push_back()`, `push_front()`, `insert()`.

`list` bietet eine Methode zum automatischen sortieren

Zusätzlich wird `sort()` unterstützt, das die Liste sortiert, sofern für die Elemente ein '<' Operator definiert ist. Für eigene Klassen kann der Programmierer diesen per *Operatorenüberladung* selbst spezifizieren (siehe Anhang C). Hier ein einfaches Beispiel zum Umgang mit `list`:

Listing 10.2: Beispiel zu `list`

```
1  #include <iostream>
2  #include <list>
3  #include <string>
4  using namespace std;
5
6  int main(){
7    list<string> kursteilnehmer;
8    kursteilnehmer.push_back("Trillian");
9    kursteilnehmer.push_back("Zaphod");
10   kursteilnehmer.push_back("Ford");
11   kursteilnehmer.push_back("Marvin");
12   kursteilnehmer.push_back("Arthur");
13   kursteilnehmer.sort();
14   kursteilnehmer.push_front("Alf");
15
16   for ( list<string>::iterator iter ( kursteilnehmer.begin() );↩
         iter !=  kursteilnehmer.end() ; iter++ )
17     cout << *iter << endl;
18
19   system("pause");
20   return 0;
21 }
```

gibt aus:

```
Alf
Arthur
Ford
Marvin
Trillian
Zaphod
```

10.3. sequenzielle Containeradapter

Die Containeradapter im Allgemeinen nehmen sich einen beliebigen sequenziellen Container und sorgen dafür, dass er sich verhält wie ein abstrakter Typ. Standardmäßig wird für die Adapter `stack` und `queue` der Container `deque` genutzt, der Adapter `priority_queue` ist jedoch als `vector` implementiert. Ist das unerwünscht besteht die Möglichkeit den Standardcontainertypen zu überschreiben.

```
1  stack< int, vector<int> > int_stack;
2  int_stack( intvec );
```

Jetzt haben wir einen Stack der als `vector` implementiert ist und eine Kopie von `intvec` erhält. Bleibt die Klammer leer oder wird sie weggelassen, wird ein leerer Stack erzeugt. Im Gegensatz zu den Containern ist noch anzumerken, dass jeder Adapter alle relationalen Operatoren(<, >, !=, usw.) unterstützt.

10.3.1. stack

`stk.pop()` löscht das Element das ganz oben auf dem Stack liegt. Es erfolgt jedoch keine Rückgabe.

`stk.top()` gibt das oberste Element zurück.

`stk.size()` gibt die Anzahl der Elemente auf dem Stack zurück.

`stk.push( element )` legt ein neues Element auf dem Stack ab.

`stk.empty()` wenn der Stack leer ist wird `true` zurück gegeben, wenn nicht `false`.

10.3.2. queue

Die `queue` arbeitet nach dem FIFO-Prinzip (first-in, first-out) und unterstützt folgende Operationen:

`que.pop()` löscht das erste Element der Warteschlange. Es erfolgt jedoch keine Rückgabe.

`que.size()` gibt die Anzahl der Elemente in der Warteschlange zurück.

`que.push( element )` fügt ein neues Element am Ende der Warteschlange an.

`que.empty()` wenn die Warteschlange leer ist wird `true` zurück gegeben, wenn nicht `false`.

`que.front()` das erste Element der Warteschlange wird zurückgegeben.

`que.back()` das letzte Element der Warteschlange wird zurückgegeben.

10.3.3. priority_queue

Die `priority_queue` ordnet die Elemente nach ihrer relativen Priorität, die durch den Operator `'<'` bestimmt wird. Es werden abgesehen von `front()` und `back()` alle Operatoren der `queue` unterstützt. Dazu kommt noch der `top()` Operator der das Element mit höchster Priorität zurück gibt.

