Programmierung mit C++ für Computerlinguisten

CIS, LMU München

Max Hadersbeck

Mitarbeiter
Daniel Bruder, Ekaterina Peters, Jekaterina Siilivask, Kiran Wallner,
Stefan Schweter, Susanne Peters, Nicola Greth

— Version Sommersemester 2018 —

Inhaltsverzeichnis

1 Einleitung 1
- 1.1 Prozedurale und objektorientierte Programmiersprachen 1
- 1.2 Encapsulation, Polymorphismus, Inheritance 1
- 1.3 Statische und dynamische Bindung 2
- 1.4 Die Geschichte von C++ 2
- 1.5 Literaturhinweise 3

2 Traditionelle und objektorientierte Programmierung 4
- 2.1 Strukturiertes Programmieren 4
- 2.2 Objektorientiertes Programmieren 5

3 Grundlagen 7
- 3.1 Installation von Eclipse und Übungsskript 7
- 3.2 Starten eines C++ Programms 7
- 3.3 Allgemeine Programmstruktur 8
- 3.4 Kommentare in Programmen 10
- 3.5 Variablen 11
  - 3.5.1 Was sind Variablen? 11
  - 3.5.2 Regeln für Variablen 12
  - 3.5.3 Grundtypen von Variablen 12
  - 3.5.4 Deklaration und Initialisierung von Variablen 13
  - 3.5.5 Lebensdauer von Variablen 14
  - 3.5.6 Gültigkeit von Variablen 15
- 3.6 Namespace 16
  - 3.6.1 Defaultnamespace 16
- 3.7 Zuweisen von Werten an eine Variable 17
- 3.8 Einlesen und Ausgeben von Variablenwerten 19
  - 3.8.1 Einlesen von Daten 19
  - 3.8.2 Zeichenweises Lesen und Ausgeben von Daten 21
  - 3.8.3 Ausgeben von Daten 22
  - 3.8.4 Arbeit mit Dateien 23
    - 3.8.4.1 Textdateien unter Unix 23
    - 3.8.4.2 Textdateien unter Windows 24
    - 3.8.4.3 Lesen aus einer Datei 25
    - 3.8.4.4 Lesen aus einer WINDOWS Textdatei 26

3.8.4.5 Schreiben in eine Datei 26
3.9 Operatoren 29
3.9.1 Arithmetische Operatoren 30
3.9.2 Relationale und Logische Operatoren 30
3.9.3 Die Wahrheitstabelle bei logischen Ausdrücken 31
3.9.4 String Operatoren 31
3.9.4.1 Zuweisungsoperator: = ... 31
3.9.4.2 Verknüpfungsoperator: + ... 31
3.9.4.3 Lexikographischer Vergleich bei Strings: 32

4 Konstanten 33

5 Strukturierung von Programmen 37
5.1 Kontrollstrukturen 37
5.1.1 Statements und Blöcke 37
5.2 compound statement 37
5.3 Selection-Statements 38
5.3.1 Selection-Statement: `if` , `if - else` 38
5.3.2 Selection-Statement: `switch` 39
5.4 Iteration-Statements 40
5.4.1 Iteration-Statement: `while` 40
5.4.2 Iteration-Statement: `do` 41
5.4.3 Iteration-Statement: `for` 42
5.4.4 Iteration-Statement: `range-for` 43
5.5 Das Wichtigste in Kürze 44

6 Einsatz von Strings 46
6.1 Einfï¿½hrung 46
6.1.1 Bitbreite der Buchstaben von Strings 46
6.1.2 Deklaration und Initialisierung von Stringvariablen mit 8-Bit Buchstabencode 47
6.2 Methoden fï¿½r Stringobjekte 47
6.3 Konstruktion eines Strings 49
6.4 Destruktion eines Strings 49
6.5 Zugriff auf die Buchstaben eines Strings 49
6.6 Alphabetischer Vergleich von Strings 50
6.7 Suchen innerhalb eines Strings 53
6.8 Modifizieren eines Strings 53
6.9 C++ Strings und C Strings 57
6.10 Das Wichtigste in Kï¿½rze 57

7 Funktionen 59
7.1 Motivation 59
7.2 Funktionstyp 61
7.3 Funktionsname 62
7.4 Funktionsargumente 62
7.5 Defaultwerte für Funktionsargumente 62
7.6 Funktionsrumpf 63
7.7 Die Argumentübergaben 63
7.7.1 Übergabe des Werts eines Arguments 64
7.7.2 Übergabe einer Referenz auf ein Argument 64
7.7.3 Beispiele Value vs. Referenz 64
7.7.4 Seiteneffekte 65
7.8 Überladen von Funktionen 67
7.9 Mehrdeutigkeiten durch Überladen der Funktionen 68

8 Internationalisierung unter C++ 70
8.1 Einführung in Kodierungen 70
8.2 Unicode Transformation Format: UTF-8 70
8.3 Datentypen für Unicode-Zeichen 72
8.4 Locales und Imbuing 73
8.4.1 Localeabhängiges Arbeiten bei UCS Codierung 74
8.5 UCS Zeichen und Datei-Streams 76
8.5.1 Konvertierung von utf8 nach ISO-Latin 76
8.5.2 Ausgabe einer UTF8- kodierten Datei 77
8.6 Das Wichtigste in Kürze 78

9 Programmieren von Klassen 80
9.1 Einführung 80
9.2 Deklaration von Klassen 80
9.3 Deklaration von Klassenobjekten 81
9.4 Beispiel mit Klassen 82
9.5 Initialisierung der Daten eines Objekts 85
9.6 Löschen der Daten eines Objekts 87
9.7 Kopieren der Daten eines Objekts 87
9.8 Das Überladen von Operatoren 88
9.9 Überladen von relationalen und logischen Operatoren 89
9.10 Überladen von unären Operatoren 89
9.11 Überladen des Output-Operators 90

10 Vererbung 92
10.1 Einleitung 92
10.2 Vererbung von Zugriffsrechten 93

10.3 Spezielle Methoden werden nicht vererbt 94
10.4 Zuweisung von Objekten einer Unterklasse an Objekte der Oberklasse . . 97
10.5 Überschreiben von Methoden/Funktionen in der abgeleiteten Klasse . . . 97
10.6 Polymorphismus 98

11 Templates 101
11.1 Generische Funktionen 101
11.2 Generische Klassen 103
11.3 Erweiterung des Beispiels memory 106

12 Standard Template Library (STL) 111
12.1 Iteratoren der STL 111
12.2 Klassen der STL für die Computerlinguistik 115
12.2.1 wchar_t 115
12.2.2 `wstring` 115
12.3 Utilities der STL für die Computerlinguistik 116
12.3.1 `pair <type1, type2>` 116
12.3.1.1 `make_pair()`- Hilfsfunktion 116
12.4 Container der STL für die Computerlinguistik 119
12.4.1 Überblick 119
12.4.2 Einsatz der Container 119
12.4.3 `vector <type>` 120
12.4.4 `list<type>` 124
12.4.5 deque<type> 127
12.4.6 set <type> 127
12.4.6.1 Beispiel mit dem Container `set` 128
12.4.7 map <type1,type2> 129
12.4.7.1 Beispiel zur Berechnung einer Frequenzliste mit `map` . . . 130
12.4.8 unordered_set<type>, unordered_map<type1,type2> 132
12.4.8.1 Implementation der Hash-Templates der STL mit dem gcc 132
12.5 STL-Algorithmen 136
12.5.1 Vorbemerkung: Laufzeiten 136
12.5.1.1 Der Headerfile `algorithm` für Algorithmen 137
12.5.1.2 `find()`: 138
12.5.1.3 `find_first_of()`: 139
12.5.1.4 `find_last_of()`: 139
12.5.1.5 `find_first_not_of()`: 141
12.5.1.6 `find_last_not_of()`: 142
12.5.1.7 `find_if()`: 142
12.5.1.8 `find_end()`: 143
12.5.1.9 `adjacent_find()`: 144
12.5.1.10 `search()`: 145

12.5.1.11 search_n(): . . . 146
12.5.1.12 count(): . . . 147
12.5.1.13 count_if(): . . . 148
12.5.1.14 equal(): . . . 149
12.5.1.15 mismatch(): . . . 150
12.5.1.16 replace(): . . . 151
12.5.1.17 replace_copy(): . . . 151
12.5.1.18 replace_copy_if(): . . . 151
12.5.1.19 replace_if(): . . . 152
12.5.1.20 unique(): . . . 152
12.5.1.21 unique_copy(): . . . 152
12.5.1.22 sort(): . . . 153
12.5.1.23 transform(): . . . 154
12.5.1.24 for_each(): . . . 155

13 Erweiterung um reguläre Ausdücke, die Bibliothek Boost::Regex 156
13.1 Integration des Pakets Boost . . . 156
13.1.1 Installation des Pakets Boost . . . 156
13.1.2 Kompilieren unter Verwendung der Bibliothek Boost::Regex . . . 157
13.1.3 Einbinden von Boost::Regex in Eclipse . . . 157
13.1.4 Verwendung von Boost::Regex . . . 158
13.2 Verwendung von Regex mit C++11 . . . 158
13.3 Suchen nach einem regulären Ausdruck: regex_match . . . 158
13.4 Suchen nach einem regulären Ausdruck: regex_search() . . . 160
13.5 Ersetzen in einem String mit einem regulären Ausdruck: regex_replace() 162
13.6 UNICODE und Lokalisierung mit boost . . . 163
13.7 Markierte Subexpressions . . . 166
13.8 Erstes Vorkommen in einer Zeichenkette finden: regex_find() . . . 167
13.9 Alle Vorkommen in einer Zeichenkette finden: regex_find_all() . . . 167
13.10Zeichenketten aufsplitten: regex_split() . . . 168
13.11Zeichenketten durchlaufen: regex_iterator() . . . 169
13.12Unicode Zeichenklassen . . . 169

14 Ausnahmebehandlung 172
14.1 Selbstdefinierte Ausnahmen . . . 173
14.1.1 Vordefinierte Ausnahmen . . . 176
14.1.1.1 Standard-Exceptions . . . 176
14.1.1.2 System-Exceptions . . . 176
14.1.1.3 Exceptions, die in der Standardlibrary ausgelöst werden . 177
14.1.1.4 Zusätzliche Laufzeitfehler . . . 178

15 Spezialthemen 182
15.1 Vorsicht bei using Direktive: Namespaces 182
15.2 Flache und Tiefe Member einer Klasse 182
15.3 Speicher Allocation (Beispiele allocation) 183
15.4 Destruktor (Beispiele in destruct) 184
15.4.1 Mit Zuweisungsoperator: 184
15.4.2 Initialisierung der Variable (Verzeichnis init_copy) 184
15.4.3 Initialisierung: Copykonstruktor (Verzeichnis init_copy) 185
15.4.4 Argumentübergabe als Wert (Verzeichnis routine_copy) 186
15.4.5 Wertrückgabe bei Routinen (Verzeichnis routine_copy) 186
15.5 Berechnung Konkordanz mit Routinen der STL 186
15.6 Hashes und Internationalisierung 190
15.7 Überladen von Operatoren, Zweiter Teil 195
15.7.1 Überladen von binären Operatoren 195
15.7.2 Die friend-Operatorfunktion 198
15.8 Überladen des Ausgabeoperators 200
15.9 Überladen des Eingabeoperators 201
15.10 Liste der überladbaren Operatoren 203
15.11 Mehrere locales mit boost 204
15.12 Spezialthemen und Beispiele 204
15.12.1 Konkordanzprogramm mit Klassen und Vektoren 204

16 Neue Standards von C++ 213
16.1 Einleitung 213
16.2 Der Compiler-Aufruf 213
16.3 Vereinheitlichte Initialisierung 213
16.4 Lambda- Funktionen 215
16.5 Das auto()-Keyword 216
16.6 Variablen Templates 218
16.7 Range-For- Schleifen 218
16.8 Neue Algorithmen 220
16.8.1 all_of() 220
16.8.2 any_of() 221
16.8.3 none_of() 222
16.8.4 copy_n() 222
16.8.5 iota() 223
16.9 Neue Datentypen für Internationalisierung 224
16.10 Hashfunktion für ungeordnete Container: Ein Benchmarkbeispiel 225
16.10.1 Jenkins-Hashfunktion 225
16.10.2 Stefan-Hashfunktion 225
16.10.3 Modifizierte Stefan-Hashfunktion 226
16.10.4 Benchmarking 226

16.10.5 Ergebnis: Jenkins-Hashfunktion 226
16.10.6 Ergebnis: Standard-Hashfunktion 226
16.10.7 Ergebnis: Stefan-Hashfunktion 226
16.10.8 Ergebnis: Stefan2-Hashfunktion 226
16.10.9 Fazit 227
16.10.10 Quelle 227
16.11 Die Zeitbibliothek `Chrono`: Ein weiteres Benchmarkbeispiel 227
16.11.1 Ergebnisse 229
16.12 Weitere Neuerungen seit C++14 229
16.13 Ausblick auf C++17 230
16.14 Literaturhinweise 230

Literaturverzeichnis 231

Abbildungsverzeichnis 233

Tabellenverzeichnis 234

1 Einleitung

1.1 Der Unterschied zwischen prozeduralen und objektorientierten Programmiersprachen

In traditionellen prozeduralen Programmiersprachen bestehen Programme aus einer Einheit von Daten und Funktionen. Es gibt lokale oder globale Daten und Funktionen, die diese Daten modifizieren.
Bei der objektorientierten Programmierung werden Daten und Funktionen, die thematisch zusammengehören, zu Gruppen zusammengefasst. Diese Gruppen heißen *Objekte*. Die Objekte sind für ihre Daten zuständig und interagieren mit anderen Objekten. Ein Programm besteht somit aus der Interaktion von Objekten. Dadurch werden Programme besser strukturiert als bisherige Programme und größere Programmieraufgaben können leichter bewältigt werden.

1.2 Encapsulation, Polymorphismus, Inheritance

In der objektorientierten Programmierung (OOP) stehen drei Begriffe im Vordergrund:

Encapsulation (Einkapselung):
: Jedes Objekt kennt seine Daten und die dafür zuständigen Operationen. Es kann seine Daten nach außen verbergen und nur über eigene Methoden modifizieren lassen. Die für die Objekte zuständigen Operationen heißen *Methoden*. Der Programmierer muss nun versuchen, die zu programmierende Aufgabe in eine Interaktion von Objekten umzuformen.

Polymorphismus:
: Methoden und Operanden, die die gleiche Aufgabe auf verschiedenen Objekten ausführen können, dürfen in der OOP den gleichen Namen haben. Diese Möglichkeit steigert die Lesbarkeit von Programmen erheblich. Beim Aufruf einer Methode wird anhand des angesprochenen Objekts die passende Methode ausgewählt.

Inheritance (Vererbung):
: Um hierarchische Zusammenhänge von Objekten widerspiegeln zu können, erlaubt es die OOP, Objekte hierarchisch zu verknüpfen. Objekte können dann zu einer Ober- oder Unterklasse gehören. Jede Unterklasse hat Zugriff auf Methoden der Oberklasse.

1.3 Statische und dynamische Bindung

Ein wichtiger Aspekt der OOP ist die Bindung der Identifier an ihre Typen. Man unterscheidet statische und dynamische Bindung. Die traditionellen Programmiersprachen sind größtenteils statisch, da schon zur Kompilationszeit eine Zuordnung der Variablennamen zu ihrem Typ vorgenommen wird. Die OOP ist in der Lage diese Zuordnung erst zur Laufzeit zu realisieren, da bei Variablen zur Laufzeit der Typ noch bekannt ist. So kann dann der Polymorphismus aufgelöst werden.
Die dynamische Bindung gibt dem Programmierer die größten Freiheiten, da er zur Definitionszeit lediglich den Ablauf einer Methode festlegen muss und zur Laufzeit die für das Objekt notwendigen Methoden ausgewählt werden.

1.4 Die Geschichte von C++

C++ wurde 1980 von Bjarne Stroustrup an den Bell Labs erfunden. Beeinflusst wurde er von verschiedenen Konzepten, die es schon in anderen Programmiersprachen gab:

SIMULA67 Klassenkonzept
ALGOL68 Deklaration von Variablen an beliebigen Stellen
ADA templates, exception handling
ML exception handling

1983 entstand die erste Version – genannt C – mit Klassen, dann wurde der Name C++ gefunden.
1989 wurde ein X3J16 ANSI-Komitee gegründet um ANSI C++ zu definieren. Dieses Komitee hat am 28. April 1995 eine Draft verabschiedet, die hauptsächlich Klassenlibraries standardisiert. Es wurden C++ Klassen für folgende Bereiche definiert:

Clause	Category
lib.language.support	Language support
lib.diagnostics	Diagnostics
lib.utilities	General utilities
lib.strings	Strings
lib.localization	Localization
lib.containers	Containers
lib.iterators	Iterators
lib.algorithms	Algorithms
lib.numerics	Numerics
lib.input.output	Input/output

Tabelle 1.1: Standardklassen in C++

2011 wurde mit C++11 (auch *C++0x*) ein neuer Standard für C++ verabschiedet, der u.a die C++ Standardlibrary stark erweitert. Im Januar 2015 erschien C++14 mit einigen Neuerungen und für 2017 ist bereits ein neuer Standard, C++17, geplant.

1.5 Literaturhinweise

Eine sehr gute und ausführliche Einführung in die Programmierung mit C++ in deutscher Sprache finden Sie in den Büchern von Ulrich Breymann, im speziellen in den Büchern Breymann: (Ulr07) und (Ulr09).
Ein ältere, aber trotzdem gute Einführung in C++ finden Sie im Buch von Herbert Schildt (Her94).
Es sind auch die Bücher vom Erfinder von C++ Bjarne Stroustrup, zu erwähnen, die sehr detailliert alle Features von C++ komplett abdecken: (Bja00) und (Bja09).
Die Standard Template Library wird am besten bei Nicolai M. Josuttis beschrieben: (Jos99)
Ein vertiefendes Buch über die Arbeit mit der Standard Template Library stellt das von Scott Meyers dar: (Sco08).
Vertiefende Literatur zu Templates finden Sie im Buch von David Vandervoorde und Nicolai M. Josuttis: (Dav02).
Literatur zur Einfühung mit der Bibliothek Boost finden sid im Buch von Björn Karlsson: (Bjo05).
Ein Buch mit vielen Beispielen zu bestimmten Aufgaben ist das C++ Kochbuch von D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis und Jeff Cogswell: (D. 06).
Bücher zur fortgeschrittenen Arbeit mit C++ sind die von Andrew Koenig und Barbara E. Moo: (And00) und von Andrei Alexandrescu: (And09).

2 Traditionelle und objektorientierte Programmierung

2.1 Strukturiertes Programmieren

Die so genannte strukturierte Programmentwurfsmethode, in den 60er Jahren entwickelt, kennt nur drei elementare Formen der Programmstruktur: Auswahl (Selektion), Reihung und Wiederholung (Iteration); es gelten hierbei folgende Vereinbarungen (vgl. Schulze 1988: 408):

1. Jeder Block ist eine selbständige funktionale Einheit mit nur einem Eingang und einem Ausgang.
2. Jeder Block steht zu einem anderen in der Überordnung [. . .] oder der Unterordnung [. . .].
3. Einem Block darf mehr als ein Block untergeordnet sein.
4. Jedem Block darf nur ein Block übergeordnet sein (so dass eine eindeutige [. . .] Baumstruktur entsteht).
5. Blöcke müssen vollkommen voneinander getrennt sein und dürfen sich nicht überschneiden.
6. Das Springen von einem Block in den anderen ist nur [. . .] über die Aus- und Eingänge zugelassen.

Derartiges Programmieren sieht Schneider (HJS86) aus arbeitsökonomischem Blickwinkel als eine:

> Programmiermethodik, bei der man eine Gesamtaufgabe so in Teilaufgaben aufteilt, dass (a) jede Teilaufgabe für einen Programmierer überschaubar ist, (b) eine Teilaufgabe weitgehend unabhängig von den anderen Teilaufgaben gelöst [. . .] werden kann und (c) der Programmierer sich und andere auf relativ einfache Weise von der Korrektheit der einzelnen Teilaufgaben und der Gesamtaufgabe überzeugen kann.

Schildt (Her94) erklärt in „Teach Yourself C++“:

> Structured programming relies on well-defined control structures, code blocks, the absence (or at least minimal use) of the GOTO, and stand-alone subroutines that support recursion and local variables. The essence of structured programming is the reduction of a program into its constituent elements.

Der Begriff des strukturierten Programmierens steht somit mit dem der funktionalen Dekomposition in Zusammenhang (Ste89, S. 37f.):

> Functional decomposition is a method for subdividing a large program into functions. [...] This design method results in a hierarchy of functions in which higher-level functions delegate lower-level functions. This method is also known as top-down design, because it starts at a high level description of the program, then works through the refinement of the design to the low-level details of the implementation.

2.2 Objektorientiertes Programmieren

Da die objektorientierte Hochsprache C++ syntaktisch und semantisch als Supermenge (superset) der Hochsprache C gilt, sollte objektorientiertes Programmieren (object-oriented programming, OOP) auch an den effizienten Prinzipien des strukturierten Programmierens ausgerichtet sein. In diesem Sinn formuliert Schildt (Her94, S. 4):

> OOP takes the best of the ideas embodied in structured programming and combines them with powerful new concepts that allow you to organize your programs more effectively. Object-oriented programming encourages you to decompose a problem into related subgroups. Each subgroup becomes a self-contained object that contains its own instructions and data that relate to that object.

Hierbei wären nun drei Wesenszüge auszuführen, die allen objektorientierten Sprachen innewohnen sollten:

1. Einkapselung (encapsulation),
2. Polymorphismus (polymorphism) und
3. Vererbung (inheritance)

Zu 1.:

> Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be bound together in such way that a self-contained „black box“ is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created.

Zu 2.:

> Polymorphism [...] is the quality that allows one name to be used for two or more related but technically different purposes. The purpose of polymorphism as it is applied to OOP is to allow one name to be used to specify a general class of actions. Within a general class of actions, the specific action to be applied is determined by the type of data. [...] in C++ it is possible to use one function name for many different purposes. This is called function overloading. More generally, the concept of polymorphism is the idea of „one interface, multiple methods“, und „Polymorphism“ can be applied to both functions and operators. Virtually all programming languages contain a limited application as it relates to the arithmetic operators. For example in C, the + sign is used to add integers, long integers, characters, and floating-point values. In these cases, the compiler automatically knows which type of arithmetic to apply. In C++ you can extend this concept to other type of data that you define. This type of polymorphism is called operator overloading.

Zu 3.:

> Inheritance is the process by which one object can acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add those features that are specific only to itself. Inheritance is important because it allows an object to support the concept of hierarchical classification. Most information is made manageable by hierarchical classification.

3 Grundlagen

3.1 Installation von Eclipse und Übungsskript

Zur Installation von Eclipse lesen Sie bitte die Anleitung im Wiki:
`http://www.cis.uni-muenchen.de/lehre/SommerSemester2016/HoehereProgrammierung`

Das Übungsskript zur Vorlesung finden Sie unter:
`http://www.cis.uni-muenchen.de/kurse/max/C++/scripten/uebung-skript`

3.2 Starten eines C++ Programms

Es gibt zwei verschiedene Arten, wie Programme auf dem Rechner ausgeführt werden:

1. Programme, die vom Interpreter der Programmiersprache Zeile für Zeile interpretiert werden, z.B. PERL, PROLOG, JAVA.

2. Programme, die vom Compiler der Programmiersprache in Maschinencode übersetzt werden und dann als eigenständige Programme arbeiten, z.B. C, PASCAL.

PERL- und PROLOG-Programme müssen in der Regel unter Angabe eines Zusatzprogramms, des Interpreters, aufgerufen werden: Möchte man z.B. das Programm `eins.perl` ausführen, dann gibt man im Terminal den Befehl `perl eins.perl` ein.
C++ Programme gehören dagegen zur zweiten Klasse von Programmen. C++ Programme werden als Textfile in Maschinencode kompiliert und als eigenständige Programme gestartet. Maschinencode enthält Befehle, die unmittelbar von der CPU ausgeführt werden können. Das macht in Compiler-Sprachen geschriebene Programme nicht nur bedeutend schneller als solche, die interpretiert werden müssen (etwa 10-20 mal schneller), sondern sie sind auch selbstständig lauffähig, benötigen also keine Zusatzprogramme. Maschinencode lässt sich auch durch eine ASSEMBLER-Programmiersprache direkt erzeugen.
Der Weg zu einem lauffähigen C++ Programm lässt sich in folgendem Schema darstellen:

Sourcecode ⇒ Objektfile ⇒ Maschinencode

1. Schritt: Schreiben des Programms mit Hilfe eines Texteditors, Sichern des Programmtextes (Sourcecode) als Datei mit der Endung `.cxx` oder `.cpp` z.B. `emacs beispiel.cxx`

2. Schritt: Kompilieren und Linken des Sourcecodes in Maschinencode mit Hilfe des C++ Compilers. Der C++ Compiler übersetzt ein C++ Programm in ein Maschinenprogramm mit dem Namen `a.out` (unter UNIX) und `a.exe` (unter Windows) z.B. `g++ beispiel.cxx` (in bestimmten Versionen auch `gcc beispiel.cxx`).
Soll das erzeugte Maschinenprogramm einen anderen Namen erhalten, dann kann dies mit der Option `-o` beim Übersetzen festgelegt werden:
z.B. `g++ -o beispiel beispiel.cxx` erzeugt ein Maschinenprogramm mit dem Namen `beispiel`

3. Schritt: Starten des Maschinenprogramms `a.out`, bzw. `a.exe` z.B. `a.out` oder `./a.out` oder `./a.exe`
(zur Erläuterung: `./` erzwingt die Suche im aktuellen Verzeichnis, damit lassen sich Fehlermeldungen, die durch voreingestellte Pfadangaben ausgelöst werden, vermeiden) oder beim Übersetzen mit der Option: `-o beispiel ./beispiel`

3.3 Allgemeine Programmstruktur

Ein C++Programm kann aus beliebigen Funktionen, die über verschiedene Dateien verteilt sind, bestehen. Eine Funktion innerhalb des Programms entspricht der Hauptfunktion und heißt `main()`. Diese Funktion stellt den Einstiegspunkt in das Programm dar. Alle weiteren Funktionen werden direkt oder indirekt von ihr aufgerufen.

Jedes C++Programmmodul hat folgende Struktur:

```
[ Präprozessor-Direktiven ]
[ Typ-, Klassendeklarationen ]
[ Definition und Deklaration von Variablen
(globale Variablen) ]
[ Deklaration von Funktionen (=Prototypen) ]
Definition von Funktionen (nur eine Funktion wird
main genannt)
```

Jede Funktion hat folgende Struktur:

```
Funktionstyp Funktionsname ( [ Parameter 1 ,...,
Parameter n ] )
Funktionsrumpf
```

Jeder Funktionsrumpf hat folgende Struktur:

```
{ [ Deklarationen von Variablen ]
Statements }
```

Beispiel 3.1: Funktion mit Hauptprogramm

```
// Präprozessor-Direktiven

type funktion ( ... );      // Prototyp

int main() {
    /* Hauptprogramm */
}

type funktion ( ... ) {     // Definition
    /* Anweisungen der Funktion */
}
```

Beispiel 3.2: Ein komplettes C++ Programm

```
// file: Grundlagen/eins.cpp
// description:
#include <iostream>    // systemweite Header
#include <string>
using namespace std;

// Prototypen: Funktionen werden deklariert:
int begruessung();
int verabschiedung();

int main() {
    const string Space(5, ' ');
    string vorname, nachname;
    string name;

    begruessung();
    cout << "Bitte Vornamen eingeben: ";
    cin >> vorname;
    cout << "Bitte Nachnamen eingeben: ";
    cin >> nachname;
    name = vorname + Space + nachname;
    cout << "Ihr Name ist " << name << endl;
    verabschiedung();
    return 0;
}

int begruessung() { // Funktionskopf
    cout << "Guten Tag!" << endl; // Funktionsrumpf
    return 0;
}

int verabschiedung() {
    cout << "Auf Wiedersehen" << endl;
    return 0;
}
```

Die `main()`-Funktion ruft zunächst die Funktion `begruessung()` auf, die „Guten Tag" gefolgt von einem Zeilenumbruch auf dem Bildschirm ausgibt. Dann wird der Benutzer aufgefordert, Vor- und Nachnamen getrennt voneinander einzugeben. Beide Eingaben werden zu einem String konkateniert, der daraufhin ausgegeben wird. Abschließend wird die Funktion `verabschiedung()` ausgeführt, die „Auf Wiedersehen" ausgibt.

Formatierungsvorschläge:

Die öffnende geschweifte Klammer eines Blockes schreiben wir nach einem Blank in dieselbe Zeile wie die zugehörige Funktion. Die korrespondierende schließende Klammer immer auf Höhe des ersten Buchstabens der Zeile mit der öffnenden Klammer. Zur logischen Gliederung eines Programmes nutzen wir leere Zeilen, z.B. zwischen verschiedenen Funktionsdefinitionen.

Beispiel 3.3: Aufrufe von Funktionen

```
void foo() {
   int I;
   double d;
   char c;
   void bar() {

   }
}
```

3.4 Kommentare in Programmen

In C++ können Kommentare an beliebiger Stelle eingestreut werden. Es gibt die Möglichkeit, Kommentare zwischen die Zeichen `/*` und `*/` zu schreiben, oder mit dem Zeichen `//` den Rest der Zeile als Kommentar zu definieren. Kommentare, die über mehrere Zeilen geschrieben werden sollen, können mit `/**` begonnen und mit `*/` beendet werden. Um die Zeilen dazwischen als Kommentar zu markieren beginnen diese mit `*`.

Beispiel 3.4: Kommentare 1

```
std::cout << "Dieser Text wird ausgegeben" << std::endl;
/* std::cout << "Dieser Text wird NICHT ausgegeben !!" << std::endl;*/
```

oder

Beispiel 3.5: Kommentare 2

```
std::cout << "Ausgabetext" << std::endl;      // Kommentar
```

oder

Beispiel 3.6: Kommentare 3

```
/** Das ist ein
* Kommentar.
* Dieser wird nicht ausgegeben.
*/
```

Übung 3.1

Schreiben Sie ein C++ Programm, das den Text `Hallo! Hier bin ich` auf dem Terminal ausgibt.

3.5 Variablen

3.5.1 Was sind Variablen?

In der Programmierung versteht man unter einer Variablen eine Bezeichnung für ein Objekt eines bestimmten Datentyps. Eine Variable kann auch als Synonym für einen Speicherbereich bezeichnet werden. In diesem Speicherbereich wird der Wert des konkreten Objekts gespeichert. Jede Variable zeigt genau auf ein konkretes Objekt und kann zu jedem Zeitpunkt nur einen Wert annehmen. Es können aber mehrere Variablen auf ein Objekt zeigen. Der Bezug einer Variablen zu einem konkreten Objekt wird bei der Deklaration realisiert.

Beispiel:

Gegeben sei der Datentyp `Zeichenkette`
Wollen wir Buchstabenketten speichern, dann benötigen wir für jede Buchstabenkette eine eigene Variable. Beim Kompilieren des Programms wird der Variablen ein eindeutiger Speicherbereich zugeordnet. Die Zuordnung der Variablen zu ihrer Adresse wird in einem Adressbuch festgehalten:

Datentyp	**Variablenname**	**Adresse (z.B.)**
`Zeichenkette`	`Vorname`	1000
`Zeichenkette`	`Nachname`	2000

Tabelle 3.1: Variablen im Adressbuch

`Zeichenkette Vorname` $\Leftrightarrow$ `(Datentyp: Zeichenkette), (Vorname, Adresse nnnn)`

`Zeichenkette Nachname` $\Leftrightarrow$ `(Datentyp: Zeichenkette), (Nachname, Adresse mmmm)`

3.5.2 Regeln für Variablen

- Jede Variable ist einem konkreten Objekt zugeordnet und muss vor dem Gebrauch deklariert werden
- Variablennamen beginnen mit Buchstaben
- Groß/Kleinschreibung bei Variablennamen ist signifikant
- Variablennamen sollten nicht mit Underscore beginnen (reserviert für Systemvariablen)
- 31 Buchstaben sind signifikant bei lokalen Variablen

3.5.3 Grundtypen von Variablen

Der Compiler der Programmiersprache C++ nimmt die Prüfung der Variablentypen sehr genau. Er testet sowohl die Datentypen, als auch die Verwendung von Konstanten. Bevor der Programmierer eine Klasse, eine Variable oder eine Funktion verwenden kann, muss er sie vorher deklariert haben. Bei Funktionen bezeichnet man die Deklaration eines Prototyps als die Definition eines Prototyps.
In der Tabelle 3.2 sind die Standarddatentypen der Programmiersprache C++ aufgeführt, in der Tabelle 3.3 die Erweiterungen dieser Datentypen.

Datentyp	Deklaration	Bits	Wertebereich	Beispiele
Ganze Zahl	int i;	32 Bit	-2**32 < I < -2**32	i = 3;
Rationale Zahl	float x;	64 Bit		x = 1.129;
ASCII-Buchstaben	char c;	7 Bit, 8 Bit		c = ’a’;
ISO-Buchstaben	unsigned char c;	8 Bit		c = ’a’;
Wahrheitswert	bool w;	8 Bit	true, false	w=true;

Tabelle 3.2: Standarddatentypen in C++

Datentyp	Deklaration	Wertebereich	Beispiele
Zeichenkette	string str;	Buchstabencodes zwischen 0 < I < 255	"Hallo wie gehts"

Tabelle 3.3: Erweiterungen von Datentypen in C++

Mit dem neuen Standard C++11 ist auch die Verwendung des `auto`-Keywords als Datentyp erlaubt. Bei diesem Keyword wird der Datentyp einer Variable zur Laufzeit durch den

Compiler erkannt. Weiterhin kann man den Typ einer Variable auch durch das Keyword `decltype` bestimmen. Weitere Informationen zu C++11 finden sich in Kapitel 16.

3.5.4 Deklaration und Initialisierung von Variablen

Jede Variable muss mit ihrem Namen und dem Namen für den Typen des Objekt vor dem Gebrauch definiert werden: Daraufhin reserviert der Compiler ausreichend Speicherplatz für das Objekt, das unter diesem Namen gespeichert wird. Die Definition einer Variablen nennt man Deklaration. Sollen mehrere Variablen eines Datentyp deklariert werden, dann können hinter dem Typnamen des Objekts mit Komma getrennt die Namen der Variablen angegeben werden.
In der Programmiersprache C++ wird die Deklarationsstelle von Variablen nicht vorgeschrieben. Variablen können außerhalb oder innerhalb von Funktionen, oder Blöcken deklariert werden.

Beispiel:

```
Datentyp      int
Variablen:    (int, zahl_1), (int, zahl_2)
Objekte:      zwei ganze Zahlen
Deklaration:  int zahl_1, zahl_2;

Datentyp      string
Variablen:    (string, name_1) ,(string, name_2)
Objekte:      zwei ISO 8-Bit Zeichenketten
Deklaration:  string name_1, name_2;
```

Beim Kompilieren des Programms werden den Variablen eindeutige Speicherbereiche zugeordnet. Die Zuordnung der Variablen zu ihren Adressen wird in einem Adressbuch festgehalten (siehe Tabelle 3.4).

Datentyp	Variablenname	Adresse (z.B.)
int	zahl_1	1000
int	zahl_2	2000

Tabelle 3.4: Adressbuch, vom Compiler erzeugt

3.5.5 Lebensdauer von Variablen

Der Begriff Lebensdauer beschreibt, wie lange ein Objekt existiert, also zu welchem Zeitpunkt es angelegt und wann es wieder zerstört wird. Die Lebensdauer einer Variable ist abhängig von ihrer Deklarationsstelle.
Die Deklarationsstelle liegt:

1. Fall: außerhalb von Funktionen (globale Variablen):
Alle dort vereinbarten Objekte (globale Objekte oder globale Variablen) und Objekte, die innerhalb von Funktionen mit dem Schlüsselwort `static` vereinbart werden (statische Objekte), existieren (leben) während der gesamten Ausführungszeit des Programms und innerhalb aller Funktionen des gesamten Programms.

2. Fall: innerhalb von Funktionen (lokale Variablen):
Alle dort vereinbarten Objekte (automatische Objekte oder lokale Variablen) existieren (leben) nur innerhalb der Funktion.

3. Fall: innerhalb eines Blocks (lokale Variablen):
Alle dort vereinbarten Objekte existieren (leben) nur, wenn der Block, zu dem sie gehören, abgearbeitet wird.

Beispiel 3.7: Lebensdauer von Variablen

```
int x; // globales x (1. Fall)

int f() {
    int x; // lokales x verbirgt globales x (2. Fall)
    x = 1; // (2.-3. Fall)
    {
        float x; // neues lokales x mit anderem Typ
        x = 2.0; // (2.-3. Fall)
    }
    return x;
}

float g() {
    float y;
    y = x - 3.0; // globales x (1. Fall)
    return y;
}

int main(void) {
    f();
    g();
}
```

Achtung: Bei Mehrfachdeklarationen mit unterschiedlicher Signatur erzeugt C++ keine Fehlermeldungen!

3.5.6 Gültigkeit von Variablen

Kurz: Die Gültigkeit einer Variablen entspricht solange der Lebensdauer der Variablen, bis innerhalb einer Funktion oder eines Blocks eine Variable mit dem gleichen Namen definiert wird.

Genauer: Der Bereich innerhalb eines Programms, in dem der Bezeichner der Variablen verwendbar - also gültig - ist, wird als Gültigkeitsbereich einer Variablen (engl.: `scope`) bezeichnet. Je nachdem an welcher Stelle eine Variable im Programm deklariert wird, ändert sich auch ihr Gültigkeitsbereich. In Bezug auf ihre Gültigkeit gibt es in C/C++ im Wesentlichen zwei unterschiedliche Arten von Variablen, die globalen und die lokalen Variablen.

1. Lokale Variablen: Eine Variable, deren Deklaration innerhalb eines Blocks steht, wird als lokal bezeichnet. Ein Block ist ein Bereich innerhalb eines Programms, der von einer geschweiften Klammer umschlossen ist, so bilden z.B. die Anweisungen innerhalb einer Funktionsdefinition einen Block. Die Gültigkeit einer lokalen Variablen beginnt an der Stelle ihrer Deklaration und ist auf den Block, in dem sie deklariert wurde, und auf alle darin enthaltenen Blöcke beschränkt.

2. Globale Variablen: Eine globale Variable wird außerhalb aller Blöcke deklariert und ist ab dem Ort ihrer Deklaration in der gesamten Datei gültig.

Solange man kleinere Programme mit nur einer Funktion (der Funktion `main()`) schreibt, spielt die Frage nach der Gültigkeit einer Variablen keine große Rolle. Sobald die Programme allerdings mit steigendem Umfang in immer mehr Module zerlegt werden, wird die Verwendung von Variablen, die an jedem Ort des Programms zugänglich sind, immer problematischer. In größeren Projekten besteht bei der Verwendung von globalen Variablen die Gefahr, dass eine Änderung einer Variablen in einem Programmteil unvorhergesehene Auswirkungen an einer völlig anderen Stelle des Programms hat. Um dieser Gefahr entgegenzuwirken, sollte der Gültigkeitsbereich einer Variablen nicht größer als unbedingt nötig sein.

Bemerkung: Durch den Einsatz unterschiedlicher Gültigkeitsbereiche ist es in C++ möglich, Variablen mit identischem Bezeichner in einem Programm einzusetzen. Aus Gründen der Übersichtlichkeit sollte man den Einsatz dieser Technik stets sorgfältig abwägen.
Folgende Regeln sind für den Einsatz gleichnamiger Bezeichner einzuhalten:

1. Zwei gleichnamige Bezeichner dürfen nicht im gleichen Block deklariert werden.

2. Zwei gleichnamige Bezeichner dürfen nicht beide global sein.

3.6 Namespace

Da in C++ sehr viele, auch globale, Variablen-, Klassen-, und Funktiondeklarationen in `include`-Files vorgenommen werden, kann es vorkommen, dass in verschiedenen `include`-Files die gleichen Namen bei Deklarationen verwendet werden. Bei der Aufgabe des Compilers, ein eindeutiges Adressbuch für alle Deklarationen zu erstellen, gäbe es Mehrdeutigkeiten, die zu Fehlern führen würden.
Um dies zu verhindern, wurde in C++ das Konzept des Namespaces eingeführt. Jede Variablen-, Klassen-, und Funktiondeklaration kann in einem eigenen Namespace definiert werden. Die Lebensdauer und Gültigkeit beschränkt sich dann automatisch auf diesen Namespace. Bei der Verwendung der Variable, Klasse oder Funktion muss dem Namen der Variable, Klasse oder Funktion der Namen des Namespaces, getrennt von zwei Doppelpunkten vorangestellt werden. Der vorangestellte Namespace legt genau fest, aus welchem Namespace welche Variable, Klasse oder Funktion verwendet werden soll. Der Compiler erzeugt also für jeden Namespace ein eigenes Adressbuch und verhindert somit Mehrdeutigkeiten.

3.6.1 Defaultnamespace

In neueren C++ Kompilern wird die Technik des Namespaces konsequent durchgeführt. Alle Standardfunktionen und Standardklassenerweiterungen von C++ sind im Default Namespace `std` deklariert. Werden sie verwendet muss dem Namen der Standardfunktion das Präfix `std::` vorangestellt werden. Möchte man einen Namespace als Default für alle Variablen, Klassen und Funktionen einstellen, so benutzt man zur Einstellung des Namespaces std die Anweisung „`using namespace std;`“
Also entweder:

```
std::string name;
std::cout << "Hallo";
```

oder:

```
using namespace std;
string name;
cout << "Hallo";
```

Bemerkung: Die Verwendung von „`using namespace std;`“ hat einen Nachteil: Man holt sich alle Namen im aktuellen Scope, selbst solche, von deren Existenz man vielleicht nicht einmal wusste. Deshalb gibt es noch einen Mittelweg: Man kann explizit die genutzten Variablen, Klassen oder Funktionen – und nur diese – in den Namespace holen:

```
#include <iostream>
using std::cout;        // std::cout hereinholen
```

Beispiel: Namespace

Die Klassendefinition string und die Funktion cout aus dem Standardnamespace werden verwendet.

```
std::string name;
std::cout << " Hallo wie gehts ";
```

Dazu ein Programm:

Beispiel 3.8: Namespaces

```
// file: Grundlagen/namespace.cpp
// description: work with namespaces

#include <iostream>
#include <string>

using namespace std;
string programname = "namespace.cpp";

namespace A {
    string name; // A::name
}

int main() {

    string name;
    cout << " Hallo! Das Programm " << programname << endl;
    cout << " Bitte geben Sie einen String ein: ";
    cin >> A::name;
    cout << " Der String ist " << A::name << endl;
    cout << " Bitte geben Sie einen zweiten String ein: ";
    cin >> name;
    cout << A::name << " und " << name << endl;
    return 0;
}
```

3.7 Zuweisen von Werten an eine Variable

Mit dem Zuweisungsoperator `=` können einer Variablen Werte zugewiesen werden. Die Sprache C++ entscheidet auf Grund des Variablentyps der rechten und linken Seite des Zuweisungsoperators wie die Zuweisung durchgeführt wird. Wurde eine Zuweisungsoperation mit den Datentypen der linken und rechten Seite definiert, dann wird diese Operation durchgeführt, ansonsten wird versucht die Datentypen mit `cast`-Operatoren anzupassen. Da der Programmierer in C++ sowohl Zuweisungsoperatoren und Castoperatoren selbst definieren kann, hat er alle Möglichkeiten offen, Werte von Variablen an andere Variablen zu übertragen. Versucht der Programmierer einer `const` Variable einen Wert zuzuweisen, bricht der Compiler mit einer Fehlermeldung ab.

Allgemein erlaubt es C++ Operatoren und ihre Funktionalität selbst zu definieren. Man spricht in diesem Fall von sog. „operator-overloading“.
Wenn man eigene Klassen definiert, ist es zum Teil sehr nützlich auch die passenden Operatoren bereitzustellen. Ein Beispiel ist die Stringkonkatenation mit +.
Das +-Symbol hat in diesem Falle eine völlig andere Funktionsweise als z.B. bei der Integeraddition, bei der 2 Zahlen addiert werden. Dem Nutzer bleibt diese jedoch verborgen. So kann man z. B.:

```
A = "aaa"
B = "bbb"
C = a + b; (C = "aaabbb")
```

ausführen.

