Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Dokumentation
Parametric Programming

Erstellung und Benutzung eines assoziativen Indexes für Programmtexte

Christian Bunk
Alexander Miller

Abgabedatum: 07.06.2011

Prof. Dr. -Ing. Horst Hansen

Inhaltsverzeichnis

1. Motivation und Problemstellung

Die Standard Template Library (STL) ist ein Teil des C++ Standards, welcher eine effiziente Möglichkeit anbietet die Daten zu verwalten bzw. mit den Daten zu arbeiten. Im Laufe der Lehrveranstaltung "Parametric Programming" wurden die einzelnen Komponenten der STL wie die Behälter (engl. containers), Iteratoren (engl. iterators) und Algorithmen (engl. algorithms) vorgestellt und an kleinen Beispielen erläutert. Als Prüfungsleistung müssen zwei Projekte erfolgreich realisiert werden, wobei das in der Vorlesung erworbene Wissen praktisch angewandt werden sollte.

In der vorliegenden Ausarbeitung handelt es sich um eine Dokumentation zum ersten Projekt. Wie in der Aufgabenstellung definiert (Anhang A.1) soll ein Programm zur "Erstellung und Benutzung eines assoziativen Indexes für Programmtexte" modelliert und entwickelt werden. Da die Algorithmen der STL auf die Effizienz der Verarbeitung ausgelegt sind, wurden diese im Laufe des Projektes umfangreich verwendet.

2. Beschreibung der Programmstruktur

Wie in der Aufgabenstellung vordefiniert wurde bei der Implementierung des Programms, welches die vorgegebene Problemstellung effektiv löst, objektorientiert vorgegangen. Aus diesem Grund besteht das Programm aus einer globalen Funktion `main` (d.h. keine weiteren globalen Funktionen vorhanden) sowie aus mehreren Klassen, deren Methoden die eigentliche Funktionalität beinhalten. Darüber hinaus wurden nur die Standard-Bibliotheken der Sprache C++ verwendet. Die interne Datenhaltung für den erstellten Index erfolgt ausschließlich durch die Container der Standard Template Library (STL). Wie die praktische Umsetzung im Detail realisiert wurde, kann der Beschreibung der einzelnen Klassen entnommen werden. Im allgemeinen wurden alle mit `new` allokierte Speicherbereiche durch die `delete` Anweisung wieder freigegeben.

2.1. Controller

Die Klasse **Controller** dient der Verwaltung der übergebenen Parametern und übernimmt die vollständige Kontrolle über das gesamte Programm. Aus diesem Grund werden für die Koordination notwendige Parameter `int argc` und `char *argv[]` an den Konstruktor übergeben. Für die Extraktion der übergebenen Optionen wird eine zusätzliche Klasse **CmdLine** verwendet, welche im nächsten Abschnitt ausführlich beschrieben ist. Für die temporäre Speicherung der übergebenen Optionen wird ein Behälter vom Typ `map<string, string>` verwendet, da dieser nicht nur eine Option, sondern auch einen bestimmten Wert beinhalten kann (z.B. -t=<dateiname>). Für die Aufbewahrung von Argumenten wird ein `vector<string>` verwendet, weil es sich um eine Auflistung von Ausgabe- und Eingabedateien handelt.

Durch das Anwenden der Algorithmen `size()` und `empty()` des Map-Behälters, wird die Anzahl der übergebenen Optionen kontrolliert und ggf. eine entsprechende Fehlermeldung ausgegeben. Um das Programm benutzerfreundlich und bedienbar zu gestalten wird bei einem falschen Programmaufruf die öffentliche Methode `printHelp()` aufgerufen, so dass eine Hilfe mit der exakten Beschreibung eines richtigen Aufrufes am Terminal ausgegeben wird.

Durch das Durchgehen des Behälters mit Hilfe eines Iterators `map<string, string>::iterator`, werden die übergebenen Optionen auf die Richtigkeit über-

prüft und ggf. die zuständigen Methoden der Klasse **Index** aufgerufen. Das Programm kann mit folgenden Übergabeparametern gestartet werden:

<program> <options> <outputfile> <inputfile>*
<program> : Program name
<options> :
-p Print index list on data terminal
-i Create an index
-q=<word> Print all indexes for word <word> on data termnal
-s=<prefixterm> Print indexes for all words with prefix term <prefixterm>
-t=<filename> Print indexes for words founded in file <filename>
–help Description of command line parameters

Im Falle einer falschen Eingabe erhält der Benutzer eine verständliche Fehlermeldung und wird auf die richtige Eingabe der Parameter hingewiesen.

2.2. CmdLine

Die Klasse **CmdLine** übernimmt das Einlesen der Übergabeparameter und unterscheidet dabei zwischen Optionen (z.B. -i) und Argumenten (z.B. output.txt, input.txt). Wie bereit in der Klasse **Controller** beschrieben werden dafür die Behältertypen `map<string, string>` und `vector<string>` verwendet. Die öffentlichen Methoden `map<string,string> getOptions();` und `vector<string> getArguments();` ermöglichen der Klasse **Controller** einen Zugriff auf die gespeicherten Daten.

2.3. Index

Die Klasse Index soll einen Index aus eingelesenen Daten erzeugen. Die Klasse Index wird vom Controller erzeugt. Sie stellt verschiedene Methoden bereit um den Index nach bestimmten Kriterien zu durchsuchen.

Der Index wird nach dem Schema `<Wort> <Datei> <Zeilen>` erstellt. Dazu wurde folgende Datenstruktur entworfen. Eine `map` speichert als `key` das Wort. Somit kann die `map` leicht nach bestimmten Wörtern durchsucht werden. Auch das Finden, Einfügen und Löschen ist bei einer `Map` relativ leicht. Als `value` wird wieder eine `map` angegeben. Ein Wort zeigt damit wieder auf eine weitere Map. In dieser inneren `map` wird als `key` der Dateiname geschrieben und als `value` ein `set` mit Zeilennummern. Damit wird eine Assoziation hergestellt, das ein Wort in einer Datei in verschiedenen Zeilennummern vorkommen kann. Ein Wort kann auch in mehreren Dateien vorkommen.