10.4. assoziative Container

10.4.1. map

Erschaffung

Da die Bibliothek standardmäßig den '<' Operator nutzt um die Schlüssel zu vergleichen, sollte dieser durch den Schlüsseltypen unterstützt werden. Es ist jedoch möglich eine eigene Vergleichsfunktion zu definieren (siehe C++ Primer Abschnitt 15.8.3 Seite 707).
Um den Container `map` nutzen zu können, muss er erst eingebunden werden:

```
#include <map>
```

Wie jedem Container muss einer `map` der Typ der Objekte übergeben werden, den er verwalten soll. Bei der `map` müssen nun aber zwei Typen angegeben werden:

```
map<Schluesseltyp, Werttyp> mapName;
```

`map<Schluesseltyp, Werttyp> mapName( map2 );` Erzeugt `mapName` als Kopie von `map2`. Hier ist zu beachten, dass mapName und `map2` dieselben Schlüssel- und Werttypen haben.

`map<Schluesseltyp, Werttyp> mapName( iter_a, iter_b );` erzeugt `mapName` als Kopie des Bereichs der durch die Iteratoren `iter_a` und `iter_b` angegeben wird.

Zugriff auf Elemente

Ein Zugriff auf die Elemente der `map` ist über `mapName[Schluessel];` möglich, es wir dann der Wert direkt zurückgegeben
Eine zweite Möglichkeit eröffnet `mapName.find( Schluessel );` diese Operation gibt einen Iterator zurück, der auf das zum Schlüssel gehörende Schlüssel-Wert-Paar zeigt.

Elemente hinzufügen

Benutzen wir `mapName[Schluessel];` um ein Element abzufragen und es existiert nicht, wird es automatisch angelegt.
`mapName[Schluessel] = 1;` Weist dem Wert des Schlüssels den Wert 1 zu. Existiert der Schlüssel noch nicht, wird ein neues Schlüssel-Wert-Paar angelegt und der Wert wird mit 1 initialisiert. Die Schlüssel-Wert-Paare sind vom Typ `pair`, dessen Elemente über die Methoden `first()` und `second()` aufgerufen werden können.

`mapName.insert( Schluessel-Wert-Paar )` Ist der Schlüssel nicht in der map enthalten wird das Paar eingefügt. Wenn doch bleibt map unverändert. Schlüssel- und Werttyp des Paares und der map müssen identisch sein.

`mapName.insert( iter_a, iter_b )` Die Iteratoren `iter_a` und `iter_b` grenzen einen Bereich ab, der aus Schlüssel-Wert-Paaren besteht. Schlüssel- und Werttyp des Paares müssen wiederum mit denen von `mapName` übereinstimmen. Jedes Element in diesem Bereich, das noch nicht in mapName vorhanden ist, wird eingetragen.

Löschen von Elementen

`mapName.erase( Schluessel )` Löscht das zum `Schluessel` gehörende Schlüssel-Wert-Paar.

`mapName.erase( iter )` Löscht das Element auf das der Iterator `iter` zeigt.

`mapName.erase( iter_anf, iter_ende )` Löscht alle Elemente zwischen den beiden Iteratoren.

Beispiel:

Listing 10.3: Beispiel zu `map`

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  using namespace std;
5
6  int main () {
7    map<char, string> tierbuch;  // erstellt eine map, die ↩
       Schlüssel sind vom Typ char, die Werte sind strings
8
9    tierbuch['a'] = "Affe";
10   tierbuch['b'] = "Baer";
11   tierbuch['c'] = "Chamaeleon";
12
13   pair< char, string > tierbucheintrag( 'e', "Giraffe" ); // ↩
       Wir legen zuerst ein Schlüssel-Wert-Paar an
14   tierbuch.insert( tierbucheintrag );   // und fügen es dann ↩
       zum Tierbuch hinzu
15
16   map< char, string >::iterator mein_iterator;  //Einen ↩
       Iterator anlegen um die Suche innerhalb der map zu ↩
       ermöglichen
17   mein_iterator = tierbuch.find('b');
18
19   cout << "tierbuch['a'] ist " << tierbuch['a'] << endl
20        << "tierbuch['b'] ist " << tierbuch['b'] << endl
21        << "tierbuch['c'] ist " << tierbuch['c'] << endl
22        << "tierbuch['d'] ist " << tierbuch['d'] << endl
23       // 'd' gibt es als Schlüssel noch nicht, der Aufruf von '↩
           d' legt aber ein solches Schlüssel-Wert-Paar an.
24        << "tierbuch['e'] ist " << tierbuch['e'] << endl;
```