Beispiel 3.9: Wertzuweisung

```
// file: Grundlagen/assignment.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {
    string str1("Hallo"), str2 = "wie gehts", str3;
    int val1 = 100, val2 = 10;
    float pi = 3.14;

    cout << "Der Wert von str1 = " << str1 << endl;

    str3 = str1;
    // genug Speicher für  str3 alloziert
    // jeder Buchstabe von str1 in str3 kopiert.

    cout << " Der Wert von str3 = " << str3 << endl;

    cout << " Der Wert von val1 = " << val1 << endl;
    val1 += val2;
    cout << " Nach val1 += val2; val1 = " << val1 << endl;

    int intPi = pi; // implizite Konvertierung
    cout << " pi : " << pi << " intPi = " << intPi << endl;

    return 0;

}
```

3.8 Einlesen und Ausgeben von Variablenwerten

C++ unterstützt das gesamte I/O-System (input/output) von C, enthält aber zudem eigene objektorientierte I/O-Routinen. Ein wichtiger Vorteil des I/O-Systems von C++ ist, dass es für selbst definierte Klassen problemlos erweitert werden kann.
Das C++I/O-System arbeitet mit Streams. Zu Beginn eines C++Programms werden automatisch folgende drei Streams geöffnet:

Stream	**Bedeutung**	**Standardgerät**	**I/O-Operator**
`cin`	standard input	Tastatur	> >
`cout`	standard output	Bildschirm	< <
`cerr`	standard error	Bildschirm	< <

Tabelle 3.5: Streams

In der Header-Datei `iostream.h` sind Klassenhierarchien definiert, die `I/O`-Operationen unterstützen.
Einen Stream kann man sich als ein Band vorstellen, auf dem die eingegebenen Zeichen zwischengepuffert werden, bis der Benutzer `<return>` eingibt. Das Zeichen `<return>` löst die Einleseoperation aus und es werden bis zum Separator alle Zeichen in die Variablen eingelesen. Nichtgelesene Zeichen bleiben auf dem Eingabeband und werden bei der nächsten Einleseoperation verarbeitet.
Als Standard-Input verwendet C++ die Tastatur. Man kann diesen Stream mit folgenden UNIX Befehlen umlenken:

```
a.out < eingabe.txt
a.out > ausgabe.txt
```

3.8.1 Einlesen von Daten

Eine Leseoperation in einem Programm hat folgende Struktur:

```
stream I/O-operator variable
```

z.B.

```
cin >> variable;
```

Beispiel 3.10: Eingabe von 2 ganzzahligen Werten über die Tastatur

```
int val1,val2;
cin >> val1 >> val2;
```

Eingabestrom >>		1		2	<return>

Eingelesen wird der Wert 1 und der Wert 2, dies entspricht der Zuweisung: `val1=1, val2=2`

Führende Leerzeichen (Blanks, Tabulatoren und Zeilenenden) werden überlesen. Das Lesen der Operatoren endet beim ersten Zeichen, das nicht mehr zum Typ des Operanden passt. Einlesen von Strings endet, wenn ein Separator als nächstes Zeichen auftaucht. Separatoren sind Leerzeichen, Tabulator oder Zeilenende. Der Separator wird dem String nicht angehängt, sondern vom Eingabestream gelöscht.

Beispiel 3.11: Eingabe eines Strings

```
string str;
cin >> str;
```

Eingabestrom >>	d	e	r	<return>
	d	e	r	

Es wird der Artikel „der“ gelesen, das `<return>`-Zeichen bleibt auf dem Eingabestrom.

Beispiel 3.12: Eingabe von zwei Strings

```
string str1,str2;
cin >> str1 >> str2;
```

Eingabestrom >>	d	e	r		d	i	e		<return>

Es wird der Artikel „der“ gelesen und der Variable `str1` zugewiesen. Das Blank bleibt auf dem Eingabestrom. Zum Einlesen des Wertes von `str2` wird als nächstes das führende Blank gelesen und der zweite Artikel „die“ der Variable `str2` zugewiesen.
Der Separator „Blank“ hinter dem „die“ bleibt wie der Zeilenterminator auf dem Eingabestrom bestehen.

Beispiel 3.13: Lesen ganzer Zeilen mit `getline()`

Soll in eine Stringvariable eine ganze Zeile gelesen werden, bei welcher Blank und Tabulatoren nicht als Separator gelten sollen, gibt es die Methode `getline()`.

Achtung: Die Methode `getline()` liest alle Zeichen im Eingabepuffer bis zum `newline`-Zeichen. Das `newline`-Zeichen bleibt auf dem Eingabestrom.

```
string zeile;
getline(cin,zeile);
```

Eingabestrom >>		d	e	r		d	i	e		<nl>

Es wird die Zeile „der die “ gelesen. Der Zeilenterminator `newline` bleibt auf dem Eingabestrom.
Zum Lesen von einzelnen Buchstaben, ohne interner Interpretation des Wertes auf dem Eingabestrom, gibt es die I/O Methode `get()`. Diese Methode kann auch bei Dateien mit Binärdaten verwendet werden, da keine Leerzeichen, Newlines usw. überlesen werden.

Achtung:

Werden `cin` und `getline()` abwechselnd verwendet, muss beachtet werden, dass `cin` das die Zeile terminierende `newline` auf dem Eingabepuffer stehen lässt. Ein nachfolgendes `cin` überliest dieses `newline` und wartet auf neue Eingabe. Die Routine `getline` überliest dieses `newline` nicht, sondern interpretiert es als leere Eingabe und fordert den Benutzer zu keiner neuen Eingabe auf. Der Benutzer hat den Eindruck, dass das `getline()` übersprungen wird. Die Lösung besteht darin, vor einem `getline` alle Buchstaben bis zum `newline` zu überlesen und dann erst aufzurufen:

```
string s1,s2,s3;
cout << "bitte ersten String eingeben>>>";
cin >> s1;
cout << "bitte zweiten String eingeben>>>";
cin >> s2;
cout << "Bitte eine ganze Zeile eingeben >>";

//1. Lösung: entfernt einen Buchstaben von cin
cin.ignore();

//2. Lösung: entfernt bis zu 26 Buchstaben, bis zum newline
cin.ignore(26, '\n');

getline(cin,s3);
```

3.8.2 Zeichenweises Lesen und Ausgeben von Daten

In C++ gibt es zwei Methoden zum zeichenweisen Ein/Ausgeben von Daten: `get()` und `put()`. Bei `get()` und `put()` handelt es sich um Methoden aus der Ein/Ausgabe Klasse. Die Notation der beiden Methoden erfolgt in einer bisher unbekannten Schreibweise:
`cin.get(Zeichen)` bzw. `cout.put(Zeichen)`
`cin` repräsentiert den Stream von dem gelesen wird, somit die **Inputklasse**, Zeichen ist das **Argument der Methode**. `cout` repräsentiert den Stream auf den geschrieben wird, somit die **Outputklasse**, Zeichen ist das **Argument der Methode**.
Das folgende Programm kopiert zeichenweise den Input in den Output:

Beispiel 3.14: Zeichenweises Kopieren

```
// file: Grundlagen/read.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {
```

```
        char zeichen;
        while (cin.get(zeichen))
            cout.put(zeichen);

        return 0;
    }
```

Tipp: Um eine Eingabe mit der Tastatur zu beenden, verwendet man die Tastenkombination `ctrl-d`.

3.8.3 Ausgeben von Daten

Eine Schreiboperation in einem Programm hat folgende Struktur:

```
cout << variable-oder-konstante  { << variable-oder-konstante }
```

Beispiel 3.15: Ausgabe von 2 ganzzahligen Werten auf das Terminal

```
cout << val1 << val2;
```

Es werden die Werte der Variable `val1` und `val2` auf dem Terminal ausgegeben. Nachfolgende Ausgaben beginnen hinter dem Wert der Variable `val2`.

Beispiel 3.16: Ausgabe von Text auf das Terminal

```
cout << " Hallo wie gehts ";
cout << " Mir gehts gut " ;
cout << " Ich heisse " << " otto ";
```

Es wird der Text der Reihe nach ausgegeben. Soll zwischen den Ausgaben ein Zeilenumbruch eingestreut werden, dann kann dies mit der vordefinierten Ausgabekonstante `endl` erreicht werden.

```
cout << " Wie geht es Dir? " << endl << " Mir gehts gut " << endl;
```

Übung 3.2

Schreiben Sie ein C++ Programm, das einen String einliest und testet, ob der erste Buchstabe des Strings ein Großbuchstabe ist.

Übung 3.3

Schreiben sie ein Programm *translate*. Dieses Programm soll eine Textzeile einlesen und alle Großbuchstaben in Kleinbuchstaben und umgekehrt konvertieren und ausgeben.

Übung 3.4

Was passiert, wenn Sie einen Text mit Umlauten eingeben?

3.8.4 Arbeit mit Dateien

Zur permanenten Speicherung von Daten verwendet man in Betriebssystemen Dateien. Dateien werden über einen Namen innerhalb eines Verzeichnisses eindeutig identifiziert. Binäre Daten speichernde Dateien werden Binärdateien und Texte speichernde Dateien werden Textdateien genannt.

3.8.4.1 Textdateien unter UNIX

Unter UNIX wird eine Textdatei als geordnete Folge oder „Strom“ (`stream`) von Zeichen (Bytes) betrachtet. Die einzelnen Zeilen sind durch ein Zeilentrennzeichen (`\n`) getrennt. Abbildung 3.1 zeigt 5 Zeilen, die mit einem Texteditor in die Datei `eingabe.txt` geschrieben werden.

```
Eins
Zwei
Drei
4 5
Ende
```

Abbildung 3.1: Datei `eingabe.txt`

Der Inhalt wird in einer Datei `eingabe.txt` gespeichert.
In der Datei werden alle Buchstaben mit ihrem 8 Bit ASCII-Code abgespeichert. Damit die Information des Zeilenumbruchs nicht verloren geht, wird in der Datei an seine Stelle ein so genannter Special Character eingefügt. Dieser Special Character hat die Aufgabe bei der Ausgabe einen Zeilenumbruch zu bewirken.
Der Special Character hat in der ASCII-Codierung die Nummer 10, (Oktalwert = 12) und heißt Newline (Abkürzung = `nl`).
Aus diesen Gründen wird auf der Festplatte der Inhalt folgendermaßen Zeichen für Zeichen abgespeichert:

```
E i n s nl Z w e i nl D r e i nl 4   5 nl E n d e nl
```

Abbildung 3.2: Datei `eingabe.txt` (Bytedarstellung)

Unter UNIX gibt es den Befehl `od` (Octal Dump) mit dem man den Inhalt einer Datei Zeichen für Zeichen darstellen (=„dumpen“) kann. Dem Befehl muss man angeben, mit welchem Format die Zeichen der Datei konvertiert werden sollen, bevor er auf dem Terminal ausgegeben wird. In der linken Spalte gibt der Befehl `od` die oktale Bytenummer des ersten Zeichens in der Datei aus; in den restlichen Spalten die Zeichen entsprechend der ausgewählten Konvertierung. (Zur weiteren Information zum Befehl `od` geben Sie den UNIX Befehl `man od` ein.) z.B. Ausgabe der Bytenummer und des Dateiinhalts als Folge von oktal- (binär-) und ASCII-Werten:

```
unixhost> od -ab eingabe.txt

0000000 E   i   n   s   nl  Z   w   e   i   nl  D   r   e   i   nl  4
        105 151 156 163 012 132 167 145 151 012 104 162 145 151 012 064
0000020 sp  5   nl  E   n   d   e   nl
        040 065 012 105 156 144 145 012
```

Abbildung 3.3: Datei `eingabe.txt` (Bytedarstellung)

Beispiel 3.17: Ausgabe der Zeichen als ASCII-Wert

```
unixhost> od -a eingabe.txt

0000000 E  i n  s nl Z w e  i nl D r e i nl 4
0000020 sp 5 nl E n  d e nl
```

Abbildung 3.4: Datei `eingabe.txt` (ASCII-Darstellung)

Befehle zur Ausgabe von Textdateien interpretieren die `nl`-Zeichen bei der Ausgabe auf das Terminal richtig und erzeugen einen Zeilenumbruch anstelle des Buchstabens `nl`.

Textbefehl `cat` zur Ausgabe einer Textdatei:

```
unixhost> cat eingabe.txt
Eins
Zwei
Drei
4 5
Ende
```

3.8.4.2 Textdateien unter Windows

Wie unter Unix wird bei Windows eine Textdatei als geordnete Folge oder „Strom“ (stream) von Zeichen (Bytes) betrachtet. Die einzelnen Zeilen werden aber von den zwei Zeichen Carriage-Return (`<cr>`) und durch das Newline (`<nl>`) getrennt.

Beispiel: Textdatei unter Windows

Abbildung 3.5 zeigt 5 Zeilen die mit einem Texteditor in die Datei `eingabe.txt` geschrieben werden.
In der Datei werden alle Buchstaben mit Ihrem 8 Bit ASCII-Code abgespeichert. Damit die Information des Zeilenumbruchs nicht verloren geht, werden in der Datei an seiner Stelle zwei Special Character eingefügt. Die Special Character haben die Aufgabe bei der Ausgabe einen Zeilenumbruch zu bewirken und haben in der ASCII-Codierung die Nummern 12 (oktalwert = 15) und 10 (Oktalwert = 12). Sie heißen Carriage Return (Abkürzung = `cr`) und Newline (Abkürzung = `nl`).

```
Eins
Zwei
Drei
4 5
Ende
```

Abbildung 3.5: Datei `eingabe.txt` unter WINDOWS

Aus diesen Gründen wird auf der Festplatte der Inhalt folgendermaßen Zeichen für Zeichen abgespeichert:

```
E i n s cr nl Z w e i cr nl D r e i cr nl 4   5 cr nl E n d e cr nl
```

Abbildung 3.6: Datei `eingabe.txt` unter WINDOWS (ASCII-Darstellung)

Beispiel 3.18: Ausgabe der Zeichen als ASCII - Wert unter WINDOWS

```
unixhost> od -a eingabe.txt
```

```
0000000 E   i   n   s   cr  nl  Z   w   e   i   cr  nl  D   r   e   i
        105 151 156 163 015 012 132 167 145 151 015 012 104 162 145 151
0000020 cr  nl  4   sp  5   cr  nl  E   n   d   e   cr  nl
        015 012 064 040 065 015 012 105 156 144 145 015 012
```

Abbildung 3.7: Datei `eingabe.txt` (ASCII-Darstellung) unter WINDOWS

Befehle zur Ausgabe von Textdateien auf das Terminal interpretieren die <cr> und <nl> Zeichen richtig und erzeugen einen Zeilenumbruch auf dem Terminal anstelle der Buchstabens <cr> <nl>.

3.8.4.3 Lesen aus einer Datei

Möchte man den Inhalt der Datei in einem C++ Programm lesen, dann muss im Programm die include-Datei `fstream` eingebunden werden. Als nächstes muss für die Datei im C++ Programm eine interne Input-file-stream-Variable definiert und der Datei zugeordnet werden.
Die Zuordnung der Datei kann entweder bei Deklaration der Input-file-stream-Variablen durch Angabe des Filenamens realisiert werden oder mit Hilfe der viel flexibleren Input/Output Methode `open()`. Da die `open()`-Methode als Argument kein Objekt der String-Klasse akzeptiert, sondern nur ein Argument vom Typ C-String, muss ein Filename des Datentyps String in einen C-String konvertiert werden. Dies kann mit der String-Methode `c_str()` realisiert werden.
Nachdem die Zuordnung vorgenommen wurde, kann von der Datei über die Input-file-stream-Variable als Eingabekanal gelesen werden.

Beispiel 3.19: Dateieingabe von der Datei `eingabe.txt`

```
include <fstream>
ifstream eingabe("eingabe.txt");
```

oder

```
include <fstream>
include <string>
ifstream eingabe;
string filename("eingabe.txt");

eingabe.open(filename.c_str());
```

3.8.4.4 Lesen aus einer WINDOWS Textdatei

Wie oben erwähnt, endet unter WINDOWS die Zeile mit `<cr>` `<nl>`. Wird eine Zeile aus einer WINDOWS Datei mit der I/O Methode `getline()` gelesen, dann wird der Special Character `<cr>` vom Eingabestrom gelesen und dem String angefügt:

Beispiel 3.20: Lesen ganzer Zeilen mit `getline()`

```
string zeilenEingabe;
getline(cin,zeilenEingabe);
```

Eingabestrom >>		d	e	r		d	i	e	<cr>	<nl>

Es wird die Zeile „der die`<cr>`“ gelesen, der Zeilenterminator `<nl>` bleibt auf dem Eingabestrom.

3.8.4.5 Schreiben in eine Datei

Das Schreiben auf eine Datei unterscheidet sich vom Lesen aus einer Datei nur dadurch, dass der externen Datei eine interne Output-file-stream Variable zugeordnet wird. Die Zuordnung der file-stream Variablen kann wie bei der Eingabe bei der Deklaration der file-stream Variable oder mit der Eingabe/Ausgabemethode `open()` realisiert werden.

```
#include <fstream>
ofstream ausgabe("ausgabe.txt");
```

oder

```
#include <fstream>
#include <string>
ofstream ausgabe;
string filename("ausgabe.txt");

ausgabe.open(filename.c_str());
```

Ist die entsprechende Ausgabedatei noch nicht vorhanden, so wird sie vom Compiler generiert. Um eine geöffnete Datei nach Gebrauch zu schließen, benutzt man den Befehl `filename.close()`.

Beispiel 3.21: Lesen und Schreiben mit Dateien

```
// file: Grundlagen/io.cpp
// description:

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    string filenameIn, filenameOut;
    string zeile;
    ifstream textfileIn;
    ofstream textfileOut;

    cout << " Name der Eingabedatei: ";
    cin >> filenameIn;
    cout << " Name der Ausgabedatei: ";
    cin >> filenameOut;

    textfileIn.open(filenameIn.c_str());
    textfileOut.open(filenameOut.c_str());

    if (!textfileIn) //siehe Status-Flags
    {
        cout << " Fileopen Error on " << filenameIn << endl;
        return(-1);
    }
    while (getline(textfileIn, zeile)) {
        cout << " Die Zeile war: #" << zeile << "#" << endl;
        textfileOut << " Die Zeile war: #" << zeile << "#" << endl;
    }
    return 0;
}
```

Ausgabe beim Lesen der Windows Datei:

```
./Beispiele/EXE/file.exe
 Name der Eingabedatei: >>>eingabe_win.txt
 Name der Ausgabedatei: >>>s
#Die Zeile war #Eins
#Die Zeile war #Zwei
#Die Zeile war #Drei
#Die Zeile war #4 5
#Die Zeile war #Ende
```

Ausgabe beim Lesen der UNIX Datei:

```
./Beispiele/EXE/file.exe
 Name der Eingabedatei: >>>eingabe_unix.txt
 Name der Ausgabedatei: >>>s
```

```
Die Zeile war #Eins#
Die Zeile war #Zwei#
Die Zeile war #Drei#
Die Zeile war #4 5#
Die Zeile war #Ende#
```

Beispiel 3.22: File-I/O mit vorgegebenen Dateinamen

```
// file: Grundlagen/fileIo.cpp
// description:

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    string filenameIn, filenameOut;
    string zeile;
    ifstream textfileIn("eingabe.txt");
    // oder: ifstream textfileIn(filenameIn.c_str() );
    ofstream textfileOut("ausgabe.txt");

    if (!textfileIn) //siehe Status-Flags
    {
        cout << " Fileopen Error " << endl;
        return(-1);
    }

    while (getline(textfileIn, zeile)) {
        cout << " Die Zeile war: " << zeile << endl;
        textfileOut << " Die Zeile war: " << zeile << endl;
    }

    texfileIn.close();
    texfileOut.close();
    return 0;
}
```

Beispiel 3.23: File-I/O mit eingelesenen Dateinamen ohne `open()`

```
// file: Grundlagen/fileIo2.cpp
// description:

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
```

```
    string filenameIn, filenameOut;
    string zeile;

    cout << " Name der Eingabedatei: ";
    cin >> filenameIn;
    cout << " Name der Ausgabedatei: ";
    cin >> filenameOut;

    ifstream textfileIn("eingabe.txt");
    ofstream textfileOut("ausgabe.txt");

    if (!textfileIn) {
        cout << " Fileopen Error " << endl;
        return(-1);
    }

    while (getline(textfileIn, zeile)) {
        cout << " Die Zeile war: " << zeile << endl;
        textfileOut << " Die Zeile war: " << zeile << endl;
    }

    return 0;
}
```

Übung 3.5: Datei-Handling

Lesen Sie die erste Zeile der Datei `eingabe.txt` und speichern Sie diese Zeile in der Datei `ausgabe.txt`.

Übung 3.6

Schreiben sie ein Programm `count_punct`!
Dieses Programm soll eine Textzeile einlesen, diese Zeile ausgeben und in der nächsten Zeile die Positionen aller Punktuationszeichen anzeigen.

3.9 Operatoren

In der Programmiersprache C++ gibt es nicht nur eine große Anzahl von vordefinierten Operatoren, die Operanden verknüpfen, sondern es gehört als zentrale Eigenschaft zu dieser Programmiersprache, dass der Programmierer eigene Operatoren für beliebige Operanden definieren kann oder existierende Operatoren für seine Operanden umdefinieren kann. Die Möglichkeit der Redefinition nennt man Überladen von Operatoren.

Der freizügige Umgang mit Operatoren erlaubt es C++ für jeden Zweck eigene Operatoren zu verwenden oder zu definieren. Die Programme werden damit viel lesbarer.

3.9.1 Arithmetische Operatoren

Binäre arithmetische Operatoren sind:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division, reellwertig
/	Division, ganzzahlig
%	Modulo Operator

Tabelle 3.6: Binäre arithmetische Operatoren

3.9.2 Relationale und Logische Operatoren

Es gibt die logischen Werte wahr (jede Zahl ungleich Null) und falsch (nur die ganze Zahl Null). Ausdrücke mit logischen Operatoren werden ihrer Priorität entsprechend ausgewertet. Operatoren mit höherer Priorität werden vor Operatoren mit niedriger Priorität ausgewertet. Bei gleicher Priorität werden die Ausdrücke von links nach rechts ausgewertet. Je niedriger die Rangfolge, desto höher die Priorität.

Logische Operatoren sind:

Zeichen	Name	Rangfolge
!	nicht	2
<	kleiner	7
<=	kleiner gleich	7
>	größer	7
>=	größer gleich	7
==	gleich	8
!=	ungleich	8
&&	und	12
||	oder	13

Tabelle 3.7: Logische Operatoren

Achtung:

Die Priorität von `&&` ist größer als die von `||`, sowohl `&&` als auch `||` haben aber eine niedrige Priorität als `>`, `<`, `>=` und `<=`. (z. B. `x < y && y < x`) Die Negation (`!`) ist einstellig und konvertiert `TRUE` (nicht Null) in `FALSE` (Null) und `FALSE` (Null) in `TRUE` (nicht Null).

Beispiel 3.24: Priorität der Operatoren

```
if ((1<lim-1) || (!valid) && (x!=4)) x=0
```

3.9.3 Die Wahrheitstabelle bei logischen Ausdrücken

bool p,q;

p	q	p || q	p && q	! p
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

Tabelle 3.8: Wahrheitswertabelle bei logischen Ausdrücken

3.9.4 String Operatoren

Für die Stringklasse wurde in C++ eine Reihe von Operatoren überladen, die den Umgang mit Strings sehr erleichtern.

3.9.4.1 Zuweisungsoperator: =

Dieser Operator schafft genügend Speicherplatz für den Zuweisungsoperanden und überträgt jeden Buchstaben in diesen String.

Beispiel 3.25: Stringzuweisung

```
string zweiter,erster="hallo";
zweiter = erster;
```

3.9.4.2 Verknüpfungsoperator: +

Dieser Operator verknüpft beliebige Strings, und hängt sie aneinander. Für den verknüpften String alloziert der Verknüpfungsoperator genügend Speicherplatz. Der verknüpfte String kann mit dem Zuweisungsoperator auch an andere Strings zugewiesen werden. In diesem Fall wird der Speicher des erzeugten Strings erweitert.

Beispiel 3.26: Stringverknüpfung

```
string vorname="Otto",nachname="Barolo";
string name;

name = vorname + " " + nachname;
cout << name << endl;
```

<	lexikographisch kleiner
<=	lexikographisch kleiner gleich
>	lexikographisch größer
>=	lexikographisch größer gleich
==	lexikographisch gleich
!=	lexikographisch ungleich

Tabelle 3.9: Lexikographischer Vergleich bei Strings

3.9.4.3 Lexikographischer Vergleich bei Strings:

Der lexikographische Vergleich von Strings stellt ein gutes Beispiel für das Überladen von Operatoren dar.

Beispiel 3.27: Lexikographischer Vergleich

```
// file: Grundlagen/lexComp.cpp
// description:

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

int main() {
    const string Ref = "Hallo";
    string eingabe;

    cout << " Bitte geben Sie einen String ein >>>";
    cin >> eingabe;

    if (eingabe > Ref)
        cout << eingabe << " groesser als " << Ref << endl;

    if (eingabe == Ref)
        cout << eingabe << " identisch zu " << Ref << endl;

    if (eingabe < Ref)
        cout << eingabe << " kleiner als " << Ref << endl;

    return 0;
}
```

4 Konstanten

In C++ können Objekte definiert werden, denen ein fester Wert zugewiesen wird, der sich im gesamten Programm nicht ändern darf. Jeder Zugriff auf das Objekt wird beim Kompilieren kontrolliert, ob er nicht den Wert ändert. Diese Art von Objekte nennt man konstante Objekte. Ein konstantes Objekt wird wie eine Variable deklariert, es muss lediglich bei der Deklaration das zusätzliche Keyword `const` dem Typbezeichner vor- oder nachgestellt werden. Einer Konstanten kann nur bei der Deklaration ein sogenannter Initialwert zugewiesen werden, der sich im gesamten Programm nicht mehr ändern darf. Konstante werden im folgenden Programm als Read-Only-Objekte behandelt.

Beispiel 4.1: Deklaration einer Konstanten

```
const double Pi = 3.1415927;
double const Pi = 3.1415927;    //gleichbedeutend

Pi += 2;    //Fehlermeldung beim Kompilieren!
```

Formatierungsvorschlag:

Wir entscheiden uns, das Keyword `const` dem Typbezeichner nachzustellen. Der Grund liegt darin, dass es eine direkte Antwort auf die Frage liefert: „Was ist konstant?“[1] . Wir lesen also `const` Ausdrücke von rechts nach links. Die `const` Variablennamen beginnen mit einem Großbuchstaben.

Beispiel 4.2: Initialisierung von Konstanten

```
int main() {
    int val = 2;
    int val2 = 3;

    int q = 0;
    // int const q=0; //Initialisierung
    int const& RefQ = q;

    //FEHLER:
    RefQ = 3; //nicht zulässig
```

1 siehe Vandevoorde, Josuttis(Dav02)

```
    //RICHTIG
    q = 3;       //zulässig

    //////////////////
    int* const I = &val; // Zeiger auf Element mit Datentyp int
                         // ist konstant!
    // RICHTIG:
    // Der Wert kann sich ändern, nicht aber die Adresse
    val = 4; //zulässig

    //FEHLER:
    I = &val2; //nicht zulässig

    //////////////////
    int const* J = &val; // Zeiger auf konstantes Element
                                         // mit Datentyp int.
    // Der Wert kann sich nicht ändern, aber die Adresse
    J = &val2; //zulässig

    //FEHLER:
    *J = 10; //nicht zulässig val2 = 10; würde funktionieren

    //////////////////
    int const* const K = &val; // Konstanter Zeiger auf ein
                         //konstantes Element, Datentyp int.
    // Der Wert und die Adresse dürfen sich nicht ändern.

    //FEHLER:
    *K = 10; //nicht zulässig

    //FEHLER:
    K = &val2; //nicht  zulässig


}
```

Ausgabe:

```
[const][matrix][max] g++ const.cpp
const.cpp: In function ?int main()?:
const.cpp:14: error: assignment of read-only reference 'RefQ'
const.cpp:25: error: assignment of read-only variable 'I'
const.cpp:36: error: assignment of read-only location '* J'
const.cpp:45: error: assignment of read-only location '*(const int*)K'
const.cpp:49: error: assignment of read-only variable 'K'
```

Zugriff auf const-Objekte mit Methoden

Da `const`-Objekte read-only sind, gibt es keine verändernde Methoden wie z.B. `setName()` für diese Objekte. Aber auch Methoden wie `getName()` oder `display()` können zunächst nicht mehr aufgerufen werden. Grund hierfür ist, dass der Kompiler an dem Keyword `const` erkennt, dass die Elemente nur lesend verarbeitet werden können.

Referenzen auf const Objekte

Wie erklärt ist eine Referenz ein anderer Name für eine Variable. Soll eine Referenz als anderer Name für eine Variable verwendet werden, die auf ein konstantes Objekt verweist, muss der Datentyp des Objekts als `const` definiert werden. Wir benötigen somit eine Referenz auf ein `const`-Objekt.

Formatierungsvorschlag:

Wenn wir mit Referenzen arbeiten, dann setzen wir zwischen dem Variablennamen und dem Referenzzeichen ein Space um klar zu trennen, was der Typ und was der Bezeichner ist.

```
int const Q = 0;
int const& RefQ = q;
Q = 1; //nicht zulässig
RefQ = 4; //nicht zulässig
```

Achtung: Umgekehrt ist es möglich, dass einer Referenz auf ein konstantes Objekt ein nicht konstantes Objekt zugewiesen wird, z.B.:

```
int q = 0;
int const &QRef = q;
q = 5; //zulässig
QRef = 3; //nicht zulässig
```

Mit der Referenz `qref` kann nun nur lesend auf `i` zugegriffen werden (sog. Read-Only-Bezeichner, der im Gegensatz zu einer „normalen“ Referenz mit einer Konstanten initialisiert werden kann).

const-Methoden:

Read-Only-Methoden sind Methoden, die nur lesend auf Daten zugreifen und für konstante Objekte aufrufbar sein sollen. Dies muss man bei Deklaration und Definition durch Anhängen des Schlüsselwortes `const` an den Funktionskopf kennzeichnen. Auch hier gilt wieder: `const` Definitionen werden von rechts nach links gelesen: „Was ist konstant?“

Beispiel 4.3: const Methoden

```
unsigned long GetNr() const;
//GetNr() ist nun read-only und für const-Objekte aufrufbar
```

Es können weiterhin für nicht konstante Objekte Read-Only-Methoden aufgerufen werden. Versucht eine `const`-Methode ein Datenelement zu verändern, erzeugt der Compiler eine Fehlermeldung. Ebenso, wenn sie eine andere Methode aufruft, die nicht als `const` deklariert wurde.

const- und nicht const-Versionen einer Methode:

Da das Schlüsselwort `const` zur Signatur einer Methode gehört, können für jede Elementfunktion zwei Versionen geschrieben werden:

- eine Read-Only-Version, die automatisch für konstante Objekte aufgerufen wird,
- eine „normale“ Version, die für nicht konstante Objekte aufgerufen wird.

Nach unserer Regel für `const`-Definitionen kommt bei `const`-Pointern das Schlüsselwort `const` immer nach dem * !

```
int *const P1 = q;      //const Pointer auf eine int-Variable
int const* P2 = q;      //Pointer auf einen const int
```

Für maximale Sicherheit wird dringend empfohlen, wenn immer es möglich ist, `const` zu verwenden! (Ausnahme: arrays)

```
ptr = buf;
ptr = &buf[0]; //äquivalent
ptr = ptr +10; // ptr++
int const& RefQ = q;
```

Formatierungshinweise:

`const` beginnen immer mit einem Großbuchstaben.

5 Strukturierung von Programmen

5.1 Kontrollstrukturen

5.1.1 Statements und Blöcke

Jeder Ausdruck, abgeschlossen von einem Semikolon, ist ein Statement. Somit ist das Semikolon kein Trennsymbol sondern das Abschlusszeichen eines Statements.
Mehrere Statements können mit geschweiften Klammern zu einem Block zusammengefasst werden. Der Block (=Block-Statement) gilt nach außen wie ein Statement und wird nicht mit einem Semikolon abgeschlossen.
Es gibt folgende Statements:

- Expression-Statement
- Compound-Statement
- Selection-Statement
- Iteration-Statement
- Jump-Statement
- Labeled-Statement

5.2 compound statement

Die geschweiften Klammern gruppieren Statements zu einem Compound-Statement (Block). Innerhalb eines Compound-Statements können an beliebiger Stelle Variablen deklariert werden. Die Lebensdauer solcher Variablen ist auf diesen Block beschränkt, da sie nach Verlassen des Blocks wieder gelöscht werden.

Beispiel 5.1: Block-Statements, 1

```
{   int a;
    a=x;
}   /* Hier kein Semikolon notwendig, da Block-Statement */
```

oder:

Beispiel 5.2: Block-Statements, 2

```
if (a==0) {
   b=1;
   c=1;
   d=1;
   cout << b << endl;
}
```

oder:

Beispiel 5.3: Block-Statements, 3

```
if (a==0) {
   int i;
   i = a + 1;
   cout << i << endl;
}
```

5.3 Selection-Statements

Selection-Statements werten eine Bedingung aus und führen auf Grund des Ergebnisses der Bedingung bestimme Anweisungen aus.

5.3.1 Selection-Statement: `if` , `if - else`

```
if (expression)
    statement1
else
    statement2
```

Das `if`-Statement testet den logischen Wert der Bedingung `(expression)`. Falls der Wert `TRUE` ist, wird `statement1` ausgeführt. Falls der Wert `FALSE` ist, wird `statement2` hinter dem `else`-Zweig (soweit vorhanden) ausgeführt.
Bei der Abarbeitung von `statement1`, bzw. `statement2` ist zu beachten, dass das Selection-Statement nur eine einzelne Anweisung ausführt. Soll `statement1` oder `statement2` aus mehreren Anweisungen bestehen, dann müssen die Anweisungen mit geschweiften Klammern zu einem einzelnen Compound-Statement zusammengefasst werden.
Bei geschachtelten `if .. else` gehört das `else` immer zum näheren `if`.

Achtung: `expression` muss immer in runden Klammern eingeschlossen werden.

Beispiel 5.4

```
if (i == 2)
    cout << "i ist gleich zwei" << endl;
else
    cout << "i ist ungleich zwei" << endl;
```

oder:

```
if ((name == "Otto") && (praemie == 1000)) {
    cout << "Otto hat 1000 Franken erwischt" << endl;
    praemie = 0;
} else
    cout << "Du armer Hund" << endl;
```

Problematisch:

```
if (i == 2)
    cout << "i ist gleich zwei" << endl;
    cout << " Das ist der if-Zweig" << endl;
```

5.3.2 Selection-Statement: `switch`

```
switch (expression) {
        case constant - expr : statements1
        case constant - expr : statements2
default: statementsn
}
```

Jedes Label hat eine oder mehrere ganzzahlige Konstanten oder Konstantenausdrücke. Entsprechend dem Wert von `expression` werden sequentiell alle `case constant-expr` ausgewertet und verglichen, ob sie mit dem Wert übereinstimmen. Bei Übereinstimmung werden die nachfolgenden Statements ausgeführt und beim nächsten `case`-Statement weiter getestet. Soll die Abarbeitung der sequentiellen Tests nach der Ausführung eines Statements abgebrochen werden, so muss dieses Statement mit `break` abgeschlossen werden. Die Abarbeitung wird dann hinter dem gesamten `switch`-Statement fortgesetzt.

Wird keine Übereinstimmung zwischen dem Wert von `expression` und `constant-expr` gefunden, so wird auf einen eventuell vorhandenen `default`-Fall verzweigt.

Beispiel 5.5

```
char c;
switch (c) {
    case '0' : case '1' :   cout << "0,1" << endl; break;
    case '2' :              cout << "2" << endl;
    case '3' :              cout << "3" << endl; break;
    default  :              cout << " Nichts davon" << endl;
}
```

5.4 Iteration-Statements

5.4.1 Iteration-Statement: `while`

```
while (expression)
       Statement
```

Solange `expression` `TRUE` (ungleich Null) ist, wird `Statement` ausgeführt. Es wird genau ein Statement ausgeführt (vgl. Selection-Statement). Sollen mehrere Statements ausgeführt werden, müssen sie mit geschweiften Klammern zu einem Compound-Statement zusammengefasst werden.
`Statement` wird solange wiederholt, bis der Wert von `expression` `FALSE` (gleich Null) ist.

Beispiel 5.6

```
// file: Strukturierung/while.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int while1(void);

int main() {
    cout << " Hallo! While loop " << endl;
    while1();
    return 0;
}

int while1(void) {
    int num;
    cout << "Gib den Countdown Wert ein >>>";
    cin >> num; /* Achtung vor Endlosschleifen */
    while (num > 0) {
        num--;
        cout << " Countdown laeuft >>" << num << endl;
    }
    cout << " Start " << endl;
    return 0;
}
```

Beispiel 5.7: *Achtung vor Endlosschleifen!*

```
int i = 5;
while (i) {// wahr ist alles, was nicht 0 ist.
   i++;
   cout << i << endl;
}
```

5.4.2 Iteration-Statement: do

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

`statement` wird ausgeführt und dann `expression` ausgewertet. Falls `expression` `TRUE` (ungleich Null) ist, wird `statement` erneut ausgeführt. Wird `expression` gleich Null (`FALSE`), terminiert die Schleife und das Programm wird hinter dem `do-Statement` fortgesetzt.

Beispiel 5.8

```
// file: Strukturierung/do.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {
    string passwd = "eins";
    string gelesenPasswd;
    int zaehler = 0;

    do {
        cout << " Enter Password >>>";
        cin >> gelesenPasswd;
        zaehler++;
    } while (gelesenPasswd != passwd && zaehler < 5);

    if (gelesenPasswd == passwd)
        cout << " Passwort gefunden " << endl;
    else
        cout << " Passwort nicht gefunden " << endl;

    return 0;
}
```

5.4.3 Iteration-Statement: `for`

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

- `expr1` : Wird vor dem Durchlauf der Schleife ausgeführt (Initialisierung)
- `expr2` : Testbedingung der Schleife; Wiederholung falls `expr2 TRUE` (ungleich Null)
- `expr3` : Wird nach einem Durchlauf von Statement ausgeführt (Inkrement)

entspricht:

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

Achtung: Im Gegensatz zu anderen Programmiersprachen ist die `for`-Schleife bei C viel flexibler, da die Komponenten `expr1, expr2, expr3` einer `for`-Loop beliebige Ausdrücke sein können, deren Werte auch innerhalb der Komponenten verändert werden können.

Beispiel 5.9: Endlosschleifen mit `for`

Es dürfen auch `expr's` der `for`-Schleife fehlen – eine Endlosschleife:

```
for (;;);
```

Der Index einer for-Schleife muss nicht unbedingt ein ganzzahliger Wert sein:

Beispiel 5.10: Indizes von `for`-Schleifen

```
// file: Strukturierung/for.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int for2(void);
int for3(void);

int main() {
    cout << " Hallo! For loop " << endl;
    for2();
    for3();
    return 0;
```

```
}

int for2(void) {
    char a;
    cout << " Die Buchstaben von a bis g lauten: ";
    for (a = 'a'; a <= 'g'; a++)
        cout << a;
    cout << endl;
}

int for3(void) {
    float x;
    cout << " Die Reihe von x - 1/x lautet: ";
    for (x = 3; x > 0.00005; x = x - 1 / x)
        cout << x << ", ";
    cout << endl;
    cout << " Eingabe der Obergrenze für for loop >>> ";
    int i, n;
    cin >> n;
    for (i = 0; i < n; i++)
        cout << " Schleifendurchlauf Nummer " << i << endl;
}
```

Komplexe Ausdrücke in `expr1, expr2 und expr3`

Als Rumpf der `for`-Schleife wird die leere Ausweisung verwendet. Sämtliche Berechnungen werden innerhalb der `for`-Schleife ausgeführt. Dies ist jedoch ein sehr komplizierter Programmstil und somit abzulehnen.

```
for (i=0,j=1,x=10; i=j*2*x; i = i*j,x--);
```

5.4.4 Iteration-Statement: `range-for`

Mit Range-For Schleifen kann man bestimmte Werte einer Variable durchlaufen. Dieses Iterations-Statement ist erst ab C++11 benutzbar. Genauere Informationen siehe Kapitel 16.

```
for (int n : {1,3,5,7,9}) {
    cout << n << endl;
}
```

Formatierungsvorschläge:

Zwischen Keywords, runden und geschweiften Klammern soll ein Whitespace stehen. z.B.

Beispiel 5.11

```
for (...) {
 //..
}
```

Ebenso bei den anderen Statements (`while`, `do-while`, `switch`, `if`) und `try` & `catch`. (siehe Kapitel 14).

5.5 Das Wichtigste in Kürze

Der `for`-Loop wird verwendet:

- wenn der Schleifenindex im Vordergrund steht. (z.B. Durchlaufen von Arrays).
- bei einfachen Interationsschritten
- falls die Iteration und das Abbruchkriterium eng zusammenhängen

Der `while`-Loop wird verwendet:

- wenn das Abbruchkriterium im Vordergrund steht
- beim Lesen von Dateien
- wenn zuerst getestet werden soll, bevor die Statements im while-Block ausgeführt werden, da sonst Fehler auftreten könnten (z.B. Arithmetischer Ausdruck um Division durch Null zu vermeiden)

Der `do`-Loop wird verwendet:

- wenn die Statements im do-Block im Vordergrund stehen.
- wenn zuerst die Statements ausgeführt werden sollen und dann die Bedingung getestet werden soll. (z.B.: Einlesen von Abbruchkriterien)

Aufgaben und Übungen zum Kapitel – Kopieren von Dateien und Erstellen von Wortlisten

Übung 5.1

Schreiben Sie ein C++ Programm `copy.cxx`.
Das Programm soll eine Datei in eine zweite Datei kopieren. Das Programm soll den Benutzer nach den Namen der alten und neuen Datei fragen.

Übung 5.2

Schreiben sie ein C++ Programm `wordlist.cxx`.
Dieses Programm soll eine Datei lesen und die Wörter der Datei zeilenweise in eine neue Datei kopieren. Die Wörter in der neuen Datei dürfen am Ende keine Punktuationszeichen haben. Sie müssen also vor dem Übertragen gelöscht werden.
Der Name der Eingabedatei soll vom Benutzer erfragt werden und die neue Datei soll den Namen der alten Datei bekommen, die Extension des Dateinamens der neuen Datei soll aber .wordlist sein.
Beispiel:

```
EINGABEFILENAME>text.txt
AUSGABEFILENAME>text.wordlist
```

Übung 5.3: Schleifen und Datei-Handlung

Schreiben Sie drei C++ Programme: `pword_for.cxx`, `pword_while.cxx` und `pword_do.cxx`
Das Programm soll den Benutzer höchstens 5 Mal nach einem Passwort fragen.
Die interne Abfrageschleife soll beim Programm `pword_for.cxx` als `for`-Schleife, beim Programm `pword_while.cxx` als `while`-Schleife und beim Programm `pword_do.cxx` als `do`-Schleife implementiert werden.
ACHTUNG: Das interne Passwort soll aus der Datei `mypasswd.txt` gelesen werden. Verwenden Sie also Routinen des FILE I/O aus Kapitel 3.8.4. Die Eingaben des Benutzers, auch die Eingaben aus den letzten Programmläufen, sollen in einer Datei `log.txt` gespeichert werden (TIPP: open for append!).

6 Einsatz von Strings

6.1 Einfï¿½hrung

In C++ sind Strings keine einfache Datenstruktur, sondern sie werden intern als eine Klasse implementiert und somit als Objekte gespeichert. Wie bei jedem Klassenobjekt, besitzt ein String Attribute, man kann spezielle String-Methoden aufrufen und String-Operationen ausfï¿½hren. Unter den Attributen von Stringobjekten speichert man die Eigenschaften des Strings, die Operationen und Methoden legen fest, wie die Attribute eines Objektes bearbeitet werden kï¿½nnen.

Beispiel:

Wir haben einen String namens `BspString` mit dem Inhalt `"Hallo Welt"` initialisiert. Nachfolgend sehen Sie einige Attribute und Methoden wie die im string-Objekt BspString gespeichert sind:

```
string BspString;
    Attribute
        Der eigentliche Text: "Hallo Welt"
        Anzahl der Buchstaben: 10
    Methoden
        data()
        size()
```

Die internen Attribute eines Objektes sind nach auï¿½en hin verborgen, und sie kï¿½nnen nur ï¿½ber vordefinierte Methoden gelesen bzw. verï¿½ndert werden. Dieses Konzept entspricht in der Objektorientierten Programmierung der sogenannten „Encapsulation“.

6.1.1 Bitbreite der Buchstaben von Strings

In der Programmiersprache C++ erlaubt die Datenstruktur `string` nur eine Folge von bis zu 8-Bit kodierten Buchstaben abzuspeichern. In der Klasse `string` verbirgt sich also ein `Array` von `unsigned char` und diese kï¿½nnen somit nur ASCII Buchstabenketten oder 8 Bit Erweiterungen (z.B.ISO Character) abspeichern. UNICODE Zeichen, die einen Buchstabencode mit mehr als 8-Bit benï¿½tigen, kï¿½nnen in Strings nicht abgespeichert werden. Fï¿½r UNICODE Zeichen gibt es eine Erweiterung von 8-Bit Buchstaben `unsigned char` auf 16 oder 32 -Bit Buchstaben `wchar_t` und eine Erweiterung von `string` auf `wstring`. Hier muss auf das Kapitel 8 „Internationalisierung“ verwiesen werden.

6.1.2 Deklaration und Initialisierung von Stringvariablen mit 8-Bit Buchstabencode

Man deklariert Strings mit:

```
string name;
```

C++ fï¿½hrt dadurch automatisch eine sogenannte **Konstruktormethode** aus, die in der Klasse String definiert wurde. Die Konstruktormethode erzeugt ein Objekt, das alle Attribute und Methoden der Klasse enthï¿½lt, und reserviert Speicher fï¿½r den Inhalt der Attribute. (siehe Kapitel 9)
Man kann einem Objekt auch bei der Deklaration bereits einen Anfangswert, den **Initialwert**, zuweisen:

```
string name = "Bauer";
```

6.2 Methoden fï¿½r Stringobjekte

Sobald ein Stringobjekt deklariert ist, hat man Zugriff auf seine Methoden.
Zum Beispiel ist es mï¿½glich, ï¿½ber eine Methode der Klasse String die Lï¿½nge eines String-Objekts zu erfahren:

```
string name;
int laenge;

name = "Bauer";
laenge = name.size(); //laenge erhï¿½lt den Wert 5.
```

Die Klasse String verfï¿½gt ï¿½ber eine ganze Reihe eingebauter Methoden, unter anderem die in Tabelle 6.1 abgebildeten.