Bei einem Wort wird zwischen Groß- und Kleinschreibung unterschieden. Das Wort 'Hallo' und 'hallo' sind zwei verschiedene Wörter und tauchen auch

seperat im Index auf. Ein Wort besteht aus den Zeichen `[A-Za-z_]([A-Za-z0-9]|-|_)*`. Der Index untersucht die Wörter nicht auf semantische Korrektheit! Alle Wörter die aus gültigen Zeichen bestehen werden akzeptiert. Die Klasse `FileUtil` wird genutzt um die Dateien zeilenweise auszulesen. Anschließend werden die Wörter aus jeder Zeile entnommen. Jedes Wort wird in die `map` eingefügt, zusammen mit dem Dateinamen und der aktuellen Zeilennummer. Ist das Wort in der `map` schon vorhanden wird nur der Dateiname und die Zeilennummer eingefügt. Ist auch schon der Dateiname vorhanden, wird nur die Zeilennummer hinzugefügt wenn diese noch nicht vorhanden ist.

2.4. IndexParser

Damit eine Index-Datei in das Programm eingelesen werden kann, wurde die Klasse **IndexParser** implementiert. Um die Daten in die bereits vorhandene Datenstruktur einzulesen, wird ein Zeiger auf das Objekt der Indexklasse dem Konstruktor übergeben. Somit besteht die Möglichkeit die bereits vorhandene Index-Datei mit der privaten Methode `void FileToLines(vector<string> *lines, string index_file);` zeilenweise einzulesen und die extrahierten Daten in einem `vector<string>` - Behälter zu speichern. Zunächst wird die Methode `string ParseLine(string linie, bool flag_wort, bool flag_file, bool flag_index, string last_word);` aufgerufen, welche die einzelnen Zeilen wortweise einliest und durch die öffentliche Methode `void addToIndex(string, string, set<int, less<int> >);` der Klasse **Index** in die Datenstruktur schreibt.

2.5. Lexic

Die Klasse **Lexic** soll für eine lexikografische Sortierung in der `map` der Klasse **Index** sorgen. Ein einfacher vergleich zweier Zeichen reicht nicht aus, da hier nur der ASCII-Wert verglichen wird. Die Funktion `lexic_compare()` unterscheidet zwar auch zwischen Groß- und Kleinschreibung allerdings werden z.B. Wörter wie 'hallo' und 'Hallo' als identisch betrachtet und tauchen dann in der `map` nur als eine der beiden Varianten auf. Beide Varianten liefern also keine zufriedenstellende Lösung an die Anforderungen der Aufgabe. Aus diesem Grund wurde eine eigene Implementierung der lexikografischen Sortierung erstellt, welche wie folgt funktioniert:
Die Klasse **Lexic** stellt lediglich einen Funktionsoperator `operator()(string s1, string s2)` zur Verfügung. Diesem werden zwei Strings zum Vergleich übergeben. Nun werden die Strings zeichenweise verglichen, beginnend bei dem ersten Zeichen. Zuerst findet ein case insensitive Vergleich aller Zeichen statt. Dazu werden die Zeichen in Kleinbuchstaben mit Hilfe der Methode `tolower(char c)` verwandelt. Unterscheidet sich ein Zeichen der beiden Zeichenketten, so ist

der Vergleich vorbei. Denn nun kann eindeutig gesagt werden ob der Buchstabe vor oder nach dem anderen Buchstaben im Alphabet kommt. sind beide Zeichen jedoch gleich wird zum nächsten Zeichen der beiden Zeichenketten gesprungen. Beide Strings werden so lange untersucht bis ein Buchstabe kleiner oder größer ist. Wenn einer der beiden Strings zu Ende ist, dann kann kein Vergleich mehr statt finden. Bis hier hin kann also gesagt werden ob beide Strings identisch sind, oder einer von beiden länger ist aber den anderen bis hier hin als Teilstring enthält (z.B. 'Welt' und 'Weltkarte'). Nun wird also untersucht ob beide Strings gleich lang sind. Wenn nicht so ist der kürzere String kleiner als der längere.

Sind beide Strings jedoch gleich lang muss eine *case sensitive* Untersuchung stattfinden. Beide Strings werden also wieder von Anfang bis Ende zeichenweise untersucht. Für eine case sensitive Untersuchung wird einfach der ASCII Wert der beiden Zeichen verglichen. Ist der Buchstabe ein Großbuchstabe, so kommt er vor dem Kleinbuchstaben. Dies ist lediglich eine Definition, welche jederzeit umgetauscht werden kann. In diesem Projekt wurde festgelegt, dass ein Großbuchstabe immer vor einem gleichen Kleinbuchstaben kommt. Dieser Vergleich findet so lange statt bis ein Buchstabe größer oder kleiner ist. Sind beide Strings zu Ende (denn beide sind hier gleich lang) sind alle Buchstaben bezüglich der Groß- und Kleinschreibung gleich. In diesem Fall liegen also zwei absolut identische Wörter vor.

A. Anhang

A.1. Aufgabenstellung

1. Belegaufgabe

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Prog. mit parametrisierten Datentypen	Sommersemester 2010	Dozent: Horst Hansen
Ausgabe: 26.4.2010	Abgabe Gruppe 1: 31.5.2011, Gruppe 2: 7.6.2011	

Lernziele:
Mit der Lösung dieser Aufgabe sollen sie zeigen, daß Sie in der Lage sind, Anwendungen unter Verwendung parametrisierter Datentypen zu entwerfen und mit Hilfe der Standard Template Library (STL) von C++ zu implementieren.