```
25    cout << "Das Tierbuch beinhaltet " << (int) tierbuch.size() ↩
          << " Tiere." << endl
26          << "Der Eintrag 'b' im Tierbuch ist: " << mein_iterator↩
              ->second << endl;
27
28    system("pause");
29    return 0;
30  }
```

gibt aus:

```
tierbuch['a'] ist Affe
tierbuch['b'] ist Baer
tierbuch['c'] ist Chamaeleon
tierbuch['d'] ist
tierbuch['e'] ist Giraffe
Das Tierbuch beinhaltet 5 Tiere.
Der Eintrag 'b' im Tierbuch ist: Baer
```

Auf Elemente der map lässt sich also mit dem `[]` Operator zugreifen. Bei nicht vorhandenen Werten fügt der aber automatisch einen neuen Leerwert hinzu, was nötig ist, damit die Zuweisung `tierbuch['a'] = "Affe"` funktioniert.

nach Schlüssel suchen

Möchte man dieses Verhalten umgehen, kann man, wie oben erwähnt, `find('b')` verwenden, dass bei Nichtexistenz des Schlüssels kein neues Schlüssel-Werte-Paar erzeugt. `find('b')` gibt jedoch nicht direkt den Wert sondern einen Iterator auf das Schlüssel-Wert-Paar zurück.

10.4.2. set

Ein Set unterstützt die selben Operationen wie eine Map, mit dem Unterschied, dass sich in dem Set nur Schlüssel befinden und daher der Indizierungsoperator nicht untertützt wird. Die Schlüssel eines Sets sind nicht mit Werten verknüpft. Ein Set ist immer nur eine Sammlung von Schlüsseln.

10.4.3. multimap und multiset

`multimap` und `multiset` unterstützen im Gegensatz zu `map` und `set` das mehrfache Vorkommen eines Schlüssels. Es werden, mit Ausnahme der Indizierungsoperation, auch dieselben Operationen wie in `map` bzw. `set` unterstützt. Jedoch mit dem Unterschied, dass mehrere Werte gehandhabt werden müssen.

`multiName.insert( Schluessel-Wert-Paar )` fügt das Paar immer hinzu.

`erase( Schluessel )` löscht alle Elemente mit diesem `Schluessel` und gibt die Zahl der gelöschten Elemente zurück.

`erase( iter )` hingegen löscht nur das Element auf das der Iterator `iter` zeigt.

Elemente suchen

In einer `map` oder einem `set` war die Suche noch sehr einfach, da ein Element enthalten sein konnte oder nicht, hier gestaltet es sich in sofern schwieriger, dass mehrere Elemente mit dem gleichen Schlüssel vorhanden sein können.

`multiName.lower_bound( Schluessel )` Es wird ein Iterator auf ein Element zurückgegeben, dessen Schlüssel nicht kleiner ist als `Schluessel`.

`multiName.upper_bound( Schluessel )` Es wird ein Iterator auf das erste Element zurückgegeben, dessen Schlüssel größer ist als `Schluessel`,.

`multiName.equal_range( Schluessel )` wie `lower_bound` und `upper_bound` in einem Operator. Gibt ein Paar aus oberer und unterer Grenze zurück.

Anhang A
Operatoren

Arithmetische Operatoren					
Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Addition / Subtraktion	`+ -`	`a + 1`		Summe bzw. Differenz	7
Multiplikation	`*`	`a * b`		Produkt	6
Division	`/`	`a / b`	Bei ganzzahligen Operanden (z.B. `int`) nur ganzzahlige Division	Quotient	6
Modulo / Rest	`%`	`a % b`		Rest	6

Vergleichsoperatoren (siehe 3.1)					
Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Gleichheit	`==`	`a == 1`	Bei double-Vergleich Ungenauigkeit der internen Darstellung beachten	`true` bei Identität (Bit für Bit), sonst `false`	10
Ungleichheit	`!=`	`a != b`	s.o.	s.o.	10
Größer als / Kleiner als	`> <`	`a < 2`		`true` / `false`	9
Größer gleich / Kleiner gleich	`>= <=`	`a >= 2`		`true` / `false`	9

Zuweisungsoperatoren (siehe 2.4)					
Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Zuweisung	`=`	`a = 2`	**Nicht zum Vergleich einsetzen!** Typen müssen kompatibel sein.	Wert des linken Operanden nach der Zuweisung	17
Kombinierte Zuweisung	`+= -= *= /= %= ...`	`a += 2`	Kurzform für: `a = a + 2`	Wert des linken Operanden nach der Zuweisung	17

Bitoperatoren					
Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Bitshift links / Bitshift rechts	`<< >>`	`a >> n`	Nur für ganzzahlige Typen	Bits des linken Operanden um n Stellen verschoben	8
Bitweise UND / ODER / XOR	`& | ^`	`a & b`	Verwendung bei Bitmasken etc. für Ganzzahlen	Bitweise Verknüpfung der Operanden	& 11, | 13, ^ 12
Bitweise NICHT	`~`	`~a`		Jedes Bit wird umgekehrt	3

Logische Operatoren					
Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Negation	`!`	`!Ausdruck`		Negierter Wert von Ausdruck	3
logisches UND	`&&`	`( a != 0 ) && ( 2 < b / a )`	*Lazy Evaluation:* Die rechte Seite wird nicht mehr ausgewertet, wenn die linke `false` ist	`true` wenn beide Operanden `true` sind	14

logisches ODER	`||`	`a || ( !a & b )`	*Lazy Evaluation:* Die rechte Seite wird nicht mehr ausgewertet, wenn die linke `true` ist	`true` wenn mindestens ein Operand `true` ist	15

Post- und Präfixoperatoren

Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Postfix Inkrement / Dekrement	`++ --`	`for( int a=0; a < 11; a++ )`	Der Operand wird um 1 erhöht/verringert, nachdem der Ausdruck ausgewertet wird	Wert des Operanden vor Inkrementierung bzw. Dekrementierung	2
Präfix Inkrement / Dekrement	`++ --`	`int b = ++a;`	Der Operand wird um 1 erhöht/verringert, bevor der Ausdruck ausgewertet wird	Wert des Operanden nach Inkrementierung bzw. Dekrementierung	3

Zugriffsoperatoren (siehe 2.9)

Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Element-operator	`.`	`Auto.ps, Auto.gibGas()`	Wählt ein Element der Klasse oder struct aus	Wert des Elements bzw. der aufgerufenen Methode	2