=, assign()	Wert zuweisen
+=, append()	Zeichen anhï¿½ngen
insert()	Zeichen einfï¿½gen
swap()	die Werte zweier Strings vertauschen
erase()	Zeichen lï¿½schen
clear()	alle Zeichen entfernen (leert den String)
resize()	Anzahl der Zeichen ï¿½ndern (Zeichen am Ende des Strings hinzufï¿½gen oder lï¿½schen)
replace()	Zeichen ersetzen
+	Strings konkatenieren
==, !=, <, <=, >, >=, compare()	Strings vergleichen
size(), length()	gibt die Anzahl der Zeichen zurï¿½ck
max_size()	gibt die grï¿½ï¿½tmï¿½gliche Anzahl von Zeichen zurï¿½ck
empty()	gibt zurï¿½ck, ob der String leer ist oder nicht
capacity()	gibt die Anzahl der Zeichen zurï¿½ck, die der String ohne erneute Speicherallokation enthalten kann
reserve()	reserviert Speicher fï¿½r eine gewisse Anzahl von Zeichen
[], at()	auf ein Zeichen im String zugreifen
», getline()	String aus einem Strom lesen
«	String auf einen Strom schreiben
data()	gibt den Inhalt des Strings als ein Character Array zurï¿½ck
c_str()	gibt den Inhalt des Strings als C-String zurï¿½ck
substr()	gibt einen Teilstring zurï¿½ck
find()	nach einem Zeichen oder Teilstring suchen

Tabelle 6.1: Methoden der Klase string

6.3 Konstruktion eines Strings

Die Konstruktoren dienen dazu, Speicherplatz zu alloziieren und neue Objekte zu erzeugen. Eine Klasse kann mehr als nur eine Konstruktorfunktion besitzen.
Auch die Klasse String verfï¿½gt ï¿½ber mehrere Konstruktoren – siehe hierzu Tabelle 6.2

`string s`	`erzeugt den leeren String s`
`string s(str)`	`erzeugt einen String s als Kopie des existierenden Strings str`
`string s(str, stridx)`	`erzeugt einen String s, der ab dem Index stridx mit den Zeichen des Strings str initialisiert wird`
`string s(str, stridx, strlen)`	`erzeugt einen String s, der ab dem Index stridx mit hï¿½chstens strlen Zeichen des Strings str initialisiert wird`
`string s(cstr)`	`erzeugt einen String s, der mit dem C-String cstr initialisiert wird`
`string s(chars, chars_len)`	`erzeugt einen String s, der mit chars_len Zeichen des Character Arrays chars initialisiert wird`
`string s(num, c)`	`erzeugt einen String s, der den Character c num-mal enthï¿½lt`

Tabelle 6.2: Konstruktoren der Klasse `string`

6.4 Destruktion eines Strings

Die Destruktoren, die in einer Klasse definiert sind, dienen dazu, Aufrï¿½umarbeiten fï¿½r nicht mehr benï¿½tigte Objekte zu leisten. Der hï¿½ufigste Zweck ist die Speicherfreigabe. In C++ werden Strings automatisch destruiert, sobald deren Lebensdauer erlischt, wenn also der Block, in dem sie deklariert wurden, verlassen wird. Der Lï¿½schmechanismus wird von einer innerhalb der Klasse definierten Destruktormethode realisiert.

6.5 Zugriff auf die Buchstaben eines Strings

Auf einzelne Zeichen im String kann man entweder mit dem Operator `[]` oder mit der Funktion `at()` zugreifen. Beide Methoden liefern das Zeichen an der Position des ï¿½bergebenen Index. Wie gewohnt hat das erste Zeichen im String den Index 0, das letzte Zeichen hat den Index `length()-1`.
Bemerkung: Der Operator `[]` ï¿½berprï¿½ft nicht, ob der ï¿½bergebene Index gï¿½ltig

ist. Die Funktion `at()` nimmt diese ï¿½berprï¿½fung vor.
Fï¿½r die konstante Version des Operators `[]` ist der Index „`numberOfCharacters`“ gï¿½ltig. An dieser Stelle gibt er fï¿½r Objekte vom Typ String `'\0'` zurï¿½ck. Fï¿½r die nicht konstante Version des Operators `[]` und die Memberfunktion `.at()` ist der Index „numberOfCharacters“ ungï¿½ltig.

6.6 Alphabetischer Vergleich von Strings

Fï¿½r Strings kï¿½nnen die gewï¿½hnlichen Vergleichsoperatoren benutzt werden. Die Operanden kï¿½nnen Strings oder C-Strings sein.
Werden die Vergleichsoperatoren `<`, `<=`, `>` oder `>=` benutzt, so werden die Zeichen der Strings lexikographisch verglichen. Die lexikographische Ordnung hï¿½ngt in diesem Fall vom aktuellen „Character Trait“ ab.
Mit der Funktion `compare()` kann man auch Substrings vergleichen. Die Substrings kï¿½nnen durch ihren Index und ihre Lï¿½nge definiert werden. `compare()` liefert keinen boolschen Wert, sondern einen Wert < 0 (kleiner als), 0 (gleich) oder einen Wert > 0 (grï¿½ï¿½er als). Zusï¿½tzlich gibt es die Funktion `lexicographical_compare()`, die nun doch einen `bool` zurï¿½ck gibt:

`lexicographical_compare():`

```
#include <algorithm>
bool lexicographical_compare( iterator start1, iterator end1,
       iterator start2, iterator end2 );
bool lexicographical_compare( iterator start1, iterator end1,
       iterator start2, iterator end2, BinPred p );
```

Die Methode `lexicographical_compare()` gibt `true` zurï¿½ck, falls der Elementbereich `[start1,end1]` lexikographisch kleiner dem Bereich `[start2,end2]` ist. Die Laufzeit ist linear. (mehr zu Laufzeiten in Kapitel 10.5.1)
Beispiel 6.1

```
// file: Stringeinsatz/stringLexComp.cpp
// description:

#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

using namespace std;

int main() {

    string word1 = "alpha";
    string word2 = "beta";
```

```
    string word3 = "gamma";


    cout << word1 << " steht " <<
            //Vergleich von word1, word2; Ausgabe
            (lexicographical_compare(word1.begin(), word1.end(),
            word2.begin(), word2.end()) ?
            "alphabetisch vor "
            :
            "nach oder an der gleichen Stelle wie "
            ) << word2 << endl;



    cout << word1 << " steht " <<
            //Vergleich von word1, word3; Ausgabe
            (lexicographical_compare(word1.begin(), word1.end(),
            word3.begin(), word3.end()) ?
            "alphabetisch vor "
            :
            "nach oder an der gleichen Stelle wie "
            ) << word3 << endl;



    cout << word2 << " steht " <<
            //Vergleich von word2, word3; Ausgabe
            (lexicographical_compare(word2.begin(), word2.end(),
            word3.begin(), word3.end()) ?
            "alphabetisch vor "
            :
            "nach oder an der gleichen Stelle wie "
            ) << word3 << endl;

    return 0;

}
```

Ausgabe:

```
alpha steht alphabetisch vor beta
alpha steht alphabetisch vor gamma
beta steht aphabetisch vor gamma
```

Compare:

```
#include <string>
int compare( const string& Str );
int compare( const char* Str );
int compare( size_type index, size_type length, const string& Str );
int compare( size_type index, size_type length, const string& Str,
                size_type index2, size_type length2 );
int compare( size_type index, size_type length, const char* Str,
```

```
                    size_type,length2 );
```

Anmerkung:
Im Programm werden ternï¿½re Operatoren fï¿½r die If-Abfragen verwendet. Dies ist eine Mï¿½glichkeit um kï¿½rzeren Programmcode zu schreiben. Die Struktur fï¿½r ternï¿½re If-Abfragen lautet `Variable = Bedingung ? If-Case : Else-Case;`. Erklï¿½rungen hierzu finden sich auch im ï¿½bungsskript. Im folgenden Codefragment werden zwei Mï¿½glichkeiten gezeigt, dieselbe Schleife auszudrï¿½cken:

Beispiel 6.2

```
if (a == b) {
   x = 1;
} else {
   x = 0;
}
```

oder kï¿½rzer:

```
x = (a == b) ? 1 : 0;
```

Beispiel 6.3

```
// file: Stringeinsatz/stringComp.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {

    string string1("Hello world");
    string string2("Hello yourself");
    string string3("Hello world");




    if (string1.compare(string2) > 0)
        cout << string1 << " alphabet. nach " << string2 << endl;
    else if (string1.compare(string2) < 0)
        cout << string1 << " alphabet. vor " << string2 << endl;
    else
        cout << string1 << " und " << string2 << " sind gleich \n";


    if (string1.compare(string3) > 0)
        cout << string1 << " alphabet. nach " << string3 << endl;
    else if (string1.compare(string3) < 0)
```

```
        cout << string1 << " alphabet. vor " << string3 << endl;
    else
        cout << string1 << " und " << string3 << " sind gleich \n";


    if (string2.compare(string3) > 0)
        cout << string2 << " alphabet. nach " << string3 << endl;
    else if (string2.compare(string3) < 0)
        cout << string2 << " alphabet. vor " << string3 << endl;
    else
        cout << string2 << " und " << string3 << " sind gleich \n";

    return 0;

}
```

Ausgabe:

```
Hello world alphabet. vor Hello yourself
Hello world und Hello world sind gleich
Hello yourself alphabet. nach Hello world
```

6.7 Suchen innerhalb eines Strings

Um bei der Suche in Strings das erste Vorkommen einer Zeichenfolge in einem String zu ermitteln, steht in C++ die Funktion `find()` zur Verfï¿½gung. Das Suchergebnis ist der Index des ersten Zeichens der Zeichenfolge (Achtung: Beginn bei 0!). Ist der gesuchte String nicht vorhanden, wird die Pseudoposition `npos = -1` zurï¿½ckgegeben. Diese Konstante ist in der Klasse `string` definiert, kann also mit `string::npos` angesprochen werden.

Beispiel:

```
string pippi("sie hat ein Haus, ein kunterbuntes Haus");
int first = pippi.find("Haus");
```

`first` erhï¿½lt den Wert `12`. Will man den letzten Buchstaben des letzten Auftretens eines Substrings ermitteln, kann man einfach die Methode `rfind()` (right find) benutzen.
Beispiel:

```
int last = pippi.rfind("Haus");
```

Hier wird `last` mit `39` initialisiert. (weiteres zu `find()`, siehe Kapitel 12, „STL Algorithmen“)

6.8 Modifizieren eines Strings

Strings kï¿½nnen mit Hilfe verschiedener Memberfunktionen und Operatoren modifiziert werden.

Zuweisungen: Mit dem Operator `=` kann man einem String einen neuen Wert zuweisen, und zwar einen String, einen C-String oder ein einzelnes Zeichen. Wird mehr als ein Argument benï¿½tigt, um den neuen Wert zu beschreiben, kann man die Funktion `assign()` benutzen:

```
const string S1("Eisen");
string s2;

s2 = S1;              // Zuweisung eines Strings
s2 = "Bahn";          // Zuweisung eines C-Strings
s2 = '?';             // Zuweisung eines einzelnen Zeichens
s2.assign(S1, 0, 2); // Zuweisung von "Eis"
```

Vertauschen von Stringinhalten: Fï¿½r Strings steht eine Spezialisierung der `swap()`-Funktion zur Verfï¿½gung. Diese garantiert, dass die Inhalte der Strings korrekt vertauscht werden.

Leeren von Strings: Es gibt mehrere Mï¿½glichkeiten, einen String zu leeren:

```
string s;
s = "";                 // Zuweisung des leeren Strings
s.erase();              // alle Zeichen lï¿½schen
```

Einfï¿½gen und Lï¿½schen von Zeichen: Um Zeichen am Ende des Strings anzuhï¿½ngen, benutzt man den Operator `+=`, die Funktion `append()` oder `push_back()`.

```
string as("Autobahn");
string s;
s += "Umlauf";                   // Inhalt von s: "Umlauf"
s.append(as, 4, string::npos); // Inhalt von s: "Umlaufbahn"
```

(`string::npos` ist der Index hinter dem letzten gï¿½ltigen Zeichens in einem String) Mit der Memberfunktion `insert()` kann man Zeichen in einen String einfï¿½gen. Diese Funktion benï¿½tigt den Index des Zeichens, hinter dem die neuen Zeichen eingesetzt werden sollen. Allerdings dï¿½rfen nie einzelne Zeichen, sondern nur Strings eingefï¿½gt werden!

```
string s("Autobahn");

s.insert(4, "renn");            // Inhalt von s: "Autorennbahn"
```

Die Funktionen `erase()` bzw. `replace()` lï¿½schen bzw. ersetzen Zeichen. (weiteres zu `replace()` ebenso in „STL Algorithmen“, ab Seite 111)

```
string s("Autobahn");

s.replace(0, 4, "Eisen");     // Inhalt von s: "Eisenbahn"
s.erase(3, 2);                // Inhalt von s: "Eisbahn"
```

Mit `resize()` kann man die Anzahl der Zeichen verï¿½ndern: ï¿½bergibt man als Argument eine Zahl, die kleiner ist als die aktuelle Stringlï¿½nge, dann werden Zeichen am Ende des Strings abgeschnitten. Ist das Argument eine Zahl, die grï¿½ï¿½er ist als die aktuelle Stringlï¿½nge, so werden am Ende Zeichen angefï¿½gt. In diesem Fall kann man angeben, welches Zeichen angehï¿½ngt werden soll, der Default ist `'\0'`.

Beispiel 6.4: Initialisierung eines Strings

```
// file: Stringeinsatz/stringInit.cpp
// description: work with Strings

#include <iostream>
#include <string>

using namespace std;

int main() {
    string str1 = "abc";
    string str2("String 2");
    string str3(10, 'c');

    cout << " Hier ist das Programm String_init " << endl;
    cout << " Der Initialwert war: " << str1 << endl;
    cout << " Der Initialwert war: " << str2 << endl;
    cout << " Der Initialwert war: " << str3 << endl;

    if (str1 == "abc") {
        string conc;
        conc = str1 + "defghi";
        cout << " Die Konkatenation = " << conc << endl;
    }
    return 0;

}
```

Wir erhalten bei Programmausfï¿½hrung folgenden Output:

```
 Hier ist das Programm String_init
 Der Initialwert war: abc
 Der Initialwert war: String 2
 Der Initialwert war: cccccccccc
 Die Konkatenation = abcdefghi
```

Beispiel 6.5: Suchen eines Buchstabens in einem String

```
// file: Stringeinsatz/searchNpos.cpp
// description:

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

```
using namespace std;

int main() {
    char buchstabe;
    string vokal = "aeiou";

    cout << " Hier ist Programm searchNpos " << endl;
    cout << " Bitte einen Buchstaben eingeben: ";
    cin >> buchstabe;

    cout << " Die Eingabe war: " << buchstabe << endl;

    if (vokal.find(buchstabe) != string::npos)
        cout << " Der Buchstabe ist ein Vokal " << endl;
    else
        cout << " Der Buchstabe ist kein Vokal " << endl;

    return 0;
}
```

Beispiel 6.6: Suchen eines Substrings

```
// file: Stringeinsatz/substring.cpp
// description: work with Strings

#include <iostream>
#include <string>

using namespace std;

int main() {
    string str1, str2;
    int pos;

    cout << " Hier ist das Programm Substring " << endl;
    cout << " Bitte geben Sie einen String ein >>>";
    cin >> str1;
    cout << " Bitte geben Sie einen zweiten String ein >>>";
    cin >> str2;

    cout << " Eingabe war: " << str1 << "  " << str2 << endl;

    pos = str1.find(str2);

    if (pos != string::npos)
        cout << " 2. String beginnt an Pos. " << pos << endl;
    return 0;
}
```

6.9 C++ Strings und C Strings

In der Programmiersprache C gibt es fï¿½r Strings keine spezielle Datenstruktur, geschweige denn Klasse, sondern man muss mit einem Array von 7 oder 8-Bit Buchstaben arbeiten. Damit die Programmiersprache C aber unterscheiden kann, ob ein „Array of char“ oder ein „Stringarray“ vorliegt, muss das letzte Element des „Array of char“ ein Element mit dem Wert 0 sein. Man spricht von der „terminierenden Null“ (=`'\0'`).

```
int anz_chars;
unsigned char array_name[3]; //Array of Char
array_name[0]='s';
array_name[1]='i';
array_name[2]='e'; // bleibt Array of Char
//Progammabsturz bei:
strlen(array_name);  // FEHLER !!!

unsigned char string_name[4];
string_name[0]='s';
string_name[1]='i';
string_name[2]='e';
string_name[3]=0; //wird zum String

//jetzt erlaubt, da terminierende Null vorhanden:
strlen(string_name);  // errechnet die Anzahl der Buchstaben
```

Erst wenn in einem Buchstabenarray die einzelnen Buchstaben des Strings ï¿½bertragen sind und eine terminierende Null als letztes Element des Arrays gespeichert ist, spricht man von einem C-String.

Konvertierung von Strings in C-Strings:

Bei vielen Anwendungen und Methoden ist es notwendig, die Zeichenkette des C++ Strings als C-String zu ï¿½bergeben. Als Beispiel sei die `open()`-Funktion zur ï¿½ffnung einer Datei genannt. Fï¿½r diese Aufgabe gibt es die Methode `c_str()`, die in der Stringklasse definiert ist:
`c_str()` gibt den Inhalt des Strings als C-String zurï¿½ck, d.h. am Ende wird ein `'\0'` angehï¿½ngt.

Beispiel 6.7: ï¿½bergabe eines C++-Strings als C-String

```
string filename = "eingabe.txt";
open(filename.c_str()); // Es wird intern ein C_String erzeugt
```

6.10 Das Wichtigste in Kï¿½rze

- Strings sind eine KLASSE mit Attributen und Methoden.
- Strings mï¿½ssen initialisiert werden: `string bspstring;`

- Die in einem String vorhandenen Methoden ruft man in folgender Weise auf:
  `anzahlzeichen = bspstring.size();`

7 Funktionen

Eine Funktion besteht aus einem Funktionskopf und einem Funktionsrumpf. Im Funktionskopf stehen der Funktionstyp, Funktionsname und falls gewünscht die Funktionsargumente.

7.1 Motivation

Funktionen sollten verwendet werden, um den Programmquelltext besser lesbar und strukturierter zu machen. Für das bessere Verständnis und zur Lesbarkeit sollte als Funktionsname unbedingt ein sprechender Name verwendet werden.
Das kommende Beispiel zeigt einen unstruktierten und vor allem schlecht lesbaren Quelltext:

Beispiel 7.1: Spaghetti Code

```
// file: Funktionen/spaghettiCode.cpp
#include <iostream>
using namespace std;

int main() {
    string s;
    int l,w,o;
    l = 0;
    w = 0;
    o = 0;
    getline(cin,s);
    for (int i=0 ; i < s.size(); i++) {
        if (s[i] > 96 && s[i] < 123 || s[i] > 64 && s[i] < 91) {
                l++;
        }
        else if (s[i] == 32) {
                w++;
        }
        else {
                o++;
        }
    }
    int x = 100.0 / (float) s.size();
    cout <<  s.size() << " : C " << l * x;
    cout << " ,W " << w * x << " ,O " << o * x << endl;
}
```

Durch eine manuelle Umstrukturierung des Quelltextes (engl. *Refactoring*) unter der Verwendung von Funktionen, wird eine viel bessere Lesbarkeit ermöglicht, welche deutlich zum besseren Verständnis des Quelltextes beiträgt:

Beispiel 7.2: Spaghetti Code Umstrukturiert

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

using namespace std;

string readLineFromTerminal( void );
bool isLetter( char );
bool isWhitespace( char );
float ruleOfThree(int, int );
void printCharacterDistributionfor( string, int, int );
void debug(char);

int main() {
    string line;
    int letterCount, whitespaceCount, otherCharacterCount;

    whitespaceCount = 0;
    letterCount = 0;
    otherCharacterCount =0;

    line = readLineFromTerminal();

    for (int position = 0 ; position < line.size(); position++ ) {
        char currentCharacter = line.at( position );
        debug(currentCharacter);
        if ( isLetter( currentCharacter ) ) {
           letterCount++;
        }
        else if ( isWhitespace( currentCharacter ) ) {
           whitespaceCount++;
           cout << "WS"<< whitespaceCount << endl;
        }
        else {
           otherCharacterCount++;
        }
    }
    printCharacterDistributionfor( "Letters",letterCount,line.size() );
    printCharacterDistributionfor( "Whitespace",whitespaceCount, line.size() );
    printCharacterDistributionfor( "Other Chars",otherCharacterCount, line.size() );
}

string readLineFromTerminal() {
    string s;
    getline(cin,s);
```

```
    return s;
}

bool isLetter(char c) {
    return c > 96 && c < 123 || c > 64 && c < 91 ;
}

bool isWhitespace(char c) {
    return  c == 32 ;
}

void printCharacterDistributionfor( string name, int count, int max) {
    cout <<  name << " " <<  ruleOfThree(max, count);
    cout << "\% of input." << endl;
}

float ruleOfThree( int all, int part) {
    float x = 100.0 /  (float) all;
    return x * part;
}

void debug(char currentCharacter) {
    cout << "Char:" << currentCharacter << " -> ";
    cout << (int) currentCharacter << endl;
}
```

7.2 Funktionstyp

Eine Funktion kann, falls gewünscht, ein Ergebnis zurückgeben, wobei der Typ des Funktionsergebnisses dann den Typ der Funktion definiert. Ergebnistypen können alle Arten von Basistypen, selbstdefinierten Typen und Objekttypen sein. Funktionen, die kein Ergebnis zurückgeben, sind vom Typ `void`.

Beispiel 7.3: Datentypen in Rückgabewerten

```
int test(), char test(), long test(), float test(),
string test(), lexicon test(), void test()
```

Das Ergebnis einer Funktion wird mit dem `return`-Statement zurückgegeben. Der zurückgegebene Wert wird entsprechend dem Funktionstyp konvertiert.

Beispiel 7.4: Rückgabewerte von Variablen und Ausdrücken

```
return;       /* Kein Ergebnis */
return a;     /* Übergabe des Werts einer Variable */
return (a*b); /* Übergabe des Werts eines Ausdrucks */

int pow(int x) {
    return (x * x)
}
```

7.3 Funktionsname

Beim Benennen von Funktionen gelten die selben Regeln wie beim Benennen von Variablen (siehe *Motivation* oben).

Formatierungsvorschlag:

Der Übersichtlichkeit halber wollen wir Verben für Funktionsnamen verwenden und diese im `lowerCamelCase` schreiben. Das heißt: es steht ein Kleinbuchstabe am Anfang des Namens und alle folgenden Wörter beginnen mit einem Großbuchstaben. z. B.:

```
void setName() {
    //...
}
```

7.4 Funktionsargumente

Die Funktionsargumente stehen eingeschlossen in runden Klammern hinter dem Funktionsnamen. Jedes Argument wird mit seinem Typ und dem Argumentnamen aufgeführt. Eine Funktion kann auch ohne Argumente definiert werden.

Beispiel 7.5: Funktionsargumente

```
int test (int a, char y, char x) { ... }
```

7.5 Defaultwerte für Funktionsargumente

Funktionen können mit Argumenten definiert werden, für die Defaultwerte bei der Definition festgelegt werden. Entfällt beim Aufruf das Argument, dann wird für das fehlende Argument der Defaultwert eingesetzt.
Der C++-Standard legt folgendes fest: Gibt es zu einer Funktion eine Prototypendeklaration, dann muss der Defaultwert eines Parameters in der Prototypendeklaration festgelegt werden.

Beispiel 7.6: Deklaration und Definition 1

```
// file: Funktionen/Deklar_Defin.cpp
#include <iostream>
using namespace std;

// Deklaration add mit  Prototyp:
int add(int arg1, int arg2 = 0);

// Deklaration add ohne  Prototyp:
int add2(int arg1, int arg2) {
  return arg1+arg2;
}

int main() {
  int erg;
  erg = add(1,3); // Aufruf mit 2 Argumenten
  cout << "add(1,3)=" << erg << endl;
  erg = add(1);   // Aufruf mit 1 Argument
  cout << "add(1)=" << erg << endl;
}
// Definition:
int add(int arg1, int arg2) {
  return arg1 + arg2;
}
```

7.6 Funktionsrumpf

Der Funktionsrumpf ist in geschweiften Klammern eingeschlossen und entspricht somit einem Block. Am Anfang des Blocks stehen im Allgemeinen die Deklarationen lokaler Variablen. Die Argumente der Funktion stehen als lokale Variablen zu Verfügung.

7.7 Die Argumentübergaben

Argumente und Rückgabewerte von Funktionen werden in C++ immer mit ihrem Wert (=value) übergeben (=call-by-value). Beim Aufruf wird eine lokale Kopie der Variablen, die als Argument übergeben wird, erzeugt. Die Funktion kennt nur den Wert der übergebenen Variablen und hat keine Möglichkeit, den Variablenwert außerhalb der Funktion zu ändern. Die lokale Kopie eines Arguments wird mit der Konstruktorfunktion des Argumenttyps realisiert.

Soll der Wert einer Variablen innerhalb der Funktion geändert werden und die Wertänderung der Variablen außerhalb der Funktion Bestand haben, dann muss die Variable der

Funktion als Referenz definiert werden.

Die Übergabe einer Variablen als Referenz hat auch einen weiteren Vorteil: Es muss keine lokale Kopie der Variablen angelegt werden, da ja auf der Adresse der Variablen selbst gearbeitet wird. Die Argumentübergabe wird somit effizenter.

Darf sich innerhalb einer Funktion der Wert des Arguments nicht ändern und der Compiler eine wertändernde Operation des Arguments innerhalb der Funktion erkennen, so muss beim Argument vor dem Datentyp das Attribut `const` geschrieben werden.

7.7.1 Übergabe des Werts eines Arguments

Für die Argumente und Rückgabewerte wird eine temporäre Variable auf dem Stack erzeugt, die mit dem Übergabewert initialisiert wird. Nach dem Verlassen der Funktion wird die temporäre Variable wieder gelöscht. Dieses Verhalten ist zu berücksichtigen, wenn Argumente komplexer Datentypen an die Funktion übergeben werden, da jedes Mal der gesamte Inhalt der Variablen kopiert werden muss, was einen beträchtlichen Performace-Verlust zur Folge haben kann.

7.7.2 Übergabe einer Referenz auf ein Argument

Soll der Wert einer Variablen innerhalb einer Funktion geändert werden, dann muss sie als Referenzfunktionsargument definiert werden. Beim Aufruf der Funktion wird der Name des Funktionsarguments zu einem zweiten Namen für die aufgerufene Variable und jede Änderung auf der Funktionsvariablen mit dem zweiten Namen ändert sofort den Wert der Variablen außerhalb der Funktion.

7.7.3 Beispiele Value vs. Referenz

Beispiel 7.7: Call-by-value vs. call-by-reference

```
// file: Funktionen/valRef.cpp
// description:

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

// Definition der Prototypen
int addString(string);
int addString2(string &wort);

int addString(string wort) {
    wort.append("heit");
    cout << " In der Funktion wort = " << wort << endl;
    return 1;
```

```
}

int addString2(string &wort) {
    wort.append("heit");
    cout << " In der Funktion wort = " << wort << endl;
    return 1;
}

string addString3(string &wort) {
    wort.append("heit");
    cout << " In der Funktion wort = " << wort << endl;
    return wort;
}

int main() {
    string wort;
    string wort1,wort2,wort3;
    string neuesWort;
    cout << " Hello, Programm valRef.cpp " << endl;
    cout << " Bitte geben Sie ein wort ein  >>";
    cin >> wort;
    wort1=wort;
    wort2=wort;
    wort3=wort;
    addString(wort1);
    cout << " nach Aufruf von  addString=" << wort1 << endl;
    addString2(wort2);
    cout << " nach Aufruf von addString2=" << wort2 << endl;
    neuesWort = addString3(wort3);
    cout << " nach Aufruf von addString3=" << neuesWort;
    cout << ", Argument=" << wort3 << endl;
}
```

7.7.4 Seiteneffekte

Seiteneffekte (oder engl. *side effects*) treten dann auf, wenn eine Variable innerhalb einer Funktion geändert wird, und diese Wertänderung auch außerhalb der Funktion Bestand hat (siehe *call-by-reference*). Ein weiteres Beispiel für Seiteneffekte wäre dieses Beispiel:

Beispiel 7.8: Seiteneffekte

```
// file: Funktionen/sideEffects.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

void addAndPrintStringByValue(string wort) {
```

```
        wort.append("heit");
        cout << " In der Funktion <addAndPrintString> wort = "
        << wort << endl;
    }

    void addAndPrintStringByReference(string &wort) {
        wort.append("heit");
        cout << " In der Funktion <addAndPrintString> wort = "
        << wort << endl;
    }

    int main() {
        string wort;

        cout << " Programm sideEffects.cpp " << endl;
        cout << " Bitte geben Sie ein wort ein  >>";
        cin >> wort;


        cout << "Funktion <addAndPrintStringByValue> wird 3x aufgerufen"
        << endl;
        addAndPrintStringByValue(wort);
        addAndPrintStringByValue(wort);
        addAndPrintStringByValue(wort);

        cout << "Funktion <addAndPrintStringByRefernce> wird 3x aufgerufen"
        << endl;
        addAndPrintStringByReference(wort);
        addAndPrintStringByReference(wort);
        addAndPrintStringByReference(wort);
    }
```

Die Ausgabe für dieses Programm mit der Eingabe Träg ist dann:

```
Programm sideEffects.cpp
Bitte geben Sie ein wort ein >> Träg
Funktion <addAndPrintStringByValue> wird nun drei mal aufgerufen
Trägheit
Trägheit
Trägheit
Funktion <addAndPrintStringByRefernce> wird nun drei mal aufgerufen
Trägheit
Trägheitheit
Trägheitheitheit
```

7.8 Überladen von Funktionen

C++ bietet die Möglichkeit, eine Funktion mehrmals unter gleichem Namen zu definieren. Dabei müssen allerdings die Anzahl und/oder die Datentypen der Parameter unterschiedlich sein. Dies wird Überladen von Funktionen genannt. Der Compiler entscheidet anhand der Argumente beim Aufruf welche der definierten Funktionen verwendet werden soll.

Beispiel 7.9: Überladen von Funktionen

```
// file: Funktionen/defaultValues.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

// Deklaration mit Prototyp
bool bestArt(string wort, string sprache = "de");
bool bestArt(string wort, int maxChars, string sprache = "de");

bool bestArt(string wort, string sprache) {
    if (sprache == "de") {
        if (wort == "der" || wort == "die" || wort == "das") {
            cout << " Deutscher Artikel = " << wort << endl;
            return true;
        }
        return false;
    }
    if (sprache == "en") {
        if (wort == "the") {
            cout << " Englischer Artikel = " << wort << endl;
            return true;
        }
        return false;
    }
}

bool bestArt(string wort, int maxChars, string sprache) {
    if (wort.length() > maxChars) {
        cout << " Artikel " << wort << " ist zu lang" << endl;
        return false;
    } else
        return (bestArt(wort, sprache) || bestArt(wort, "en"));
}

int main() {
    string wort;
    cout << " Bitte geben Sie einen bestimmten Artikel ein  >>";
    cin >> wort;
```

```
    if (bestArt(wort, 4))
        cout << " bestArt " << wort << " gefunden" << endl;
    else {
        cout << " bestArt " << wort << " nicht gefunden" << endl;
        return 1;
    }
}
```

7.9 Mehrdeutigkeiten durch Überladen der Funktionen

In einigen Fällen können durch das Überladen von Funktionen Mehrdeutigkeiten bei Funktionsaufrufen entstehen. Gründe hierfür können sein:

- Typumwandlung
- Referenzparameter
- Default-Argumente

Folgende Beispiele sollen das verdeutlichen. Hier ist noch anzumerken, dass offenbar nicht jeder Compiler beim Übersetzen der Programme die Mehrdeutigkeiten erkennt. Der korrekte Ablauf eines so übersetzten Programms ist nicht mehr vorhersagbar.
Wird eine Funktion mit einem Argument aufgerufen, das zwar nicht vom selben aber von einem verwandten Datentyp zum deklarierten Übergabetyp ist, so wird automatisch eine Typumwandlung durchgeführt. Dadurch kann ein Funktionsaufruf mehrdeutig werden, wie in folgendem.

Beispiel 7.10: Mehrdeutigkeiten bei Funktionsaufrufen

```
#include <iostream>
using namespace std;
// Definition der Prototypen
float f(float i);
double f(double i);

float f(float i) // Funktion 1
{
    return i / 2.0;
}

double f(double i) // Funktion 2
{
    return i / 3.0;
}
int main() {
    ...
    float x = 10.09;
    double y = 10.00;
```

```
    cout << f(x) << endl; // eindeutig - f(float) aufrufen
    cout << f(y) << endl; // eindeutig - f(double) aufrufen
    cout << f(10) << endl; // mehrdeutig - 10 nach double oder float
                           // konvertieren?
    ...
}
```

Bei unbedachter Überladung von Funktionen können Mehrdeutigkeiten entstehen, die aber vom Compiler aufgedeckt werden.

8 Internationalisierung unter C++

8.1 Einführung in Kodierungen

Das UNICODE Konsortium, ein Projekt der ISO (International Organization for Standardisation), hat eine Erweiterung des ISO - Standards entwickelt, der die Kodierung aller Zeichen umfasst, die in verschiedenen Nationalitäten vorkommen. Der Standard heißt ISO 10646 und definiert das Universal Character Set (UCS) aller vorkommenden Zeichensätzen. Das UNICODE Konsortium hat einen UCS Zeichensatz entwickelt, der jedem Zeichen aller existierenden Sprachen eine Nummer zuweist. Die ersten 127 Nummern des UCS Zeichensatz sind identisch zum ASCII Standard. Der ISO 10646 war ursprünglich ein 31-bit Characterset. Die Untermenge der Zeichen, die nur aus den ersten 16 Bit bestehen, heißt „Basic Multilingual Plane (BMP)“, oder Plane 0 des UCS. Zeichen, die außerhalb der nullten Ebene liegen, sind hauptsächlich für spezialisierte Anwendungen reserviert. Das Ziel des UNICODE Consortiums ist es, dass der Standardzeichensatz maximal 21-Bit lange Zeichen benutzt.
Mit der Wahl der Kodierung legt ein Benutzer fest, wie ein Text in einer Datei gespeichert wird. Wählt der Benutzer die UCS-Codierungstabelle, dann kann er zwischen einer festen und einer variablen Bitbreite der Zeichennummer wählen:
Kodierungen mit festen Bitbreiten sind die UCS-2 (16 Bit) und UCS-4 (32-Bit) Kodierungen. Kodierungen mit variablen Bitbreiten sind die Unicode Transformation Formate.

8.2 Unicode Transformation Format: UTF-8

Hinter dem Unicode Transformation Format steckt die Idee die UCS-Codes der UNICODE Zeichen möglichst speichersparend abzuspeichern. Des weiteren sollten UTF-Dateien und Dateien, die nur aus ASCII Zeichen bestehen, identisch sein. Hinter dem UTF steckt ein Verfahren, das jeden beliebigen UCS Code eines UNICODE Zeichens auf ein bis sechs nachfolgende Bytes verteilen kann. In Abhängigkeit von der Größe des UCS Codes errechnet das Transformationsverfahren des UTFs die Anzahl der Bytes, die notwendig sind, um diesen Code wieder zu rekonstruieren. Die nacheinanderfolgenden Bytes nennt man eine Multibyte-folge, die UTF Codierung einen Multibyte Zeichensatz.

Mit dieser Technik können im UTF-8 Format alle 2 (hoch 21) Zeichen des UNICODE Zeichensatzes dargestellt werden.

Die Sortierreihenfolge entspricht der von Bigendian UCS-4.

Die Bytes von 0xFE bis 0xFF werden nie für ein in UTF-8 kodiertes Zeichen benutzt.

Die folgenden Bytefolgen werden benutzt um ein Zeichen darzustellen, wobei die zu benutzende Sequenz von der Unicode-Nummer des Zeichens abhängt.

U-00000000 - U-0000007F:	0xxxxxxx
U-00000080 - U-000007FF:	110xxxxx 10xxxxxx
U-00000800 - U-0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 - U-001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 - U-03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 - U-7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Tabelle 8.1: Bytefolgen UTF-8

Beispiele:

Das Unicode-Zeichen U+00A9 = 1010 1001 (Â©) in UTF-8 Kodierung:
`11000010 10101001 = 0xC2 0xA9`

und U+2260 = 0010 0010 0110 0000 (ungleich) in UTF-8:
`11100010 10001001 10100000 = 0xE2 0x89 0xA0`

Beispiel mit verschiedenen Kodierungen:

In den folgenden Tabellen sieht man den Text „München“ in verschiedenen Kodierungen als Stream of Byte.

ISO_8859-1							
hexadezimal	4d	fc	6e	63	68	65	6e
octalcode	115	374	156	143	150	145	156
ASCII-Zeichen	M	ü	n	c	h	e	n

Tabelle 8.2: Bytefolge mit fester Länge ISO_8859-1: ISO-LATIN1 (8-Bit)

ucs2														
hexadezimal	4d	0	fc	0	6e	0	63	0	68	0	65	0	6e	0
octalcode	115	0	374	0	156	0	143	0	150	0	145	0	156	0
ASCII-Zeichen	M	nul	|	nul	n	nul	c	nul	h	nul	e	nul	n	nul

Tabelle 8.3: Bytefolge mit fester Länge UCS-2: 2-byte Universal Character Set

UTF-8								
hexadezimal	4d	c3	bc	6e	63	68	65	6e
octalcode	115	303	274	156	143	150	145	156
ASCII-Zeichen	M	C	<	n	c	h	e	n

Tabelle 8.4: Bytefolge mit variabler Länge UTF-8: 8-bit Unicode Transformation Format

UTF-16																
hexadezimal	ff	fe	4d	0	fc	0	6e	0	63	0	68	0	65	0	6e	0
octalcode	377	376	115	0	374	0	156	0	143	0	150	0	145	0	156	0
ASCII-Zeichen	del		M	nul	|	nul	n	nul	c	nul	h	nul	e	nul	n	nul

Tabelle 8.5: Bytefolge mit variabler Länge UTF-16: 16-bit Unicode Transformation Format

8.3 Datentypen für Unicode-Zeichen

Die Zeichen des UCS Charactercodes benötigen mindestens 21 Bit lange Zahlen. Deshalb wurde für C++ ein neuer Datentyp für Character eingeführt:

Datentyp für ein Zeichen des UCS Charactercodes:

`wchar_t` ... Dieser Typ umfasst 32 Bit (UNIX) und 16 Bit (Windows)

Datentyp für eine Zeichenkette mit Zeichen des UCS Charactercodes:

`wstring` ... Dieser Typ umfasst 32 Bit Zeichen (UNIX) und 16 Bit (Windows)

Aus einem String wird ein UCS String, durch Voranstellen des Buchstabens L (long) vor die sich öffnenden Hochkommata eines Strings:

```
std::wstring name = L"Müller Hans";
std::wstring farbe(L"grün");
```

Zur Speicherung der 21-Bit breiten UCS-Codes der Zeichen reichen die 8 Bit des `unsigned char` Datentyp nicht aus. Deshalb wurden neue Datentypen zur Speicherung von Zeichen: `wchar_t` und Zeichenketten: `wstring` eingeführt. Außerdem wurden neue, passende Wide-Char und Wide-Strings Ein/Ausgabe Routinen definiert:

`wcin` Lesen von wstrings, entsprechend der globalen Kodierung

`wcout` Schreiben von wstring, entsprechend der globalen Kodierung

`getline (wcin,line)` Lesen einer wstring Zeile, entsprechend der Kodierung des Streams wcin.

8.4 Locales und Imbuing

Damit das C++ Programm die UCS Daten, die in eine Datei kopiert/gelesen, bzw. auf dem Terminal ausgeben/gelesen werden sollen, im richtigen Format bearbeitet, muss das C++ -Programm den `stream` mit der richtigen Kodierungsumgebung bearbeiten. Der Benutzer deklariert ein locale Objekt passend zur Kodierung und kann entweder einzelnen Routinen bzw. `streams` ihr eigenes Kodierungsobjekt mitgeben, oder für alle Routinen bzw. `Standard-streams` das Kodierungsobjekt auf `global` setzen.

```
Schritt1:
// Deklaration einer Kodierungsumgebung passend zur Kodierung:
// z.B. de_DE.utf-8
// Speicherung im Kodierungobjekt mylocale
std::locale mylocale("de_DE.utf-8");
```

Bemerkung:
Im Argument des locale-Konstruktors für das Kodierungsobjekt steht das gewünschte Locale. Wird das vom System voreingestellte Locale verwendet, dann kann der Konstruktor auch mit leerem String als Argument aufgerufen werden:

```
Schritt1 (Variante mit Defaultkodierung) :
// Deklaration einer Kodierungsumgebung passend zur Kodierung:
// z.B. de_DE.utf-8
setlocale(LC_ALL,"");
```

Damit alle im Hintergrund verwendeten Templatefunktionen das gewünschte Locale verwenden, muss dieses Locale global gesetzt werden.

Beispiel 8.1

```
// file: Internationalisation/locale.cpp
// description:

#include <iostream>
#include <locale>

using namespace std;

int main() {
  unsigned char line[256];
  int result;
  wstring name = L"Müller Hans";
  wstring farbe(L"grün");
  wstring fullname, str;

  setlocale(LC_ALL, "");

  wcout << L"Der Name lautet " << name << endl;
  wcout << L"Die Farbe ist " << farbe << endl;
```

```
    wcout << "Vor- und Nachnamen eingeben: ";
    getline(wcin, fullname);
    wcout << " Ihr Name ist " << fullname << endl;


    // oder als Schleife,

    while (wcout << "Eingabe :" << flush, getline(wcin, str)) {
      wcout << str << endl;
    }
    wcout << endl;

    return 0;
}
```

Achtung:

Beim momentanen Stand des g++ (Version 4.6) müssen Sie beachten, dass Sie im selben Programm `cout` und `cin` niemals gemischt mit `wcout` und `wcin` verwenden! Der erste Zugriff auf den Stream definiert sein Verhalten. Nachfolgende Zugriffe werden entweder falsch bearbeitet oder übergangen.

8.4.1 Localeabhängiges Arbeiten bei UCS Codierung

Die Programmiersprache C++ unterstützt locale-abhängige Tests und Konvertierungen von Buchstaben. Die klassischen Routinen der Buchstabentests und Konvertierungen in der C-Library werden in C++ um gleichlautende Routinen erweitert, denen im zweiten Argument das aktuelle locale mitgegeben wird.

Die Routinen erwarten im ersten Argument den wide-Character und im zweiten ein Localeobjekt, in dem die spezifischen Lokaleeinstellungen gespeichert sind

Beispiel 8.2

```
if ( isupper( str[0], mylocale ) )
```

Darüber hinaus bietet C++ Buchstabenroutinen für wide-Character. Sie funktionieren wie die klassischen Buchstabenroutinen der C-Library, berücksichtigen aber das mit `global` im Hintergrund eingestellte Locale.