Spezifische Ziele:

- Verwendung parametrisierter Datentypen beim Entwurf von Programmen
- Benutzen der Behälterklassen der STL
- Benutzen von Algorithmen aus der STL
- Benutzen der Ein-/Ausgabefunktionen von C++
- Trennen von Anwendungsfunktionalität und Benutzungsschnittstelle (Kommandozeile!)
- Dokumentation von Programmen

Aufgabe: Erstellung und Benutzung eines assoziativen Indexes für Programmtexte

Es soll ein Programm zum Indizieren von Texten erstellt werden. Es wird für jedes Wort ein Index erstellt, der angibt, in welcher Zeile an das Wort auftritt. Der erstellte Index wird in vorgegebener Form (siehe unten) in eine Textdatei und/oder am Terminal ausgegeben. Anschließend kann der Index zum Beantworten von Fragen verwendet werden.

Funktionale Anforderungen

Um Programme mit vergleichbaren Ergebnissen zu erhalten, gelten verbindlich die folgenden Definitionen für die Lösung der Aufgabe:

- Das Programm indiziert die Wörter einer oder mehrerer Eingabedatei(en) und gibt den errechneten Index in die Ausgabedatei und optional am Terminal aus.
- Das Programm verwendet den erstellten Index zum Beantworten von Anfragen des Benutzers.
- Ein Wort ist eine zusammenhängende Folge von Zeichen, die mit einem Unterstrich oder einem Buchstaben beginnt und anschließend Buchstaben, Ziffern, Bindestriche oder Unterstriche enthält.

  Als regulärer Ausdruck: `[A-Za-z_]([A-Za-z0-9]|-|_)*` .
- Alle anderen Zeichen sind *Trennzeichen*, d.h. sie beenden ein Wort.

Anforderungen an die Benutzungsschnittstelle

- Die Steuerung des Programms erfolgt ausschließlich über die Kommandozeile.
- Das Programm soll mit den folgenden Kommandozeilenparametern gestartet werden:
  `<program> <options> <outputfile> <inputfile>*`
  `<program>` : Programmname
  `<options>` :
  - `-p` Ausgabe der Indexliste am Terminal
  - `-i` Erstellen des Indexes
  - `-q=<wort>` : Ausgeben des vollständigen Indexes zum Wort `wort` am Terminal
  - `-s=<wortanfang>` : Ausgeben des vollständigen Indexes zu allen Wörtern mit dem Wortanfang `wortanfang` am Terminal
  - `-t=<dateiname>` : Ausgeben der Indizes zu allen Wörtern, die in der Datei `dateiname` vorkommen, am Terminal
  - Die Anfragen verwenden nicht die Eingabedateien, sondern lesen einen zuvor erstellten Index ein! In diesem Fall ist also `outputfile` die einzulesende Datei und es gibt kein `inputfile`!

  `<outputfile>` : Dateiname der Ausgabedatei mit der Indexliste
  `<inputfile>*` : Liste von Eingabedateien mit zu indizierendem Text
- Ausgabe von aussagekräftigen Meldungen bei fehlerhaften Eingaben auf der Kommandozeile
- Verhindern des Überschreibens von Ausgabedateien (leicht)
- Die Ausgabe des vom Programm erzeugten Wortindexes am Terminal und in die Ausgabedatei soll so aufgebaut sein:
  `<wort> BLANK <dateiindex>+` , wobei
  `<dateiindex> ::= <dateiname> (BLANK <zeilennummer>)*` ist.
  Dabei beginnen Wörter eine neue Zeile, jeder weitere Dateiname beginnt ebenfalls eine neue Zeile, wird aber mindestens ein Zeichen weit eingerückt.
- Die Ausgabe der Wortliste erfolgt stets lexikografisch sortiert.
- Die Ausgabe der Zeilennummern erfolgt in aufsteigender Reihenfolge.

Anforderungen an die Implementierung

- Als Kodierung der Zeichen in den Textdateien und im Index wird *ISO Latin1* verwendet.
- Die programminterne Datenhaltung soll mittels der Behältertypen und der Algorithemn der STL implementiert werden.
- Es dürfen nur Standardbibliotheken von C++ für die Implementierung verwendet werden.
- Die Verwendung globaler Variablen oder von Sprunganweisungen ist und bleibt verboten.
- Die Anwendung wird objektorientiert implementiert: Es gibt also nur Klassen und die globale Funktion `main`.
- Wer noch weitere globale Funktionen benötigt, muß dies schriftlich ausführlich begründen!

- Es wird ein Makefile bereitgestellt für
  - das Erzeugen des ausführbaren Programms
  - das Erzeugen der Programmdokumentation
  - das Löschen aller aus den Programmquellen erzeugten Daten
  - das Überprüfen der Speicherverwendung des Programms mittels des Werkzeugs `valgrind`
- Zum Testen des Programms stehen Testdateien in der Datei `Testdaten.zip` auf der Seite zur Lehrveranstaltung zur Verfügung.

Benotung:
Um eine sehr gute Note erreichen zu können, müssen Sie außer den *Muß-Kriterien* auch alle *Soll-Kriterien* erfüllen. Die Erfüllung einzelner *Soll-Kriterien* führt zu einer schrittweisen Verbesserung der Ausgangsnote *drei* für die Bewertung.
Lösungen, die sich nicht an die unter *Anforderungen an die Implementierung* genannten Verbote halten, werden mit **null** Punkten bzw. der Note 5 (**ungenügend**) bewertet!
Beachten Sie bitte dazu auch die auf der Seite zur Lehrveranstaltung veröffentlichte Notenskala!