Element-operator (Zeiger)	`->`	`Auto* myAuto = new Auto; myAuto->gibGas();`	Wählt aus einem per Pointer referenzierten Objekt ein Element aus	s.o.	2
Dereferenzierung	`*`	`( *myAuto ).ps`	Löst Pointer auf in referenziertes Objekt	Referenziertes Objekt	3
Addressoperator	`&`	`int* zeiger = &variable;`		Adresse der Variable	3
Scope-Zugriffs-Operator	`::`	`std::cout`	Wählt aus einem Namespace ein Objekt aus. Siehe Kap. 6		1
Array-Zugriffs-Operator (s. 2.11)	`[ ]`	`int a = arr[12]`	Wählt aus einem Array das spezifizierte Element aus. Führt keine Längenprüfung im Array durch!	n-tes Element des Arrays	2

Sonstige

Bezeichnung	Operator	Beispiel	Anmerkungen	Rückgabewert	Prior.
Cast-Operator	`( )`	`( int ) a`	Zur Typumwandlung	Variable vom in Klammern spezifizierten Typ mit Wert des rechten Operanden	4
New-Operator	`new`	`Auto* car = new Auto();`	Ruft den Konstruktur der Klasse auf. Siehe 7.2	Erstellt ein neues Objekt auf dem Heap und gibt Zeiger auf dieses zurück	3
Delete-Operator	`delete`	`delete car;`	Gibt Speicher von Objekt auf dem Heap wieder frei		3

Prioritäten sind Listenmäßig zu verstehen, das heißt: Operatoren mit der **niedrigeren** Prioritätszahl werden **zuerst** ausgeführt.
Quelle der hier angegebenen Prioritäten ist die MSDN Library.

Anhang B

Escape-Sequenzen

Einige Zeichen können nicht direkt in Zeichenketten angegeben werden, weil sie entweder nicht direkt *druckbar* sind oder eine besondere Bedeutung im Zusammenhang mit Literalen (siehe 2.6) besitzen. Die wichtigsten finden Sie in folgender Tabelle:

`\a`	alert (BEL)	gibt ein akustisches Signal aus
`\b`	backspace	löscht letztes Zeichen
`\n`	Zeilenvorschub (auf Unix-Systemen: Neue Zeile)	kann bei `cout` durch `endl` ersetzt werden: `cout << "text" << endl`, dadurch wird Systemunabhängigkeit erlangt
`\t`	Horizontaler Tabulator	
`\"`	Doppelte Anführungszeichen	" sind Zeichenketten-Literalbegrenzer
`\'`	Einfache Anführungszeichen	' sind Begrenzer für einzelne `char`
`\0`	0-Byte Zeichen	Abschluß-Zeichen bei C-Strings
`\\`	Backslash	\ ist selbst Präfix für Escape-Sequenzen
`\r`	Wagenrücklauf	Auf einigen Systemen in Verbindung mit `\n` notwendig, um eine neue Zeile anzufangen
`\123`	Das Zeichen Nr. 123_8 im jeweiligen Zeichensatz	Abhängig von Eingabedatei-Zeichenkodierung, häufig UTF-8 oder ISO-8859-1, dies kann aber speziell unter Windows sehr häufig abweichen
`\xfe8`	Wie oben, aber Zeichen nicht oktal, sondern hexadezimal ($fe8_{16}$).	s.o.

Wie oben erwähnt ist die Zeichenkodierung nicht einheitlich. Für Konsolenanwendungen auf neueren Windowssystemen hier eine kleine Sammlung häufig verwendeter Sonderzeichen:

`\x81`	ü
`\x84`	ä
`\x8E`	Ä
`\x94`	ö
`\x99`	Ö
`\x9A`	Ü
`\xe1`	ß

Listing B.1: Beispiel zur Verwendung von `Escape-Sequenzen`