Beispiel 8.3: Buchstabenroutinen für wchar_t

```
if( iswupper( str[0] ) )
```

Beispiel 8.4: Buchstabentests unter Berücksichtigung des `locale()`

```
// file: Internationalisation/WcharTests.cpp
// description: Test einiger Charactertypen in
// Abhängigkeit des Locales

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

int main() {
    wstring str;
    setlocale(LC_ALL, "");
    while (wcout << ">>" << flush, getline(wcin, str)) {
        // Test for uppercase:
        if (iswupper(str[0])) { //oder: if( isupper( str[0], locale() ) )
            wcout << str[0] << " is upper case" << endl;
            wcout << "to lowercase:  ";
            wcout << (wchar_t)towlower(str[0]) << endl;
            //oder: (wchar_t)towlower( str[0], locale() )
        }
        if (iswlower(str[0])) { //oder:  if( islower( str[0], locale() ) )
            wcout << str[0] << " is lower case" << endl;
            wcout << "to uppercase :  ";
            wcout << (wchar_t)towupper(str[0]) << endl;
            // oder : (wchar_t)toupper( str[0], locale() )
        }
        if (iswalnum(str[0])) { //oder:  if( isalnum( str[0], locale() ) )
            wcout << str[0] << " is alnum Char" << endl;
        }
        if (iswalpha(str[0])) { //oder: if( isalpha( str[0], locale() )
            wcout << str[0] << " is alpha Char " << endl;
        }
        if (iswdigit(str[0])) { //oder: if( isdigit( str[0], locale() ) )
            wcout << str[0] << " is digit Char " << endl;
        }
        if (iswpunct(str[0])) { //oder: if( ispunct( str[0], locale() ) )
            wcout << str[0] << " is punct Char " << endl;
        }
        if (iswspace(str[0])) { //oder: if( isspace( str[0], locale() ) )
            wcout << str[0] << " is space Char " << endl;
        }
    }
}
```

8.5 UCS Zeichen und Datei-Streams

Damit die Daten von Dateien gelesen/geschrieben werden können, die im UCS Charactercode gespeichert sein sollen, stellt der C++ Standard wide-streams zur Verfügung:

`wifstream, wofstream, wfstream`

Zur Konvertierung der UCS-Daten in die gewünschte Codierung der Datei, muss der wide-stream auf die entsprechende Codierung eingestellt werden:
Es wird eine Kodierungsumgebung, passend zur Kodierung deklariert und der stream wird mit der gewünschten Kodierung „eingefärbt“ (auf engl. „imbued“):

8.5.1 Konvertierung von utf8 nach ISO-Latin

Beispiel 8.5

```
// file: Internationalisation/convToISO.cpp
// description:

#include <iostream>
#include <fstream>
#include <locale>

using namespace std;

int main() {

  setlocale(LC_ALL, "");

  string filenameIso = "iso.txt";
  wstring isoLine;
  int i;
  wifstream fIso;

  // Construct locale object
  locale isolocale("de_DE.ISO-8859-1");
  locale utf8locale("de_DE.UTF-8");

  wcout << L"Open ISOlatin File" << endl;

  fIso.open(filenameIso.c_str());
  fIso.imbue(isolocale);

  if (!fIso) {
    wcerr << "Error opening " << endl;
    exit(1);
  }
```

```
    i = 0;
    while (getline(fIso, isoLine))
      // iso: getline(<ifstream>,<string>)
    {
      // wcout mit <string>
      wcout << L"[" << i << L"]  " << isoLine << endl;
      i++;
    }
    fIso.close();
    return 0;

  }
```

8.5.2 Ausgabe einer UTF8- kodierten Datei

Beispiel 8.6

```
// file: Internationalisation/printUTF8File.cpp
// description:

#include <iostream>
#include <fstream>

using namespace std;

void printUTF8File(string file) {
  setlocale(LC_ALL, "");

  locale utf8locale("de_DE.UTF-8");

  wifstream utf_8_text(file.c_str());
  utf_8_text.imbue(utf8locale);

  wstring utf_8_string;
  getline(utf_8_text, utf_8_string);

  wcout << L"Encoding:" << L"UTF-8" << endl;
  wcout << L"Anzahl der Zeichen:" << utf_8_string.size() << endl;
  wcout << utf_8_string << endl;
  for (int i = 0; i < utf_8_string.size() ; i++) {

    wcout << (int) utf_8_string.at(i) << "\t";
    wcout << utf_8_string.at(i) << endl;
  }
}

int main() {
  printUTF8File("data/utf-8_umlaut_short.txt");
}
```

8.6 Das Wichtigste in Kürze

- Kanäle für Ein- und Ausgabe werden in C++ mit dem Befehl `imbue` mit einer Kodierung (`locale`) versehen.

Aufgaben und Übungen zum Kapitel – Arbeiten mit locale

Wir arbeiten mit UTF-8 und UMLAUTEN!!! Achten Sie darauf, dass Ihr Terminal mit UTF-8 arbeitet.

Übung 8.1

Schreiben Sie ein Programm, das ein Wort von der Tastatur einliest, die Anzahl der Buchstaben des Wortes ausgibt und jeden zweiten Buchstaben des Wortes ausgibt.
Achtung: Verwenden Sie die Objektklasse wstring und wcin, wcout usw.

Übung 8.2

Schreiben Sie ein Programm, das eine Zeile von der Tastatur einliest, die Anzahl der Buchstaben des Wortes ausgibt und die Anfangsbuchstaben der Wörter der Zeile ausgibt.

Übung 8.3

Schreiben Sie ein Programm, das eine Zeile von der Tastatur einliest und ausgibt, jedoch sollen alle großen Buchstaben in kleine gewandelt werden, alle kleinen in große.

```
z.B.
EINGABE> Übermorgen
AUSGABE> üBERMORGEN
```

Hilfe zum Arbeiten unter Windows mit UTF-8 in der Konsole

Tip for using the Windows Command Prompt with UTF-8
Subject: Tip for using the Windows Command Prompt with UTF-8 List-id: ICU C/C++ Support List
Hi All,
Since several people have wondered how you can use the Windows command prompt with UTF-8 instead of a codepage, like ibm-437, I thought I'd give the group some instructions for doing this.

1. Open a command prompt window
2. Change the properties of the window to use something besides the default raster font. The Lucida Console True Type font seems to work well.

3. Run „chcp 65001“ from the command prompt

You should now be able to type a UTF-8 file to the screen now. I've tried this on Windows 2000 and Windows XP.

I should warn you that some DOS based commands don't work well with UTF-8, like the more command, but some things do work, like the type command.

George Rhoten

IBM Globalization Center of Competency/ICU San JosÃ©, CA, USA

Sollte utf-8 unter einer Windowseingabeaufforderung nicht funktionieren, versucht anstatt von der Tastatur, über piping und stdin von einer Datei zu lesen! - Max

9 Programmieren von Klassen

9.1 Einführung

Zentrales Konzept in der Objektorientierten Programmierung ist die Kapselung von Daten und Elementfunktionen, die sogenannte „Encapsulation“. In Objektorientierten Programmiersprachen können Daten und Elementfunktionen an die Objekte gebunden werden, für die sie zuständig sind. Der Programmierer legt genau fest, wie die Daten modifiziert werden können.

- Will man erreichen, dass auf die Daten oder Elementfunktionen eines Objekts zugegriffen werden kann, so müssen diese als `public` definiert werden.
- Will man erreichen, dass die Daten oder Elementfunktionen eines Objekts nach außen verborgen bleiben, so müssen diese als `private` definiert werden.

9.2 Deklaration von Klassen

Im Deklarationsteil von Klassen wird der Name der Klasse, die Objekte und die Prototypen der Elementfunktionen des Objekts festgelegt. Außerdem wird definiert, wie die Daten und Elementfunktionen nach außen freigegeben werden.

Beispiel 9.1: Klassendeklaration

```
// Deklarationsteil der myclass.hpp

class MyClass {
public:
    public functions and data MyClass

private:
    private functions and data MyClass
};
```

Diejenigen Elementfunktionen und Daten, die innerhalb einer derartigen Klassendeklaration deklariert werden, heißen Mitglieder (members) dieser Klasse: „Memberfunctions“ und „Member“.

Der Spezifizierung `private` kann auch weggelassen werden, da die Voreinstellung des Zugriffsrecht `private` ist. `private`-Members sind ausschließlich von den anderen Mitgliedern

dieser Klasse erreichbar.

Um Funktionen und Daten einer Klasse auch von anderen Programmteilen, die Objekte diese Klasse verwenden, erreichbar zu machen, müssen sie als `public` definiert werden.

Innerhalb des Deklarationsteils sollen nur die Deklaration der Member und der Memberfunktionen stehen.
Der Implementationsteil der Memberfunktionen, also der Programmcode, wird unterhalb des Deklarationsteils in der Deklarationsdatei hinzugefügt.

Beispiel 9.2: Implementationsteil

```
// Implementationsteil der myclass.hpp

ergebnistyp ClassName::function - name(parameter - list) {
    ... // body of function
}
```

Die Deklarationsdatei mit Implementationsteil soll in der Regel die Extension `hpp` tragen.

Hinweis: Es wäre auch möglich den Deklarationsteil und Implementationsteil in zwei Dateien auszulagern. Hierbei würde die Deklarationsdatei die Dateiendung `.h` bekommen und nur den Deklarationsteil der Klasse umfassen. Die Implementation der dort enthaltenen Memberfunktionen wird dann außerhalb der Deklarationsdatei in einer Implementationsdatei vorgenommen. Diese würde dann die Dateiendung `.cpp` bekommen. In der Implementationsdatei müsste dann noch die Deklarationsdatei am Anfang mittels `#include` Statement eingebunden werden. Der Übersicht halber wird auf diese Trennung hier allerdings verzichtet; deswegen enthält die Deklarationsdatei in unserem Beispiel schon den Implementationsteil.

Damit im Implementationsteil eindeutig definiert ist, zu welcher Klasse eine Memberfunktion gehört, muss dem Namen der Memberfunktion der Klassenname mit dem Resolutionsoperator `::` vorgestellt werden.

9.3 Deklaration von Klassenobjekten

Mit der Definition von Klassen wird lediglich die Datenstruktur im Speicher festgelegt, es wird aber noch kein Speicher für ein Objekt dieser Klasse zur Verfügung gestellt. Dies wird mit der Objektdeklaration erledigt, dem Objektnamen wird der Klassenname vorgestellt:

```
ClassName object_name;
```

Bei der Deklaration wird Speicher für dieses neue Objekt alloziiert und die `public` definierten Daten und Funktionen sind über den Objektnamen zugänglich.

9.4 Beispiel mit Klassen

Beispiel 9.3: Eine Klasse substantiv

```
// file: Klassenprogrammierung/substantiv.hpp
// description:

#include <iostream>

#ifndef SUBSTANTIV_HPP
#define SUBSTANTIV_HPP

class Substantiv {
public:
    void intoGrundform(std::string &vollform);
    void druckeLexikoneintrag();
private:
    bool genitiv(std::string &wort);
    std::string vollform;
    std::string grundform;
    std::string morphem;
};

void Substantiv::intoGrundform(std::string &wort) {
    vollform = wort;
    if (genitiv(wort))
        std::cout << " Genitiv entdeckt " << std::endl;
    else
        std::cout << " Endung nicht erkannt " << std::endl;
}

bool Substantiv::genitiv(std::string &wort) {
    int lastChar = wort.length() - 1;
    if (wort[lastChar - 1] == 'e' && wort[lastChar] == 's') {
        morphem = "es";
        grundform.assign(wort, 0, lastChar - 1);
        return true;
    }
    return false;
}

void Substantiv::druckeLexikoneintrag() {
    if (grundform != "")
        std::cout << " Grundform = " << grundform << std::endl;
    if (morphem != "")
        std::cout << " Morphem = " << morphem << std::endl;
    if (vollform != "")
        std::cout << " Vollform = " << vollform << std::endl;
}
#endif
```

```
// file: Klassenprogrammierung/mainSubstantiv.cpp
// description: Anwender der Substantiv Klasse

#include "substantiv.hpp"
#include <iostream>

using namespace std;

int main() {
    string wort;
    Substantiv subst;

    cout << " Hello, Programm mainSubstantiv.cpp " << endl;
    cout << " Genitiv eines Substantivs eingeben: ";
    cin >> wort;
    subst.intoGrundform(wort);
    subst.druckeLexikoneintrag();
    return 0;
}
```

Übersetzen des Beispiels:

```
g++ -o mainSubstantiv  mainSubstantiv.cpp
```

Aufgaben und Übungen zum Kapitel – Programmieren von Klassen

Übung 9.1

Schreiben Sie die Implementation der Klasse substantiv aus dem Skript ab und erweitern Sie die Klasse substantiv.

Schreiben Sie eine Deklarationsdatei mit Implementationsteil : `substantiv.hpp`

Schreiben Sie ein Hauptprogramm : `mainSubstantiv.cpp`

Übung 9.2

Wie lautet der Kompilierungsbefehl?

Übung 9.3

Es sollen folgende private Methoden zur Klasse substantiv hinzugefügt werden:

```
plural(string &wort)
```

Es soll die Pluralendung `'en'` erkannt werden.

Übung 9.4

`tolower()` Es soll der private Eintrag der Vollform in ein klein geschriebenes Wort konvertiert werden.

Übung 9.5

Die Methode `into_grundform` soll jeden Neueintrag automatisch mit der `private`-Methode `tolower()` in ein kleingeschriebenes Wort konvertieren und so unter `vollform` speichern.

Übung 9.6

Testen Sie das Programm mit den Wörtern „Hauses“ und „Sonnen“.

9.5 Initialisierung der Daten eines Objekts

Bei der Deklaration eines Objekts wird in C++ automatisch Speicher für die Daten des Objekts besorgt. Die Speicheralloziierung wird vom „Default-Konstruktor“ realisiert. Dieser Konstruktor besorgt nur Speicher, weist den Objektdaten aber keine Defaultwerte zu. Möchte der Programmierer bei der Deklaration den Objektdaten spezielle Werte zuweisen, muss er innerhalb der Klassendeklaration eine eigene Konstruktorfunktion definieren, die den Default-Konstruktor überschreibt.

In C++ können über Konstruktoren auch spezielle Initialisierungswerte für die Daten der Objekte definiert werden. Die Werte können bei der Deklaration der Objekte als Argumente übergeben werden. Anzahl und Art der Argumente entscheidet welche Konstruktorfunktion aufgerufen wird.

Beachte:

Da selbstdefinierte Konstruktoren die Initialisierungswerte der Objekte nicht verändern dürfen, müssen die Konstruktoren mit `const` Argumenten definiert werden.

Damit beim Aufruf der Konstruktorfunktion die Argumente nicht unnötig auf den STACK kopiert werden, empfiehlt es sich, die Argumente als Referenz zu definieren.

Beispiel 9.4: Verwendung von Konstruktorfunktionen

```
// file: Klassenprogrammierung/constructor.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {
    string wort1; // Defaultkonstruktor
    string wort2("wie gehts"); // weiterer Konstruktor

    cout << " Hello, Programm Konstruktor " << endl;
    cout << " Default Konstruktor: " << wort1 << endl;
    cout << " Default Konstruktor: " << wort2 << endl;
    return 0;
}
```

Beispiel 9.5: Eigene Konstruktorfunktionen

```
// file: Klassenprogrammierung/constructorSelf.hpp
// description: Deklarationsdatei für die Klasse lexikon
```

```
#ifndef CONSTRUCTORSELF_HPP
#define CONSTRUCTORSELF_HPP

#include <iostream>

class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);

private:
    std::string land;
    std::string umlaute;
};

Lexikon::Lexikon() {
    land = "garkeins";
    umlaute = "";
};

Lexikon::Lexikon(const std::string &MyLand, const std::string &MyUmlaute) {
    land = MyLand;
    umlaute = MyUmlaute;
};


Lexikon::Lexikon(const std::string &MyLand) {
    land = MyLand;
};

void Lexikon::druckeEinstellungen() {
    std::cout << " Einstellungen = " << std::endl;
    std::cout << " Land = " << land << std::endl;
    std::cout << " Umlaute = " << umlaute << std::endl;
};

#endif
```

```
// file: Klassenprogrammierung/mainConstructorSelf.cpp
// description: Anwender der Klasse lexikon

#include "constructorSelf.hpp"
#include <iostream>

using namespace std;

int main() {
    Lexikon meins;
    Lexikon englisch("englisch");
```

```
    Lexikon deutsch("deutsch", "äöü");

    cout << " Hello, mainConstructorSelf " << endl;
    cout << " Konstruktor " << endl;
    meins.druckeEinstellungen();

    cout << " Konstruktor " << endl;
    englisch.druckeEinstellungen();
    cout << " Konstruktor " << endl;
    deutsch.druckeEinstellungen();
    return 0;
}
```

9.6 Löschen der Daten eines Objekts

Wenn die Lebensdauer einer temporären Variable erlischt, wird sie automatisch gelöscht. Arbeitet man mit einer Variablen einer selbstdefinierten Klasse, dann wird beim Löschen der Variable in der Klassendefinition eine Destruktorfunktion gesucht und aufgerufen. Falls die Klasse keine eigene Destruktorfunktion definiert hat, wird ein sogenannter Defaultkonstruktor aufgerufen. Der Aufruf eines Defaultdestruktors kann zu Problemen bei komplexen Daten führen, da nur die Adressen der Variablen (=flache Memberdaten) gelöscht werden, aber nicht die Daten, auf die die Variable zeigt (=tiefe Memberdaten). Diese tiefen Memberdaten bleiben als „Speicherleaks“ erhalten.

```
class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);
    ~Lexikon();   // Destruktor der Klasse Lexikon

private:
    std::string land;
    std::string umlaute;
};
```

9.7 Kopieren der Daten eines Objekts

Wenn ein Objekt der Klasse zusammengesetzten privaten Speicher verwaltet, muss man bei der Kopie eines Objekts beachten, dass nur die Adressen der privaten Daten kopiert werden, nicht deren dahinterliegenden Daten. Es wird eine *flache Kopie* und keine *tiefe Kopie* des Objekt realisiert. Damit ein Objekt beim Kopieren auch ihre privaten Daten kopiert auf die sie zeigt, muss im `private`public Teil der Klassendefinition eine sogenannte **Copy-Constructor** Funktion definiert werden. Die Copy-Constructor Funktion hat eine genau vorgeschriebene Syntax: Er hat keinen Return-Wert und hat als Argument eine `const` Referenz eines Klassenobjekts.

```
class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);
    ~Lexikon();   // Destruktor der Klasse Lexikon
    Lexikon(const Lexikon &l) // Copy Constructor

private:
    std::string land;
    std::string umlaute;
};
```

9.8 Das Überladen von Operatoren

Mit dieser Eigenschaft können in C++ Operatoren im Bezug auf eine Klasse neu definiert werden, wobei bisherige Definitionen des Operators nicht geändert werden. Wenn ein Operator überladen werden soll, kann eine `member operator` Funktion oder `friend operator Funktion` relativ zu einer Klasse definiert werden. Werden Objekte solcher Klassen mit dem neu definierten Operator aufgerufen, wird die jeweilige Operator Funktion aufgerufen.

Die member-Operatorfunktion

unter `public` :

```
ClassName operator#(arg-list)
```

die Funktionsdefinition :

```
return-type ClassName::operator#(arg-list)
{
// Auszuführende Operation
}
```

Der `return-type` kann beliebig sein, die `arg-list` hängt von der Stelligkeit des Operators ab.

Einschränkungen:

- Die Präzedenz eines Operators kann nicht geändert werden
- Die Stelligkeit eines bereits definierten Operators kann nicht geändert werden
- Operatorfunktionen dürfen kein Default-Argument haben
- Folgende Operatoren können nicht überladen werden : `.` `::` `.*` `?`
- Präprozessoroperatoren können nicht überladen werden

9.9 Überladen von relationalen und logischen Operatoren

Die Operatoren werden wie bei binären Operatoren übergeben. Hier muss das Ergebnis der Operation ein ganzzahliger `true`- oder `false`-Wert sein.

Beispiel 9.6

```
// Vergleichsoperator

int StrType::operator==(const StrType &Str) {
    return Str.text == text; // Vergleich bei Strings ist definiert !
}
```

9.10 Überladen von unären Operatoren

Bei neuen Compilern wird bei der Operatordefinition unterschieden, ob der Operator als Prä- oder Postoperator vorliegt, und es kann jeweils eine andere Operation aufgerufen werden.

1. *Fall Präoperator:* `++s1`
   Hier wird die Operatorfunktion ohne Parameter definiert.

2. *Fall Postoperator:* `s1++`
   Zur Unterscheidung wird bei der Operatordefinition ein fiktives Argument angegeben.

Beispiel 9.7

```
////////////////////////////////////////////////
//aus der  Deklarationsdatei:
StrType &StrType::operator++();
StrType &StrType::operator++(int notused);
////////////////////////////////////////////////
```

```
//aus dem  Implementationsteil:

// Prae-Operator Left Operator

StrType &StrType::operator++() {
    int i;
    for (i = 0; i < anz_char; i++) {
        if (islower(text[i])) text[i] = toupper(text[i]);
        else
            text[i] = tolower(text[i]);
    }
}

// Post - Operator Right Operator

StrType &StrType::operator++(int notused) {
    int i;
    for (i = 0; i < anz_char; i++) {
        if (isupper(text[i])) text[i] = tolower(text[i]);
        else
            text[i] = toupper(text[i]);
    }
}
/////////////////////////////////////////////
// aus dem Hauptprogramm:
StrType s1;

++s1;
s1++;
```

9.11 Überladen des Output-Operators

Da selbstdefinierte Klassen beliebige Daten in beliebigen Datenstrukturen zusammenfassen können, ist es selbstverständlich, dass Objekte einer Klasse nicht mit dem Standard-Output-Operator ausgegeben werden können. Um dies zu ermöglichen, muss man in der Klasse den Output-Operator << überladen.

Beispiel 9.8

```
ostream& operator<<(ostream& os, const StrType &str)
{
    os << "Land=" << str.land << ", Umlaute sind:" << str.umlaute;
    return os;
}
```

Formatierungsvorschläge:

Klassennamen schreiben wir im UpperCamelCase, d.h. das gesamte Wort beginnt mit einen Großbuchstaben und auch jedes Teilwort. Dasselbe wollen wir für Konstanten, Structures,

Enumerations (Aufzählungen) und Typedefs verwenden. z.B.:

```
enum BackgroundColour  {
   Cyan,
   Magenta,
   Yellow
};

const int FixedWidth = 1;
```

Compound Types, also Klassen und structs oder Typedefs, die Objekte definieren, sollten als Namen ein Nomen bekommen. z.B.:

```
class Costumer{
 //..
};
```

(Collections bekommen einen Plural als Namen)

10 Vererbung

10.1 Einleitung

Vererbung (inheritance) ist ein Mechanismus, mit dem Klassen hierarchisch aufgebaut werden können. Aus einer Oberklasse werden Unterklassen abgeleitet. Die Unterklasse kann die Attribute und Methoden der Oberklasse übernehmen (also „erben“) und gleichzeitig eigene Eigenschaften definieren.
Eine Hierarchie von Klassen führt dadurch von der allgemeinsten Klasse hin zur spezifischsten. Diejenige Klasse, deren Eigenschaften vererbt werden, heißt Basisklasse `(base class)` oder Oberklasse. Diejenige Klasse, die diese Eigenschaften erbt, heißt abgeleitete Klasse `(derived class)` oder Unterklasse. Bei der Vererbung spricht man von einer „is-a“ Beziehung: Die Unterklasse „is-a“ Oberklasse.

Beispiel 10.1: Beziehung zwischen Ober- und Unterklasse

Oberklasse : Tier
Unterklasse: Esel
Jeder Esel ist ein Tier: ein Esel „is-a“ Tier. Jede Unterklasse erbt die Eigenschaften und Methoden der Oberklasse. In der Unterklasse werden nur die neu hinzukommenden Eigenschaften und Methoden definiert. Beispiel:
Klasse Tier: Ein Tier hat einen bestimmten Namen, Eigenheit, Geschlecht und kann einen Laut machen:

```
class Tier {
public:
    Tier(std::wstring tierName):name(tierName) {}

    void eigenheiten() {
        std::wcout << L"Mein Name ist: " << name << std::endl;
    }
    void macheLaut();
    void istEin();

private:
    std::wstring name;
    std::wstring geschlecht;

protected:
    std::wstring laut;
};
```

Ein Esel ist ein Tier, erbt also alles von einem Tier, und hat zusätzliche Eigenschaften (bestimmer Laut und bestimmte Eigenheiten):

```
class Esel : public Tier {
public:
    Esel(std::wstring eselName) : Tier(eselName){
        laut = L"Iah! Iah! Iah!";
    }
};

Esel::eigenheiten() {
    std::wcout << L"Ich bin ein Esel und heisse " <<
     this->name << std::endl;
    std::wcout << L"Ich kann einen tollen Laut machen: "
     << this->laut << std::endl;
    std::wcout << L"Und ich bin furchtbar stur!" << std::endl;
}
```

Jetzt muss überlegt werden, wie die Unterklasse auf die Daten der Oberklasse zugreifen kann.

10.2 Vererbung von Zugriffsrechten

Zur Wiederholung:
Grundsätzlich legt bei der Konstruktion einer Klasse der Programmierer das Zugriffsrecht auf seine Eigenschaften und Methoden fest.
Er kann bisher unter zwei Arten von Zugriffsrechten unterscheiden:

```
public:
    Die Daten nicht geschützt. Man kann von außen auf sie zugreifen.

private:
    Die Daten sind innerhalb der Klasse selbst und aller als friend
    deklarierten Klassen verfügbar.
```

Beim Einsatz der Vererbung kommt nun noch ein neuer Zugriffsmodus dazu: der „protected“ Mode:
Bei diesem Zugriffsmode kann auf die Daten innerhalb der Klasse zugegriffen werden. Dazu kann auf die Daten auch von allen als `public` abgeleiteten Klassen zugegriffen werden.

```
protected:
    Nur die Klasse selbst und von ihr abgeleitete Klassen
    haben Zugriff auf die Daten.
```

Wie eine Hierarchie Oberklasse → Unterklasse den Zugriff auf seine Daten und Methoden definiert, legt der Programmierer bei der Definition fest:

1. Fall: <access specifier> = public
   class Esel : public Tier { ... }

Die Unterklasse Tier erbt alle Eigenschaften und Methoden der Oberklasse, die in der Oberklasse als `public` erklärt sind. Private Eigenschaften und Methoden der Oberklasse sind in diesem Fall nur auf die Oberklasse beschränkt und von der Unterklasse aus nicht zugreifbar. Damit die Möglichkeit besteht Attribute oder Methoden der Oberklasse auch und nur für die Unterklassen freizugeben, muss der Programmierer das Zugriffsattribut `protected` verwenden.

Zusammenfassung:
Eine Unterklasse, die mit dem access specifier public auf eine Oberklasse zugreift hat somit Zugriff auf alle `public` und `protected` definierten Attribute oder Methoden einer Oberklasse.

2. Fall: `<access specifier> = private`

   `class Esel : private Tier` { ...}

   Die Unterklasse erbt keine einzige Eigenschaft oder Methode der Oberklasse, außer bei `friend` deklarierten Unterklassen.

3. Fall: `<access specifier> = protected`

   `class Esel : protected Tier` { ...}

   Die Unterklasse erbt nur die im protected Bereich der Oberklassen definierten Eigenschaften oder Methoden.

Es ergibt sich folgendes Schema:

Zugriffsrecht in der Unterklasse <access specifier>	**Zugriffsrecht in der Oberklasse**
private	kein Zugriff
protected	protected
public	public und protected

Tabelle 10.1: Zugriffsrechte in den Klassen

10.3 Spezielle Methoden werden nicht vererbt

Bei der Vererbung werden

- Konstruktoren
- Kopierkonstruktoren
- Destruktoren
- Zuweisung von Objekten

nicht weitervererbt. Jede Ober/Unterklasse muss diese Methoden selbst definieren.

Beispiel 10.2: Methodendefinition

```
class Tier {
public:
    Tier(std::wstring tierName):name(tierName) {
    }

protected:
    std::wstring name;
};
```

Da jede Unterklasse ein anonymes Objekt der Oberklasse beinhaltet, muss die Unterklasse bei der Konstruktion eines Objekts, das anonyme Objekt der Oberklasse mitkonstruieren. Aus Performancegründen bietet sich die Initialisierungsliste dazu an:

Beispiel 10.3: Initialisierungsliste

```
class Esel : public Tier {
public:
     Esel(std::wstring eselName) : Tier(eselName) {
     }
};

// file: Vererbung/tier.hpp
// description:

#ifndef TIER_HPP
#define TIER_HPP

#include <iostream>
#include <stdlib.h>

class Tier {
public:
  Tier(std::wstring tierName) : name(tierName) {}
  Tier();
  void eigenheiten() {
    std::wcout << L"Mein Tiername ist: " << name << std::endl;
  }
  void macht();

  void rufen();
  void istEin();
private:
  std::wstring geschlecht;
  std::wstring geschlechtBestimmen();

protected:
  std::wstring laut;
  std::wstring name;
```

```
};

Tier::Tier() {
  this->geschlecht = geschlechtBestimmen();
}

void Tier::rufen() {
  std::wcout << this->name << L" komm her!" << std::endl;
}

void Tier::macht() {
  std::wcout << this->laut << std::endl;
}

void Tier::istEin() {
  std::wcout << this->name << L" ist ein " << this->geschlecht << std::endl;
}

std::wstring Tier::geschlechtBestimmen() {
  if (rand() % 2 == 0) {
    return L"Weiblich";
  }
    return L"Maennlich";
}

#endif /* TIER_HPP */
```

//

```
// file: Vererbung/esel.hpp
// description:

#ifndef ESEL_HPP
#define ESEL_HPP

#include "tier.hpp"

class Esel : public Tier {
public:
  Esel(std::wstring eselName) : Tier(eselName) {
    laut = L"Iah! Iah! Iah!";
  }
  Esel();
  void eigenheiten();
};

Esel::Esel() : Tier() {
  this->laut = L"Iah! Iah! Iah!";
}

void Esel::eigenheiten() {
  std::wcout << L"Ich bin ein Esel und heisse " << this->name << std::endl;
```

```
  std::wcout << L"Ich kann einen tollen Laut machen: " << this->laut << std::endl;
  std::wcout << L"Und ich bin furchtbar sturr!" << std::endl;
}

#endif /* ESEL_HPP */
```

//

```
// file: Vererbung/mainEsel.cpp
// description:

#include <cstdlib>
#include "Esel.hpp"

using namespace std;

int main() {

    setlocale(LC_ALL,"");

    Esel e(L"Mister Sturr");
    e.eigenheiten();

    return 0;
}
```

10.4 Zuweisung von Objekten einer Unterklasse an Objekte der Oberklasse

Da eine „is-a“ Beziehung zwischen Ober- und Unterklasse in C++ vorliegt, ist es möglich Objekte einer Unterklasse an Objekte der Oberklasse zuzuweisen. Spezialisierungen der Unterklasse, werden nicht übernommen. Umgekehrt ist eine Zuweisung von Objekten einer Oberklasse an ein Unterklasse nicht möglich.

Beispiel 10.4: Zuweisung an Objekte der Oberklasse

```
int main() {
  Tier t;
  Esel e(L"Sturr");

  t = e;
}
```

10.5 Überschreiben von Methoden/Funktionen in der abgeleiteten Klasse

Ähnlich wie beim Operatoroverloading wird in der Objektorientierten Programmierung Wert darauf gelegt Methoden mit gleicher Funktionalität unter gleichem Namen zu de-

finieren. An Hand der Argumente soll das Programm selbst erkennen, welche Methode zuständig ist. Dieser Grundsatz wird bei der Vererbung beibehalten. Wird eine Methode aufgerufen, die unter dem gleichen Namen in der Oberklasse definiert ist, dann ist genau festgelegt, dass immer die Methode der Unterklasse aufgerufen wird.

```
int main() {
  Tier t(L"Boris");
  Esel e(L"Milka);
  t.eigenheiten();
  e.eigenheiten();
}
```

10.6 Polymorphismus

Polymorphismus heißt auf Deutsch „Vielgestaltigkeit“. In der Objektorientierten Programmierung bedeutet Polymorphismus, dass erst zur Laufzeit entschieden wird, welche Methode in der Vererbungshierarchie verwendet wird. Der Polymorphismus kann nur dann funktionieren, wenn der Methode zur Laufzeit Information geliefert wird, auf welches Objekt sie angewendet wird. Diese Information wird einer Methode nur dann geliefert, wenn sie bei der Definition als eine „virtuelle“ Methode definiert wurde. Damit aber erst zur Laufzeit definiert wird welches Objekt der Unterklasse verwendet wird, muss mit Pointern auf die Oberklasse gearbeitet werden.
In der Oberklasse wird dann die überlagerte Methode „virtual“ genannt:

Beispiel 10.5: Polymorphismus

```
// file: Vererbung/tierVirt.hpp
// description:

#ifndef TIERVIRT_HPP
#define TIERVIRT_HPP

#include <iostream>
#include <cstdlib>

class Tier {
public:

    Tier(std::wstring tierName):name(tierName) {
        geschlecht = geschlechtBestimmen();
    }

    virtual void eigenheiten();

    void macht();
    void rufen();
    void istEin();

protected:
    std::wstring name;
    std::wstring geschlecht;
    std::wstring laut;
    std::wstring geschlechtBestimmen();
};

void Tier::eigenheiten() {
    std::wcout << L"Mein Tiername ist: " << name << std::endl;
}

void Tier::rufen() {
    std::wcout << this->name << L" komm her!" << std::endl;
}

void Tier::macht() {
    std::wcout << this->laut << std::endl;
}

void Tier::istEin() {
    std::wcout << this->name << L" ist ein " << this->geschlecht << std::endl;
}

std::wstring Tier::geschlechtBestimmen() {
    if ( rand() %2 == 0) {
```

```
        return L"Weiblich";
    }
    else {
       return L"Maennlich";
    }
}

#endif /* TIERVIRT_HPP */
```

Hinweis: Alle abgeleiteten Klassen von Tier müssen jetzt, da wir eine modifizierte Variante von „Tier“ benutzen tierVirt.hpp einbinden!

```
// file: Vererbung/tierVirtMain.cpp
// description:

#include "EselVirt.hpp"
#include "tierVirt.hpp"

using namespace std;

int main() {

    setlocale(LC_ALL,"");

    Tier t(L"Tierchen");
    Esel e(L"Sturrchen");

    Tier * ptt;

    ptt = &e;
    ptt->eigenheiten();

    ptt = &t;
    ptt->eigenheiten();
}
```

11 Templates

11.1 Generische Funktionen

In sehr vielen Anwendungen verwendet man die gleiche Anweisungsabfolge für unterschiedliche Daten. So ist z.B. das Vorgehen, zwei Objekte zu vertauschen immer der gleiche. Sei es bei ganzen Zahlen oder bei reellen Zahlen. In herkömmlichen Programmiersprachen musste man z.B. zum Vertauschen zweier Objekte für jeden Objekttyp eine eigene Funktion schreiben.
Vertauschen der Werte zweier Zahlen:

```
void swap(int & a1, int & a2) {
    int a = a1; // Zwischenvariable a wird mit a1 gleichgesetzt
    a1 = a2; //a1 ist gleich a2
    a2 = a; //a2 ist gleich Zwischenvariable a
}
```

Vertauschen der Werte zweier Strings:

```
void swap(string & a1, string & a2) {
    string a = a1;
    a1 = a2;
    a2 = a;
}
```

In der Programmiersprache C++ kann man eine Rechenvorschrift als „Schablonenfunktion“ definieren. Die Vorgehensweise der Rechenvorschrift wird festgelegt, die konkreten Objekttypen aber noch offen gelassen. Will man eine so definierte „Schablonenfunktion“ für konkrete Objekte einsetzen, dann deklariert man sie mit dem konkreten Objekttyp. Man instantiiert die konkrete Funktion. Erst zu diesem Zeitpunkt kreiert der Compiler aus der Schablonenfunktion eine konkrete Rechenvorschrift optimiert für diesen Objekttyp.
Schablonenfunktionen unterscheiden sich von normalen Funktionen dadurch, dass dem Funktionsnamen das Schlüsselwort `template` und die Platzhalter für den Datentyp vorgestellt werden:

```
template <class Ttype> ret-type func-name(parameter list) {
    // body of function &
}

template <class MyType> bool store_element (mytype &elem)
template <class MyType> bool find_element (mytype &elem)
```

Beispiel 11.1: Schablonenfunktion für das Vertauschen von Objekten

```
template <class X> void swap(X &a, X &b) {
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

Die in der Funktion verwendeten Objekttypen, die erst mit der Instantiierung eine konkrete Bedeutung bekommen, werden Platzhalter-Datentypen genannt.
Es ist möglich in einer Funktion mehr als einen Platzhalter-Datentyp zu verwenden.
Der Unterschied zum Überladen besteht darin, dass hier der Hauptteil der Funktion gleich ist. Es ist nun möglich eine generische Funktion zu überschreiben, um eine Abweichung für einen konkreten Fall zu ermöglichen, im Allgemeinen wird man aber nur eine echte Funktionen überschreiben.
Beispiel 11.2: Arbeiten mit Templates

```
// file: Templates/templates.cpp
// description: work with Templates

#include <iostream>
#include <string>

using namespace std;

template <class X> void my_swap(X &a, X &b) {
    X temp;
    temp = a;
    a = b;
    b = temp;
}

main() {
    int i = 10, j = 20;
    float x = 10.1, y = 23.3;
    string one = "first";
    string two = "second";

    cout << " Hier ist das Programm templates " << endl;
    cout << " Before i=" << i << " j=" << j << endl;
    my_swap(i, j); //swap integer
    cout << " After i=" << i << " j=" << j << endl;

    cout << " Before x=" << x << " y=" << y << endl;
    my_swap(x, y); //swap floats
    cout << " After x=" << x << " y=" << y << endl;

    cout << " Before one=" << one << " two=" << two << endl;
    my_swap(one, two); // swap strings
```

```
    cout << " After one=" << one << " two=" << two << endl;

}
```

11.2 Generische Klassen

Die Technik, den Datentyp bis zur Instantiierung offen zu lassen ist auch bei der Definition von Klassen möglich. Man definiert eine „Schablonen-Klasse“. Bei der Instantiierung erzeugt der C++ Kompiler eine konkrete Klasse, die für den festgelegten Objekttypen vorbereitet ist.

```
template <class Ttype> class class-name {
    // ...
}
```

Genau wie bei der Definition einer Klasse ohne Templates, soll die Deklaration einer Klasse mit Templates in einer `.hpp` Datei geschehen, die neben dem Deklarationsteil auch die Implementation der Memberfunktionen einer Klasse enthält.
Damit eindeutig definiert ist, welcher Datentyp des Templates in welcher Memberfunktion verwendet wird, muss dem Klassennamen der Memberfunktion der Name des Templatedatentyps nachgestellt werden.
Struktur einer Memberfunktion einer Klasse mit Templatedatentyp:

```
template <class Ttype> resulttype class-name<Ttype>::member_funktion {
    // ...
}
```

Somit ergibt sich folgende Modularisierung der Dateien bei der Arbeit mit Templates:

```
1. Datei: Deklarationsdatei der Klassen mit Templates inkl.
Implementationsteil der Klasse mit Templates
    Endung .hpp,
    z.B. memory.hpp

2. Datei: Anwenderdatei der Klasse mit Templates
    Endung: .cpp
    z.B: main_memory.cpp
```

Struktur der Deklarationsdatei inkl. Implementationsteil:

```
/////////////// Deklaration: memory.hpp //////////////////
// Deklarationsdatei fuer die Klasse memory
//
// Deklaration der Templates mit Klassen ....

/////////////// Deklaration ENDE     ////////////////////

/////////////// Implementationsteil /////////////////////
```

```
// Implementation fuer die Klasse mit Template

Implementations - Code ...

/////////////// Implementationsteil ENDE     ////////////
```

Struktur der Anwendungsdatei:

```
///////////////// Hautprogramm: mainMemory.cpp////////////
// Anwedner  der Klasse mit Template
// Einbinden der Header Datei:
#include "memory.hpp"

....

int main() {

    ... Anwenderprogramm

}
////////////// Hautprogramm ENDE     /////////////////////
```

Beispiel 11.3: Templates

```
// file: Templates/memory.hpp
// description: work with Templates

#ifndef MEMORY_HPP
#define MEMORY_HPP

#include <iostream>
#include <string>

const int MaxAnz = 10;

template <class X>
class memory {
public:
    bool storeElement(X &element);
    bool findElement(X &element);

    memory() {
        numOfElements = 0;
    };
private:
    X allElements[MaxAnz];
    int numOfElements;
};

template<class X>
bool memory<X>::storeElement(X &element) {
    std::cout << " Store " << element << std::endl;
```

```
    if (numOfElements < MaxAnz) {
        allElements[numOfElements] = element;
        numOfElements++;
        std::cout << " Success " << std::endl;
        return true;
    } else {
        std::cout << " Memory is full " << std::endl;
        return false;
    }
}

template<class X>
bool memory<X>::findElement(X &element) {
    int index = 0;
    std::cout << " find " << element << std::endl;
    while (index < numOfElements &&
            allElements[index] != element) {
        index++;
    }
    if (allElements[index] == element) {
        std::cout << " Found " << std::endl;
        return true;
    } else {
        std::cout << " Not Found " << std::endl;
        return false;
    }
}

#endif

// file: Templates/mainMemory.cpp
// description:

#include <iostream>
#include <string>

// Einbinden der Header Datei:
#include "memory.hpp"

using namespace std;

int main() {
    string wort;
    memory<string> stringMemory;

    cout << " Programm mainMemory.cpp " << endl;
    do {
        cout << " Bitte geben Sie ein Wort ein: ";
        cin >> wort;
        stringMemory.storeElement(wort);
    } while (stringMemory.findElement(wort));
```

```
    return 0;
}
```

Übersetzen des Beispiels memory:

```
g++ -o main_memory  mainMemory.cpp
```

Erweitern Sie das Programm `mainMemory.cpp` aus Kapitel 9 um folgende Memberfunktionen:
Übung 11.1

```
remove_element(int &element)
```

Die Memberfunktion soll ein int-Element im Memory finden, und falls es vorhanden ist aus dem memory-Array entfernen.

Übung 11.2

```
reverse_memory()
```

Die Memberfunktion soll die Reihenfolge aller Elemente im memory-Array umdrehen.

Übung 11.3

```
print_memory()
```

Die Memberfunktion soll das gesamte memory ausdrucken.

11.3 Erweiterung des Beispiels memory

Das vorherige Beispiel soll um eine Sortiermethode erweitert werden. Zum linearen Sortieren ist es sehr nützlich, wenn es zwei Methoden gibt:

```
// vertauscht das i-te und j-te Element des memory:
swap(int i, int j)

// sucht ab dem Element from das kleinste Element im memory:
find_min_elem(X referenz, int from)
```

Das Suchen wird dann folgendermaßen implementiert:
Im Speicher sind `anz_elemente` Elemente eingelesen.

```
int i = 0

do {
    suche das kleinste Element ab dem Element i.
            Falls es ein kleineres gibt, dann vertausche es mit dem Element i
            erhöhe i um eins
} while (i < anz_elemente)
```

Beispiel 11.4: Auszug aus dem Implementationsteil

```
/////////////////Deklaration: memory_sort.h
// Filename: memory.hpp
// Autor : Max
// Here: work with Templates

#include <iostream>
#include <string>

const int MaxAnz = 5;

template <class X>
class memory {
public:
    bool store_element(X &element);
    bool find_element(X &element);
    void print_elements();
    void sort();
    int find_index_of_min(X &referenz_elem, int from_index);
    void swap(int index_elem1, int index_elem2);

    memory() {
        num_of_elements = 0;
    };

private:
    X all_elements[MaxAnz];
    int num_of_elements;
};

///////////////// Deklaration ENDE

//Einbinden des Implementationsteils

//Erweiterung von memory.cpp

template<class X>
int f<X>::find_index_of_min(X &referenz_elem, int from_index) {
    /*
      Eingabe :
      X &referenz_elem   ...    Referenz Element
      int from_index     ...    Ab diesem Index wird gesucht
      Ausgabe :
      return value ....
        =-1, falls alle Element größer gleich als die Referenz sind
        =index des kleinsten Elements beginnend from_index
     */
    int index = from_index;
```

```
    X min_element = referenz_elem;
    print_elements();

    int min_index = -1;
    while (index < num_of_elements) {
        if (all_elements[index] < min_element) {
            min_index = index;
            min_element = all_elements[index];
        }
        index++;
    }
    return min_index;
}

template<class X>
void memory<X>::swap(int index_elem1, int index_elem2) {
    // Vertauscht die Elemente im Memory,
    // die unter dem Index  index_elem1 und  index_elem2
    // gespeichert sind.

    X hilf_element = all_elements[index_elem1];

    all_elements[index_elem1] = all_elements[index_elem2];
    all_elements[index_elem2] = hilf_element;

    return;
}

template<class X>
void memory<X>::sort() {
    // Sortiert die Elemente im Memory,
    int index_min;
    int index = 0;

    for (index = 0; index < num_of_elements - 1; index++) {
        index_min = find_index_of_min(all_elements[index], index + 1);
        if (index_min > 0) {
            swap(index, index_min);
        }
    }
}
///////////////// Implementation ENDE

///////////////// Hautprogramm: mainMemorySort.cpp
// Anwender der Klasse memory mit Template
// autor: max
// file: mainMemorySort.cpp


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

```
// Einbinden der Header Datei:
#include "memorySort.hpp"

using namespace std;

int main() {
    string wort;
    memory<string> string_memory;
    int word_num = 0;
    int index = 0;

    cout << " Hello, Programm mainMemorySort.cpp " << endl;
    cout << " Bitte geben Sie " << MaxAnz << " Wörter ein " << endl;
    do {
        cout << " Bitte geben Sie das " << word_num + 1 << ".te Wort ein >>";
        cin >> wort;
        string_memory.store_element(wort);
        word_num++;
    } while (word_num < MaxAnz);

    cout << " Sie haben folgende Wörter eingegeben " << endl;
    string_memory.print_elements();
    string_memory.sort();
    cout << " Sortiert sind die Wörter Wörter folgendermaßen :" << endl;
    string_memory.print_elements();

    return 0;
}
///////////////// Hauptprogramm ENDE     ///////////////
```

Aufgaben und Übungen zum Kapitel – Klassen III und Templates.

Übung 11.4

Implementieren Sie das Programm `mainMemorySort.cpp` (Siehe Anhang)

Übung 11.5

Verwenden Sie das Programm `mainMemorySort` (aus dem Skript, Kapitel 11) zum Einlesen und Sortieren von Buchstaben. Wie lautet das Hauptprogramm dazu?

Übung 11.6

Schreiben Sie ein Programm, das ein Wort einliest und die Buchstaben des Wortes alphabetisch sortiert. Verwenden Sie dazu die `<string>` Klasse und die Templateklasse memory aus dem Script.

Übung 11.7

Und nun nutzen Sie die Möglichkeiten die Templates bieten und ändern das Programm mainMemorySort auch zum Sortieren von strings. Was ändert sich am Programm? Implementieren Sie das Programm.