Als Lösung sind abzugeben:

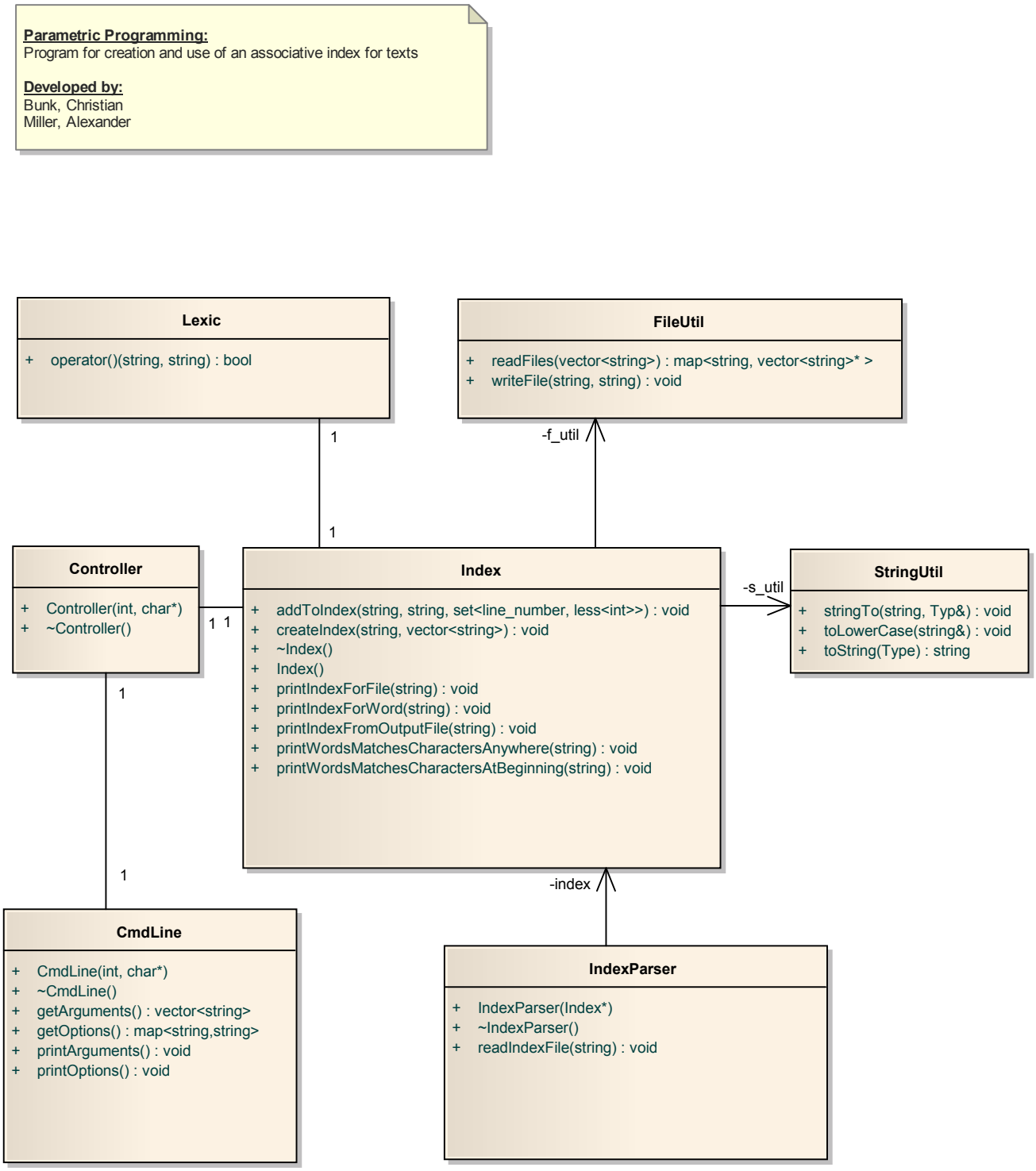
- ein **Ausdruck** des Klassendiagramms Ihrer Lösung mit den öffentlichen Methoden
- eine vollständige mit `doxygen` erstellte Dokumentation Ihres Programms (**ohne Ausdruck!!!**)
- ein **Ausdruck** einer schriftlichen Beschreibung der Programmstruktur unter besonderer Berücksichtigung der gewählten Behältertypen und Algorithmen aus der STL
- ein **Ausdruck** des kommentierten Programms (z.B. erstellt mittels `pp`)
- ein **Ausdruck** der Indexinformation zu den Texten `Indextest0.txt` und `Indextest1.txt`

Die Abgabe der Lösung erfolgt durch jede Gruppe von Studierenden (maximal 2 Personen pro Gruppe) persönlich im Rahmen der entsprechenden oben genannten Übungsstunde an den Dozenten. Bei der Abgabe der Lösung an einem Rechner im Übungslabor muß das unter Linux funktionsfähige Programm übersetzt und vorgeführt werden. Beide Studierende einer Gruppe müssen alle Fragen des Dozenten zum Programm beantworten können.