```
1 #include <iostream>
2 using namespace std;
3 int main () {
4 cout << "Escape\x2dSequenzen k\x94nnen f\x81r die Textausgabe ↩
      sehr hilfreich sein. \nEs wird n\x84mlich m\224glich ↩
      verschiedene Sonderzeichen darzustellen.!\b " << endl;
5 }
```

gibt aus:

```
Escape-Sequenzen können für die Textausgabe sehr hilfreich sein.
Es wird nämlich möglich verschiedene Sonderzeichen darzustellen.
```

Anhang C

Dinge von denen nicht die Rede war...

Dieses Kompendium kann nicht alle Aspekte der Programmierung mit C++ abdecken. Daher gibt es hier einen Überblick über die wichtigsten Konzepte, die bewusst nicht behandelt wurden. Diese Liste soll dem geneigten Leser als Anhaltspunkt für weitere Recherche im *C++ Primer* dienen.

Operatorenüberladung Das Ziel des Klassenkonzept ist es, den Umgang mit Klassen möglichst intuitiv zu gestalten. Aus diesem Grund ist es häufig wünschenswert, eigene Operatoren für seine Klassen einzuführen. Etwa um den `<<` Operator von `std::cout` so zu erweitern, dass er auch mit Objekten der eigenen Klasse umgehen kann, oder um einer String-Klasse die Möglichkeit zu geben, zwei Strings per `+=` aneinander zu hängen. Tatsächlich sind Operatoren nichts anderes als Methoden, die bestimmte Objekte akzeptieren. Siehe hierzu: `http://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading`. C++ Primer, Kap. 14, S. 597

Casting Mit *Casting* bezeichnet man die Umwandlung eines Objekts in einen anderen Typ. Es gibt so genannte Casting-Operatoren, die auch überladen werden können. So kann man der eigenen Klasse auch die Möglichkeit geben, etwa in eine Zahl (z.B. eine Prüfsumme) oder eine Zeichenkette (etwa die Beschreibung des aktuellen Objektzustands) umgewandelt zu werden. C++ Primer, Kap. 14.9, S. 632

Konstruktoren Obwohl angerissen, deckt dieses Kompendium bei weitem nicht alle Aspekte von Konstruktoren ab: es gibt so gennante *Copy-Contructors*, die dazu dienen, dem Programmierer die Möglichkeit zu geben, beim Duplizieren eines Klassenobjekts nicht nur automatisch eine bitweise Kopie anzulegen, sondern beispielsweise Prüfnummern anzupassen, oder auch nur als Zeiger referenzierte Objekte zu kopieren. Dies ist beispielweise dann notwendig, wenn sonst vom Originalobjekt aus Versehen etwas per `delete` (siehe Kap. 6 hier im Kompendium) gelöscht wird, was auch noch in einer anderen Instanz benutzt wird.

C++ Primer, Kap. 12.4, S. 535 , C++ Primer, Kap. 13, S. 563

Destruktoren Wie der Name andeutet: C++ bietet die Möglichkeit, eine Methode zu definieren, die immer dann automatisch aufgerufen wird, wenn ein Objekt (entweder automatisch am Ende seiner Gültigkeit, oder per `delete`) zerstört wird. Hier können dann der Speicher von nicht mehr benutzten Objekten befreit, Dateien geschlossen oder sonstige sinnvolle Operationen erledigt werden. Destruktoren sind Methoden, welche wie die Klasse heißen, deren Namen aber '~' vorangestellt ist und welche keine Argumente akzeptieren.

C++ Primer, Kap. 13.3, S. 573

***rein* virtuelle Methoden** Beim Einsatz von Polymorphismus ist es manchmal wünschenswert, eine Klasse zu definieren, die zumindest in einigen Aspekten unvollständig ist. Diese Klasse besitzt dann Methoden, die sie zwar deklariert, aber nicht implementiert sind. Die Implementierung wird den Tochterklassen überlassen, damit diese dabei individuell zutreffende Wege gehen *müssen*. Eine Klasse, die mindestens eine rein virtuelle Methode enthält, wird abstrakte Basisklasse genannt.

Templates Gerade beim Verwalten von Objekten verschiedener Typen wünscht man sich als C++ Programmierer gelegentlich, dass man beispielsweise eine Klasse schreiben könnte, die Objekte eines bestimmten Typs behandelt, ohne dass man diesen Typ vorher kennt. Auch soll diese Klasse dann für einen anderen Typ wieder Verwendung finden können.

C++ hat dafür einen recht komplexen Weg gefunden: *Templates*. Diese verlassen aber das Blickfeld dieses Kompendiums. C++ Primer, Kap. 16, S. 731

Fehlerbehandlung C++ bietet ein Konzept, Fehler beim Programmablauf abzufangen, oder an die aufrufende Funktion weiterzugeben (`catch`/`throw`). `try.../catch()` bieten hier die Möglichkeit, Programme auszuführen, bei denen eine Ausnahmesituation (Beispiel: Puffer voll!) auftreten kann, und für diesen Fall gleich ein Verhalten zu spezifizieren. C++ Primer, Kap. 17.1, S. 804