Übung 11.8

Testen sie das neue Programm um Wörter einzulesen und die Wörter alphabetisch sortiert auszugeben.

12 Standard Template Library (STL)

Die Standard Template Library (STL) ist eine Sammlung von Containern, Iteratoren, speziellen Klassen und Algorithmen, die als Templates (=Schablonen) definiert sind. Dem Benutzer bleibt die interne Implementation verborgen, die Schnittstelle zum Programmierer ist so gut es geht einheitlich. Der Benutzer kann sich für sein Programm aus dieser abstrakten Sammlung konkrete Implementationen der Templates instantiieren. Die Standard Template Library erlaubt dem Programmierer Objekte in Containern zu speichern. Um auf die Elemente in den Containern zugreifen zu können bietet die STL Iteratoren. Zur Manipulation des Containers und der Objekte gibt es in der STL Algorithmen. Im anstehenden C++0X Standard werden zahlreiche neue Implementationen aus der public domain Library 'boost' übernommen, wie z.B. Programme zur Arbeit mit regulären Ausdrücken. Die Standard Template Library wird am besten im Buch von Nicolai M. Josuttis beschrieben (Jos99). Ein weiteres, vertiefendes Buch über Arbeit mit der Standard Template Library stellt das von Scott Meyers (Sco08) dar. Beschrieben wird sie auch im Internet unter der Adresse: `http://www.sgi.com/tech/stl/index.html`

12.1 Iteratoren der STL

Um auf Elemente, die innerhalb eines Containers gespeichert sind, zuzugreifen benötigt man eine sogenannte Projektionsfunktion. Bei Arrays ist die Projektionsfunktion z.B. die Indexfunktion `[]`. Innerhalb der STL soll der Zugriff auf die Elemente eines Containers unabhängig vom Containertyp immer gleich sein. Die interne Struktur des Containers soll nach außen verborgen bleiben. Es wurde in der STL das Konzept der Iteratoren entwickelt, die diese Forderung erfüllen. Iteratoren erlauben auf die Elemente in Containern schreibend und lesend zuzugreifen.
Iteratoren sind Verweise auf Objekte einer Sequenz (Container), mit denen man über die einzelnen Elemente eines Containers „laufen“ kann. Der Iterator kann auf das erste Element des Containers gesetzt werden und von einem Element zum nächsten „wandern“ (iterieren). Der Iterator kann überprüfen, ob das letzte Element des Containers erreicht ist.

Aufbau von Iteratoren:

Es gibt 3 Grundoperationen bei Iteratoren

1. Dereferenzierung mittels * : gibt das markierte Objekt als Ergebnis zurück
2. Inkrementierung mittels ++ : setzt den Iterator zum nächsten Objekt des Containers
3. Überprüfung der Gleichheit (Ungleichheit) mittels == (!=) : liefert als Ergebnis, ob zwei Iteratoren auf das selbe Objekt eines Containers verweisen (Ausnahme: Output-Iteratoren)

Damit die Iterator-Schnittstelle zu den Containern der STL unabhängig vom Datentyp arbeiten kann, gibt es für in jeden Container entsprechende Elementfunktionen: Die wichtigsten Elementfunktionen sind `begin()` und `end()`. `begin()` liefert einen Iterator, der auf das erste Objekt eines Containers zeigt, `end()` liefert einen Iterator, der die Position hinter dem letzten Element des Constainers repräsentiert.

Beispiel 12.1: Einsatz von Iteratoren

```
//alle Werte eines List Containers ausgeben:

#include<list>    //bietet Container mit Iteratoren

using namespace std;

//Instantiierung der Liste mit <int>
list<int> myList;

//Erzeugen des Iterators für den Container list
list<int>::iterator iter;

// von einem Element zum nächsten Laufen:
// Durchlaufen vom ersten bis zum letzen Element
// und den Wert ausgeben.
for (iter = myList.begin(); iter != myList.end(); ++iter) {
        cout << "Element:"  << *iter << endl;
    }
```

Es gibt 5 Kategorien von Iteratoren:

- Output-Iteratoren für das Schreiben von Daten. Die einzelnen Objekte einer Sequenz werden hier durch Dereferenzieren beschrieben, der Iterator kann nur mittels `++` vorwärts bewegt werden.

  Beispiel:

  ```
  *p = x; //Das Objekt, auf das Iterator p zeigt
          //übernimmt den Wert von x
  ```

- INPUT-ITERATOREN für das Lesen von Daten. Die einzelnen Objekte werden durch die Dereferenzierungsoperation * gelesen, der Iterator kann ebenfalls nur vorwärts bewegt werden.

  Beispiel:

```
x = *p; //Die Variable x übernimmt Wert
        //des dereferenzierten Iterators p
        //("->" stellt den Zugriff auf das Objekt dar)
```

- FORWARD-ITERATOREN ermöglichen Lesen und Schreiben und den Vorwärtsdurchlauf, sie sind also eine Kombination aus Input- und Output-Iterator.

- BIDIREKTIONALE ITERATOREN ermöglichen das Laufen in beide Richtungen mittels `++` und -- (die Präfix-Schreibweise `--iter` ist hier grundsätzlich schneller).

  Sie werden von folgenden Containern bereitgestellt:

```
std::list
std::map
std::multimap
std::set
std::multiset
```

- RANDOM-ACCESS-ITERATOREN bieten zu allen schon genannten Operationen zusätzlich:
  - direkten Zugriff auf Elemente mit dem Indexoperator `[]`
  - direkte Verschiebung durch `iter += n, iter -= n;`
  - arithmetisches Rechnen wie z.B. `iter + n, iter - n;`
  - Abstände wie z.B. `iter1 - iter2`
  - relationale Operationen: `<, >, <=, >=`

  Sie sind in folgenden Containern vorhanden:

```
std::vector
std::deque
std::string
std::array
std::valarray (für Numerik, optimierter Vektor)
```

Außer der Verschiebung des Iterators um ein Element mittels `++` bzw. -- , gibt C++ dem Programmierer durch die Funktion `advance()` die Möglichkeit, alle Iteratoren über eine bestimmte Anzahl von Positionen laufen zu lassen. Argumente dieser Funktion sind der Iterator und an zweiter Stelle die Anzahl der Positionen, also `advance(iter,n)`. Bei bidirektionalen und Random-Access-Iteratoren, kann `n` auch negative Werte annehmen, um die Iteratoren rückwärts zu verschieben.

Beispiel 12.2: Einsatz von advance()

```
std::list<int>::iterator iter; // Listeniterator, bidirektional!
advance(iter,4);               // vier Positionen vorschieben
advance(iter,-5);              // fünf Positionen zurück
```

Die Anzahl der Elemente eines Bereichs, der durch zwei Iteratoren eingegrenzt ist, lässt sich durch `distance(iter1,iter2)` ausgeben. Sehr praktisch ist diese Funktion z. B. auch, um die Position eines Elements in einem Container zu bestimmen, indem man einfach als ersten Parameter den Iterator begin() verwendet.
Schlussendlich lassen sich auch noch mit der Methode `iter_swap(iter1,iter2)` zwei Elemente vertauschen.
Interessantes Beispiel:

```
list<int> myList;
iter_swap(myList.begin(), --myList.end());
```

Hier werden das erste und letzte Element der Liste vertauscht, da der `end()`-Iterator hinter das letzte Element zeigt.
In jeder der o.g. Klassen sind die Datentypen `iterator` für nicht konstante Objekte und `const_iterator` für konstante Objekte definiert. (Siehe dazu auch das Schlüsselwort `const` in Kapitel 4 ab Seite 33)
Wo es möglich ist, sollte const verwendet werden!

Zu besseren Lesbarkeit des Programms gibt es in C++ die Möglichkeit die Typfestlegung des Templates in einer speziellen `typedef`-Anweisung unter einem eigenen Namen vorzunehmen. Der selbstdefinierte Typname kann dann bei der Deklaration von Objekten verwendet werden.

```
#include <vector>
typedef vector <string> MyVector;

int main() {
    MyVector substantiv;
    MyVector verb;
    ...
}
```

Seit dem neuen Standard C++11 gibt es unter anderem für Iteratoren das Keyword `auto`. Mit diesem Keyword ist es nicht mehr nötig den Datentyp des Iterators explizit anzugeben. Weitere Informationen zum `auto`-Keyword können in Kapitel 16 gefunden werden.

12.2 Klassen der STL für die Computerlinguistik

12.2.1 wchar_t

Mit dem Datentypen `char` ist es nicht möglich, die Zeichensätze aller nationalen Sprachen zu repräsentieren, da er mit seiner 8- Bit Breite nur $2^8 = 256$ Zeichen zu codieren erlaubt. Deshalb wurde ein MultiByte-Datentyp `wchar_t` (wide character) eingeführt, dessen Bitbreite jedoch wieder plattformabhängig ist. In C++ ist dieser Typ ein integraler Datentyp, wie auch `char, int`, etc. Um ein Zeichen als `wchar_t` zu deklarieren, wird ein großes `L` (für long) direkt vor das Hochkomma gestellt.

Beispiel 12.3

```
std::wchar_t umlaut = L'ü';
std::wchar_t s[] = L"Eine kurze Zeichenkette";
```

Der Unicode Standard 4.0 sagt:
„The width of wchar_t is compilerspecific and can be as small as 8 bits. Consequently, programs that need to be portable across any C or C++ compiler should not use wchar_t for storing Unicode text. The wchar_t type is intended for storing compiler-defined wide characters, which may be Unicode characters in some compilers.“
Beim GNU c++ Compiler hat `wchar_t` die Bitbreite 32, beim Windows c++ die Bitbreite 16.

12.2.2 `wstring`

(ebenso: siehe hierzu auch das Kapitel zur Internationalisierung ab Seite 70)
Der Datentyp für eine Zeichenkette mit Zeichen des UCS Charactersets nennt sich `wstring`, das `w` steht hier ebenfalls für wide und ebenso wird ein String zu einem wide string durch Voranstellen eines großen `L`’s.

Beispiel 12.4: Datentyp wstring

```
std::wstring heute = L"so schön, so schön";
std::wstring stadt(L"münchen");
```

`string und wstring` besitzen die gleichen Eigenschaften und Schnittstellen. Alle (!) Standard-String-Funktionen aus ANSI-C sind auch in einer `<wstring>` Version verfügbar. Die Header-Datei zum Einbinden heißt `wchar.h` und enthält z.B. Funktionen wie `wcslen()` (wide character string length), die `wc`-Funktionen sind sogar auch in `<string>` deklariert.
Sämtliche Bibliotheksfunktionen, die mit Zeichenketten arbeiten, können bei Verwendung von `wchar_t` bzw. `wstring` auch nur noch mit vorne angehängtem `w` verwendet werden!

12.3 Utilities der STL für die Computerlinguistik

12.3.1 pair <type1, type2>

Zur Speicherung von Schlüssel/Objekt-Paaren ist in der C++Standardbibliothek (STL) ein Klassen-Template pair<const Key,T> mit zwei public-Datenelementen first und second, sowie Default- und Kopierkonstruktor definiert.
Key ist der Typ des Schlüssels, T ist der Typ T für Objekte. first speichert den Schlüssel und second das zugehörige Objekt.
Die zwei Elemente des pairs sind völlig unabhängig voneinander und können von beliebigem Datentyp sein.
Um ein Objekt des Datentyps pair zu definieren, werden hinter dem Datentyp in spitzen Klammern die beiden Datentypen, die im pair gespeichert werden sollen, angeben, dahinter folgt wie üblich der Variablenname. Wird dann nichts weiter angegeben, werden die beiden Elemente des pairs mit Defaultwerten initialisiert (z. B. 0 bei int).

Beispiel 12.5: Klassen-Template pair

```
std::pair<int,int> emptyPair;                   // Defaultwert=(0,0)
std::pair<string,float> cfPair("text",3.14F);   // Defaultkonstruktor
std::pair<string,float> copyPair(cfPair);       // Kopierkonstruktor
```

Außer einfachen Datentypen können in einem pair auch Objekte abgespeichert werden. Allerdings mit folgenden Einschränkungen:

- die Objekte müssen kopierbar sein (die Objektklasse muss den Kopierkonstruktor definieren)
- die Objekte müssen zuweisbar sein (d.h. Das Verhalten des Zuweisungsoperators muss definiert sein; ggf. muss die Objektklasse dazu den Zuweisungsoperator = überladen)
- die Objekte müssen vergleichbar sein(die Objektklasse muss < und == durch friend functions überladen; <=, != etc. werden aus diesen beiden Operatoren gebildet)

Um ein Objekt einem pair zuzuweisen, gibt man innerhalb der spitzen Klammern die Klasse des Objekts an. Die im pair gespeicherten Daten können hierbei beliebig aus Objekten, Objektpointern, Standarddatentypen und Pointern zusammengesetzt sein. Ebenso ist ein weiteres pair als Datentyp erlaubt.

12.3.1.1 make_pair()- Hilfsfunktion

In der STL ist ein Funktions-Template make_pair() definiert, dass es ermöglicht, ein pair ohne explizite Angabe der Datentypen zu erstellen.

Beispiel 12.6: Funktions-Template make_pair

```
std::pair<string,float> myPair;
myPair = std::make_pair("euler",2.71828F);
//gleichbedeutend mit:
std::pair<string,float> myPair("euler",2.71828F);
```

Vorsicht bei `pairs` mit Strings:

Beispiel 12.7: Make_pair mit string

```
std::pair<std::string,int> myPair;
myPair = std::make_pair("string",1);              //Fehler!
MyPair = std::make_pair(std::string("string"),1); //O.K  :)
```

`Maps` und `Multimaps` speichern Paare von sortierbaren Keys und Objekten. Ist `pos` die Position eines Objekts in einer `Map` bzw. `Multimap`, so kann mit `pos->first` bzw. `myMap.first` der Schlüssel und mit `pos->second` bzw. `myMap.second` das zugehörige Objekt angesprochen werden.

Beispiel 12.8: map make_pair

```
// file: STL/makeP.cpp
// description:

#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <algorithm>

using namespace std;

typedef map<int, wstring> wstring_map;
typedef wstring_map::iterator it;

int main() {
    wstring_map m; //Anlegen der Map
    it pos; //Iterator


    m.insert(wstring_map::value_type(1, L"Hans"));
    m.insert(wstring_map::value_type (1, L"Helmut"));
    m.insert(wstring_map::value_type (3, L"Herbert"));
    m.insert(wstring_map::value_type (7, L"Hubert"));

    wcout << L"Hier die String-Map: " << endl;

    for (pos = m.begin(); pos != m.end(); ++pos)
        wcout << pos->first << L" " << pos->second << endl;
```

```
        wcout << endl;

        pos = m.find(3); //Paar zum Schluessel suchen
        if (pos != m.end())
            wcout << pos->first << L" " << pos->second << endl;

        int key = 1; //Anzahl der Objekte bestimmen:
        wcout << L"Zum Schluessel " << key << L" gibt es ";
        wcout << m.count(key) << L" Objekte " << endl;

        return 0;
    }
```

Ausgabe:

```
Hier die Multimap:
1 Hans
1 Helmut
3 Herbert
7 Hubert
3 Herbert
Zum Schluessel 1 gibt es 2 Objekte
```

12.4 Container der STL für die Computerlinguistik

12.4.1 Überblick

Unter dem Datentyp Container versteht man in der STL eine Datenstruktur zur Zusammenfassung gleichartiger Objekte, die entsprechend dem Containertyp gespeichert und bearbeitet werden können.
Die STL unterscheidet verschiedene Container, die je nach Anwendung gewählt werden können.
Der Container `vector` simuliert einen eindimensionalen Vektor aus der Mathematik, der Container `set` entspricht einer Menge, bei der keine identischen Objekte vorkommen dürfen. Mit der Datenstruktur `map` wird die Datenstruktur `set` erweitert, da bei dieser Datenstruktur zwei-elementige Objekte abgespeichert werden können.
Die Datenstruktur `multiset` und `multimap` erweitern `set` und `map` um die Eigenschaft, dass identische Objekte abgespeichert werden können. Die Datenstruktur `unordered_map` und `unordered_set` stellen eine andere Implementation des `set` und `map` dar: Hier werden die Objekte in einem „hash“ und nicht in einem Baum abgespeichert.
Die STL bietet dem Programmierer Templates, die die Datenstrukturen implementieren und dem Benutzer Memberfunktionen, die den Zugriff und die Modifikation der Objekte definieren.

12.4.2 Einsatz der Container

Jeder Container kann in einem Programm benutzt werden, wenn das Container-Template mit `#include` eingebunden wird und ein Objekt des Containertemplates deklariert wird. Bei der Deklaration muss der Datentyp der Objekte definiert werden, die im Template abgespeichert werden.
Über die `public` Memberfunktionen des Containers kann dann auf die Objekte zugegriffen werden.

Beispiel 12.9: Vector <string>

Wir wollen Wörter abspeichern und verwenden dazu `string` Vektoren:

```
#include <vector>

int main() {
    vector<string> worte;
    worte.push_back("hallo");
}
```

12.4.3 vector <type>

```
#include <vector>
```

Das Template `vector` simuliert einen eindimensionalen Vektor aus der Mathematik.

Die Elemente sind sequentiell abgespeichert, am Ende des Vektors können Objekte angefügt oder gelöscht werden, über eine Indexfunktion kann auf jedes Element zugegriffen werden (Random-Access-Iterator).

Vektor Zugriffsfunktionen	**Beschreibung**
begin()	Liefert den Iterator auf das erste Element
end()	Liefert den Iterator auf das Element hinter dem letzten Element
push_back(...)	Fügt Element an das Ende des Vektors
pop_back()	Löscht das letzte Element des Vektors
swap(.)	Vertauscht zwei Elemente des Vektors
insert(,)	Fügt neues Element ein
size()	Anzahl der Elemente eines Vektoren
capacity()	Kapazität des Vektors
empty()	Leert den Vektor
[]	Zugriff auf ein Element über einen Index

Tabelle 12.1: STL vector Zugriffsfunktionen

Beispiel 12.10: Vector

```
// file: STL/vector.cpp
// description: Beispiel mit Vektoren

#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main() {
    vector<string> worte;
    string eingabe;

    cout << " Hallo! Bitte geben Sie einen String ein >>>";
    while (cin >> eingabe) {
        worte.push_back(eingabe);
        cout << " Naechsten String eingeben oder ^D >>>";
    }

    for (int i = 0; i < worte.size(); i++)
        cout << " Element[ " << i << " ] = " << worte[i] << endl;
}
```

Beispiel 12.11: Beispiel mit typedef und Iteratoren

```
// file: STL/vectorIt.cpp
// description: Beispiel mit Vektoren

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

typedef vector <string> MyVector;

int main() {
    MyVector strVec;
    MyVector::iterator pos;
    int i;

    ifstream infile("eingabe.txt");
    string line;

    while (getline(infile, line)) {
        cout << " gelesen :" << line << endl;
```

```
        strVec.push_back(line);
    }
    cout << " Alles gelesen " << endl;

    for (pos = strVec.begin(); pos != strVec.end(); ++pos) {
        cout << " Element = " << *pos << endl;
    }

    // oder

    for (i = 0; i < strVec.size(); i++)
        cout << " Element[" << i << "]= " << strVec[i] << endl;


    return 0;

}
```

Beispiel 12.12: Beispiel mit einem reverse- Iterator

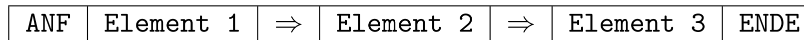
```
// file: STL/vectorRit.cpp
// description:

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

int main ()
{
  vector<int> myvector;

  // Speichert Werte von 1 bis 10
  for (int i=1; i<=10; i++) {
      myvector.push_back(i);
  }

  cout << "Jetzt kommt der normale Iterator:" << endl;

  // normaler Iterator
  vector<int>::iterator it;
  for ( it=myvector.begin() ; it < myvector.end(); ++it ) {
    cout << " " << *it << endl;
  }

  cout << "Jetzt folgt der reverse Iterator:" << endl;

  // reverse Iterator
  vector<int>::reverse_iterator rit;
  for ( rit=myvector.rbegin() ; rit < myvector.rend(); ++rit ) {
    cout << " " << *rit << endl;
  }

  return 0;
}
```

Der Vector wird nun mittels Reverse- Iterator von hinten durchlaufen, die Reihenfolge wird somit beim Durchlaufen umgekehrt.

12.4.4 list<type>

```
#include <list>
```

Das Template `list` simuliert wie das Template `vector` einen eindimensionalen Vektor aus der Mathematik.
Die Elemente sind als zweifach verkette Liste abgespeichert. Im Gegensatz zu Vektoren erlaubt diese Datenstruktur Objekte auch innerhalb und am Anfang des Containers sehr effizient einzufügen oder zu löschen.



ANF	Element 1	⇒	Element 2	⇒	Element 3	ENDE

Tabelle 12.2: Datenstruktur `list`

list Zugriffsfunktionen	Beschreibung
begin()	Liefert den Iterator auf das erste Element
end()	Liefert den Iterator auf das Element hinter dem letzten Element
push_back(...)	Fügt Element an das Ende der list
pop_back()	Löscht das letzte Element der list
push_front(...)	Fügt Element am Anfang der list ein
pop_front(...)	Löscht das erste Element der list
swap(.)	Vertauscht zwei Elemente der list
insert(,)	Fügt neues Element ein
erase(.)	Löscht Elemente
size()	Anzahl der Elemente einer Liste
capacity()	Kapazität der Liste
empty()	Leert die Liste
sort()	Sortiert die Liste

Tabelle 12.3: STL list Zugriffsfunktionen

Beispiel 12.13: Container list

```
// file: STL/list.cpp
// description:

#include <iostream>
#include <fstream>
#include <string>
#include <list>

using namespace std;

typedef list <string> MyList;

int main() {
    MyList strList;
    MyList::iterator pos;
    int i;
    ifstream infile("eingabe.txt");
    string line;

    while (getline(infile, line)) {
        cout << " gelesen :" << line << endl;
        strList.push_back(line);
    }
    cout << " Alles gelesen " << endl;

    for (pos = strList.begin(); pos != strList.end(); ++pos) {
        cout << " Element = " << *pos << endl;
    }
    // oder
    //for (i = 0; i < strList.size(); i++)
    //    cout << " Element[" << i << "]= " << strList[i] << endl;
    //return 0;
}
```

Aufgaben und Übungen zum Kapitel „Standard Template Library“ (STL).

Normalerweise speichert der list-container die Elemente nicht alphabetisch. Das soll dieses Programm ändern.Verwenden Sie dazu den STL Container list <string> und die Iterator-Funktionen.
Erweitern Sie das Beispielprogramm zu den Listen aus dem Script um folgende Funktionalität:

Übung 12.1

insert_sorted(list wort_liste, string wort)
Die Funktion soll den string wort in eine wort_liste an der alphabetisch richtigen Stelle einfügen, so, dass die einfach verkettete Liste alphabetisch sortiert bleibt.

Übung 12.2

print_list(list wort_liste)
Die Funktion soll die gesamte Liste ausdrucken.

Übung 12.3

Es soll ein main Programm geschrieben werden, das alle Wörter eines Textes aus einem Textfile sortiert in die oben definierte Wortliste einfügt und die sortierte Liste wieder ausgibt. Verwenden Sie dazu Ihre Funktionen.
!!! Achtung: Das Programm muss mit UTF-8 und deutschen Umlauten arbeiten.

12.4.5 deque<type>

`#include <deque>`
„Double Ended Queue“
Das Template `deque` simuliert wie das Template `vector` einen eindimensionalen Vektor aus der Mathematik.
Die Elemente sind aber als dynamisches Array gespeichert. Bei der Implementation wurde großer Wert darauf gelegt, dass Änderungen am Anfang und Ende sehr effizient möglich sind. Zu den Vorteilen, an beliebiger Stelle einfügen und löschen, erlaubt diese Datenstruktur Objekte über einen Index anzusprechen.

ANF	Element 1	⇌	Element 2	⇌	Element 3	ENDE

Tabelle 12.4: Datenstruktur deque

deque Zugriffsfunktionen	**Beschreibung**
begin()	Liefert den Iterator auf das erste Element
end()	Liefert den Iterator auf das Element hinter dem letzten Element
push_back(...)	Fügt Element an das Ende des deque
pop_back()	Löscht das letzte Element des deque
push_front(...)	Fügt Element am Anfang des deque ein
pop_front()	Löscht das erste Element des deque
swap(.)	Vertauscht zwei Elemente des deque
insert(,)	Fügt neues Element ein
size()	Anzahl der Elemente eines deque
capacity()	Kapazität des deque
empty()	Leert den deque
[]	Zugriff auf ein Element über einen Index

Tabelle 12.5: STL deque Zugriffsfunktionen

Eine Anwendung aus der Computerlinguistik, die eine Konkordanz erzeugt, kann sehr gut mit Hilfe einer `deque` implementiert werden. Im Kapitel 15.5 sehen Sie ein Beispiel dazu.

12.4.6 set <type>

`#include <set>`
Der Containertyp set entspricht in der Mathematik der geordneten Menge. Die Elemente des `set` sind in einem geordneten Baum gespeichert. Damit die Elemente des `set`-Containers einer Ordnung entsprechend gespeichert werden können, muss der Programmierer, falls er im `set` keine Objekte von Standardtypen speichert, auch eine Vergleichsfunktion definieren, die die lineare Ordnung der Objekte festlegt.

Set Member Funktionen	Beschreibung
begin()	Liefert den Iterator auf das erste Element
end()	Liefert den Iterator auf das Element hinter dem letzten Element
swap(.)	Vertauscht zwei Elemente des set
insert(,)	Fügt neues Element ein
size()	Anzahl der Elemente eines set
max_size()	Maximale Anzahl der Elemente eines set
capacity()	Kapazität des set
empty()	Leert den set

Tabelle 12.6: STL set Zugriffsfunktionen

12.4.6.1 Beispiel mit dem Container set

Beispiel 12.14: Container set

```
// file: STL/set.cpp
// description:

#include <iostream>
#include <fstream>
#include <string>
#include <set>

using namespace std;

typedef set<string> MySet;

int main() {
    MySet strSet;
    MySet::iterator pos;

    ifstream infile("wordlist.txt");
    string line;

    while (getline(infile, line)) {
        cout << " gelesen :" << line << endl;
        strSet.insert(line);
    }
    cout << " Alles gelesen " << endl;

    for (pos = strSet.begin(); pos != strSet.end(); ++pos) {
        cout << " Element = " << *pos << endl;
    }
    return 0;
}
```

12.4.7 map <type1,type2>

```
#include <map>
```

`Map` stellt eine Erweiterung des `set`-Containers dar.
`Map` ist wie `set` ein sortierter Container, der im Gegensatz zum `set` nicht einelementige Objekte, sondern beliebige Paare von Objekten speichern kann. Der `map`-Container eignet sich hervorragend zum Speichern von Key-Value-Paaren und ist nach den Keys sortiert. Jeder Key darf nur einmal vorkommen. Mit diesem Datentyp kann ein Assoziatives Array simuliert werden, da der Key auch ein `wstring` sein kann. Wie beim `set`-Container muss beim `map`-Container eine Vergleichsfunktion für die Keys angegeben werden, falls kein Standardvergleich existiert.

Map Member Funktionen	**Beschreibung**
begin()	Liefert den Iterator auf das erste Element
end()	Liefert den Iterator auf das Element hinter dem letzten Element
swap(.)	Vertauscht zwei Elemente der map
insert(,)	Fügt neues Element ein
size()	Anzahl der Elemente einer map
max_size()	Maximale Anzahl der Elemente einer map
capacity()	Kapazität der map
empty()	Leert die map
[]	Zugriff auf ein Element über den Key

Tabelle 12.7: STL map Zugriffsfunktionen

12.4.7.1 Beispiel zur Berechnung einer Frequenzliste mit map

Beispiel 12.15: Frequenzliste mit map

```
// file: STL/mapCount.cpp
// description:

#include <iostream>
#include <fstream>
#include <map>
#include <string>

using namespace std;

typedef map<string, int> AssocativeArray;

////////////////// functions
int myChomp(string &str);

int main() {

    fstream fs;
    ifstream infile("wordlist.txt");
    string word;
    string token;
    AssocativeArray tokenData;
    AssocativeArray::iterator im;

    //  Read each line: every line: one word !!
    while (getline(infile, token)) {
        myChomp(token); // remove carriage return at the end
        cout << "gelesen " << token;
        if (tokenData[token]) {
            //        update value
            cout << " und gefunden " << endl;
            tokenData[token]++;
        } else {
            //        Store token and set value to 1
            cout << " und nicht gefunden " << endl;
            tokenData[token] = 1;
        }

    }

    // Print out all tokens in the map container
    for (im = tokenData.begin(); im != tokenData.end(); ++im) {
        cout << " Element = #" << im->first << "#" << endl;
        cout << " Value = " << im->second << endl;
    }

    // Now read Elements from the MAP container
```

```
    do {
        cout << "Enter word to search in the map >>> ";
        getline(cin, word);

        if (word != "") {
            //      Use the find function to get an iterator position
            im = tokenData.find(word);
            if (im != tokenData.end()) {
                cout << " Found \"" << im->first;
                cout << "\"  = " << im->second << endl;
            } else {
                cout << "tokenData.find(...) no token matches \"";
                cout << word << "\"" << endl;
                cout << "Lookup tokenData[\"" << word;
                cout << "\"] would have given ";
                cout << tokenData[word] << endl;
            }
        }
    } while (word != "");

    return 0;

}

int myChomp(string &str) {
    int lastCharPos = str.length() - 1;
    if (str[lastCharPos] == '\r' || str[lastCharPos] == '\n')
        str.resize(lastCharPos);
}
```

Der `multi_set`- Container bietet sich für das Sortieren von Frequenzlisten an.

12.4.8 unordered_set<type>, unordered_map<type1,type2>

Während bei `set`- und `map`-Containern die Objekte in einem linear geordneten Baum abgespeichert werden, verwenden `unordered set/map` der STL zur Speicherung und zum Zugriff auf die Objekte eine Hashfunktion. Die Objekte sind also ungeordnet gespeichert. Im neuen C++0X Standard gibt es folgende Templates, die die Objekte über Hashfunktionen speichern und zugreifbar organisieren: `unordered_set` und `unorderd_map`.
Mit dem Einsatz von Hashfunktionen erhöht sich die Performance des Programms beträchtlich.
Die Hasherweiterung kann nur verwendet werden, wenn der Programmierer einen C++ Kompiler verwendet, der den C++0X Standard unterstützt. Ältere Kompiler bieten den Benutzern keine standardisierten Extensionen an, die HASH-Templates einzusetzen erlauben.
Zur Berechnung des Hashwertes kann der Benutzer die eingebaute Hashfunktion verwenden. Diese Hashfunktion verarbeitet aber nur C-Strings. Deshalb muss für andere Datentypen wie z.B. `wstring` eine eigene Hash-Konvertierungsfunktion angegeben werden, die intern die eingebaute Hashfunktion überschreibt.

12.4.8.1 Implementation der Hash-Templates der STL mit dem gcc

Die Include-Files finden sich unter
`<unordered_set>`
bzw.
`<unordered_map>`
Die Templatedefinition des `unordered_set` benötigt ein Argument:

1. Argument: Datentyp des "key" - Elements (hier: `string`)

Die Definition des HASH- und des Vergleichoperators bei `string` key - Elemente ist vorhanden.

Die Templatedefinition des `unordered_map` benötigt zwei Argumente:

1. Argument: Datentyp des "key" - Elements (hier: `string`)
2. Argument: Datentyp des "value" - Elements (hier: `int`)

Auch hier ist die Definition des HASH- und des Vergleichoperators bei `string` key - Elementen vorhanden.
Soll ein Hash für Stringelemente definiert werden ergibt dies folgende Definition:

```
typedef unordered_set<string>  HashSet;
```

Soll ein Hash für Stringelemente, deren Häufigkeit bestimmt werden soll, definiert werden ergibt dies folgende Definition:

```
typedef unordered_map<string, int> HashMap;
```

Beispiel für einen unordered_set Container

Beispiel 12.16: unordered_set Container

```
// autor: max
// file:unordered_set.cxx
// g++ -std=c++0x unordered_set.cpp

#include <iostream>
#include <fstream>
#include <string>
#include <unordered_set>

using namespace std;

// Definition of the HashSet Template, C++0X Standard
typedef unordered_set<string>  HashSet;

int main() {
  HashSet string_set;
  HashSet::iterator pos;

  ifstream infile ("eingabe.txt");
  string line;

  while (getline(infile,line)) {
      cout << " gelesen :" << line << endl;
      string_set.insert(line); // oder string_set[line]
    }
  cout << " Alles gelesen " << endl;

  for (pos = string_set.begin();pos != string_set.end(); ++pos) {
      cout << " Element = " << *pos << endl;
    }

  return 0;
}
```

Beispiel für einen unordered_map Container

Beispiel 12.17: unordered_map Container

```
// File: unordered_map.cxx
// g++ -std=c++11 unordered_map.cpp
// starten: cat text.txt | ./a.out

#include <iostream>
#include <locale>
#include <unordered_map>
#include <map>

// Hash zum Aufbau der Frequenzliste
```

```
typedef std::unordered_map <std::wstring,int> lexikon_t;
// Multimap Container zum Sortieren definieren
typedef std::multimap <int,std::wstring> mm_lexikon_t;


void print_sorted_freq(lexikon_t &words_freq) {
  mm_lexikon_t word_mm;
  for (auto  lex_it = words_freq.begin();
       lex_it != words_freq.end(); ++ lex_it)  {
    std::wcout << "Insert" << lex_it->first << "=" << lex_it->second << std:: endl;
    //Paar einfuegen:
    // zum Sortieren der Woerter nach der Frequenz die Paare <...,...>
    // in ... ueberfuehren,  Container fuellen
    word_mm.insert(std::pair<int,std::wstring>( lex_it->second, lex_it->first ));
  }

  std::wcout << std::endl << std::endl
         << L"Frequenzen in ... sind (sortiert):" << std::endl;

  //- ReverseIterator fuer Container deklarieren
  //-Schleife mit ReverseIterator
  for (auto rev_mm_it = word_mm.rbegin();
       rev_mm_it != word_mm.rend(); ++rev_mm_it)

    // Ausgeben des Worts mit der Frequenz
    std::wcout  << rev_mm_it->second << "=" << rev_mm_it->first << std::endl;
}

void print_sorted_freq_range_for(lexikon_t &words_freq) {
  mm_lexikon_t word_mm;
  // neues Range for: durchlaeuft von Anfang bis Ende
  for (auto &key_value_pair : words_freq) {
    std::wcout << "Insert" << key_value_pair.first << "="
                 << key_value_pair.second << std:: endl;
    word_mm.insert(std::pair<int,std::wstring>( key_value_pair.second,
                 key_value_pair.first ));
  }

  std::wcout << std::endl << std::endl
               << L"Frequenzen in ... sind (sortiert):" << std::endl;

  //-Schleife mit ReverseIterator (Range for gibt es nicht fÃ¼r reverse)
  for (auto rev_mm_it = word_mm.rbegin(); rev_mm_it != word_mm.rend(); ++rev_
mm_it)
    std::wcout  << rev_mm_it->second << "=" << rev_mm_it->first << std::endl;

}

int main() {

  std::setlocale(LC_ALL, "");
```

```
  lexikon_t lexikon;
  int num_of_words=0;
  std::wstring word;

  while (std::wcin >> word) {
    lexikon[word]++;
    num_of_words++;
  }
  std::wcout << "Inserted " << num_of_words << " Words " << std::endl;

  //    print_sorted_freq(lexikon);
  print_sorted_freq_range_for(lexikon);

  return 0;
}
```

12.5 STL-Algorithmen

12.5.1 Vorbemerkung: Laufzeiten

Der Begriff Laufzeit (runtime) meint die Zeitspanne, während der ein Programm ausgeführt wird. Da sich die konkrete Zeitdauer, die zur Lösung einer Aufgabe benötigt wird, oft nur durch Ausprobieren bestimmen lässt, begnügt man sich mit Abschätzungen. Zum Beispiel kann man mit Hilfe der „O-Notation“ (groß „oh“, nicht Null), auch Landau-Notation genannt, eine ungefähre Abschätzung, wie viel Zeit zwischen Programmstart und -ende vergehen würde, vornehmen. Bei dieser Notation wird nur die Größe der Eingabedaten beachtet und man beschreibt ein ungefähres Wachstumsverhalten der Laufzeit für größere Eingaben.
Man unterscheidet die folgenden Varianten zur Laufzeitabschätzung:

- `worst-case`-Laufzeit (schlechtester Fall): gibt an, wie lange der Algorithmus maximal braucht. Für viele Algorithmen gibt es nur wenige Eingaben, die die `worst-case`-Laufzeit erreichen, weshalb diese Abschätzung nicht unbedingt realistisch ist.
- `average-case`-Laufzeit (durchschnittlicher Fall): gibt die erwartete Laufzeit bei einer gegebenen Verteilung der Eingaben an. Da diese allerdings nicht immer bekannt ist, ist die Berechnung der `average-case`-Laufzeit in diesen Fällen nur unter einschränkenden Annahmen möglich.
- best-case-Laufzeit (bester Fall): gibt an, wie lange der Algorithmus in jedem Fall braucht, also selbst für ideale Eingaben. Diese untere Schranke wird nur selten angegeben, da sie nur für wenige Fälle zutrifft und die `best-case`-Laufzeit in der für die schlechteren Fälle enthalten ist.

Typische Zeitkomplexitäten:

- lineares Wachstum: $O(n)$
  Ein Programm, das in O(n) läuft, macht pro eingegebener Zahl eine konstante Anzahl von Rechenschritten. Verdoppelt sich hier die Anzahl der eingegebenen Zahlen, verdoppelt sich auch die Ausführungsdauer.
- quadratisches Wachstum: $O(n^2)$
  Ein Programm dieser Klasse macht pro eingegebener Zahl eine konstante Anzahl von Durchläufen durch alle Elemente. Verdoppelt man die Eingabedaten, kommt es nun also etwa zu einer Vervierfachung der Ausführungsdauer.
- exponentielles Wachstum: $O(2^n)$
  In diesem Fall verdoppelt sich mit jeder weiteren Zahl (ungefähr) die Laufzeit. Verhältnismäßig kleine Eingabegrößen können also schon zu extrem langen Laufzeiten führen. (Solch einen Zeitverbrauch hätte z. B. ein Sortieralgorithmus, der jede der möglichen Reihenfolgen auf Sortiertheit überprüft.) Katastrophal!

Deshalb versucht man, Verfahren mit exponentieller Laufzeit zu vermeiden. Ob dies jedoch überhaupt möglich ist, ist nicht bewiesen/ ungeklärt! Also strebt man Verfahren mit polynomieller oder noch besser logarithmischer Laufzeit $O(\log_n)$ bzw. $O(n \log_n)$ an. Diese logarithmische Ausführungszeit wird von heutigen Sortierverfahren erreicht.

Einige Beispiele:

sequentielle Suche: $O(n)$
binäre Suche: $O(\log_n)$
QuickSort : $O(n \log_n)$
Multiplikation von 2 n-stelligen Zahlen: $O(n \cdot n)$
Matrixmultiplikation: $O(n \cdot n \cdot n)$ (kubisch)

12.5.1.1 Der Headerfile `algorithm` für Algorithmen

`#include <algorithm>`
Es existieren 60 Algorithmen, die in 8 Klassen eingeteilt werden können:

1. Nichtmodifizierende Sequenzoperationen - diese extrahieren Informationen, suchen Positionen, modifizieren keine Elemente, z. B. `find()`.
2. Modifizierende Sequenzoperationen - Vielfältige Operationen, die Elemente, auf die sie jeweils zugreifen, verändern, z. B. `swap()`, `fill()`.
3. Sortieren - Sortieren und Prüfen der Konsistenz (Gültigkeit von Indizes),
   z.B. `sort()`, `lower_bound()` .
4. Mengenalgorithmen - Erzeugen von sortierten Strukturen, Mehrmengenoperationen, z.B `set_union()`, `set_intersection()`.
5. Heap-Operationen z. B. `make_heap()`, `push_heap()`, `sort_heap()`.
6. Minimum und Maximum - z. B. `min()`, `max()`, `min_element()`, `max_element()`.
7. Permutationen - z. B. `next_permutation()`, `prev_permutation()`.
8. Numerische Algorithmen - z. B. `partial_sum()`.

Für größere Flexibilität kann man den Algorithmen keine kompletten Container, sondern nur durch zwei Iteratoren ausgewählte Bereiche übergeben (der erste Iterator zeigt hierbei auf den Anfang und der zweite direkt hinter der das Ende der zu bearbeitenden Daten, siehe unten).
Eine kleine Auswahl an praktischen Algorithmen folgt nun:

12.5.1.2 find():

Um bei der Suche in Strings das erste Vorkommen einer Zeichenfolge in einem String zu ermitteln, steht in C++ die Funktion `find()` zur Verfügung. Das Suchergebnis ist der Index des ersten Zeichens der Zeichenfolge (Achtung: Beginn bei 0!). Ist der gesuchte String nicht vorhanden, wird die Pseudoposition `npos = -1` zurückgegeben. Diese Konstante ist in der Klasse string definiert, kann also mit `string::npos` angesprochen werden.

Beispiel 12.18: Funktion find()

```
string pippi("sie hat ein Haus, ein kunterbuntes Haus");
int first = pippi.find("Haus");
```

`first` erhält den wert 12.
Will man den letzten Buchstaben des letzten Auftretens eines Substrings ermitteln, kann man einfach die Methode `rfind()` (right find) benutzen.

Beispiel 12.19: Funktion rfind()

```
int last = pippi.rfind("Haus");
```

Hier wird `last` mit 35 initialisiert.
Um nur in einem bestimmten Teil des Containers zu suchen, kann man ihn mit 2 Iteratoren `start` und `end` auswählen: (Achtung: `start` zeigt hier wieder auf das erste Element und `end` hinter das Letzte des gewählten Bereichs.)

```
iterator find( iterator start, iterator end, const TYPE& val );
```

Der `find()`-Algorithmus sucht nun ein Element zwischen `start` und `end`, das dem Wert von `val` entspricht. Ist ein solches Element gefunden, wird ein Iterator, der auf dieses Element zeigt, zurückgegeben. Anderenfalls ist der Rückgabewert ein Iterator auf das Ende.

```
size_t find ( const string& str, size_t pos = 0 ) const;
size_t find ( const char* s, size_t pos, size_t n ) const;
size_t find ( const char* s, size_t pos = 0 ) const;
size_t find ( char c, size_t pos = 0 ) const;
```

Die Funktion `find()` durchsucht den String nach dem in `str`, `s` oder `c` gegebenen Inhalt und gibt die Position des ersten Auftretens im String zurück. Wenn `pos` angegeben ist, läuft die Suche nur auf Elementen ab einschließlich dieser Position ab, alle vorhergegangenen Elemente werden in diesem Fall ignoriert. Ist `pos` gleich 0, wird der gesamte String durchsucht. Rückgabewert ist das erste Auftreten des gesuchten Inhalts im String. Falls der gesuchte Wert nicht gefunden wurde, wird `npos` zurückgegeben. Anders als bei `find_first_of()` reicht es hier nicht, wenn mehr als ein Character gesucht wird, dass nur einer von ihnen matcht!

12.5.1.3 `find_first_of()`:

```
iterator find_first_of( iterator start, iterator end, iterator find_start,
                        iterator find_end );
iterator find_first_of( iterator start, iterator end, iterator find_start,
                        iterator find_end, BinPred bp );
```

Die Funktion `find_first_of()` sucht nach dem ersten Auftreten irgendeines der Elemente zwischen `find_start` und `find_end`. Die durchsuchten Daten liegen zwischen `start` und `end`. Wird eines der Elemente zwischen `find_start` und `find_end` gefunden, wird ein Iterator auf ebendieses Element zurückgegeben. Anderenfalls ist der Rückgabewert ein Iterator, der auf `end` zeigt.