Bewertungskriterien:
Bewertet werden neben der Vorführung mit Erläuterung (siehe oben):

- die korrekte Funktion des Programms
- die objektorientierte Struktur des Programms
- die Robustheit des Programms
- die Lesbarkeit des Programmtextes
- die Beschreibung der Programmstruktur
- die Einhaltung der Programmierrichtlinien
- die Form der schriftlichen Dokumente
- die Extras

A.2. Klassendiagramm


Parametric Programming:
Program for creation and use of an associative index for texts
Developed by:
Bunk, Christian
Miller, Alexander
Lexic
+ operator()(string, string) : bool
FileUtil
+ readFiles(vector<string>) : map<string, vector<string>* >
+ writeFile(string, string) : void
-f_util
1
1
Controller
+ Controller(int, char*)
+ ~Controller()
1 1
Index
+ addToIndex(string, string, set<line_number, less<int>>) : void
+ createIndex(string, vector<string>) : void
+ ~Index()
+ Index()
+ printIndexForFile(string) : void
+ printIndexForWord(string) : void
+ printIndexFromOutputFile(string) : void
+ printWordsMatchesCharactersAnywhere(string) : void
+ printWordsMatchesCharactersAtBeginning(string) : void
-s_util
StringUtil
+ stringTo(string, Typ&) : void
+ toLowerCase(string&) : void
+ toString(Type) : string
1
1
-index
CmdLine
+ CmdLine(int, char*)
+ ~CmdLine()
+ getArguments() : vector<string>
+ getOptions() : map<string,string>
+ printArguments() : void
+ printOptions() : void
IndexParser
+ IndexParser(Index*)
+ ~IndexParser()
+ readIndexFile(string) : void

A.3. Index aus Testdaten

Index aus indextest0.txt
auch Indextest0.txt 5
Dies Indextest0.txt 1 5
eine Indextest0.txt 1
es Indextest0.txt 7
geht Indextest0.txt 6
ist Indextest0.txt 1
noch Indextest0.txt 5
So Indextest0.txt 6
Testdatei Indextest0.txt 5
zweite Indextest0.txt 2

Index aus indextest1.txt
Alles Indextest1.txt 5
auch Indextest1.txt 4
Dies Indextest1.txt 1 4
eine Indextest1.txt 1
erste Indextest1.txt 2
ist Indextest1.txt 1
noch Indextest1.txt 4
prima Indextest1.txt 5
Testdatei Indextest1.txt 4

Index aus indextest0.txt und indextest1.txt
Alles Indextest1.txt 5
auch Indextest0.txt 5
  Indextest1.txt 4
Dies Indextest0.txt 1 5
  Indextest1.txt 1 4
eine Indextest0.txt 1
  Indextest1.txt 1
erste Indextest1.txt 2
es Indextest0.txt 7
geht Indextest0.txt 6
ist Indextest0.txt 1
  Indextest1.txt 1
noch Indextest0.txt 5
  Indextest1.txt 4
prima Indextest1.txt 5
So Indextest0.txt 6

Testdatei Indextest0.txt 5
  Indextest1.txt 4
zweite Indextest0.txt 2

A.4. Quellcode

Listing A.1: „main.cpp“

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

#include "controller.hpp"

using namespace std;

/**
* @mainpage Index - Project for Parametric Programming
* @section Programmaufruf
* Creates an index from input files into one output file.
* @code
* index -i <outputFile> <inputFile>+
* @endcode
* Print the index from given index file.
* @code
* index -p <indexFile>
* @endcode
* Print the index only for this word
* @code
* index -q=<word> indexFile
* @endcode
* Print index where the given word is at the beginning.
* @code
* index -s=<word> indexFile
* @endcode
* Print the index the given word is anywhere in another word.
* @code
* index -s=<*word> indexFile
* @endcode
* Print the index for words occuring in that file.
* @code
* index -t=<file> indexFile
* @endcode
*
* @section Copyright
* @authors Alexander Miller, Christian Bunk
*/
int main (int argc, char *argv[]) {
    Controller *ctr = new Controller(argc, argv);
    delete ctr;

    return EXIT_SUCCESS;
}
```

Listing A.2: „controller.hpp“

```
#ifndef _CONTROLLER_H
#define _CONTROLLER_H

//#include <algorithm>

#include "cmdline.hpp"
#include "index.hpp"
#include "indexparser.hpp"

using namespace std;
```

```
11
12 /** Class Controller
13  *
14  *  This class controlls the flow of this programm.
15  *
16  *  author  Alexander Miller, Christian Bunk
17  *  date    3.5.2011
18  *  version 0.1
19  */
20 class Controller {
21 private:
22
23 public:
24     /**
25      * Takes the paramters from main function call.
26      * @param argc Nummber of parameters.
27      * @param argv[] Array with c-strings with all given paramters.
28      */
29     Controller(int argc, char *argv[]);
30
31     /**
32      * Destructor.
33      */
34     ~Controller();
35
36 private:
37
38     /**
39      * Print help information about how to use this program.
40      */
41     void printHelp();
42
43 };
44
45 #endif  /* _CONTROLLER_H */
```

Listing A.3: „controller.cpp“

```
1  #include "controller.hpp"
2
3  Controller::Controller(int argc, char *argv[]) {
4
5      // read command line parameters
6      CmdLine params(argc, argv);
7
8      map<string, string> options = params.getOptions();
9      vector<string> arguments = params.getArguments();
10 /*
11     cout << "Options: " << endl;
12     params->printOptions();
13
14     cout << endl << "Arguments: " << endl;
15     params->printArguments();
16 */
17     // params is no longer needed
18     //delete params;
19
20     Index *index = new Index();
21
22     #ifdef Two_Options
23         if (options.size() > 1) {
24             if (options.size() == 2) {
```