Beispiel 12.20: Funktion find_first_of()

```
int main(){
  const string* WS = "\t\n ";
  const int n_WS = strlen(WS);

  string s1 = "Dieser Satz hat fuenf Woerter.";
  string s2 = "Eins";


  string end1 = find_first_of(s1, s1 + strlen(s1),WS, WS + n_WS);
  string end2 = find_first_of(s2, s2 + strlen(s2),WS, WS + n_WS);

  cout << "Erstes Wort von s1: " << s1 << " " << end1 - s1 << endl;
  cout << "Erstes Wort von s2: " << s2 << " " << end2 - s2 << endl;
}
```

12.5.1.4 `find_last_of()`:

```
size_t find_last_of ( const string& str, size_t pos = npos ) const;
size_t find_last_of ( const char* s, size_t pos, size_t n ) const;
size_t find_last_of ( const char* s, size_t pos = npos ) const;
size_t find_last_of ( char c, size_t pos = npos ) const;
```

Die Funktion `find_last_of()` durchsucht den String vom Ende her nach irgendeinem der Character aus `str`, `s` oder `c` und gibt die Position des ersten Auftretens im String zurück. Falls `pos` spezifiziert ist, wird die Suche nur auf Charactern bis einschließlich zu dieser Position ausgeführt, mögliche Character nach dieser Position werden in diesem Fall ignoriert. Wird nichts gefunden, ist der Rückgabewert `npos`. Es reicht, wenn irgendeines der Zeichen in dem String mit einem der gesuchten übereinstimmt. Um nach einer ganzen Folge vom Ende her zu suchen, gibt es `rfind()`. (siehe oben)

Beispiel 12.21: Funktion find_last_of() 1

```
// file: STL/find_last_of.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {

     string string1("Dies ist ein Test!");
     int location;

     location = string1.find_last_of("ist");
     cout << "\nfind_last_of() hat '" << string1[location]
             << "' gefunden an Pos: " << location << endl;

     return 0;
}
```

Ausgabe:

```
find_last_of() hat "t" gefunden an Pos: 16
```

Beispiel 12.22: Funktion find_last_of() 2

```
// file: STL/find_last_of2.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {

     string lower("abcdefghijklmnopqrstuvwxyz");
     string upper("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
     //string any(lowerChar + upperChar + ' ');
     string street("Bogenstrasse 17");

     cout << "last UCase: " << street.find_last_of(upper) << endl;
     cout << "last LCase: " << street.find_last_of(lower) << endl;
     return 0;
}
```

Ausgabe:

```
last UCase: 0
last LCase: 11
```

12.5.1.5 find_first_not_of():

```
size_t find_first_not_of ( const string& str, size_t pos = 0 ) const;
size_t find_first_not_of ( const char* s, size_t pos, size_t n ) const;
size_t find_first_not_of ( const char* s, size_t pos = 0 ) const;
size_t find_first_not_of ( char c, size_t pos = 0 ) const;
```

Die Funktion `find_first_not_of()` sucht nach dem ersten Character im Objekt, der weder in `str`, `s` oder `c` auftritt. Rückgabewert ist dessen Position. Wenn `pos` angegeben ist, findet die Suche nur auf Charactern ab einschließlich dieser Position statt. Alle vorhergehenden Elemente werden dabei ignoriert. Rückgabewert ist die Position des ersten Characters im Objekt, der nicht Teil der verglichenen Character ist. Wird kein Character gefunden, auf den dies zutrifft, wird `npos` zurückgegeben.

Beispiel 12.23: Funktion find_first_not_of()

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

int main ()
{
  string str ("suche keine Buchstaben...");
  size_t found;

  found=str.find_first_not_of("abcdefghijklmnopqrstuvwxyz"
                              "ABCDEFGHIJKLMNOPQRSTUWXYZ ");
  if (found != string::npos)
  {
    cout << "Erstes Zeichen das kein Buchstabe ist: " << str[found];
    cout << ", an Position " << int(found) << endl;
  }
  return 0;
}
```

Ausgabe:

```
Erstes Zeichen das kein Buchstabe ist: ., an Position 22
```

12.5.1.6 `find_last_not_of()`:

```
size_t find_last_not_of ( const string& str, size_t pos = npos ) const;
size_t find_last_not_of ( const char* s, size_t pos, size_t n ) const;
size_t find_last_not_of ( const char* s, size_t pos = npos ) const;
size_t find_last_not_of ( char c, size_t pos = npos ) const;
```

Die Funktion `find_last_not_of()` durchsucht den String vom Ende her nach dem ersten Character im Objekt, der weder in `str`, `s` oder `c` auftritt. Rückgabewert ist dessen Position. Falls `pos` spezifiziert ist, wird die Suche nur auf Charactern bis einschließlich zu dieser Position ausgeführt, mögliche Character nach dieser Position werden in diesem Fall ignoriert. Wird nichts gefunden, ist der Rückgabewert `npos`. Es reicht, wenn irgendeines der Zeichen in dem String nicht mit einem der Gesuchten übereinstimmt. Um nach einer ganzen Folge vom Ende her zu suchen, gibt es `rfind()`. (siehe oben).
Beispiel 12.24: Funktion find_last_not_of()

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

int main(){

  string lowerChar("abcdefghijklmnopqrstuvwxyz");
  string upperChar("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
  string anyChar(lowerChar+upperChar+' ');
  string street("Bogenstrasse 17");

  cout<<"Letztes Nichtbuchstaben- Zeichen an Position: "
         <<street.find_last_not_of(anyChar)<<endl;

  return 0;
}
```

Ausgabe:

```
Letztes Nichtbuchstaben- Zeichen an Position: 14
```

12.5.1.7 `find_if()`:

`iterator find_if( iterator start, iterator end, UnPred up );`
Die Funktion `find_if()` sucht das erste Element zwischen `start` und `end`, für das das unäre Prädikat `up` true zurückgibt. Wenn ein solches Element gefunden wird, wird ein Iterator auf dieses Element zurückgegeben. Anderenfalls ist der Rückgabewert ein Iterator, der auf `end` zeigt. Im folgenden Beispiel wird mit Hilfe von `find_if()` und einem unären „greater-than-zero“-Prädikat die erste positive Zahl größer Null in einer Liste von Integern gesucht.

Beispiel 12.25: Funktion find_if()

```
bool greaterThanZero (int i) {
return (i > 0);
}

vector<int> numsVector;
vector<int>::iterator it;
int nums[] = { 0, -1, -2, -3, -4, 342, -5 };
int end = 7;
numsVector.assign(nums, nums+end);

it = find_if(numsVector.begin(), numsVector.end(), greaterThanZero);
if( it == numsVector.end() ) {
    cout << "Keine Zahl groesser Null gefunden" << endl;
  } else {
    cout << "Zahl gefunden: " << *it << endl;
  }
```

12.5.1.8 find_end():

```
iterator find_end(iterator start, iterator end, iterator seq_start,
                    iterator seq_end );
iterator find_end(iterator start, iterator end, iterator seq_start,
                    iterator seq_end, BinPred bp );
```

Die Funktion `find_end()` sucht nach der Sequenz, die von `seq_start` und `seq_end` bestimmt ist. Wenn eine solche Sequenz zwischen `start` und `end` gefunden ist, wird ein Iterator auf das erste Element der zuletzt gefundenen Sequenz zurückgegeben. Wird keine solche Sequenz gefunden, dann ist der Rückgabewert ein Iterator auf `end`.
Falls das binäre Prädikat `bp` gegeben ist, wird es verwendet, um zu überprüfen wenn Elemente übereinstimmen.

Beispiel 12.26: Funktion find_end()

```
vector<int> numsVector;
vector<int>::iterator it;
int nums[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 };
int end = 11;
numsVector.assign(nums,nums+end);
int target1[] = { 1, 2, 3 };
it = find_end(numsVector.begin(), numsVector.end(), target1, target1+3);
if( it == numsVector.end() ) {
  cout << "Keine Übereinstimmung für { 1, 2, 3 }gefunden" << endl;
}
```

```
else {
  cout << "Die letzte Übereinstimmung ist an Position " <<
    int(it-numsVector.begin()) << endl;
}

int target2[] = { 3, 2, 3 };
it = find_end(numsVector.begin(), numsVector.end(), target2, target2+3);
if( it == numsVector.end() ) {
  cout << "Keine Übereinstimmung für { 3, 2, 3 }" << endl;
}
else {
  cout << "Die letzte Übereinstimmung ist an Position " << *it << endl;
}
```

Im ersten Teil des Quellcodes wird das letzte Auftreten von „1 2 3“ gefunden, im Zweiten wird die Sequenz, nach der gesucht wurde, nicht gefunden.

12.5.1.9 `adjacent_find()`:

```
iterator adjacent_find( iterator start, iterator end );
iterator adjacent_find( iterator start, iterator end, BinPred pr );
```

Die Funktion `adjacent_find()` sucht zwischen `start` und `end` benachbarte, identische Elemente. Falls das binäre Prädikat `pr` angegeben ist, wird es verwendet, um zu überprüfen, ob zwei Elemente identisch sind oder nicht. Der Rückgabewert ist ein Iterator, der auf das erste der zwei gefundenen Elemente zeigt. Werden solche Elemente nicht gefunden, zeigt der zurückgegebene Iterator auf `end`.

Beispiel 12.27: Funktion adjacent_if()

```
#include <iostream>
#include <algorithm>
#include <cassert>
#include <functional>
#include <deque>
using namespace std;

int main(){
  deque<string> names(5);
  deque<string>::iterator i;
  names[0] = "anne";
  names[1] = "Anne";
  names[2] = "Anna";
  names[3] = "anna";
  names[4] = "anna";
  i = adjacent_find(names.begin(), names.end());
  cout << *i;
  return 0;
}
```

Ausgabe: `anna`

12.5.1.10 search():

```
iterator search( iterator start1, iterator end1, iterator start2,
                             iterator end2 );
iterator search( iterator start1, iterator end1, iterator start2,
                             iterator end2, BinPred p );
```

Der `search()`-Algorithmus sucht die Elemente des Bereichs `[start2,end2]` im Bereich `[start1,end1]`. Falls das optionale binäre Prädikat `p` gegeben ist, wird es verwendet, um Vergleiche auf den Elementen auszuführen. Falls `search()` einen übereinstimmenden Teilbereich findet, wird ein Iterator auf den Beginn dieses Bereiches zurückgegeben. Wird keine Übereinstimmung gefunden, wird ein Iterator auf `end1` zurückgegeben. Im average case läuft `search()` in linearer Zeit, im worst case in quadratischer.

Beispiel 12.28: Funktion search()

```
// file: STL/search.cpp
// description:

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool myPredicate(int i, int j) {
    return (i == j);
}

int main() {
    vector<int> myVec;
    vector<int>::iterator it;

    for (int i = 1; i < 10; i++) myVec.push_back(i * 10);
    //Vektor: 10 20 30 40 50 60 70 80 90

    //mit normalem Vergleich:
    int match1[] = {40, 50, 60, 70};
    it = search(myVec.begin(), myVec.end(), match1, match1 + 4);

    if (it != myVec.end()) {
        cout << "match1 gefunden, Position ";
        cout << int(it - myVec.begin()) << endl;
    } else
        cout << "match1 nicht gefunden" << endl;

    //Vergleich mit Praedikat:
    int match2[] = {20, 30, 50};
    it = search(myVec.begin(), myVec.end(), match2, match2 + 3, myPredicate);
```

```
    if (it != myVec.end())
        cout << "match2 gefunden, Position " << int(it - myVec.begin()) << endl;
    else
        cout << "match2 nicht gefunden" << endl;
    return 0;
}
```

Ausgabe:

```
match1 gefunden an Position 3
match2 nicht gefunden
```

12.5.1.11 search_n():

```
iterator search_n( iterator start, iterator end, size_t num,
                            const TYPE& val );
iterator search_n( iterator start, iterator end, size_t num,
                            const TYPE& val, BinPred p );
```

Die Methode `search_n()` sucht nach `num` Auftreten von val im Bereich `[start,end]`. Falls `num` aufeinander folgende Kopien von `val` gefunden sind, wir ein Iterator auf den Anfang dieser Sequenz zurückgegeben. Anderenfalls ist der Rückgabewert ein Iterator auf `end`. Ist das optionale binäre Prädikat `p` gegeben, wird es zum Vergleich von Elementen benutzt. `search_n()` läuft in linearer Zeit.

Beispiel 12.29: Funktion search_n()

```
// file: STL/search_n.cpp
// description:

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool myPred(int i, int j) {
    return (i == j);
}

int main() {
    int myInts[] = {10, 20, 30, 30, 20, 10, 10, 20};
    vector<int> myVec(myInts, myInts + 8);

    vector<int>::iterator it;

    //mit normalem vergleich:
    it = search_n(myVec.begin(), myVec.end(), 2, 30);

    if (it != myVec.end()) {
        cout << "zwei mal 30 gefunden an Position ";
```

```
            cout << int(it - myVec.begin()) << endl;
        } else
            cout << "nicht gefunden" << endl;

        // Vergleich mit Praedikat:
        it = search_n(myVec.begin(), myVec.end(), 2, 10, myPred);

        if (it != myVec.end()){
            cout << "zwei mal 10 gefunden an Position ";
        cout << int(it - myVec.begin()) << endl;
        } else
            cout << "nicht gefunden" << endl;

        return 0;
}
```

Ausgabe:

```
zwei mal 30 gefunden an Position 2
zwei mal 10 gefunden an Position 5
```

12.5.1.12 count():

`size_t count( iterator start, iterator end, const TYPE& val );`

Mit der Methode `count()` kann man feststellen, wie viele Schlüssel/Objekt-Paare (key/value) im Container enthalten sind. Als Argument wird ein Schlüssel übergeben, gezählt werden dann die zu diesem Schlüssel passenden Objekte. Bei Maps liefert die Methode 0 oder 1, je nachdem ob es ein entsprechendes Paar gibt oder nicht. Bei Multimaps kann der zurückgelieferte Wert natürlich auch > 1 sein.
Die `count()`-Funktion gibt die Anzahl der Elemente zwischen `start` und `end` zurück, die mit dem Wert von `val` übereinstimmen.

Beispiel 12.30: Funktion count()

```
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
  string str1("Count ist praktisch fuer Strings");
  unsigned int i;
  i = count(str1.begin(), str1.end(), 'i');
  cout << "Es gibt " << i << " i's in str1" << endl;
  return 0;
}
```

Ausgabe:

```
Es gibt 3 i's in str1
```

12.5.1.13 count_if():

`size_t count_if( iterator start, iterator end, UnaryPred p );`
Die Funktion `count_if()` gibt die Anzahl der Elemente zwischen `start` und `end` zurück, für welche das Prädikat `p` true wird.

Beispiel 12.31: Funktion count_if()

```
// filename: STL/count_if.cpp
// description:

#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
#include <iterator>

using namespace std;

bool greater7(int);

int main() {
    ostream_iterator< int > output(cout, " ");
    int a2[ 10 ] = {99, 27, 13, 7, 5, 27, 3, 87, 93, 7};
    vector< int > v2(a2, a2 + 10); // Kopie von a2
    cout << "Der Vektor enthaelt: ";
    copy(v2.begin(), v2.end(), output);
    int r = count_if(v2.begin(), v2.end(), greater7);
    cout << " \nAnzahl der Elemente groesser 7: " << r;
    cout << endl;
    return 0;
}

bool greater7(int value) {
    return value > 7;
}
```

Ausgabe:

```
Der Vektor enthaelt: 99 27 13 7 5 27 3 87 93 7
Anzahl der Elemente groesser 7: 6
```

12.5.1.14 equal():

```
bool equal( iterator start1, iterator end1, iterator start2 );
bool equal( iterator start1, iterator end1, iterator start2, BinPred p );
```

Die `equal()`-Funktion gibt true zurück, falls die Elemente in den zwei Bereichen identisch sind. Der erste Bereich geht hierbei von start1 bis start2, der Zweite hat dieselbe Größe und beginnt bei start2. Ist das binäre Prädikat `p` angegeben, wird es anstatt von `==` verwendet, um jedes Paar von Elementen zu vergleichen.

Beispiel 12.32: Funktion equal()

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
   v1.push_back( i );
}

vector<int> v2;
for( int i = 0; i < 10; i++ ) {
   v2.push_back( i );
}

if( equal( v1.begin(), v1.end(), v2.begin() ) ) {
   cout << "v1 und v2 sind gleich." << endl;
} else {
   cout << "v1 und v2 sind ungleich." << endl;
}
```

Hier werden mit Hilfe von `equal()` die Elemente zweier Vektoren v1 und v2 verglichen.

12.5.1.15 mismatch():

```
pair <iterator1,iterator2> mismatch(iterator start1, iterator end1,
                 iterator start2 );
pair <iterator1,iterator2> mismatch(iterator start1, iterator end1,
                 iterator start2, BinPred p );
```

Der Algorithmus `mismatch()` vergleicht die Elemente im Bereich `[start1,end1]` mit den Elementen in einem Bereich selber Länge, beginnend bei `start2`. Der Rückgabewert ist die erste Position, an der sich die zwei Bereiche unterscheiden. Falls das optionale binäre Prädikat `p` gegeben ist, wird es verwendet, um die Elemente der zwei Bereiche zu vergleichen. `mismatch()` läuft in linearer Zeit.

Beispiel 12.33: Funktion mismatch()

```
int A1[] = {3,1,4,1,5,9};
int A2[] = {3,1,4,2,6,9};
const int N = sizeof(A1) / sizeof(int);

pair<int*,int*> result = mismatch(A1,A1+N,A2);
cout << "Der erste Unterschied ist an Position ";
cout << result.first - A1 << endl;
cout << "Die Werte sind: " << *(result.first) << ", ";
cout << *(result.second) << endl;
```

12.5.1.16 replace():

Die Methode `replace()` überschreibt einen Teilstring mit einem anderen String. Die Längen der Strings dürfen hierbei unterschiedlich sein. `replace()` hat drei Argumente. In den ersten Beiden werden die Startposition und die Länge des zu ersetzten den Strings angegeben, das dritte Argument ist der String, durch den ersetzt werden soll.

Beispiel 12.34: Funktion replace()

```
string s1("Hier komme ich!"), s2("st du");
s1.replace(9,5,s2); //ersetzt 5 Zeichen ab Index 9 durch s2
```

`s1` enthält jetzt den String "Hier kommst du!".
mit `find()`:

```
string s1("Hier kommen wir!"), s2("Peter und Paul");
int pos = s1.find("wir");
if(pos != string::npos)
  s1.replace(pos,3,s2);  //an pos 3 Zeichen ("wir") durch s2 ersetzen
```

Hier ist `s1` nun "Hier kommen Peter und Paul!".
Soll nur ein Teil eines Strings eingefügt werden, kann man dies mit Übergabe von zwei weiteren Argumenten erreichen. Das neue vierte Argument enthält dann die Startposition des Substrings, das Fünfte dessen Länge.

```
string s1("Da kommt Klaus!"), s2("Meine Maus?");
s1.replace(9,5,s2,1,9);
```

Anschließend enthält s1 den String "Da kommt eine Maus!".

```
void replace( iterator start, iterator end, const TYPE& old_value,
const TYPE& new_value );
```

Die Funktion `replace()` ersetzt jedes Element des Bereichs `[start,end`, das gleich `old_value` ist, durch `new_value`. Diese Funktion läuft in linearer Zeit.

12.5.1.17 replace_copy():

```
iterator replace_copy( iterator start, iterator end, iterator result,
            const TYPE& old_value, const TYPE& new_value );
```

Die Funktion `replace_copy()` kopiert die Elemente des Bereichs `[start,end]` nach result. Jegliche Elemente des Bereichs, die identisch mit `old_value` sind, werden durch `new_value` ersetzt.

12.5.1.18 replace_copy_if():

```
iterator replace_copy_if( iterator start, iterator end, iterator result,
                Predicate p, const TYPE& new_value );
```

`replace_copy()` kopiert die Elemente des Bereichs `[start,end)` nach `result`. Jedes Element, für das das Prädikat `p` true ist, wird hierbei durch `new_value` ersetzt.

12.5.1.19 replace_if():

```
void replace_if( iterator start, iterator end, Predicate p,
                      const TYPE& new_value );
```

Der Algorithmus `replace_if()` weist jedem Element des Bereichs `[start,end)`, für das das Prädikat `p` true zurückgibt, den Wert von `new_value` zu. Diese Funktion läuft in linearer Zeit.

12.5.1.20 unique():

Der `unique()`-Algorithmus entfernt alle aufeinander folgenden mehrfach auftretenden Elemente aus dem Bereich `[start,end)`, die Laufzeit ist linear.

```
iterator unique( iterator start, iterator end );
iterator unique( iterator start, iterator end, BinPred p );
```

Falls das binäre Prädikat `p` gegeben ist, dann wird es verwendet, um zu testen, ob zwei Elemente Duplikate voneinander sind. Der Rückgabewert von `unique()` ist ein Iterator auf das Ende des modifizierten Bereichs.

12.5.1.21 unique_copy():

```
iterator unique_copy(iterator start, iterator end, iterator result );
iterator unique_copy(iterator start, iterator end, iterator result, BinPred p );
```

Die Funktion `unique_copy()` kopiert den Bereich `[start,end)` nach `result` und entfernt dabei alle Duplikate. Ist das binäre Prädikat `p` gegeben, wir es verwendet, um zu überprüfen, ob zwei Elemente identisch sind. Der Rückgabewert ist ein Iterator auf das Ende des neuen Bereichs `(result)`. `unique_copy()` läuft in linearer Zeit.

Beispiel 12.35: Funktion unique_copy()

```
#include <algorithm>
#include <iostream>
#include <iterator>

using namespace std;

 int main(){
  int f[] = {'a','a','c','c','c','b','b','e','a','\0'}, g[10];
  int* z = unique_copy(f, f + 10, g);
  unique_copy(g,z,ostream_iterator<char>(cout,"  "),equal_to<char>());
  return 0;
}
```

g enthält nun: a c b e a und wird auch ausgeben.

12.5.1.22 sort():

Der `sort()`-Algorithmus basiert auf dem `QuickSort`-Ansatz und ist im Schnitt der schnellste Sortier-Algorithmus, er läuft in $O(n \log(n))$ für average und worst case. Das ist schneller als polynomiell, aber noch langsamer als lineare Zeit, jedoch kann das Verfahren bei ungünstigen Eingaben quadratische Laufzeit haben.
Quicksort arbeitet nach dem Prinzip „Divide and Conquer“, es wird also zunächst die zu sortierende Liste durch ein so genanntes Pivotelement (engl pivot = Achse, Drehpunkt) in zwei Teillisten (links, rechts) zerlegt. Elemente, die kleiner dem Pivotelement sind, kommen in die linke Teilliste, die Größeren in die rechte, gleich große Elemente werden beliebig verteilt und somit sind nach der Aufteilung die Elemente der linken Liste kleiner gleich denen der rechten Teilliste. Anschließend werden nach dem selben Mechanismus die beiden Teillisten sortiert (Rekursion! Abbruch bei Länge einer Teilliste von 0 oder 1).
Idealerweise wählt man das Pivotelement so, dass sich zwei etwa gleichlange Teillisten ergeben.

```
void sort( iterator start, iterator end );
void sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

Der `sort()`-Algorithmus sortiert die Elemente des Bereichs `[start,end)` nach aufsteigender Ordnung. Falls dabei 2 oder mehr gleiche Elemente auftreten, gibt es keine Garantie, in welcher Reihenfolge sie angeordnet werden (siehe `QuickSort` oben).

Beispiel 12.36: Funktion sort()

```
#include <iostream>
#include <algorithm>

using namespace std;
int main(){
    const string hello("Hello world!");
    string s(hello.begin(),hello.end());
    string::iterator pos;
    for (pos = s.begin(); pos != s.end(); ++pos) {
        cout << *pos;    }
    cout << endl;

    sort (s.begin(), s.end());

    cout << "sortiert: " << s << endl;
    return 0;
}
```

Ausgabe:

```
Hello world!
sortiert:  !Hdellloorw
```

12.5.1.23 transform():

Der `transform()`-Algorithmus wendet eine Funktion auf einen gewählten Bereich von Elementen an. Das Ergebnis jeder Anwendung der Funktion wird in einem Iterator gespeichert.

```
iterator transform(iterator start, iterator end, iterator result,
                   UnaryFunction f );
iterator transform(iterator start1, iterator end1, iterator start2,
                   iterator result, BinaryFunction f );
```

Die erste Version von `transform()` wendet die unäre Funktion `f` auf jedes Element des Bereichs `[start,end)` an und weist das erste Ergebnis/die erste Ausgabe der Funktion an den Iterator `result` zu, das zweite an `(result+1)`, etc.
Die zweite Version arbeitet ähnlich. Der Unterschied ist, das ihr zwei Bereiche durch Iteratoren angegeben werden und sie eine binäre Funktion `f` auf einem Paar von Elementen aufruft.

Beispiel 12.37: Funktion transform()

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int incr (int i) { return ++i; }
int sum (int i, int j) { return i + j; }

int main() {
    vector<int> first;
    vector<int> second;
    for (int i = 1; i < 6; i++) {
            first.push_back(i*100);
    }
    second.resize(first.size());

    // Erste Methode -> Funktion incr wird auf ersten Vector angewandt
    // Ergebnis wird im zweiten Vektor gespeichert
    transform (first.begin(), first.end(), second.begin(), incr);
    // Werte im second- Vector: 101, 201, 301, 401, 501

    // Zweite Methode -> Funktion sum wird auf ersten
    // und zweiten Vector angewandt
    // Ergebnis wird im ersten Vektor gespeichert
    transform (first.begin(), first.end(), second.begin(), first.begin(), sum);

    // Ausgabe des ersten Vektor wäre: 201, 401, 601, 801, 1001
    return 0;
}
```

Der `transform()`-Algorithmus verändert somit die Collection; er ist also *mutable*.

12.5.1.24 `for_each()`:

Der `for_each()`-Algorithmus dient zum Durchlaufen einer Collection. Das Ergebnis der verwendeten Funktion wird verworfen. Somit verwendet der Algorithmus nicht die Collection und ist somit *immutable*.

Beispiel 12.38: Funktion for_each()

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

void putOut(string);
list<string> buildList();

int main() {
    list<string> myList = buildList();

    for_each(myList.begin(),myList.end(),putOut);
}

void putOut(string ding) {
    cout << ding << endl;
}

list<string> buildList() {
    list<string> myList = list<string>();
    myList.push_back("1");
    myList.push_back("2");
    myList.push_back("drei");
    myList.push_front("null");

    return myList;
}
```

13 Erweiterung um reguläre Ausdücke, die Bibliothek Boost::Regex

In ANSI C++ und in der STL gab es lange keine Bibliotheksfunktionen zur Arbeit mit regulären Ausdrücken. John Maddock (Bjo05) hatte in den letzten Jahren eine sehr leistungsfähige C++-Library zur Arbeit mit regulären Ausdrücken entwickelt.

Diese `boost_regex`-Library ist seit C++11 direkt integriert und muss nicht mehr extra eingebunden werden. Die große Leistung dieses Pakets liegt in der Integration der Konzepte der Internationalisierung und einer API, die die Verwendung im Stile der STL-Routinen erlauben.

Bei Boost (http://www.boost.org/) handelt es sich um eine Sammlung von Bibliotheken für C++ . Eines der Ziele des Projekts ist die gute Integration in der STL. `Boost::Regex` ist eine Bibliothek des Public Domain Pakets Boost.

13.1 Integration des Pakets Boost

Sollten Sie bereits C++11 verwenden, müssen Sie das Paket Boost für die Verwendung von regulären Ausdrücken nicht installieren.

13.1.1 Installation des Pakets Boost

Sofern Ihr Compiler oder ihr Betriebssystem das Paket Boost nicht schon automatisch eingebunden hat, muss dieses Paket nachtäglich installiert werden. Der beste und sicherste Weg ist die Verwendung eines Paketmanagers:

Beispiel 13.1: Installation des Pakets Boost

apt unter Debian (und -Derivaten wie Ubuntu):

```
shell> sudo apt-get install libboost-dev
```

yum unter Red Hat (und -Derivaten wie Fedora):

```
shell> yum install boost
```

port unter MAC OSX

```
shell>  sudo port install boost
```

pacman für Arch Linux

```
shell>  pacman -S boost
```

Hinweis: Falls die oben genannten Befehle das Boost- Paket nicht finden, muss manuell danach gesucht werden:

Beispiel 13.2: Manuelle Suche nach dem Paket Boost

Beispielsweise unter Debian (und -Derivaten wie Ubuntu):

```
shell> aptitude search boost
shell> apt-cache search boost
```

Unter Red Hat (und -Derivaten wie Fedora):

```
shell> yum list boost
```

Unter Arch Linux:

```
shell> pacman -Ss boost
```

Steht keines dieser Tools zur Verfügung müssen die Sources selbst kompiliert werden. Da der Vorgang relativ komplex ist verweise ich hier auf die Installationsanleitung:
http://www.boost.org/more/getting_started/index.html

13.1.2 Kompilieren unter Verwendung der Bibliothek `Boost::Regex`

Beim Kompilieren muss man die Bibliothek einbinden, dies geschieht indem man

```
shell> g++ programm.cpp -lboost_regex
```

in die shell eingibt. Sind die Pfade falsch gesetzt, kann man sich behelfen indem man zusätzlich die Option

```
-I /ort_der_Bibliotheksfunktion/boost-Version/
```

mitgibt.

13.1.3 Einbinden von `Boost::Regex` in Eclipse

Um mit `Boost::Regex` auch in Eclipse zu arbeiten, soll man zuerst die Bibliothek zum Projekt einbinden: Project → Properties → C/C++ General → Paths and Symbols → GNU C++
Unter Includes soll man das Verzeichnis einbinden, wo die Boost Bibliothek sich befindet (unter Linux normalerweise `/usr/includes/boost`) und unter Library `boost_regex` als neue Bibliothek hinzufügen.

13.1.4 Verwendung von Boost::Regex

Damit die Prototypen der `boost_regex` Library bekannt sind, muss in das Programm die `boost_regex` Header Datei eingebunden werden.

```
#include <boost/regex.hpp>
```

Als nächstes muss für den regulären Ausdruck, den man verwenden will ein „Regular Expression Objekt“ deklariert werden:

```
boost::regex re("(\\w+|\\s)+[.?!]$");  // erzeugt eine regex Objekt
```

oder mit Optionen:

```
boost::regex re("(\\w+|\\s)+[.?!]$",boost::regex::icase); //ignore case
```

Achtung: Es müssen hier zwei Backslashes verwendet werden! Nur ein Backslash würde in C++ die Escape- Sequenz für spezielle Zeichencodes einleiten.

13.2 Verwendung von Regex mit C++11

Seit C++11 ist die Boost Regex Bibliothek direkt in C++eingebunden. Diese kann mit `include <regex>` verwendet werden.

13.3 Suchen nach einem regulären Ausdruck: regex_match

Um in einem String nach dem regulären Ausdruck zu suchen, gibt es eine `regex_match` und eine `regex_search` Funktion des Pakets. Beiden übergibt man den String, den man durchsuchen will und das „Regular Expression Objekt“.

```
regex_match(ein_String,RegexObjekt)
```

Diese Funktion übernimmt den String und ein RegexObjekt und liefert als Ergebnis einen `bool`-Wert zurück:
`true` wenn der reguläre Ausdruck mit der Zeichenkette übereinstimmt, `false` wenn nicht.

Beispiel 13.3

```
// file: Regex/regexMatch.cpp
// description:  regex_match()

#include <iostream>
#include <boost/foreach.hpp>
#include <boost/regex.hpp>

using namespace std;
using namespace boost;
int main() {
    string namen[] = {"Leon","Lucas","Ben","Finn","Jonas"};
```

```
    // entspricht regex dreiBuchstaben("\\w\\w\\w" , boost::regex::perl)
    regex dreiBuchstaben("\\w{3}");
    //regex beginntMitL("^L"); würde nicht matchen, da submatch
    regex beginntMitL("^L.*");

    BOOST_FOREACH( string name, namen) {

        if ( regex_match(name,dreiBuchstaben ) ) {
            cout << "Name: " << name <<  " hat drei Buchstaben." << endl;
        }
        if ( regex_match(name,beginntMitL ) ) {
                cout << "Name: " << name <<  " beginnt mit L." << endl;
        }
    }
}
```

Neuer Standard C14:
Beispiel 13.4

```
// file: Regex/regexMatch.cpp
// description:  regex_match()

#include <iostream>
#include <array>
#include <regex>

using namespace std;

int main() {
  array <string,5> namen {"Leon","Lucas","Ben","Finn","Jonas"};

    // entspricht regex dreiBuchstaben("\\w\\w\\w" , boost::regex::perl)
    regex dreiBuchstaben("\\w{3}");
    //regex beginntMitL("^L"); würde nicht matchen, da submatch
    regex beginntMitL("^L.*");

    for (auto name: namen)  {

        if ( regex_match(name,dreiBuchstaben ) ) {
            cout << "Name: " << name <<  " hat drei Buchstaben." << endl;
        }
        if ( regex_match(name,beginntMitL ) ) {
                cout << "Name: " << name <<  " beginnt mit L." << endl;
        }
    }
}
```

13.4 Suchen nach einem regulären Ausdruck: regex_search()

Im Gegensatz zu `regex_match` sucht die Methode `regex_search` nach Substrings in einem String. Diese Methode übernimmt beim Aufruf zwei Iteratoren, die den Suchbereich bestimmen, also `iterator1` markiert den Beginn, `iterator2` das Ende des Suchbereichs. Das `MatchObjekt` beinhaltet die Zeichenfolgen, auf die das `RegexObjekt` passt.

Beispiel 13.5

```
// file: Regex/regexSearch.cpp
// desription: regex_search(),
//             Ausgabe aller Wörter einer Zeile

#include <fstream>
#include <iostream>
#include <fstream>
#include <locale>
#include <string>
#include <boost/regex.hpp>

using namespace std;
using namespace boost;

void search_in_wstring(wregex,wstring);

int main() {
    string file("eingabe.txt");
    setlocale(LC_ALL, "de_DE.UTF-8");

    wifstream utf_8_text(file.c_str());
    utf_8_text.imbue(locale("de_DE.UTF-8"));

    wstring utf_8_string;
    getline(utf_8_text, utf_8_string);

    wregex words(L"\\b(\\w+)\\b");
    wregex nonWhitespace(L"\\S+");
    wregex russianCharacters(L"???");


    search_in_wstring(words, utf_8_string);
    search_in_wstring(nonWhitespace, utf_8_string);
    search_in_wstring(russianCharacters, utf_8_string);
}

void search_in_wstring(wregex re, wstring line) {
    wsmatch aMatch;

    wstring::const_iterator startOfSearch = line.begin();
    wstring::const_iterator endOfSearch = line.end();
```

```
        //regex_search(Startposition, Endposition, MatchVariable, Regex)
        while ( regex_search(startOfSearch, endOfSearch, aMatch, re) ) {
                wcout << L"Matched: " << aMatch[0] << endl;
                startOfSearch = aMatch[0].second;
        }
        wcout << endl << endl;
}
```

Neuer Standard C14:
Beispiel 13.6

```
// file: Regex/regexSearch_C14.cpp
// desription: regex_search(),
//             Ausgabe aller Wörter einer Zeile
// compilieren: (ab GNU G++ Version 5.1)
// g++ -std=c++11 regexSearch_C14.cpp

#include <fstream>
#include <iostream>
#include <fstream>
#include <locale>
#include <string>
#include <regex>

using namespace std;

void search_in_wstring(wregex,wstring);

int main() {
  //    string file("eingabe.txt");
  string file("eingabe_ru.txt");
  setlocale(LC_ALL, "de_DE.UTF-8");

  wifstream utf_8_text(file.c_str());
  utf_8_text.imbue(locale("de_DE.UTF-8"));

  wstring utf_8_string;
  getline(utf_8_text, utf_8_string);

  wregex words(L"\\b(\\w+)\\b");
  wregex nonWhitespace(L"\\S+");
  wregex russianCharacters(L"Ñ Ñ Ð¾");


  search_in_wstring(words, utf_8_string);
  search_in_wstring(nonWhitespace, utf_8_string);
  search_in_wstring(russianCharacters, utf_8_string);
}

void search_in_wstring(wregex re, wstring line) {
  wsmatch matches;
```

```
    regex_search(line,matches,re);

    for (auto x:matches) {
      wcout << L"Matched: " << x << endl;
    }
    wcout << endl << endl;
  }
```

13.5 Ersetzen in einem String mit einem regulären Ausdruck: regex_replace()

Um mit Hilfe eines regulären Ausdrucks in einem String etwas zu ersetzen, gibt es eine `regex_replace()`-Funktion im Paket.
Diese Funktion übernimmt den String, ein RegexObjekt, einen Ersetzungsstring und liefert als Ergebnis einen `bool`-Wert zurück. Es werden im String alle Zeichenfolgen, die auf RegexObjekt zutreffen durch die in Ersetzung definierten Angaben ersetzt.

```
regex_replace(Eingabe,RegexObjekt, Ersetzung, flag)
```

Beispiel 13.7

```
String bool;

result =  boost::regex_replace(string("abc"),boost::regex("..."),string("ABC"))
```

Ersetzt zwischen `iterator1` und `iterator2` alle Zeichenfolgen, auf die RegexObjekt zutrifft und schreibt das Ergebnis nach Ausgabe. Ausgaben können ein Stream oder String sein.

Neuer Standard C14:

Beispiel 13.8

```
// file: Regex/regexReplace_C14.cpp
// desription: regex_replace(),
//             Ausgabe aller Wörter einer Zeile
// compilieren: (ab GNU G++ Version 5.1)
// g++ -std=c++11 regexReplace_C14.cpp

#include <iostream>
#include <locale>
#include <string>
#include <regex>


int main() {

  setlocale(LC_ALL, "");
```

```
    std::locale mylocale("de_DE.utf-8");

    std::wstring text (L"Würden Sie bitte entsprechend entscheiden");
    std::wregex words(L"\\b(\\w+)\\b");
    std::wregex de_words;
    std::wregex prefix(L"\\bent");
    std::wregex bigram(L"\\b(\\w+)\\s(\\w+)\\b");

    de_words.imbue(mylocale);
    //  de_words.assign(L"[[:lower:]]");
    de_words.assign(L"[äöü]");

    std::wcout << regex_replace(text,words,L"...") << std::endl;
    std::wcout << regex_replace(text,de_words,L"_") << std::endl;
    std::wcout << regex_replace(text,prefix,L"___") << std::endl;
    std::wcout << regex_replace(text,bigram,L"$2  $1") << std::endl;
  }
```

13.6 UNICODE und Lokalisierung mit boost

Die Bibliothek boost unterstützt den Datentyp `wchar_t` und somit auch wstrings. Jedes locale definiert Buchstabenklassen z.B. `\w` und `\W`, die die locale nur berücksichtigen, wenn das richtige locale gesetzt wurde.

Beispiel 13.9

```
// file: Regex/regexWString.cpp
// description: regex_match() mit Internationalisierung

#include <boost/regex.hpp>
#include <fstream>
#include <iostream>
#include <locale>
#include <string>

using namespace std;

int main() {

    setlocale(LC_CTYPE,"");
    locale mylocale("de_DE.utf-8");
    wstring zeile;
    wifstream datei;
    string filename;

    // Regulaerer Ausdruck für wstring
    boost::wregex re1(L"(\\s*ueber\\s*)", boost::wregex::icase);
    boost::wregex re2(L"(\\s*\\w+\\s*)", boost::wregex::icase);
    boost::wsmatch what;
```

```
    wcout.imbue(mylocale); //  wcout mit dem locale 'einfaerben'

    for (int i = 1; i < argc; ++i) {
        filename = argv[i];
        datei.open(filename.c_str());
        datei.imbue(mylocale);
        int j = 0;
        while (getline(datei, zeile)) {
            j++;
            //getline(datei,zeile);
            // Methode Regex_Match
            if (regex_match(zeile, what, re1)) {
                wcout << L"re1 found. Line= " << j << endl;
            }
            if (regex_match(zeile, what, re2)) {
                wcout << L"re2 found, Line= " << j << endl;
            }

        }
        datei.close();
    }
    return 0;
}
```

Neuer Standard C14:
Beispiel 13.10

```
// file: Regex/regexWString.cpp
// description: regex_match() mit Internationalisierung

#include <fstream>
#include <iostream>
#include <locale>
#include <string>
#include <regex>

using namespace std;

int main(int argc, const char* argv[]) {

    setlocale(LC_CTYPE,"");
    locale mylocale("de_DE.utf-8");
    wstring zeile;
    wifstream datei;
    string filename;

    // RegulÃ¤rer Ausdruck für wstring
    std::wregex re1;
    std::wregex re3;  //(L"(\\s*\\w+\\s*)");
    std::wregex re2;
```

```
    re1.imbue(mylocale);
    re1.assign(L"(\\süber\\s)");
    re2.imbue(mylocale);
    re2.assign(L"(\\sdie\\s)");

    std::wsmatch what;

    wcout.imbue(mylocale); //  wcout mit dem locale 'einfärben'

    for (int i = 1; i < argc; ++i) {
        filename = argv[i];
        datei.open(filename.c_str());
        datei.imbue(mylocale);
        int j = 0;
        while (getline(datei, zeile)) {
            j++;

        wcout << "IN:" << zeile << endl;
            //getline(datei,zeile);
            // Methode Regex_Match
            if (regex_find(zeile, re1)) {
                wcout << L"re1 found. Line= " << j << endl;
            }
            if (regex_match(zeile, re2)) {
                wcout << L"re2 found, Line= " << j << endl;
            }

        }
        datei.close();
    }
    return 0;
}
```

Inhalt der Datei: `text.txt`

```
shell> cat text.txt
das ist ein Test
das wäre zweite Zeile
der
über
für
dass
Das ist die dritte Zeile
```

Aufruf des Programms:

```
shell> ./a.out text.txt
re2 found, Line= 3
re1 found. Line= 4
re2 found, Line= 4
re2 found, Line= 5
re2 found, Line= 6
```

13.7 Markierte Subexpressions

Wie aus Perl bekannt, kann man die Matches von regulären Ausdrücken in einer Variablen speichern. In Perl heißt diese Methode *capturing regular expressions*. Dadurch erreicht man erstens eine Gruppierung von Ausdrücken und zweitens einen Zugriff auf Teile des gesamten Treffers (Substrings). In Perl benutzt man dafür die Variablen \$1,...,\$n, die man auf mit `()` geklammerte Ausdrücke anwendet. Boost arbeitet nach dem gleichen Prinzip, wobei der Zugriff auf die Substrings etwas anders funktioniert. Hat man einen Ausdruck geklammert, können nun verschiedene Funktionen auf die Substrings angewendet werden:

```
regex_replace:
```

interne Variablen `$1,$2,...`

```
boost::wregex re3(L"\\s*(\\w)(\\w)(\\w)(\\w)\\s*");
wstring zeile, erg;

while (getline(datei, zeile)) {
    j++;
    if (regex_match(zeile, re3)) {
        wcout << L"re3 found " << j << zeile << endl;
        erg = regex_replace(zeile, re3, L"# $4 - $3 - $2 - $1 #");
        wcout << L"replaced:" << j << erg << endl;
    }
}
datei.close();
```

`regex_match` über ein zusätzliches Arrayargument. Die Subexpressions stehen im Array:

```
boost::wregex re(L"(\\s*(ü)be(r)\\s*)", boost::wregex::icase);
boost::wsmatch what;


while (getline(datei, zeile)) {
    j++;
    //getline(datei,zeile);
    // Methode Regex_Match
    if (regex_match(zeile, what, re)) {
        wcout << L"re found " << j << L" $0=" << what[0] << endl;
        wcout << L" $1=" << what[1] << endl;
        wcout << L" $2=" << what[2] << L" $3=" << what[3] << endl;
    }

}
datei.close();
```

13.8 Erstes Vorkommen in einer Zeichenkette finden: regex_find()

Um in einer Zeichenkette das erste Vorkommen eines bestimmten Strings zu finden, gibt es eine regex_find()- Funktion in der Boost- Bibliothek:

Beispiel 13.11

```
#include <string>
#include <vector>
#include <iostream>
#include <boost/foreach.hpp>
#include <boost/algorithm/string.hpp>

using namespace std;
using namespace boost;

int main() {

        setlocale(LC_ALL,"");

        wstring text = L"Die Torheit tritt auf und spricht:";

        vector< iterator_range<wstring::iterator> > findVector;
        find_iterator<wstring::iterator> sfi;

        for(
                // sfi (String find iterator) auf erstes auftreten setzen
                sfi = make_find_iterator(text, first_finder("au", is_iequal())) ;
                // Solange nicht leer
                sfi != find_iterator<wstring::iterator>();
                // Auf nächste Position
                sfi++
        ) {
                wcout << copy_range<wstring>(*sfi) << endl;
        }
}
```

13.9 Alle Vorkommen in einer Zeichenkette finden: regex_find_all()

Um alle Vorkommen eines bestimmten Strings zu finden, gibt es eine regex_find_all()- Funktion in der Boost- Bibliothek:

Beispiel 13.12

```
#include <string>
#include <vector>
#include <iostream>
#include <boost/foreach.hpp>
#include <boost/algorithm/string.hpp>
```

```
using namespace std;
using namespace boost;


int main() {

        setlocale(LC_ALL,"");

        wstring text = L"Die Torheit tritt auf und spricht:";

        vector< iterator_range<wstring::iterator> > findVector;

        //find_all( wohinSpeichern, woSuchen, "Was suchen" ) case sensitive
        find_all( findVector, text, L"mir" );

        // iterator_range als Rückgabetyp , mit BOOST_FOREACH
        BOOST_FOREACH( iterator_range<wstring::iterator> it, findVector) {
                wcout << it << endl;
        }
}
```

Bemerkung: `find_all()` unterscheidet zwischen Klein- und Großschreibung. Soll das vernachlässigt werden bietet Boost die Möglichkeit `ifind_all()` zu verwenden!

13.10 Zeichenketten aufsplitten: regex_split()

Die Boost- Bibliothek ermöglicht auch ein Aufsplitten von Zeichenketten. Als Zeichenkette des Funktionsarguments der Methode `is_any_of()` können die gewünschten Zeichen verwendet werden, an denen der Text aufgesplittet werden soll.
Hinweis: Die Aufsplittung erfolgt hier nur an einzelnen Zeichen (wird hier zum Beispiel ein regulärer Ausdruck verwendet, so funktioniert das nicht!).

Beispiel 13.13

```
#include <vector>
#include <iostream>
#include <boost/foreach.hpp>
#include <boost/algorithm/string.hpp>

using namespace std;
using namespace boost;

int main() {
        setlocale(LC_ALL,"");

        wstring text = L"Die Torheit tritt auf und spricht:";

        vector<wstring> words;
```

```
        /*split( wohinSpeichern, woSuchen, "an was splitten" )*/
        split( words, text, is_any_of(L" ,.:"), token_compress_on );

        BOOST_FOREACH( wstring word, words) {
                wcout << word << endl;
        }
}
```

13.11 Zeichenketten durchlaufen: regex_iterator()

Eine Zeichenkette kann wie folgt mit einem Regex- Iterator durchlaufen werden:

Beispiel 13.14

```
#include <iostream>
#include <boost/foreach.hpp>
#include <boost/regex.hpp>

using namespace std;
using namespace boost;

int main() {
        wstring text = L"German Europa, Software + das Internet: International";

        wregex re(L"[A-ZÖÄÜ]\\S+");

        regex_token_iterator<wstring::const_iterator> aMatch(text.begin(),
        text.end(),re);
        regex_token_iterator<wstring::const_iterator> noMatch ;

        while (aMatch != noMatch ) {
                wcout << * aMatch << endl;
                aMatch++;
        }
}
```

13.12 Unicode Zeichenklassen

Betrachtet man folgendes Beispiel, wird man feststellen, dass standardmäßig der reguläre Ausdruck nicht passen würde:

```
#include <iostream>
#include <boost/regex.hpp>

int main()
{
    std::setlocale(LC_ALL, "");

    boost::wregex condition(L"\\p{u}");
```

```
    std::wstring test_word(L"Ü");

    if (boost::regex_match(test_word, condition)) {
        std::wcout << L"Matches!" << std::endl;
    }
}
```

Das liegt daran, dass `boost::wregex` genauso wie z.B. ein `wide-stream` erst eingefärbt werden muss. Dasselbe muss man ebenfalls machen, falls als regulärer Ausdruck z.B. die `[[:upper:]]` Zeichenklasse verwendet werden soll:

Beispiel 13.15

```
#include <iostream>
#include <boost/regex.hpp>

int main()
{
    std::setlocale(LC_ALL, "");

    boost::wregex condition;
    condition.imbue(std::locale(""));
    condition.assign(L"\\p{u}");

    std::wstring test_word(L"Ü");

    if (boost::regex_match(test_word, condition)) {
        std::wcout << L"Matches!" << std::endl;
    }

    boost::wregex condition2;
    condition2.imbue(std::locale(""));
    condition2.assign(L"[[:upper:]]");

    if (boost::regex_match(test_word, condition2)) {
        std::wcout << L"Matches!" << std::endl;
    }
}
```

Die Regex-Bibliothek besitzt desweiteren noch Variablen und Funktionen für Unicode reguläre Ausdrücke:

```
#include <iostream>
#include <boost/regex.hpp>
#include <boost/regex/icu.hpp>

int main()
{
    std::setlocale(LC_ALL, "");
```

```
    boost::u32regex condition3 = boost::make_u32regex(L"\\p{u}");

    if (boost::u32regex_match(test_word, condition3)) {
        std::wcout << L"Matches using lib icu!" << std::endl;
    }
}
```

Das ganze muss dann mit:

```
shell> g++ programm.cpp -lboost_regex -licuuc
```

kompiliert werden!

14 Ausnahmebehandlung

Es gehört in die Kategorie „schlechter Programmierstil“, wenn ein Programm vorzeitig abbricht und keine Fehlermeldung ausgibt.

In einem C++ Programm kann festgelegt werden, dass bei einem Laufzeitfehler nicht das Programm terminiert, sondern eine selbstdefinierte Routine aufgerufen wird. Dann liegt es beim Programmierer, in dieser Routine eine sachgemäße Fehlerbehandlung durchzuführen und dem Benutzer detailliert vom Laufzeitfehler zu berichten und eventuell sogar Korrekturen zu veranlassen.

Zum Abfangen von Laufzeitfehlern gibt es in C++ die sogenannte Ausnahmebehandlung.

Der Mechanismus, der dahinter steckt ist sehr einfach:

1. der Programmierer definiert mit einer „`try`“ Anweisung, dass im nächsten Block auftretende Exceptions beachtet werden.
2. der Programmierer definiert eine „`catch`“ Anweisung, bei der nach einer eventuell auftretenden Exception weitergemacht wird.

„The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user.“
Vertiy Stob. „Catch as catch can: A light-hearted look at exception handling“, The Register, 11 January 2006.

Der Programmierer kann in seinen Programmen auch selbst Exceptions generieren, die den Programmablauf sofort unterbrechen. Dazu gibt es die sogenannte „`throw`“ Anweisung. Stößt das Programm auf eine „`throw`“ Anweisung, wird der Programmablauf sofort unterbrochen und nach einer „`catch`“ Anweisung gesucht, bei der das Programm fortgesetzt wird.

Wichtig ist, dass es verschiedene Arten von Exceptions gibt, und dass es für jede Exception die passende „`catch`“ Anweisung gibt.

Durch Ausnahmebehandlung ist es möglich, auftretende Laufzeitfehler, die unter Umständen zum Abbruch des Programms führen würden, abzufangen. Dies hat zum Vorteil, dass man erstens selbst festlegen kann, was bei einem Laufzeitfehler passieren soll und zweitens,

dass man bei Auftreten eines Fehlers erkennen kann, wo dieser aufgetreten ist. Zudem kann man an anderer Stelle auf den Fehler reagieren, als wo dieser ausgelöst wurde.

14.1 Selbstdefinierte Ausnahmen

In C++ sind bei der Ausnahmebehandlung drei Schlüsselwörter von Bedeutung: `try`, `catch` und `throw`.
Will man eine oder mehrere Anweisungen auf Fehler überwachen, schließt man diese in einen `try`-Block ein. Tritt in diesem Block ein Fehler auf, so wird eine so genannte Ausnahme ausgelöst (mittels `throw`), welche dann von `catch` abgefangen wird. Alles, was im `try`-Block nach der Anweisung, die eine Ausnahme ausgelöst hat, steht, wird nicht ausgeführt!
Bei `catch` handelt es sich ebenfalls um einen Block, der den Code enthält, welcher beim Abfangen der Ausnahme ausgeführt wird.

```
try {
    anweisungen(auch Funktionsaufrufe)
} catch (typ1 arg) {
    anweisungen
} catch (typ2 arg) {
    anweisungen
} catch (...) {
    //optionaler, genereller Handler für alle übrigen Exceptions
}
```

[Die drei Punkte sind reguläre C++Syntax!]

Wie oben gezeigt, kann es mehrere `catch`-Blöcke geben. Welcher davon ausgeführt wird, hängt vom Typ der Ausnahme ab. Wird keine Ausnahme ausgelöst, oder passt kein Ausnahmetyp zu irgendeinem `catch()` (dies kann nur passieren, wenn kein genereller Handler mit `(...)` vorhanden ist), dann werden diese Blöcke einfach übersprungen.

Da der `catch(...)`-Block wirklich alle Exceptions fängt, muss er als letzter der `catch`-Blöcke aufgeführt werden, da diese sequentiell durchlaufen werden und sonst immer nur er aufgerufen werden würde. Ein nicht zu vernachlässigender Nachteil ist hier, dass man den Typ der aufgetretenen Exception nicht kennt und deshalb nicht genau reagieren kann. Wenn vom Programmierer nicht anders spezifiziert (z.B. durch `exit()`), wird das Programm mit der Anweisung nach `catch` fortgesetzt.
Erzeugt werden Ausnahmen durch das Schlüsselwort `throw`:

```
throw ausnahme;
```

Der Datentyp der von `throw` erzeugten Ausnahme ist wichtig. Es können auch eigene Datentypen für Ausnahmen verwendet werden, meistens wirft man aber eine Instanz einer Exceptionklasse.

Beispiel 14.1

```
// file: Exception/exception.cpp
// description:

#include <iostream>
#include<string>

using namespace std;

int main() {

    class Fehler : public exception {
    };

    locale mylocale("de_DE.utf-8");
    locale::global(mylocale);

    try {
        throw Fehler();
        throw 666; // int, wird abgefangen
        // wird nicht mehr ausgeführt:
        throw L"stone";
        wcout << L"kein Fehler aufgetreten!\n";
    } catch (int i) {
        wcout << L"int-Fehler " << i << " abgefangen\n";
    } catch (wstring str) {
        wcout << L"Stringfehler " << str << " abgefangen\n";
    } catch (Fehler &f) {
        wcout << L"Fehler, eine Klasse" << endl;
        wcout << "was ist los : " << f.what() << endl;
    }
    return 0;
}
```

Es ist auch möglich, Ausnahmen in Funktionen, die innerhalb des `try`-Blocks aufgerufen werden, auszulösen.
Zugriff auf die Fehlermeldung erhält man dann durch die Methode `what()`:

```
cout << e.what() << endl;
```

Mit der Funktion `uncaught_exception()` aus `<exception>` kann man feststellen, ob gerade eine Exception aktiv ist. Dies kann man z.B. zum **Loggen von Fehlern** verwenden.

Beispiel 14.2

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

class Logger {
public:

    ~Logger() {
        if (uncaught_exception()) {
            cout << "Funktion wurde durch einen Fehler beendet\n";
        }
    }
};

void foo() {
    Logger log;
    throw "Fehler";
}

int main() {
    try {
        foo();
    } catch (...) {
    }
}
```

Merksatz:
Eine Exception wird immer

- by value geworfen (`throw Exception();`)
- by reference gefangen (`catch(Exception& e);`).

14.1.1 Vordefinierte Ausnahmen

14.1.1.1 Standard-Exceptions

Diese Exceptions lassen sich generell in drei Gruppen unterteilen:

1. Exceptions zur Unterstützung der Sprache C++ (System-Exceptions)
2. Exceptions die in der Standard-Bibliothek verwendet werden (`logic_error`, `ios_base::failure`)
3. Exceptions die auf Grund von Berechnungen auftreten (`runtime_error`)

Alle Exception-Klassen besitzen eine public-Memberfunktion `what()`, um Informationen über die aufgetretene Exception zu erhalten. `what()` liefert einen 0-terminierten Byte-String zurück. (Sein Inhalt ist implementierungsabhängig.)

14.1.1.2 System-Exceptions

System Exceptions sind ein fester Bestandteil des C++ Standards:

- Die `bad_alloc`-Exception (in der Header-Datei `new` definiert) wird bei einem misslungenen Versuch zur dynamischen Speicher-Allokation ausgelöst. D.h. jeder Aufruf des `new`-Operators kann damit eine Exception auslösen!
- Die `bad_cast`-Exception (in der Header-Datei `typeinfo` definiert) wird durch den `dynamic_cast`-Operator (siehe unten) ausgelöst, wenn eine Typkonvertierung einer Referenz fehlschlägt. Eine fehlerhafte Konvertierung eines Zeigers liefert jedoch einen NULL-Zeiger zurück.
- Die `bad_typeid`-Exception (in der Header-Datei `typeinfo` definiert) wird durch den typeid-Operator ausgelöst, wenn die Typinformationen eines dereferenzierten NULL-Zeigers ermittelt werden soll. Die alleinige Angabe des NULL-Zeigers löst aber keine `bad_typeid`-Exception aus, da ja die Typinformation des Zeigers selbst ermittelt werden kann!
- Die `bad_exception`-Exception (in der Header-Datei `exception` definiert) kann ausgelöst werden, wenn eine (Member-)funktion eine Exception auslöst, die nicht in ihrer Exception-Spezifikation aufgeführt ist.

Bemerkung `dynamic_cast`:
Wie alle anderen Cast-Operatoren dient auch der `dynamic_cast` zur Konvertierung eines Ausdrucks in einen anderen Typ. Allerdings für polymorphe Typen:
Die Syntax ist: `dynamic_cast<target_type> (expr)`
`target_type` ist der Zieltyp des Ausdrucks `expr` (also der Typ in den `expr` umgewandelt werden soll). Für `dynamic_cast` können dies nur Pointer- bzw. Referenztypen sein.

Die anderen Cast-Operatoren:

- `(type) expr` wie in C,
  expr wird in den Typ `(type)` gecastet

- `const_cast<type> (expr)`
  konvertiert Konstanten in normale Variablen; Vorsicht! (siehe auch „Konstanten“, Kapitel 4)

- `static_cast<type> (expr)`
  entspricht dem klassischen Cast-Operator aus C, nicht-polymorphe Typkonvertierung

- `reinterpret_cast<type> (expr)`
  wandelt beliebige Typen um (z.B. auch `int` in einen Pointer); unchecked und nicht portabel, deshalb gefährlich!

14.1.1.3 Exceptions, die in der Standardlibrary ausgelöst werden

Die oben genannten Exceptions kann man aber auch in eigenen Programmen auslösen. Jedoch sollte hier stets der Kontext der Exception beachtet werden: eine `out_of_range`-Exception sollte man nur dann auslösen, wenn auch wirklich eine Bereichsüberschreitung vorliegt.

Die nächsten exemplarischen Exceptions liegen im namespace `std` und benötigen die Header-Datei `stdexcept`:

- Die `invalid_argument`-Exception zeigt ein falsches Argument an. Diese Exception wird z. B. von der Klasse `bitset` verwendet, wenn z. B. ein String in ein bitset umgewandelt werden soll, aber innerhalb des Strings andere Zeichen als 0 und 1 liegen.

- Die `length_error`-Exception weist darauf hin, dass versucht wird, ein Objekt zu erzeugen, das größer ist als seine maximal erlaubt Größe. Dieser Fall tritt z. B. innerhalb der Standard-Bibliothek dann auf, wenn ein String nach einer Operation mehr als `max_size()` Zeichen enthalten würde. (siehe auch Operationen für Strings, Abschnitt 6.2)

- Da Streams beim Misslingen von Operationen Exceptions auslösen können, gibt es die Exception `ios_base::failure`. Die Auslösung von Exceptions durch Streams muss explizit durch Aufruf der Stream-Memberfunktion `exceptions(..)` freigegeben werden. Als Parameter erhält `exceptions(..)` eines oder mehrere der Flags `eofbit`, `failbit`, `badbit`.

- `ios_base::failure` ist zwar in der Header-Datei `ios` definiert, dieser Header muss jedoch nicht extra eingebunden werden, da dies automatisch durch den entsprechenden Stream-Header (z.B. `fstream`) erfolgt! (siehe Status-Flags)

14.1.1.4 Zusätzliche Laufzeitfehler

Während des Programmablaufs können Ausnahmen auftreten, die unvorhergesehen sind und nicht innerhalb von Standardroutinen verwendet wurden. Diese Exceptions nennt man „run-time-error-Exceptions“.

Zur Verfügung stehen hier: `range_error`, `overflow_error`, `underflow_error`. Alle liegen im Namespace `std` und benötigen die Headerfile `stdexcept`.
Will man eine dieser Standardexceptions auslösen, muss man dem Konstruktor der Exception eine Referenz auf einen C-String übergeben.

Beispiel 14.3: Beispiel für den Einsatz von Exceptions beim Verarbeiten einer Datei

```
std::ifstream InFile;

// Alten Exception-Status retten:
std::ios_base::iostate eOld = InFile.exceptions();

// Exceptions aktvieren:
InFile.exceptions(std::ios::failbit | std::ios::eofbit);

// Nun Datei verarbeiten
try {
    InFile.open(pszFILE); // Datei öffnen
    for (;;) { // Endlosschleife
        InFile.getline(acLine, nSIZE); //zeilenweise einlesen und ausgeben
        cout << acLine << endl;
    }

} catch (std::ios_base::failure& e) //Exceptions fangen
{
    // ...
}
```

Beispiel 14.4: Vollständiges Beispiel mit Exceptions

```
// file: Exception/bigException.cpp
// description:

#include <iostream>
#include <fstream>
#include <locale>
#include <cerrno>
#include <cstdlib>
#include <cstring>
#include <string>

using namespace std;

int small_exception();
int read_from_file(wifstream &wfile);
int read_from_file_with_exception(wifstream &wfile);

int main() {
    wstring line;
    locale mylocale("de_DE.utf-8");
    locale::global(mylocale);
    wifstream wfile("in.txt");
    wfile.imbue(mylocale);
    int result;
    bool test1 = false;
    bool test2 = false;
    bool test3 = true;


    if (test1) {
        result = small_exception();
        wcout << "Small Exception: " << result << endl;
        exit(0);
    }

    wcout << "We set File 'in.txt' to utf8 " << endl;
    if (wfile.fail()) {
        wcout << "ERROR: open wfile " << endl;
        exit(0);
    }

    if (test2) {
        result = read_from_file(wfile);
        wcout << "read_from_file: " << result << endl;
        exit(0);
    }

    if (test3) {
```

```
            result = read_from_file_with_exception(wfile);
            wcout << "read_from_file_with_exception: " <<
                    result << endl;
            exit(0);
        }

}

int read_from_file_with_exception(wifstream &wfile) {
    //  wfile.exceptions(ios::eofbit|ios::failbit|ios::badbit);
    //Definiert Standard Exceptions.

    class Lesefehler : public exception {
    };
    wstring line;
    ios::iostate status;

    do {
        try {
            getline(wfile, line);
            if (wfile.fail()) throw Lesefehler();
            if (wfile.eof()) throw Lesefehler();
            if (wfile.bad()) throw Lesefehler();
            wcout << "--->IN>>" << line << endl;
        } catch (const Lesefehler& Error) {
            wcout << L"Lesefehler Error" << endl;
            wcout << "error.what(): " << Error.what() << endl;
            wcout << "strerror:" << errno << "=" << strerror(errno) << endl;
            if (wfile.fail()) {
                wcout << "Fail situation, we try to clear it" << endl;
                wfile.clear();
                wcout << "test eof Bit" << endl;
                if (wfile.eof()) {
                    wcout << "Still eof Bit is set" << endl;
                } else {
                    wcout << "eof bit cleared, skip 1 byte" << endl;
                    wfile.clear();
                    wfile.seekg(3, ios::cur);
                    wfile.clear();
                }
            }
        } catch (const exception& Error) {
            wcout << "Standard Exception" << endl;
            wcout << "error.what(): " << Error.what() << endl;
        } catch (const char *Code) {
            wcout << "Default Exception CODE" << endl;
        } catch (...) {
            wcout << "Default Exception" << endl;
        };

        wcout << "--->IN: #" << line << "#" << endl;
```

```
        wcout << "eof() bit " << wfile.eof() << endl;
        wcout << "fail() bit " << wfile.fail() << endl;
        wcout << "bad() bit " << wfile.bad() << endl;
        wcout << "strerror: " << strerror(errno) << endl;

        status = wfile.rdstate();
        wcout << L" Status des Lesens: " << status << endl;

        /*     if (!status)
{
  wfile.setstate(ios::goodbit);
  wcout << "IN: #" << line << "#" << endl;
}
         */
    } while (!wfile.eof());

    return status;
}

int small_exception() {

    class Fehler : public exception {
    };

    try {
        throw Fehler();
        throw 666; // int, wird abgefangen
        // wird nicht mehr ausgeführt:
        throw L"stone";
        wcout << L"kein Fehler aufgetreten!\n";
    } catch (int i) {
        wcout << L"int-Fehler " << i << " abgefangen\n";
    } catch (wstring str) {
        wcout << L"stringfehler " << str << " abgefangen\n";
    } catch (Fehler &f) {
        wcout << L"Fehler, eine Klasse" << endl;
        wcout << "was ist los : " << f.what() << endl;
    }
    return EXIT_SUCCESS;
}

int read_from_file(wifstream &wfile) {
    wstring line;
    while (getline(wfile, line)) {
        wcout << "--->IN>>" << line << endl;
    };
    wcout << "Error strerror :=" << strerror(errno) << endl;
    return EXIT_SUCCESS;
}
```

15 Spezialthemen

15.1 Vorsicht bei `using` Direktive: Namespaces

Sehr gefährlich ist der Gebrauch der Direktive

```
using XYZ;
```

Wird ein `using namespace` verwendet, dann gilt diese Einstellung während des gesamten restlichen Programms. Wird mit der Präprozessordirektive `#include` ein File eingebunden, in dem auch eine `using namespace` Direktive verwendet wird, dann gilt die `using` Direktive in der eingebundenen Datei und anschließend auch im restlichen Programm, also z.B. immer `std::`.
Das ist sehr gefährlich, deshalb wird empfohlen, *immer* `std::` vor die entsprechenden Aufrufe zu stellen.

15.2 Flache und Tiefe Member einer Klasse

Beispiel 15.1: Flache Daten

```
private:
    {
        char d;
        int anzahl;
    }
```

Beispiel 15.2: Statische Daten, die auch als flach gesehen werden

```
private:
    {
        char data[1024];
        int anzahl;
    }
```

Beispiel 15.3: Tiefe Daten in einer Klasse, realisierbar nur mit Pointern

```
    {
        char *data;
        int anzahl;
    }
```

Mit der Deklaration eines Pointers steht noch kein Speicher für die Elemente des Members zur Verfügung. Es muss dann dementsprechend der Konstruktor neu definiert werden:

Beispiel 15.4: Deep-Copy-Constructor

```
class c_s {
public:
    // Konstruktoren
    c_s::c_s(int size) {
        data = new char[size];  // ... alle Buchstaben der c_s Daten
        anzahl = 0;
    }
}
```

Im Headerfile kann auch die Defaultgröße festgelegt werden.
Im `.hpp` File, (nicht im Implementationsfile!)

```
c_s(int size=MAX_SIZE);    // ... legt ein c_s-Objekt an
```

15.3 Speicher Allocation (Beispiele allocation)

Wenn kein Speicher alloziiert wird, dann stürzt das Programm ab:
Fehler: `g++ -DERROR c_s_main.cpp`
Korrekt: `g++ c_s_main.cpp`

⇒ Es muss also Speicher alloziiert werden.

Zuweisung und Initialisierung:

```
c_s::c_s(int size) {
    data = new char[size]; // ... alle Buchstaben der c_s Daten
    anzahl = 0;
}
```

15.4 Destruktor (Beispiele in destruct)

Wenn die Lebensdauer einer temporären Variable erlischt, wird sie automatisch gelöscht. Zum Löschen wird der Defaultdestruktor aufgerufen, der nur flache Memberdaten löscht. Tiefe Memberdaten werden als „Speicherleaks“ behalten.

Vorführung ohne Destruktor, bei der sich beobachten lässt, wie der Speicher stetig wächst:

```
[max] g++ c_s_main.cpp
ps -uax | grep max | grep a.out
```

immer 50 MB mehr...
⇒ Destruktor einführen:

```
c_s.hpp File:
  // Destruktoren
  ~c_s();  // Destruktor: gleicher Name, mit Tilde, ohne Argument

c_s.cpp File:
c_s::~c_s() {
  std::cout << "Delete" << std::endl;
  delete [] data;   // eckige Klammern sind wichtig
  anzahl = 0;
}
```

Vorführung mit Destruktor, wobei der Speicher gleich bleibt

```
[max] g++ -DDESTR c_s_main.cpp
ps -uax | grep max | grep a.out
```

Beim Programmieren einer Klasse muss man sich zum Konstruktor unbedingt auch einen Destruktor definieren!

15.4.1 Mit Zuweisungsoperator:

```
s2 = "Hans";
```

Definition eines Zuweisungsoperators:

```
c_s &c_s::operator=(std::string const &str) {
    this->set_data(str);
    return *this;
}
```

Achtung: Dazu mit `set_data()` auch Speicher besorgen!

15.4.2 Initialisierung der Variable (Verzeichnis `init_copy`)

Der `copy`-Konstruktor wird benötigt bei:

- Initialisierungen `c_s a=b;`

- Funktionaufrufen mit call-by-value Argumenten (ohne Referenz darf sie ja nicht auf dem Original arbeiten) `x(c_s a)`
- Returnwert bei call-by-value

Beispiel 15.5: Tiefer Konstruktor

```
c_s s4 = s1;
```

Terminiert:

```
[max] g++ -DERR1 c_s_main.cpp
```

```
c_s s1("Huelle");

#ifdef COPY
c_s s4=s1;
#endif
c_s s2;
s2="Hans";
```

Terminiert nicht:

```
[max] g++ -DERR2 c_s_main.cpp
```

```
c_s s1("Huelle");
c_s s2;
 s2="Hans";
#ifdef COPY
c_s s4=s1;
#endif
```

Vorführung mit dem `gdb`, `skip` beim `return 0` Statement, er stürzt bei der Freigabe der Variablen ab.

15.4.3 Initialisierung: Copykonstruktor (Verzeichnis `init_copy`)

Bei Initialisierungen wird eine flache Kopie angelegt, das muss verhindert werden. Es gibt in C++ einen speziellen Konstruktor, der bei Initialisierung aufgerufen wird: den Copykonstruktor

```
c_s::c_s(c_s const& Str)        // Default Copykonstruktor,
                                 // braucht Namen der Klasse und
                                 // const-Referenz auf in Objekt
{
   data = new char[Str.anzahl]; // ... alle Buchstaben der c_s Date
   strcpy(data,Str.data);
```

Wegen des Fehlens der tiefen Kopie zeigen zwei Variablen auf denselben Speicher.
Korrektur:

```
g++ -DCOPY c_s_main.cpp
```

15.4.4 Argumentübergabe als Wert (Verzeichnis routine_copy)

Wegen des Fehlens der tiefen Kopie:

```
[ss2009/3][matrix][max] g++ -DROUTINE c_s_main.cpp
```

Korrektur:

```
g++ -DCOPY c_s_main.cpp
```

15.4.5 Wertrückgabe bei Routinen (Verzeichnis routine_copy)

Wegen des Fehlens der tiefen Kopie:

```
[ss2009/3][matrix][max] g++ -DROUTINE c_s_main.cpp
```

Korrektur:

```
g++ -DCOPY c_s_main.cpp
```

15.5 Berechnung Konkordanz mit Routinen der STL

Eine Anwendung aus der Computerlinguistik, die Erzeugung einer Konkordanz, kann sehr gut mit Hilfe einer deque implementiert werden.

Beispiel 15.6: Konkordanz mit deque

```
// file: STL/dequeConc.cpp
// description: Concordance mit deque

#include <fstream>
#include <iostream>
#include <deque>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

typedef deque<string> MyDeque;
typedef vector<string> MyVec;

const int DequeSize = 5;

//////////// FUNCTIONS:
int fillDeque(MyDeque &wordDeque, string &word);
int examineDeque(MyDeque &wordDeque, string &searchWord);
bool alnum(char c);
bool notAlnum(char c);
```

```
int split(MyVec &words, string &line);
int printFormattedDeque(MyDeque &wordDeque);

/////////////// MAIN

int main() {

    int rline, lnum;
    string filename = "eingabe.txt";
    fstream fs;
    string word, line, nextWord, searchWord;
    int i, numOfWords;

    MyDeque wordDeque(DequeSize);
    MyVec words;

    //  End of declarations ...

    cout << " Hello concordance with deque " << endl;
    cout << " Input from Textfile=" << filename << endl;

    //  Open the file for reading
    cerr << " Open File " << filename << endl;
    fs.open(filename.c_str(), ios::in);
    if (!fs) {
        cerr << " Fileopen Error on " << filename << endl;
        exit(-1);
    }

    cout << " Enter Concondance word " << endl;
    cin >> searchWord;

    //  Read each line from the file, split it, and insert it
    do {
        if (getline(fs, line)) {
            split(words, line);
            for (i = 0; i < words.size(); i++) {
                fillDeque(wordDeque, words[i]);
                examineDeque(wordDeque, searchWord);
            }
        }
        words.clear();
    } while (!fs.eof());
    return ( EXIT_SUCCESS);
}


//////////////////////////////////////////

int fillDeque(MyDeque &wordDeque, string &nextWord) {
    // append to deque
```

```
    wordDeque.push_back(nextWord);
    // Delete first Element from deque
    wordDeque.erase(wordDeque.begin());
    return 1;
}

////////////////////////////////////

int examineDeque(MyDeque &wordDeque, string &searchWord) {
    int i = 0;
    char numOfChars = 0;

    if (searchWord == wordDeque[DequeSize / 2]) {
        /*      for (i=0; i < DequeSize;++i)
         {
                 cout  << wordDeque[i] << " " ;
         }
        cout << endl;
         */
        printFormattedDeque(wordDeque);
    }
    return 1;
}
////////////////////////////////////

int split(MyVec &result, string &line) {
    string::iterator from, to;
    string word;

    from = line.begin();
    while (from != line.end()) {
        from = find_if(from, line.end(), alnum);
        to = find_if(from, line.end(), notAlnum);

        if (from != line.end()) {
            result.push_back(string(from, to));
        }
        from = to;
    }
    return 1;
}

////////////////////////////////////

bool notAlnum(char c) {
    return !isalnum(c);
}

////////////////////////////////////

bool alnum(char c) {
```

```
    return isalnum(c);
}

////////////////////////////////////

int printFormattedDeque(MyDeque &wordDeque) {
    int i, numOfChars = 0;
    int numOfColumns = 50;

    cout << "|";
    for (i = 0; i < DequeSize / 2; i++) {
        if (wordDeque[i] != "") {
            cout << wordDeque[i] << " ";
            numOfChars += wordDeque[i].size();
        } else {
            cout << "-";
        }
    }
    cout << " ++" << wordDeque[DequeSize / 2] << "++  ";
    numOfChars += wordDeque[DequeSize / 2].size();
    for (i = DequeSize / 2 + 1; i < DequeSize; ++i) {
        cout << wordDeque[i] << " ";
        numOfChars += wordDeque[i].size();
    }
    while (numOfChars < numOfColumns) {
        cout << "-";
        ++numOfChars;
    }

    cout << "|" << endl;
}
```

15.6 Hashes und Internationalisierung

Selbstverständlich können die Hash-Erweiterungen auch mit `wstring` Datentypen realisiert werden. Das einzige Problem besteht darin, dass die interne Hashfunktion nur mit einem nullterminierten Bytebuffer (C-String) aufgerufen werden darf. Verwendet man die eingebaute `c_str()`-Funktion um einen `wstring` in einen C-Buffer zu konvertieren, werden die einzelnen `wchar_t`-Buchstaben in einen `wchar_t`-Buffer gespeichert, nicht aber in den notwendigen `char_t`-Buffer. Eine Lösung besteht darin, sich eine eigene HASH-Funktion zu schreiben, die einen `wstring` verwendet, oder den `wstring` in einen nullterminierten Bytebuffer zu konvertieren, mit dem man dann die eingebaute Hashfunktion aufruft.
Im nachfolgenden Beispiel sehen Sie den Header einer Funktion zur Konvertierung eines wstrings in einen nullterminierten C-String. Die Implementation der Funktion sehen Sie nächsten Programmbeispiel.

Beispiel 15.7: Konvertierungsfunktion in C-String

```
int into_byte_buf(wstring ws,char *buf,int buf_len)
```

Die Vergleichsfunktion der wstring Elemente muss auf die wstring Datentypen eingestellt sein.

```
class HashWStringFunction
{
public:
    int operator()(const std::wstring &wstr) const
    {
        into_byte_buf(wstr,utf8bytes,BYTEBUFSIZE);
        return stdext::hash<const char*>() (utf8bytes);
    }
};

class HashWStringEqual
{
public:
    bool operator()(const std::wstring &s1,const std::wstring  &s2) const
    {
        return s1 == s2;
    }
};
```

Verwendet man eine eigene HASH-Funktion für wstrings, ist die Konvertierung in einen nullterminierten Bytebuffer überflüssig. Als mögliche HASH-Funktion bietet sich die sehr leistungsfähige HASH-Funktion von Jenkins an. Der typedef des `hash_map` templates muss dann auf diese Funktion verweisen.

Beispiel 15.8: HASH-Funktion von Jenkins

```
// Definition des  HashMap Templates
typedef hash_map<wstring, int, HashWStringFunction, HashWStringEqual> WhashMap;

// Hash struct for hashMap

struct jenkinsHash
{
  /**
   * Computes the hash value of a string.
   * The hash function is the one-at-a-time hash algorithm taken from wikipedia.
   *
   * @see http://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time
   * @param str the string for which the hash is computed.
   * @return the hash value of the string.
   */
  uint operator()(const std::wstring& str) const
  {
    uint h = 0;
    for(size_t i=0; i<str.length(); i++)
      {
    h += str[i];
    h += (h << 10);
    h ^= (h >> 6);
      }
    h += (h << 3);
    h ^= (h >> 11);
    h += (h << 15);
    return h;
  }
};

typedef std::set<freqPair, compFreqPair> freqSet;
typedef ::__gnu_cxx::hash_map<std::wstring, uint, jenkinsHash> hashMap;
```

Beispiel zur Berechnung eines Hashs mit wstring Elementen

Beispiel 15.9: Hash mit wstring

```
// file: STL/hash_mapUTF8.cpp
// description: Works with utf-8 Input

#include <iostream>
#include <fstream>
#include <string>
#include <locale>
#include <ext/hash_map>

using namespace std;

// namespace of hash-functions varies from one compiler and gcc version
// in gcc v3.1/v3.2 and later: namespace __gnu_cxx.
// see: http://gcc.gnu.org/onlinedocs/libstdc++/faq/index.html#5_4
// Here it is renamed to stdext.
// Namespace alias to catch hash_map classes

namespace stdext = ::__gnu_cxx;


// Auxiliary memory and Function for the Hash function:
// Task: Convert wstring to char buffer
// buffer:
// static buffer for conversion
#define BYTEBUFSIZE 1024
static char utf8bytes[BYTEBUFSIZE];

// function header:
int into_byte_buf(wstring ws, char *buf, int buf_len);

// For containers like hash_map the equal & hash classes must be defined!!!
// This class' function operator() generates a hash value for a key.
// Unique hash values (indices) give quickest access to data.

class HashWStringFunction {
public:

    int operator()(const wstring &WStr) const {
        into_byte_buf(WStr, utf8bytes, BYTEBUFSIZE);
        return stdext::hash<const char*>() (utf8bytes);
    }
};

// This class' function operator() tests if any two keys are equal.

class HashWStringEqual {
public:
```

```
    bool operator()(const wstring &S1,
            const wstring &S2) const {
        return S1 == S2;
    }
};



/////////////////////////////////
// Definition of the HashMap Template
// using namespace stdext   (Standard Extensions)
typedef stdext::hash_map<wstring, int, HashWStringFunction,
HashWStringEqual> WHashMap;

int main() {
    WHashMap wstring_hashmap;
    WHashMap::iterator pos;

    wstring line;
    // Construct locale object with the user's default preferences:
    locale mylocale("de_DE.utf-8");
    wcout.imbue(mylocale); //  wcout mit dem locale 'einfärben'
    locale::global(mylocale); // das Locale global machen

    wifstream infile("eingabe_utf8.txt");
    infile.imbue(mylocale);

    while (getline(infile, line)) {
        if (!wstring_hashmap[line]) {
            wcout << "new-->" << line << "<--" << endl;
            wstring_hashmap[line] = 1;
        } else {
            ++wstring_hashmap[line];
            wcout << "upd-->" << line << "<-- New count= ";
            wcout << wstring_hashmap[line] << endl;
        }
    }
    wcout << "File is read" << endl << endl << endl;
    wcout << "Content of the HASH " << endl;

    for (pos = wstring_hashmap.begin();
            pos != wstring_hashmap.end(); ++pos) {
        wcout << "#" << pos->first << "#  = " << pos->second << endl;
    }
    return 0;

}

// Converts a wstring into a zero terminated byte Buffer
```

```
int into_byte_buf(wstring ws, char *buf, int buf_len) {
    // Input:
    //     wstring ws ... wide string to convert
    //     buf_len    ... Number of Bytes of the output Buffer buf
    // Output:
    // char *buf      ... Pointer to a buffer with buf_len bytes,
    //                    to hold the result
    //
    int i, j;
    int index = 0;
    int buf_len_m1 = buf_len - 1;

    union wstr_byte {
        unsigned int zahl32;
        unsigned char buf8[4];
    } u;

    j = 0;
    while (j < ws.size() && index < buf_len_m1) {
        u.zahl32 = ws[j];
        i = 0;
        while (u.buf8[i] && index < buf_len_m1) {
            buf[index] = u.buf8[i];
            index++;
            i++;
        }
        j++;
    }
    buf[index] = 0;
}
```

15.7 Überladen von Operatoren, Zweiter Teil

15.7.1 Überladen von binären Operatoren

Binäre Operatorfunktionen werden mit einem Argument definiert. Innerhalb der selbst definierten Operatorfunktion wird der rechte Operand der Operation über das Argument übergeben. Der linke Operand, der den Operatoraufruf generiert, wird an die Operatorfunktion direkt übergeben.

Beispiel 15.10: Überladen von binären Operatoren

```
// file: Klassenprogrammierung/strType.hpp
// description:

#include <iostream>

#ifndef STRTYPE_HPP
#define STRTYPE_HPP

class StrType {
public:
    // Konstruktoren:
    StrType(); // StrType s;
    StrType(const std::string); // StrType s1("string");
    StrType(const StrType &); // StrType s2(s1);

    // Destruktor:

    ~StrType() {
        anzChar = 0;
    }

    // Zuweisungsoperatoren:
    StrType operator=(const StrType); // s = s1;
    StrType operator=(const std::string); // s = "string";
    // Druckt die Anzahl der Buchstaben aus:
    int getAnzChars();

    friend std::ostream & operator<<(std::ostream&, const StrType&);


    // Verknüpfungsoperator:
    StrType operator+(const StrType); // s1 + s2;
    StrType operator+(const std::string); // s1 + "string";
    // "string" + s1;
    friend StrType operator+(const std::string, const StrType);

private:
```

```
    std::string text;
    int anzChar;

};

StrType::StrType() {
    anzChar = 0;
    text = "";
}

// Konstruktor für Deklaration : StrType s1("string")

StrType::StrType(const std::string Str) {
    anzChar = Str.size();
    text = Str;
}

// Copy-Konstruktor für Deklaration : StrType s2(s1)

StrType::StrType(const StrType & Obj) {
    anzChar = Obj.anzChar; // Laenge von obj
    text = Obj.text; // kopiere obj.text des alten Objekts
}


// Ausgabeoperator für StrType:

std::ostream & operator<<(std::ostream &stream, const StrType &Obj) {
    stream << Obj.text;
    /* schicke text an den Stream
     * (obj.text kann er verarbeiten
     * im Gegensatz zum ganzen Objekt)
     */
    return stream; // Stream an  aufrufende Funktion zurück
}

// Zuweisung: s = s1;

StrType StrType::operator=(StrType Str) {
    text = Str.text;
    anzChar = text.size();
    return *this; // Objekt, an das zugewiesen wurde, zurück
}

// Zuweisung: s = "string";

StrType StrType::operator=(const std::string Str) {
    text = Str; // kopiere die Zeichenkette
    anzChar = text.size();
    return *this; // Objekt, an das zugewiesen wurde, zurück
}
```

```
// Verknüpfung: s1 + s2

StrType StrType::operator+(const StrType Str) {
    StrType tmp;

    tmp = text + Str.text;
    tmp.anzChar = tmp.text.size();
    return tmp; // gib verknüpftes Objekt zurück
}

// Verknüpfung: s1 + "string"

StrType StrType::operator+(const std::string Str) {
    StrType tmp;
    /* Temporäres Objekt
     * (+ ändert nicht die Werte seiner Operanden,
     * -> zusätzliches Objekt für Zuweisung nötig)
     */
    tmp.anzChar = Str.size() + anzChar;
    tmp.text = Str + text;

    return tmp; // gibt verknuepftes Objekt zurück
}
// Verknüpfung: "string" + s1

StrType operator+(const std::string Str, const StrType Obj) {
    StrType tmp;
    /* Temporäres Objekt
     * (+ ändert nicht die Werte seiner Operanden,
     * -> zusätzliches Objekt für Zuweisung nötig)
     */

    tmp.text = Str + Obj.text; //  text an 2. Operanden
    tmp.anzChar = tmp.text.size();

    return tmp; // gib verknüpftes Objekt zurück
}
//Druckt die Anzahl der Buchstaben aus:

int StrType::getAnzChars() {
    return anzChar;
}

#endif

// file: Klassenprogrammierung/mainStrType.cpp
// description:

#include "strType.hpp"
#include <iostream>
```

```
using namespace std;

int main() {
    StrType s1("Franz Hans Max"), s2(s1), s3, s4, s5, s6, s7;

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    s3 = s1 + " " + s2;
    cout << "s3 = s1 + \" \" + s2: " << s3 << endl;
    s4 = s2;
    cout << "s4 = s2: " << s4 << endl;
    s5 = "Sepp";
    cout << "s5 :" << s5 << endl;

    s6 = "Das sind die Namen: " + s2;
    cout << "s6 :" << s6 << endl;
    cout << "Chars in s6 = " << s6.getAnzChars() << endl;

    return 0;
}
```

15.7.2 Die friend-Operatorfunktion

Da der linke Operand die Auswahl der Operatorfunktion trifft, ist bei den bisherigen Möglichkeiten der Operandendefinition folgender Ausdruck nicht möglich:

Beispiel 15.11: `friend`-Operatoren

```
(1)     elem1 = 10 + elem2; // bisher nicht möglich
(2)     elem1 = elem2 + 10; // bisher möglich
```

Es müssten der Operandenfunktion beide Operanden übergeben werden. Bei Memberoperatorfunktionen wird aber nur der rechte Operand als Argument übergeben, der linke entpricht dem `this()`-Zeiger.

Abhilfe bieten die `friend`-Operatorfunktionen: Wird eine Operatorfunktion als `friend` definiert, dann werden die beiden Operanden als Argumente übergeben und können einzeln bearbeitet werden. Mit dieser Technik können nun für den Ausdruck (1) und (2) eigene `friend`-Operatorfunktion definiert werden.

Bei unären `friend`-Operatorfunktionen müssen die Argumente als Referenzen übergeben werden, da bei friend-Funktionen kein Zugriff auf den `this()`-Pointer möglich ist. Post- und Präfixoperatoren können bei `friend`-Operatorfunktionen auch definiert und unterschieden werden.

Beispiel 15.12: Programmausschnitt `friend`-Operatoren

```
////////////////////////////////////////////////
//aus der Deklarationsdatei:
friend StrType operator+(StrType, int);
friend StrType operator+(int, StrType);


////////////////////////////////////////////////
//aus dem Implementationsteil:
// Addiere Zahl auf ASCII Wert der Buchstaben

StrType operator+(const StrType Str, int add) {
    StrType temp = Str;
    int i;
    temp.anz_char = Str.anz_char;
    for (i = 0; i < Str.anz_char; i++)
        temp.text[i] = Str.text[i] + add;
    return temp;
}

// Addiere auf Zahl der ASCII Wert der Buchstaben

StrType operator+(int add, const StrType Str) {
    StrType temp = Str;
    int i;
    temp.anz_char = Str.anz_char;
    for (i = 0; i < Str.anz_char; i++)
        temp.text[i] = Str.text[i] + add;
    return temp;
}

/////////////////////////////////////////////
// aus dem Hauptprogramm:
sn = 3 + s1;
sn = sn - 3;
```

Gefahren bei der Rückgabe von Referenzen

In C++ werden standardmäßig alle Argumente von Funktionen kopiert und beim Verlassen der Funktion mit der Destruktorfunktion gelöscht. Das hätte bei Funktionen mit Referenzen als Rückgabewert, wie z.B. bei Zuweisungsoperationen, zur Folge, dass die Operanden gelöscht werden. Eine Abhilfe bietet die Übergabe von Objekten als Referenzen, da hier keine Kopie angelegt wird und somit auch kein Destruktor aufgerufen wird.

Problematisch wird es aber bei der Rückgabe einer Referenz, die auf eine temporäre Variable innerhalb der Funktion zeigt. Nach dem Verlassen der Routine wird temporärer Speicher automatisch gelöscht, die Referenz zeigt dann auf Speicher, der nicht mehr exis-

tiert. Eine Abhilfe bietet die Speicherklasse `static`.

Jede temporäre Variable der Speicherklasse `static` lebt während der ganzen Lebensdauer des Moduls:

Beispiel 15.13: Verknüpfungsoperator

```
// Verknuepfungsoperator:
StrType & operator+(const StrType &); // s1 + s2;

// Verknüpfung: s1 + s2

StrType &StrType::operator+(const StrType &str) {
    static StrType tmp;

    tmp = str.text + text;
    tmp.anz_char = tmp.text.size();
    return tmp; // gib verknüpftes Objekt zurück
}
```

15.8 Überladen des Ausgabeoperators

In C++ kann für eine selbstdefinierte Klasse der Ausgabeoperator überladen werden. So können selbstdefinierte Objekte mit dem `<<`-Operator ausgegeben werden:

Beispiel 15.14: Selbstdefinierte Klassen ausgeben

```
Mytype obj;

cout << obj << endl;
```

Die Ausgabeoperation wird in C++ als Einfügung (insertion) bezeichnet, `<<` wird Einfügeoperator genannt. Bei der Ausgabe eigener Objekte wird innerhalb der Klassendefinition eine sogenannte Einfügefunktion gesucht, der als erstes Element der linke Operand und als zweites Element der rechte Operand des Ausgabeoperators übergeben wird.

Damit die Operandenfunktion auf beide Elemente über Parameter Zugriff hat, muss die Ausgabeoperationsfunktion als `friend`-Operatorfunktion definiert werden:

```
friend ostream &operator<<(ostream &stream, Mytype &ob);
```

Eine Einfügefunktion kann kein Mitglied der Klasse sein, auf die sie sich beziehen soll, da der linke Operand ein Stream, und daher kein Objekt der Klasse ist.

Beispiel 15.15: Allgemeine Form einer Ausgabefunktion