```
25                  if (options.find("p") == options.end() ||
                        options.find("i") == options.end() ) {
26                      cout << "Two options are only allowed with -i and -p
                            !" << endl;
27                      delete index;
28                      return;
29                  }
30              } else {
31                  cout << "Error: please type in just one option i.e. -i or
                        -q=value" << endl;
32                  delete index;
33                  return;
34              }
35          }
36      #else
37          // prÃ¼fen wie viele options angegeben wurden und entsprechend
                Meldung ausgeben
38          if (options.size() > 1) {
39              cout << "Error: please type in just one option i.e. -i or
                    -q=value" << endl;
40              delete index;
41              return;
42          }
43      #endif
44
45      if (options.empty()) {
46          cout << "Error: please type in one option" << endl;
47          this->printHelp();
48
49          delete index;
50          return;
51      }
52
53      // creates new IndexParser
54      IndexParser parser(index);
55
56      //  create new index
57      map<string,string>::iterator position = options.find("i");
58      if (position != options.end()) {
59          // need at least two arguments (one the output file and at least
                one input file)
60          if (arguments.size() >= 2) {
61              // read index file, so it will not be over-written
62              //parser->readIndexFile(*arguments.begin());   // Previous
                    version for parsing of existing index file
63              string file = *arguments.begin();
64              ifstream in(file.data()) ; // Try to open given IndexFile
65              // Test if file exist. true --> Warning and Exit the program;
                    false --> create index and write it into the file
66              if (!in){
67                  string out = *arguments.begin();
68
69                  // delete the first argument, the others are the input
                        files
70                  arguments.erase(arguments.begin());
71
72                  // create the index, read the input files and write the
                        index in the output file
73                  index->createIndex(out, arguments);
74                  in.close() ; // Datei schlieÃen
75                  cout << "Created index successful written in file " <<
                        file << " \n";
76              }
77              else{
```

```
78                  cout << "Output file already existing!\n";
79                  in.close() ; // Datei schlieÃen
80                  return;
81              }
82          }
83          else {
84              cout << "Error: not enough arguments. Need at least one output
                    file and one input file i.e. -i out.txt in.txt" << endl;
85          }
86      }
87
88      // print the entire index
89      position = options.find("p");
90      if (position != options.end()) {
91          // need at least one argument (the output file to be read in for
                the parser)
92          if (!arguments.empty()) {
93              parser.readIndexFile(*arguments.begin());
94
95              // TODO: how many arguments
96              index->printIndexFromOutputFile(*arguments.begin());
97          }
98      }
99
100     // print index for the word
101     position = options.find("q");
102     if (position != options.end()) {
103         // need at least one argument (the output file to be read in for
                the parser)
104         if (!arguments.empty()) {
105             parser.readIndexFile(*arguments.begin());
106
107             cout << "suche nach dem Wort: " << position->second << endl;
108             index->printIndexForWord(position->second);
109         } else {
110             cout << "Error: need output file as an argument i.e. -q=word
                    out.txt" << endl;
111         }
112     }
113
114     // print index for words who matches characters
115     position = options.find("s");
116     if (position != options.end()) {
117         if (!arguments.empty()) {
118             parser.readIndexFile(*arguments.begin());
119
120             if (*position->second.begin() == '*') {
121                 position->second.erase(position->second.begin());
122                 index->printWordsMatchesCharactersAnywhere(position->second);
123             } else {
124                 index->printWordsMatchesCharactersAtBeginning(position->second);
125             }
126
127
128         } else {
129             cout << "Error: need output file as an argument i.e. -s=word
                    out.txt" << endl;
130         }
131     }
132
133     // print index file
134     position = options.find("t");
135     if (position != options.end()) {
136         if (!arguments.empty()) {
```

```
137             parser.readIndexFile(*arguments.begin());
138
139             index->printIndexForFile(position->second);
140         } else {
141             cout << "Error: need output file as an argument i.e. -t=word
                    out.txt" << endl;
142         }
143     }
144
145     // print index file
146     position = options.find("-help");
147     if (position != options.end()) {
148         this->printHelp();
149     }
150
151     //delete parser;
152     delete index;
153 }
154
155 Controller::~Controller() {}
156
157 void Controller::printHelp() {
158     cout <<
            "*************************************************************************"
            << endl;
159     cout << "Creation and use of an associative index for program texts"
            << endl;
160     cout <<
            "*************************************************************************"
            << endl;
161     cout << "This program can be started with following command line
            parameters:" << endl;
162     cout << "<program> <options> <outputfile> <inputfile>*" << endl;
163     cout << "<program> :         Program name" << endl;
164     cout << "<options> :         " << endl;
165     cout << "  -p                Print index list on data terminal" << endl;
166     cout << "  -i                Create an index" << endl;
167     cout << "  -q=<word>         Print all indexes for word <word> on data
            termnal" << endl;
168     cout << "  -s=<prefixterm>   Print indexes for all words with prefix
            term <prefixterm>" << endl;
169     cout << "  -t=<filename>     Print indexes for words  founded in file
            <filename>" << endl;
170     cout << "  --help                Description of command line
            parameters" << endl;
171     cout << "<outputfile> :      Filename of output file with created index
            list" << endl;
172     cout << "<inputfile>* :      List of input files for indexing" << endl;
173 }
```

Listing A.4: „cmdline.hpp“

```
1  #ifndef _CMDLINE_H
2  #define _CMDLINE_H
3
4  #include <vector>
5  #include <iterator>
6  #include <string>
7  #include <map>
8  #include <iostream>
9  //#include <sstream>
10
11 using namespace std;
```

```
12
13 typedef map<string, string> options; // Options are strings. Each option
       can have a value
14 typedef vector<string> arguments; // A List with arguments.
15
16 /** Class CmdLine
17  *
18  *  This class split the command line parameters into options and
        arguments.
19  *
20  *  author  Christian Bunk
21  *  date    3.5.2011
22  *  version 0.1
23  */
24 class CmdLine {
25 private:
26     options *opt; // Alle Parameter die mit einem Bindestrich beginnen
          wie: "-p"
27     arguments *arg; // alle anderen Parameter ohne Bindestrich, z.B.
          Dateinamen "Index.txt"
28
29     void insertToOptions(string option, string value);
30
31 public:
32     /**
33      * Create new CmdLine.
34      * @param argc number of parameter.
35      * @param argv parameter in an array.
36      */
37     CmdLine(int argc, char *argv[]);
38
39     /**
40      * Destructor.
41      */
42     ~CmdLine();
43
44     /**
45      * Print all options.
46      */
47     void printOptions();
48
49     /**
50      * Print all Arguments.
51      */
52     void printArguments();
53
54     /**
55      * Return the options in a map.
56      * @return map<string, string> The option with a value. The value is
            optional.
57      */
58     map<string,string> getOptions();
59
60     /**
61      * Return the arguments in a vector.
62      * @return vector<string> The arguments in a vector.
63      */
64     vector<string> getArguments();
65 };
66
67 #endif  /* _CMDLINE_H */
```

Listing A.5: „cmdline.cpp“