```
ostream & operator <<(ostream &stream, ClassName ob) {
    // Definition der Einfügefunktion
    return stream;
}
```

Innerhalb einer Einfügefunktion kann jede beliebige Prozedur ausgeführt werden, man sollte sich aber auf Ausgabeoperationen beschränken. Einfügefunktionen sollten so allgemein wie möglich definiert werden.

15.9 Überladen des Eingabeoperators

Der >>-Eingabeoperator wird Extraktionsoperator genannt und kann ebenfalls überladen werden. Eine Funktion, die ihn überlädt, wird als Extraktionsfunktion (extractor) bezeichnet.

Beispiel 15.16: Allgemeine Form einer Extraktionsfunktion

```
istream & operator >>(istream &stream, ClassName &ob) {
    // Definition der Extraktionsfunktion
    return stream;
}
```

Eine Extraktionsfunktion kann, ebenso wie eine Einfügefunktion, keine Mitgliedsfunktion sein.

Beispiel 15.17: Eingabe und Ausgabeoperator

```
// Definition:
// Ausgabeoperator für die Klasse StrType:
friend ostream & operator<<(ostream&, const StrType&);
// Eingabeoperator für die Klasse StrType:
friend istream & operator>>(istream&, StrType&);
// ------------------------------------------------------------------------

//Implementationsteil:
// Ausgabeoperator für StrType:

ostream & operator<<(ostream &stream, const StrType &Obj) {
    stream << Obj.text;
    /* schicke text an den Stream
       (Obj.text kann er im Gegensatz zum ganzen Objekt verarbeiten */
    return stream; // gib den Stream an die aufrufende Funktion zurück
}

// Eingabeoperator für StrType
```

```
istream & operator>>(istream &stream, StrType &Obj) {
    stream >> Obj.text;
    /* schicke Eingabe des stream zu text
       (Obj.text kann er im Gegensatz zum ganzen Objekt verarbeiten) */
    Obj.anz_char = Obj.text.size();
    return stream; // gib den Stream an die aufrufende Funktion zurück
}

// -----------------------------------------------------------------------
// Aufruf im main Progamm:

cout << "Bitte String eingeben>>";
cin >> s7;
cout << "Das war die Eingabe : " << s7 << endl;
cout << "Die Anzahl der Buchstaben ist: " << s7.get_anz_chars() << endl;
```

15.10 Liste der überladbaren Operatoren

Operator	Meaning
+	Addition, unary plus
&	Address of
[]	Array subscript
&=	Assign bitwise and
≙	Assign bitwise exclusive or
|=	Assign bitwise or
-=	Assign difference
«=	Assign left shift
=	Assignment
*=	Assign product
/=	Assign quotient
%=	Assign remainder
»=	Assign right shift
+=	Assign sum
&	Bitwise and
|	Bitwise or
–	Decrement
/	Division
==	Equal
( )	Function call
>	Greater than
>=	Greater than or equal
++	Increment
«	Left shift
<	Less than
<=	Less than or equal
&&	Logical and
!	Logical complement
||	Logical or
->*	Member reference
->	Member reference
*	Multiplication, dereference
!=	Not equal
%	Remainder
»	Right shift
,	Serial evaluation
-	Subtraction, negation

Tabelle 15.1: Überladbare Operatoren

15.11 Mehrere locales mit boost

Soll die boost Bibliothek mit mehreren locales gleichzeitig arbeiten können, dann müssen die regulären Ausdrücke mit diesen locales eingefärbt werden. Damit boost die Methode des Einfärbens beherrscht, muss die boost Bibliothek beim Kompilieren entsprechend konfiguriert worden sein: siehe: http://lists.boost.org/Archives/boost/2003/12/57365.php `define BOOST_REGEX_USE_CPP_LOCALE` in `boost/regex/user.hpp`, and rebuild everything. `cpp_regex_traits` is now the default traits class, so:

```
boost::regex r1;
boost::regex r2;
r1.imbue(std::locale("en_GB"));
r1.assign("\\w+"); // UK English
r1.imbue(std::locale("fr_FR"));
r1.assign("\\w+"); // French
```

15.12 Spezialthemen und Beispiele

15.12.1 Konkordanzprogramm mit Klassen und Vektoren

Als weiteres Beispiel für Vererbung sei hier ein Indexprogramm mit Konkordanzausgabe angeführt.
Dazu kann für einen beliebigen Text (*wide-string*) oder für eine Textdatei ein sogenannter Index erzeugt werden. Dazu wird der Text bzw. die Textdatei anhand von *Whitespaces* getrennt; so wird der Text bzw. die Textdatei in Wörter zerlegt. Mittels der *find()* Funktion wird das zu suchende Wort vorher festgelegt. Sobald das Programm dieses Wort findet, wird die Position des Wortes im Text ausgegeben, sowie eine festgelegte Anzahl von Wörtern, die vor bzw. nach dem gefundenen Wort stehen (*Konkordanz*).

Beispiel 15.18: Die `main()` Klasse

```
// file: Vererbung/mainIndex.cpp
// description:

#include <iostream>
#include "index.hpp"
#include "text.hpp"

using namespace std;

int main() {

    setlocale(LC_ALL, "");

    Index myIndex;

    // Text aus Datei
    Text a = Text("test.txt");
    // Text als String uebergeben
    Text b = Text("String als Text", L"Hallo, hier ein Test.");
    Text c = Text("String als Text 2", L"Ist das denn noetig?");

    // Texte zum Index hinzufuegen
    myIndex.add(a);
    myIndex.add(b);
    myIndex.add(c);

    // Woerter im Index mit Konkordanz
    myIndex.find(L"ein");
    myIndex.find(L"der");
    myIndex.find(L"noetig?");
    myIndex.find(L"###");

    // Geschachtelt.
    myIndex.add(Text("Doch im Index", L"### und noch ein paar ###"));
    myIndex.find(L"###");
}
```

Beispiel 15.19: Die Index- Klasse

```
// file: Vererbung/index.hpp
// description:

#ifndef INDEX_HPP
#define INDEX_HPP
#include "text.hpp"

using namespace std;

class Index {
public:
    Index();
    void add(Text);
    bool find(std::wstring);
private:
    std::vector<Text> texts;
};

Index::Index() {}

/*
Text zum Index hinzufuegen.
*/
void Index::add(Text t) {
    texts.push_back(t);
}

/*
Token in Text finden.
*/
bool Index::find(wstring wortString) {
    vector<Text>::iterator i;

    bool nicht_im_index = true;

    for (i = texts.begin(); i != texts.end(); i++) {
        if (i ->find(wortString) ) {
            wcout << endl << L"Gefunden in Text: " << i->getName() << endl;
            i->findAndPrint(wortString);
            nicht_im_index = false;
        }
    }

    if ( nicht_im_index) {
        wcout << endl << wortString << L": Nicht im Index!" << endl << endl;
    }
}
#endif /* INDEX_HPP */
```

Beispiel 15.20: Die Text- Klasse

```
// file: Vererbung/text.hpp
// description:

#ifndef TEXT_HPP
#define TEXT_HPP

#include <vector>
#include <map>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include "wort.hpp"

class Text {
public:
    Text();
    Text(std::string);
    Text(std::string, const std::wstring);
    void printOnTerminal();
    std::vector<Wort> getText();
    std::wstring getName();
    Text & operator=(Text);
    bool find(std::wstring);
    void findAndPrint(std::wstring);

private:
    std::wstring name;
    std::vector<Wort> text;
    std::map< std::wstring, std::vector<int> > wordPositions;
    void readWordsfromFile( std::wifstream &);
    void readWordsFromString( std::wstring &);
    void readWordsFromStream( std::wistream &);
    void buildPositionsMap(std::vector<Wort> &);
    void buildPositionsMapAlternativeSyntax(std::vector<Wort> &);
    void printConcordance(int, int);
    std::wstring string2wstring(std::string);

};

// Constructor
Text::Text() {
}

/*
Text aus einer Datei erstellen.
*/
Text::Text(std::string filename) {
```

```
    this -> name = string2wstring(filename);
    std::wifstream file(filename.c_str());
    readWordsfromFile(file) ;
}

/*
Text aus einem String erstellen. Name des Textes ist das erste Argument.
*/
Text::Text(std::string name, std::wstring textAsString) {
    this -> name = string2wstring(name);
    readWordsFromString(textAsString);

}

/*
Wörter aus eine FileStream lesen.
*/

void Text::readWordsfromFile(std::wifstream & file) {
    // Zeile auf dem Mac auskommentieren.
    file.imbue(std::locale("de_DE.UTF-8"));
    if (file.good()) {
        readWordsFromStream(file);
    }
    else {
        std::wcout << "Problem mit der Datei" << std::endl;
    }
}

/*
Wörter aus einem StringStream lesen.
*/

void Text::readWordsFromString(std::wstring & textString) {
    std::wistringstream stringAsStream(textString);
    readWordsFromStream(stringAsStream);
}

/*
Wörter von einem Stream lesen
*/

void Text::readWordsFromStream(std::wistream & inStream) {
    std::wstring wortString;
    while (inStream >> wortString) {
        text.push_back( Wort(wortString) );
    }
    buildPositionsMap(text);
}

/*
```

```
Container Map mit Wortpositionen für Konkordanz erstellen.
*/
void Text::buildPositionsMap(std::vector<Wort> & words) {
    std::vector<Wort>::iterator i;

    for ( i= words.begin(); i != words.end(); i++ ) {
        std::wstring wortString = i -> asString();
        int iteratorPos = distance(words.begin(),i);

        // Es gibt schon einen Eintrag
        if (wordPositions.find( wortString ) != wordPositions.end()) {
            // Vector dereferenzieren
            wordPositions.at(wortString).push_back( iteratorPos );
        }
        // Es gibt noch keinen Eintrag
        else {
            std::vector<int> v;
            v.push_back(iteratorPos);

            wordPositions.insert( std::map< std::wstring,
            std::vector<int> >::value_type(wortString,v) );
        }
    }
}

/*
Wort innerhalb des Textes finden. und Position ausgeben.
*/

void Text::findAndPrint(std::wstring wortString) {
    if( wordPositions.find(wortString) != wordPositions.end()) {
        std::vector<int> v = wordPositions.at(wortString);
        std::vector<int>::iterator i;

        for (i = v.begin() ;i != v.end(); i++) {
            std::wcout << L"\t" << wortString <<": gefunden an Position -> "
            << * i << L": ";
            printConcordance(* i, 3);
        }

    }
    else {
       std::wcout << L"Wort: " << wortString << L" ist in Text: ";
        std::wcout << name;
        std::wcout << L" nicht enthalten."<< std::endl;
    }
}

bool Text::find(std::wstring wortString) {
    if( wordPositions.find(wortString) != wordPositions.end()) {
        return true;
```

```
    }
    else {
        return false;
    }
}

std::wstring Text::getName() {
    return this -> name;
}

/*
Konkordanz drucken.
*/
void Text::printConcordance(int textPosition, int windowSize) {
    int begin = textPosition - windowSize;
    int end = textPosition + windowSize;

    if (textPosition - windowSize < 0) {
        begin = 0 ;
    }

    if ( textPosition + windowSize >= text.size() ) {
        end = text.size() -1 ;
    }

    while (begin <= end) {
        if (begin != textPosition) {
            std::wcout << text.at(begin).asString() << " ";
        }
        else {
            std::wcout << L"\t>" << text.at(begin).asString() << L"<\t";
        }
        begin++;
    }
    std::wcout << std::endl;

}

Text & Text::operator=(Text other) {
    this -> text = other.text;
    this -> name = other.name;
    return * this;
}

/*
Ganzen Text auf Terminal ausgeben
*/
void Text::printOnTerminal() {

    std::wcout << L"Text: \"" << name << L"\"" << std::endl;
```

```
    std::vector<Wort>::iterator i = text.begin();
    while (i != text.end() ) {
        std::wcout << i -> asString() << L" ";
        i++;
    }

    std::wcout << std::endl << std::endl;
}

std::wstring Text::string2wstring(std::string s) {
    std::wstring sWide(s.length(), L' ');
    std::copy(s.begin(), s.end(), sWide.begin());
    return sWide;
}

#endif /* TEXT_HPP */
```

Beispiel 15.21: Die Wort- Klasse

```
// file: Vererbung/wort.hpp
// description:

#ifndef WORT_HPP
#define WORT_HPP

#include <string>

class Wort {
public:
    Wort(std::wstring);
    int laenge();
    const std::wstring asString();
private:
    std::wstring text;
};

Wort::Wort(std::wstring text) {
    this -> text = text;
}

int Wort::laenge() {
    return this -> text.length();
}

const std::wstring Wort::asString() {
    return this -> text ;
}

#endif /* WORT_HPP */
```

16 Neue Standards von C++

16.1 Einleitung

Am 11. Oktober 2011 hat die Standardisierungsorganisation *ISO/IEC* die erste überarbeitete Version von C++ seit 1998 veröffentlicht: C++11 (Codename: C++0x). Zu den wesentlichen Neuerungen gehören u.a. `Lambda`- Funktionen, eine verbesserte Standardbibliothek, eine direkte Datenfeld- Initialisierung und neue Typen für die Internationalisierung. Außerdem wurde die Regex Bibliothek Boost nun direkt in C++integriert.

Die aktuelle Fassung, C++14 (Codename: C++1y), wurde am 15. Dezember 2014 veröffentlicht. Sie dient hauptsächlich als Erweiterung von C++11 und beinhaltet einige kleine Verbesserungen und Bug Fixes.

Die nächste Version von C++ ist für 2017 geplant: C++17 (Codename: C++1z).

Im Folgenden werden einige Funktionen aus C++11 und C++14 erklärt und Beispiele zum besseren Verständnis gegeben.

16.2 Der Compiler-Aufruf

Um seine C++11-Programme fehlerfrei zu kompilieren, muss man für den GNU C++ Compiler einen bestimmten Flag setzen. Ab der GCC-Version 4.3 bis 4.6 kann dieser Flag verwendet werden:

```
g++ -std=c++0x main.cpp
```

Ab der GCC-Version 4.7 kann darüberhinaus noch folgendes Flag benutzt werden:

```
g++ -std=c++11 main.cpp
```

Ab der GCC-Version 4.8 kann auch C++14 mit folgendem Aufruf verwendet werden:

```
g++ -std=c++14 main.cpp
```

16.3 Vereinheitlichte Initialisierung

Im neuen C++11-Standard wird die Benutzung der geschweiften Klammern bei der Initialisierung vereinheitlicht und erweitert. Folgende Beispiele zeigen den Einsatz der neuen Syntax:

Beispiel 16.1

```
class Basis {

public:
    Basis(std::string name, int id) {
    std::cout << name << " " << id << std::endl;
    }
};

int main() {
    // Konstruktoraufrufe
    Basis base{"Hallo", 2};
    Basis base2 = {"Hello", 5};
    Basis * base3 = new Basis{"Hi", 10};

    // weitere Möglichkeiten
    int val{4};
    int val2 = {5}; // Das "=" ist optional

    string test("Hallo"); // "Hallo" ist hier ein Konstruktorargument
    string tes2 = { "Hello" }; // "Hello" ist hier eine Member-Initialisierung
    return {0}; // ;)
}
```

Des Weiteren ist nun eine komfortable Container-Zuweisung möglich:
Beispiel 16.2

```
// file: c11/container.cpp
// description:

#include <vector>
#include <map>
#include <string>
#include <iostream>

using namespace std;

int main() {
    vector<string> stringVector = { "Wert", "Neuer Wert", "Ganz neuer Wert"};
    vector<int> intVector = { 1, 2, 3, 5, 6 };
    map<int,string> listVector = { {1, "Max"}, {2, "Andi"}, {3, "Stefan"} };

    map<int,string>::iterator it;

    for(it = listVector.begin(); it != listVector.end(); ++it) {
        cout << it->first << " " << it->second << endl;
    }
}
```

Weitere Anwendungsfälle sind beispielsweise Return-Werte:
Beispiel 16.3

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

std::pair<std::string, int> func(std::string name, int val) {
        return { name, val*23};
    }

int main() {

    // Deklaration + Initialisierung
    std::pair<std::string, int> test{func("Hallo", 13)};

    // Und wie greife ich jetzt auf die beiden Rückgaben zu....
    std::cout << test.first << " " << test.second << std::endl; // So ;)

    return 0;
}
```

Hinweis: Natürlich funktionieren auch weiterhin die bisher bekannten Konstrukte!

16.4 Lambda- Funktionen

Im neuen C++11-Standard lassen sich auch anonyme Funktionen, sogenannte `Lambda-` Funktionen verwenden. Eine `Lambda-` Funktion ist in der Regel von der Form:

```
[]()->{}
```

Die eckige Klammern stellen die Bindung an die lokalen Variablen des Bereichs dar, die runden Klammern enthalten die (optionalen) Argumente der Funktion und die geschweiften Klammern geben den Funktionskörper an.

Eine erweiterte Variante wäre zum Beispiel von der Form:

```
auto isGreaterOrEqual = [](int left, int right){ return left >= right; };
```

Verwendbar u.a. so:

```
if(isGreaterOrEqual(5, 2)) {
    ...
}
```

16.5 Das auto()-Keyword

Verwendet man das `auto`-Keyword in Variablendefinitionen mit direkter Zuweisung, so muss man keine explizite Angabe des genauen Variablentyps vornehmen. Desweiteren kann man das Keyword `decltype` verwenden, um den Typ eines Ausdruckes zur Laufzeit zu erhalten. Folgende Beispiele sollen die Funktionsweisen von `auto` und `decltype` demonstrieren:

Beispiel 16.4

```
// file: c11/auto.hpp
// description:
using namespace std;

int main() {
    auto i = 0; // i ist ist vom Typ integer
    auto d = 1.337; // d ist vom Typ double
    auto c = 'c'; // c ist vom Typ char

    int irgendein_int;
    decltype(irgendein_int) andere_int = 5;
    // somit ist andere_int vom Typ integer
}
```

`auto` lässt sich ebenfalls in Verbindung mit Containern verwenden, um wesentlich kompakteren und kürzeren Code zu schreiben:
Beispiel 16.5

```
// file: c11/container2.hpp
// description:

#include <map>
#include <iostream>

using namespace std;

int main() {
    map<int,string> listVector = { {1, "Max"}, {2, "Andi"}, {3, "Stefan"} };

    for (auto it = listVector.begin(); it != listVector.end(); ++it) {
        cout << it->first << " " << it->second << endl;
    }
}
```

Im Vergleich einmal der Quelltext, den man ohne C++11 Unterstützung hätte verwenden müssen:
Beispiel 16.6

```
// file: c11/containerold.hpp
// description:

#include <vector>
#include <map>
#include <string>
#include <iostream>

using namespace std;

int main() {

    map<int,string> listVector;
    map<int,string>::iterator it;
    listVector.insert(map<int,string>::value_type(1, "Max"));
    listVector.insert(map<int,string>::value_type(2, "Andi"));
    listVector.insert(map<int,string>::value_type(3, "Stefan"));

    for(it = listVector.begin(); it != listVector.end(); ++it) {
        cout << it->first << " " << it->second << endl;
    }
}
```

`decltype` bietet sich desweiteren noch für Templates an, und berechnet so den Typ eines ganzen Ausdruckes:
Beispiel 16.7

```
#include <iostream>

template<typename A, typename B>
auto add(A a, B b) -> decltype(a+b) {
    return a+b;
}

int main() {

    // Testcases:
    std::cout << add(1,2) << std::endl;
    std::cout << add(1.2,2.8) << std::endl;
    std::cout << add(1, 3.6) << std::endl;
}
```

Seit C++14 ist es auch möglich `decltype` mit dem Rückgabetyp `auto` zu verbinden und Typen der Form `decltype(auto)` zu erzeugen.

Ab C++14 kann das `auto`-Keyword nicht mehr nur für lambda-Funktionen, sondern für alle Funktionen verwendet werden. Hierfür verwendet man dieses statt des Rückgabetyps der Funktion. Sollten mehrere return-Ausdrücke in einer Funktion implementiert werden, so müssen alle den selben Rückgabetyp haben. Auch rekursive Funktionen können mit dem `auto`-Keyword implementiert werden. Hierfür ist jedoch wichtig, dass der rekursive Aufruf erst nach mindestens einem return-Ausdruck in der Definition der Funktion passiert:

Beispiel 16.8

```
auto fakultaetRichtig(int i) {
    if (i==1) {
        return i;
    }
    else {
        return i * fakultaetRichtig(i-1);
    }
}

auto fakultaetFalsch(int i) {
    if (i!=1) {
        return i * fakultaetFalsch(i-1);
    }
    else {
        return i;
    }
}
```

16.6 Variablen Templates

In früheren Versionen von C++gab es bereits Templates für Funktionen oder Klassen. Seite C++14 ist es nun auch möglich Templates für Variablen zu erstellen. Als Beispiel gilt die Variable pi, die sowohl als Integer Typ (3), aber auch als mit entsprechender Anzahl von Nachkommastellen (3,14159...) als float, double oder sogar long double dargestellt werden kann.

16.7 `Range-For-` Schleifen

`Range-For` erlaubt das Programmieren von kürzeren Schleifen, ist leichter verständlich und beansprucht weniger Quelltext:

Beispiel 16.9

```
// file: c11/range-for.hpp
// description:

#include <iostream>
```

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

void printVector(vector<int> vekki) {
    for(auto element : vekki) {
        cout << element << endl;
    }
}

void printZahlen() {
    for(int n : { 1,4,7,8,9,3333,455435,5423,232,1337 }) {
        cout << n << endl;
    }
}

int main() {
    vector<int> test = { 1,2,3,4 };
    printVector(test);
    printZahlen();
}
```

Statt mit Iteratoren eine Collection zu durchlaufen, kann nun auch ein Range-For benutzt werden:

Beispiel 16.10

```
// file: c11/iterator-range-for.hpp
// description:

#include <iostream>
#include <vector>
#include <map>

using namespace std;

typedef map<wstring, int> frequencyMap;
frequencyMap::iterator it;

void printFrequencyIt(frequencyMap &frq) {
    for(auto iter = frq.begin(); iter != frq.end(); ++iter) {
        wcout << iter->first << L" kommt ";
        wcout << iter->second << L" Mal vor!" << endl;
    }
}

void printFrequencyRange(frequencyMap &frq) {
    for(auto word : frq) {
        wcout << word.first << L" kommt ";
        wcout << word.second << L" Mal vor!" << endl;
    }
}
```

```
int main() {
    // Beachte die wide- strings!!!
    frequencyMap frq = { {L"Max", 5}, {L"Andi", 4}, {L"Stefan", 1} };

    printFrequencyIt(frq);
    wcout << endl;
    printFrequencyRange(frq);

    return 0;
}
```

16.8 Neue Algorithmen

Der C++11 Standard erweitert die STL durch neue Funktionen der `algorithm`-Klasse. Die fünf wichtigsten Neuerungen werden hier kurz mit Beispielen vorgestellt:

16.8.1 `all_of()`

```
template< class InputIterator, class UnaryPredicate >
bool all_of( InputIterator first, InputIterator last, UnaryPredicate p );
```

`all_of()` gibt `true` zurück, wenn ein Prädikat für alle Elemente erfüllt ist. Ein konkreter Anwendungsfall dafür wäre ein Vector, welcher Zahlen enthält. Nun soll geprüft werden, ob alle Elemente positiv sind:

Beispiel 16.11

```
// file: c11/all_of.cpp
// description:

#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int main() {
    vector<int> intVector = { 1, 2, 3, 5, -6 };
    if(all_of(intVector.begin(), intVector.end(), [](int a){return a >=0;})) {
        cout << "Alle Zahlen sind positiv!" << endl;
    }
    else {
        cout << "Leider sind nicht alle Zahlen positiv!" << endl;
    }
}
```

Das Beispiel verwendet für die Überprüfung, ob eine Zahl größer oder gleich Null ist, eine Lambdafunktion. Die Alternative dazu wäre ein `bool-structure` zu verwenden.

Ausgabe:

```
Leider sind nicht alle Zahlen positiv!
```

16.8.2 `any_of()`

```
template< class InputIterator , class UnaryPredicate >
bool any_of( InputIterator first , InputIterator last , UnaryPredicate p );
```

`any_of()` gibt `true` zurück, wenn für mindestens eines der Elemente das Prädikat erfüllt ist. Ein konkreter Anwendungsfall dafür wäre ein Vector, der Zahlen enthält. Nun soll geprüft werden, ob mindestens ein Element größer als 5 ist:

Beispiel 16.12

```
// file: c11/any_of.cpp
// description:

#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int main() {
    vector<int> intVector = { 1, 2, 3, 5, 6, 0 };
    if(any_of(intVector.begin(), intVector.end(), [](int a){return a>5;})) {
        cout << "Mindestens eine Zahl ist größer als 5" << endl;
    }
    else {
        cout << "Leider ist keine Zahl größer als 5!" << endl;
    }
}
```

Ausgabe:

```
Mindestens eine Zahl ist größer als 5
```

16.8.3 none_of()

```
template< class InputIterator, class UnaryPredicate >
bool none_of( InputIterator first, InputIterator last, UnaryPredicate p );
```

none_of() gibt true zurück, wenn ein Prädikat für keins der Elemente erfüllt ist. Ein Anwendungsfall dafür wäre ein Vector, welcher Zahlen enthält. Nun soll geprüft werden, ob es ein Element gibt, welches kleiner Null ist:

Beispiel 16.13

```
// file: c11/none_of.cpp
// description:

#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int main() {
    vector<int> intVector = { 1, 2, 3, 5, -6 };
    if(none_of(intVector.begin(), intVector.end(), [](int a){return a <0;})) {
        cout << "Keine Zahl ist kleiner als Null" << endl;
    }
    else {
        cout << "Leider sind nicht alle Zahlen größer als Null!" << endl;
    }
}
```

Ausgabe:

```
Leider sind nicht alle Zahlen größer als Null!
```

16.8.4 copy_n()

```
template< class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size count, OutputIterator result);
```

copy_n() kopiert eine bestimmte Anzahl an Werten einer Auflistung in eine andere Auflistung. Ein Beispiel wäre:

```
int values[5] = { 1, 2, 3, 5, 6 };
int newValues[5];

// Kopiere 5 Elemente ins Ziel
copy_n(values, 5, newValues);
```

16.8.5 `iota()`

```
template< class ForwardIterator , class T >
void iota( ForwardIterator first , ForwardIterator last , T value );
```

Ändert eine Auflistung dahingehend, dass jedes Element ausgehend von einem Startwert jeweils um Eins erhöht wird. Die Rückgabe selbst erfolgt in der ursprünglich übergebenen Auflistung. Ein Beispiel erläutert diese Funktion:

Beispiel 16.14

```
// file: c11/iota.cpp
// description:

#include <algorithm>
#include <numeric>
#include <iostream>

using namespace std;

int main() {
    int a[5] = {0}; // initialisiertes Array

    // Startwert soll 42 sein
    iota(a, a+5, 42);

    // Das ganze machen wir jetzt mal mit Buchstaben
    // Um das komplette ABC zu erzeugen!
    char c[26] = {0};
    iota(c,c+26, 'a');

    // Gebe uns die Werte mal aus! Range- based loop in c++11
    for (int &i : a) {
        cout << i << endl;
    }

    // Werte des c- Arrays:
    for (char &c2 : c) {
        cout << c2 << endl;
    }
}
```

Dieses Programm ändert die Auflistung so, dass am Ende in dem Integer-Array die Zahlen 42 bis 46 gespeichert sind. Außerdem sind in dem Char-Array nun alle Buchstaben von a bis z gespeichert! Hinweis: Im Header muss neben `algorithm` auch noch `numeric` eingebunden werden!

16.9 Neue Datentypen für Internationalisierung

Mit dem neuen C++11-Standard wurden auch neue Datentypen für die Internationalisierung eingeführt:

Die Zeichentypen `char16_t` und `char32_t` (Neu: Mit fixer Bit-Breite!).

Daraus gebildet werden die Stringtypen:
`u16string` und `u32string`.

Die neuen Unicode-Stringliterale können mittels Präfix `u`, `U` und `u8` direkt im Quelltext benutzt werden:

```
char16_t a = u'h';
char32_t b = U'i';
u16string text = u"Ich bin ein utf-16 kodierter toller String!";
u32string test = U"Ich bin ein utf-32 kodierter, noch tollerer String!");
string haha = u8("Ich bin ein utf-8 kodierter String!");
string pii = u8"\u03c0";
// Was zu Fehlern führt:
u16string text = u"utf-16 string";
u32string s32;
s32 = text; // Ohje!
s32 = "Geht auch nicht!";
s32 = text.c_str(); // Das erst recht nicht!
s32.assign(text.cbegin(), text.cend()); // Nicht möglich!
```

16.10 Hashfunktion für ungeordnete Container: Ein Benchmarkbeispiel

Der C++11-Standard bringt neue, ungeordnete Container mit, für die sich eigene Hashfunktionen implementieren lassen. Das folgende Benchmarkbeispiel verwendet fünf Hashfunktionen:

16.10.1 Jenkins-Hashfunktion

Die Jenkins-Hashfunktion wurde etwas modifziert und wird in folgender Form implementiert:

```
struct jenkins_hash {
    size_t operator()(const std::wstring &word) const {
        size_t h = 0;
        for(size_t i = 0; i < word.length(); i++) {
            h += word[i];
            h += (h << 10);
            h ^= (h >> 6);
        }
        h += (h << 3);
        h ^= (h >> 11);
        h += (h << 15);
        return h;
    }
};
```

16.10.2 Stefan-Hashfunktion

Hier eine kurze Implementation einer eigenen Hashfunktion:

```
struct bad_hash {
    size_t operator()(const std::wstring &word) const {
        size_t h = 0;
        for(size_t i = 0; i < word.length(); i++) {
            h += i * (unsigned char)word[i];
        }
        return h;
    }
};
```

16.10.3 Modifizierte Stefan-Hashfunktion

Hier nun eine modifizierte Version aus dem Kapitel `Algorithms and Data Structures` aus dem Buch `The Practice of Programming` von Brian W. Kernighan und Rob Pike (BWK99):

```
struct verybad_hash {
    size_t operator()(const std::wstring &word) const {
        size_t h = 0;
        const wchar_t * s_ptr = word.c_str();
        for (const wchar_t * p = s_ptr; *p != L'\0'; ++p) {
            h = 31 * h + (unsigned wchar_t)*p;
        }
        return h;
    }
};
```

16.10.4 Benchmarking

Nun wird mittels `time`-Kommandozeilenaufruf gemessen, wie lange das Einlesen einer rund 600MB großen Wortliste dauert. Der Test erfolgt in einer virtuellen Maschine (KVM) -> AMD FX-8150, zugewiesen 4GB RAM, vier Kerne. Es wurden jeweils immer drei Durchläufe für die jeweilige Hashfunktion gemacht und der schnellste Durchlauf notiert:

16.10.5 Ergebnis: Jenkins-Hashfunktion

```
real 0m29.072s
user 0m28.887s
sys 0m0.174s
```

16.10.6 Ergebnis: Standard-Hashfunktion

```
real 0m25.912s
user 0m25.677s
sys 0m0.172s
```

16.10.7 Ergebnis: Stefan-Hashfunktion

```
real 0m54.742s
user 0m54.539s
sys 0m0.189s
```

16.10.8 Ergebnis: Stefan2-Hashfunktion

```
real 0m25.230s
user 0m25.017s
sys 0m0.199s
```

16.10.9 Fazit

Für die Speicherung von wide-strings ist die modifizierte Stefan2-Hashfunktion am schnellsten und überholt Standardhashfunktion der unordered_map. Etwas dahinter ist die one-at-a-time Jenkins-Hashfunktion. Die Stefan-Hashfunktion liegt deutlich abgeschnitten auf dem letzten Platz. Am besten vertraut man auf die Standardhashfunktion der STL. Sofern man vorhat, eine eigene Funktion zu schreiben, sollte man diese natürlich auch ausführlich testen - unbedingt auch mit großen Datenmengen!

16.10.10 Quelle

Weitere Informationen zu diesem Vergleich und alle Quelltexte stehen unter https://schweter.eu/2012/12/eigene-hashfunktionen.html zur freien Verfügung!

16.11 Die Zeitbibliothek Chrono: Ein weiteres Benchmarkbeispiel

Die neue C++11-Zeitbibliothek `chrono` besteht aus den Komponenten Zeitdauer (`duration`), dem Zeitpunkt (`time_point`) und den Zeitgebern (`system_clock`, `steady_clock` und `hight_resolution_clock`). In Version 4.6 des GCC-Compilers wird nur `system_clock` unterstützt, deswegen verwendet das folgende Beispiel auch nur diesen Zeitgeber.

Neben der Möglichkeit auf Kommandozeilenebene mittels `time` die Ausführdauer eines Programmes zu messen, bietet die Chrono-Bibliothek somit ebenfalls Möglichkeiten der (granularen) Zeitmessung an. Im nächsten Beispiel soll nun die Zeit gemessen werden, wie lange das Programm braucht um eine große Textdatei zeilenweise in einen STL-Container einzulesen. Als Zweites soll ausgegeben werden, wie lange das Programm braucht um diesen STL-Container zu durchlaufen. Anhand dieses Beispiels kann man danach sehr gut ablesen, welche Methode die schnellste ist, um einen Container zu durchlaufen.

Beispiel 16.15

```
// file: c11/benchmarkChrono.hpp
// description:

#include <chrono>
#include <locale>
#include <fstream>
#include <iostream>
#include <tr1/unordered_map>
#include <boost/foreach.hpp>

int main() {

    setlocale(LC_ALL, "");
```

```
    std::wstring line;

    std::locale de_utf8("de_DE.utf8"); // Sollte bitte vorhanden sein!
    std::wifstream inputFileStream("out.txt");
    inputFileStream.imbue(de_utf8);

    std::tr1::unordered_map<std::wstring, long> normal_map;

    auto startReading = std::chrono::system_clock::now();

    // Fire, fire!!!
    while(getline(inputFileStream, line)) {
        normal_map[line]++;
    }

    auto endReading = std::chrono::system_clock::now();

    auto readingDuration = endReading - startReading;

    std::chrono::seconds readingDurationInSeconds(std::chrono::duration_
cast<std::chrono::seconds>(readingDuration));

    std::cout << "Das Einlesen der Datei dauerte ";
    std::cout << readingDurationInSeconds.count();
    std::cout << " Sekunden" << std::endl;

    auto startIteration = std::chrono::system_clock::now();
    auto counter = 0;

    /*
    for (auto it = normal_map.begin(); it != normal_map.end(); ++it) {
        counter++;
    }
    */
    /*
    for (const auto x : normal_map) {
        counter++;
    }
    */

    BOOST_FOREACH(const auto x, normal_map ) {
        counter++;
    }

    auto endIteration = std::chrono::system_clock::now();
    auto iterationDuration = endIteration - startIteration;
    std::chrono::seconds iterationDurationInSeconds(std::chrono::duration_
cast<std::chrono::seconds>(iterationDuration));

    //std::cout << "Die Iteration mittels For- Schleife dauert: ";
    //std::cout << "Die Iteration mittels Range-For-Schleife dauert: ";
```

```
    std::cout << "Die Iteration mittels BOOST-For-Each dauert: ";
    std::cout << iterationDurationInSeconds.count();
    std::cout << " Sekunden" << std::endl;
    std::cout << "Iterationsdurchläufe: " << counter << std::endl;

    return 0;
}
```

16.11.1 Ergebnisse

Die Ergebnisse dieses kleinen Benchmarks werden nun aufgeführt. Das Einlesen der Textdatei dauert überall gleich:

```
[stefan@politeia benchmark]$ g++ -std=c++0x benchmarkChrono.cpp
[stefan@politeia benchmark]$ ./a.out
Das Einlesen der Datei dauerte 36 Sekunden
```

Nun folgen die Ergebnisse der Container-Durchläufe:

```
Die Iteration mittels For-Schleife dauert: 4 Sekunden
Iterationsdurchläufe: 12356630

Die Iteration mittels Range-For-Schleife dauert: 7 Sekunden
Iterationsdurchläufe: 12356630

Die Iteration mittels BOOST-For-Each dauert: 7 Sekunden
Iterationsdurchläufe: 12356630
```

Man kann also feststellen, dass das Durchlaufen eines Containers mittels Iterator deutlich schneller geht, als die anderen vorgestellten Varianten!

16.12 Weitere Neuerungen seit C++14

Seit C++14 ist es nun möglich Binärzahlen zu verwenden. Hierfür werden die Prefixes `0b` oder `0B` verwendet.

Beispiel 16.16

```
int binary = 0b11110010101;
```

Des Weiteren können einzelne Hochkommata nun als Zahlentrenner in Integer und Float-Werten verwendet werden. Dies dient schlicht der Vereinfachung des Lesens.

Beispiel 16.17

```
int bigNumber = 1'000'000'000;
float smallNumber = 0.000'000'000;
```

16.13 Ausblick auf C++17

Für den neuen Standard C++17 sind nach aktuellem Stand (Februar 2016) unter anderem folgende Features geplant:

- Neue Regeln für das `auto`-Keyword
- Verschachtelte Namespace Definitionen
- UTF-8 Buchstaben
- Neue Funktionen zum Einfügen in Maps und Unordered_Maps
- Einheitlicher Container Zugriff

16.14 Literaturhinweise

Zwei hervorragende Bücher von Rainer Grimm (Rai12) und Torsten Will (Tor12) zum neuen C++11-Standard dienten u.a. als Anregung für die Beispiele in diesem Kapitel. Dort finden sich auch weiterführende Informationen zu Themen, wie Smart Pointern, Multithreading, Generischer Programmierung usw. Die vierte Ausgabe des C++-Standardwerkes von Stroustrup wurde nun ebenfalls um den C++11-Standard ergänzt (Bja13)!

Literaturverzeichnis

[And00] Andrew Koenig, Barbara E. Moo: *Accelerated C++*, Addison-Wesley (2000) (Zitiert auf Seite 3)

[And09] Andrei Alexandrescu: *Modern C++ Design, Generic Programming and Design Patterns Applied*, Addison-Wesley (2009) (Zitiert auf Seite 3)

[Bja00] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley Longman, 11 Aufl. (2000) (Zitiert auf Seite 3)

[Bja09] Bjarne Stroustrup: *Programming: Principles and Practice Using C++*, Addison-Wesley Longman, 1 Aufl. (2009) (Zitiert auf Seite 3)

[Bja13] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley Longman, 4 Aufl. (2013) (Zitiert auf Seite 230)

[Bjo05] Bjoern Karlsson: *Beyond the C++ Standard Library. An Introduction to Boost*, Pearson (2005) (Zitiert auf Seiten 3 and 156)

[BWK99] Brian W. Kernighan, Rob Pike: *The Practice of Programming*, Addison-Wesley Longman (1999) (Zitiert auf Seite 226)

[D. 06] D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, Jeff Cogswell: *C++ Kochbuch*, O'Reilly (2006) (Zitiert auf Seite 3)

[Dav02] David Vandevoorde, Nicolai M. Josuttis: *C++ Templates: The Complete Guide*, Addison-Wesley Longman (2002) (Zitiert auf Seiten 3 and 33)

[Her94] Herbert Schildt: *Teach Yourself C++*, McGraw-Hill, 2 Aufl. (1994) (Zitiert auf Seiten 3 and 5)

[HJS86] Hans Jürgen Schneider: *Programmiersprachen: Konzepte und Trends*, SEL, Stuttgart (1986) (Zitiert auf Seite 4)

[Jos99] Josuttis, Nicolai M.: *The C++ Standard Library*, Addison-Wesley Longman (1999) (Zitiert auf Seiten 3 and 111)

[Rai12] Rainer Grimm: *C++11*, Addison-Wesley (2012) (Zitiert auf Seite 230)

[Sco08] Scott Meyers: *Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library*, Addison-Wesley Professional Computing, 11 Aufl. (2008) (Zitiert auf Seiten 3 and 111)

[Ste89] Stephen C. Dewhurst, Kathy T. Stark: *Programming in C++ (Paperback)*, Pearson Ptr (June 1989) (Zitiert auf Seite 5)

[Tor12] Torsten T. Will: *C++11 programmieren*, Galileo Computing (2012) (Zitiert auf Seite 230)

[Ulr07] Ulrich Breymann: *C++, Einführung und professionelle Programmierung*, Hanser, 9 Aufl. (2007) (Zitiert auf Seite 3)

[Ulr09] Ulrich Breymann: *Der C++ Programmierer*, Hanser (2009) (Zitiert auf Seite 3)

Abbildungsverzeichnis

3.1 Datei `eingabe.txt` 23
3.2 Datei `eingabe.txt` (Bytedarstellung) 23
3.3 Datei `eingabe.txt` (Bytedarstellung) 24
3.4 Datei `eingabe.txt` (ASCII-Darstellung) 24
3.5 Datei `eingabe.txt` unter Windows 25
3.6 Datei `eingabe.txt` unter Windows (ASCII-Darstellung) 25
3.7 Datei `eingabe.txt` (ASCII-Darstellung) unter Windows 25

Tabellenverzeichnis

1.1 Standardklassen in C++ 2

3.1 Variablen im Adressbuch 11
3.2 Standarddatentypen in C++ 12
3.3 Erweiterungen von Datentypen in C++ 12
3.4 Adressbuch, vom Compiler erzeugt 13
3.5 Streams 19
3.6 Binäre arithmetische Operatoren 30
3.7 Logische Operatoren 30
3.8 Wahrheitswertabelle bei logischen Ausdrücken 31
3.9 Lexikographischer Vergleich bei Strings 32

6.1 Methoden der Klase `string` 48
6.2 Konstruktoren der Klasse `string` 49

8.1 Bytefolgen UTF-8 71
8.2 Bytefolge mit fester Länge ISO__8859-1: ISO-LATIN1 (8-Bit) 71
8.3 Bytefolge mit fester Länge UCS-2: 2-byte Universal Character Set 71
8.4 Bytefolge mit variabler Länge UTF-8: 8-bit Unicode Transformation Format 72
8.5 Bytefolge mit variabler Länge UTF-16: 16-bit Unicode Transformation Format 72

10.1 Zugriffsrechte in den Klassen 94

12.1 STL vector Zugriffsfunktionen 120
12.2 Datenstruktur `list` 124
12.3 STL list Zugriffsfunktionen 124
12.4 Datenstruktur deque 127
12.5 STL deque Zugriffsfunktionen 127
12.6 STL set Zugriffsfunktionen 128
12.7 STL map Zugriffsfunktionen 129

15.1 Überladbare Operatoren 203

Index

Übungsskript, 7
`swap()`, 101

ASCII, 23

break, 39

C-String, 50

Daten
    Ausgabe, 22
    Einlesen, 21
Datentypen, 12
    als Rückgabewerte, 61

Encapsulation, 46
Endlosschleife, 41, 42

Formatierungsvorschläge, 62
fstream
    close(), 26
    ifstream, 25
    ofstream, 26
    open(), 26
Funktionen, 59–69
    Aufruf von Funktionen, 10
    Defaultwerte für Funktionsargumente, 62–63
    Funktionsargumente, 59, 62–65
        Übergabe von Werten, 64
        Übergaben *by value*, 64
        Übergaben *per Referenz*, 64
    Funktionskopf, 59
    Funktionsname, 59, 62
    Funktionsrumpf, 59, 63
    Funktionstyp, 59, 61–62
    main(), 8
    Motivation, 59
    Seiteneffekte, 65
    Überladung, 67–69
        Mehrdeutigkeiten, 64, 68

if, 38
Inheritance, 92
Installation von Eclipse, 7

Kommentare, 10
Konstanten, 33–36
    const, 33
    Initialisierung, 33
    konstantes Objekt, 33
    read-only, 35
    read-only-Methoden, 35
    Referezen, 35
Kontrollstrukturen, 37

Namespace, 16
    Defaultnamespace, 16
neue-standards, 213–230
    ausblick, 230
    auto, 216–218
    Chrono, 227–229
    Datentypen, 224
    decltype, 217
    Initialisierung, 213–215
    Lamda-Funktionen, 215–216
    Range-For, 218
    variablen-templates, 218
    weitere-neuerungen, 229–230

Operatoren, 29–32

arithmetische, 30
Ausgabeoperator, 19
Eingabeoperator, 19
für Strings, 31
relationale und logische, 30
Wahrheitstabelle, 31
Vergleichsoperatoren, 32, 50
Verknüpfungsoperator, 31
Zuweisungsoperator, 31

Polymorphismus, 98
private, 94
Programm starten, 7
Compiler, 7
Programmierung, 3
Programmstruktur, 8
protected, 94
public, 93

Speicher, 49
Statement, 37
compound, 37
Iteration
do, 41
for, 42
rangefor, 43
while, 40
selection, 38
String, 46–58
C-String, 25, 57
c_str(), 25, 57
Destruktoren, 49
Einlesen von strings, 20
Initialisierung, 47
Konstruktoren, 49
Konstruktormethode, 47
Methoden, 47
Modifizierung, 53
Vergleich, 50
Zugriff auf Elemente, 49
switch, 39

Template
`template`, 101
Templates, 101–110
generische Funktionen, 101
generische Klassen, 103
Initialisierung, 102
Schablonenfunktion, 101

Unix
Textdatei, 23
unsigned char, 46

Variablen, 11
Deklaration, 13
globale Variable, 14, 15
Gültigkeitsbereich, 15
Initialisierung, 13
Lebensdauer, 14
lokale Variable, 14, 15
Variablenregeln, 12
Variablentypen, 12
Zuweisung von Werten, 17
Vererbung, 92
spezielle Methoden, 94

Windows
cr, 26
Textdatei, 24
wstring, 46

Zugriffsrecht, 93
Zuweisungsoperator=, 17