```
1
2  using namespace std;
3
4  #include "cmdline.hpp"
5
6  CmdLine::CmdLine(int argc, char *argv[]) {
7      opt = new options;
8      arg = new arguments;
9
10     // if there is at least one parameter
11     if (argc > 1) {
12         size_t found_minus;
13
14         // go through each parameter
15         for (int i = 1; i < argc; i++) {
16             // get next parameter
17             string parameter = string(argv[i]);
18             string::iterator p_it = parameter.begin();
19
20             // now ceck all characters of parameter
21
22             // if param is an option
23             found_minus = parameter.find('-');
24
25             if (found_minus == 0) {
26                 // check if there is an equal sign
27                 size_t found_equal = parameter.find('=');
28
29                 // if there is an equal sign
30                 if (found_equal != string::npos) {
31                     string option;
32                     option.assign(parameter, 1, found_equal-1);
33
34                     string value;
35                     value.assign(parameter, found_equal+1, string::npos);
36                     this->insertToOptions(option, value);
37                 } else {
38                     // if there is now equal sign
39                     string option;
40                     option.assign(parameter, 1, string::npos); // leave
                           the minus, and get all character after minus til
                           the end
41
42                     // insert option to map with no value
43                     this->insertToOptions(option, "");
44                 }
45
46
47             } else {
48                 this->arg->push_back(parameter);
49             }
50         }
51     }
52 }
53
54 CmdLine::~CmdLine() {
55     delete opt;
56     delete arg;
57 }
58
59 void CmdLine::insertToOptions(string option, string value) {
60     this->opt->insert(pair<string,string>(option, value));
61 }
```

```
62
63 void CmdLine::printOptions() {
64     for (map<string, string>::iterator m_it = opt->begin();
           m_it!=opt->end(); m_it++) {
65         cout << "key: " << m_it->first << "; value: " << m_it->second <<
               endl;
66     }
67 }
68
69 void CmdLine::printArguments() {
70     for (vector<string>::iterator it = arg->begin(); it!=arg->end(); ++it)
           {
71         cout << *it << endl;
72     }
73 }
74
75 map<string,string> CmdLine::getOptions() {
76     return *opt;
77 }
78
79 vector<string> CmdLine::getArguments() {
80     return *arg;
81 }
```

Listing A.6: „index.hpp“

```
1  #ifndef _INDEX_H
2  #define _INDEX_H
3
4  #include <fstream>
5  #include <iostream>
6  #include <iterator>
7  #include <vector>
8  #include <string>
9  #include <map>
10 #include <set>
11 #include <sstream>
12 #include <algorithm>
13
14 #include "stringutil.hpp"
15 #include "fileutil.hpp"
16 #include "lexic.hpp"
17
18 using namespace std;
19
20 /**
21  * word is a string
22  */
23 typedef string word;
24
25 /**
26  * file is a string
27  */
28 typedef string file;
29
30 /**
31  * line_number is an int
32  */
33 typedef int line_number;
34
35 /**
36  * line_numbers is a set containing int's
37  */
```

```
38 typedef set<line_number, less<int> > line_numbers;
39
40 /**
41  * files is a map containing the file(name) and a set of line number's
42  */
43 typedef map<file, line_numbers* > files;
44
45 /**
46  * words is a map containing the word an another map (with files and line
        number's)
47  */
48 #ifdef Lexic_Compare
49     typedef map<word, files*, Lexic > words;
50 #else
51     typedef map<word, files*> words;
52 #endif
53
54 /** Class Index
55  *
56  *  This class creates an index of several input files.
57  *
58  *  author  Christian Bunk
59  *  date    3.5.2011
60  *  version 0.1
61  */
62 class Index {
63 private:
64     // this map stores for each word, another map
65     words *_word_index; // stores a pointer to map_files
66
67     StringUtil *s_util; // reference to StringUtil
68
69     FileUtil *f_util; // reference to FileUtil
70
71     /**
72      * Doing some initial stuff.
73      */
74     void init();
75
76     // Read and write files:
77
78     /**
79      * Read content (from given files).
80      * @param files
81      */
82     void readContent(vector<file> files);
83
84     /**
85      * Write content (into file).
86      * @param out_file
87      */
88     void writeContent(file out_file);
89
90
91     // Build the index:
92
93     /**
94      * Read all lines within a file.
95      * @return vector<string>* All lines within the file.
96      */
97     vector<string>* readAllLines(file f);
98
99     /**
100     * Exctract all words from a line.
```

```
101       * Rules: each empty character will be ignored
102       */
103      vector<word> extractAllWordsFromLine(string line);
104
105      /**
106       * Add a word into the index, corresponding to a file and a line
              number.
107       * This mehtod is used by addToIndex(vector<word>, file, line_number)
108       * @param w
109       * @param f
110       * @param l
111       */
112      void addToIndex(word w, file f, line_number l);
113
114      /**
115       * Add a vector of words into the index, corresponding to a file and a
              line number.
116       * Use the method addToIndex(word, file, line_number)
117       * This method is used when the index is created.
118       * @param words
119       * @param f
120       * @param l
121       */
122      void addToIndex(vector<word> words, file f, line_number l);
123
124      /**
125       * Add a new Word and return the iterator to that position.
126       * If the word is in map already, then return the iterator with
              position of that word.
127       * @return map<word, map_files>::iterator with the position of the
              word in map.
128       */
129      words::iterator addWord(word w);
130
131      /**
132       * Add a file to a word and returns the iterator to the position of
              the file key
133       * If the file is already in map, then the iterator points to that
              file.
134       *
135       * @param it the iterator pointing to the word where the fill will be
              added.
136       * @return map<file, list<line> >::iterator pointing to the file.
137       */
138      files::iterator addFile(words::iterator it, file f);
139
140      /**
141       * Adds a line to a given file (iterator).
142       * @param file_it
143       * @param l
144       * @return
145       */
146      void addLine(files::iterator file_it, line_number l);
147
148
149      // Check if word is correct:
150
151      /**
152       * Check if the word is valid.
153       * @param w
154       * @return bool
155       */
156      bool isWordValid(word w);
157
```

```
158     /**
159      * Check if character is a valid character.
160      * @param c
161      * @return bool
162      */
163     bool isCharValid(char c);
164 
165     /**
166      * Check if the character is valid to be the first character of the
           word.
167      * @param c
168      * @return bool
169      */
170     bool isFirstCharValid(char c);
171 
172     /**
173      * Removes invalid characters at the beginning of the word.
174      * @param word The word to be corrected.
175      */
176     void correctWord(string &word);
177 
178 
179     // Get information about the index:
180 
181     /**
182      * Get a string with the index of all words.
183      * @return string
184      */
185     string allToString();
186 
187     /**
188      * Convert a word at current iterator position to string. This
           includes all files and line numbers.
189      * Use fileToString()
190      * @parma w_it The current iterator,
191      * @return string String representation of the word and all files and
           lines where it occurs.
192      */
193     string wordToString(words::iterator w_it);
194 
195     /**
196      * Converts a file at current iterator position to string. This
           includes all lines.
197      * Use line_numberToString()
198      * @param f_it The current iterator.
199      * @return string String representation of file and all lines.
200      */
201     string fileToString(files::iterator f_it);
202 
203     /**
204      * Converts a line number at current iterator position to string.
205      * @param l_set The current iterator.
206      * @return string String representation of this line number.
207      */
208     string line_numberToString(line_numbers::iterator l_set);
209 
210 public:
211     /**
212      * Create an object.
213      */
214     Index();
215 
216     /**
217      * Destroy the object.
```

```
218       */
219      ~Index();
220
221      /**
222       * This method insert a word occuring in a file at a line to the index.
223       * @param w The word to insert to the index.
224       * @param f The file the word occurs in.
225       * @param lines The line in the file where the word occurs.
226       */
227      void addToIndex(word w, file f, line_numbers lines);
228
229      /**
230       * Create a new Index.
231       * @param out The outputfile to write the index in.
232       * @param in_files Reads from these input files to create an index.
233       */
234      void createIndex(file out, vector<file> in_files);
235
236      /**
237       * Prints the index for the given word.
238       * Called with parameter -q
239       * @param w The word the index is printed for.
240       */
241      void printIndexForWord(word w);
242
243      /**
244       * Print all words which matches with the first characters the given
              string.
245       * Called with parameter -s
246       * @param characters The letters the word must fit at the beginning.
247       */
248      void printWordsMatchesCharactersAtBeginning(string characters);
249
250      /**
251       * Print all words which matches the given string anywhere.
252       * This is not a required method! Called with parameter -s
253       * @param characters The characters to find in the index.
254       */
255      void printWordsMatchesCharactersAnywhere(string characters);
256
257      /**
258       * Print all words in given file in shell.
259       * Called with parameter -t
260       * @param f Print all words occuring in the given file.
261       */
262      void printIndexForFile(file f);
263
264      /**
265       * print the index for a given outputfile.
266       * Called with parameter -p
267       * @param out_file The output file to be printed.
268       */
269      void printIndexFromOutputFile(file out_file);
270  };
271
272  #endif  /* _INDEX_H */
```

Listing A.7: „index.cpp“

```
1  using namespace std;
2
3  #include "index.hpp"
4
```

```
5  void Index::init() {
6      // create new map
7      this->_word_index = new words();
8      this->s_util = new StringUtil();
9  }
10
11 Index::Index() {
12     this->init();
13 }
14
15 Index::~Index() {
16     for (words::iterator w_it = _word_index->begin(); w_it !=
           _word_index->end(); w_it++) {
17         files *f_map = w_it->second;
18
19         for (files::iterator f_it = f_map->begin(); f_it != f_map->end();
               f_it++) {
20             line_numbers *l = f_it->second;
21             delete l;
22         }
23
24         delete f_map;
25     }
26
27     delete _word_index;
28
29     delete s_util;
30 }
31
32 void Index::createIndex(file out, vector<file> in) {
33     //cout << "call createIndex()" << endl;
34
35
36     this->readContent(in);
37     this->writeContent(out);
38 }
39
40 void Index::readContent(vector<file> files) {
41     /*//cout << "call readFiles()" << endl;
42     for (vector<file>::iterator f_it = files.begin(); f_it != files.end();
           f_it++) {
43         this->readFile(*f_it);
44     }*/
45         map<string, vector<string>* > file_content_map =
               this->f_util->readFiles(files);
46
47         // NOW WE HAVE ALL FILES AND THEIR LINES
48     for (map<string, vector<string>*>::iterator f_c_it =
           file_content_map.begin(); f_c_it != file_content_map.end();
           f_c_it++) {
49         string current_file = f_c_it->first;
50         vector<string> *lines = f_c_it->second;
51
52         int line_number = 0;
53
54         // for each line in vector
55         for (vector<string>::iterator line_it=lines->begin() ; line_it !=
               lines->end() ; line_it++) {
56             string line = *line_it;
57
58             // extract all words from line
59             vector<string> words = this->extractAllWordsFromLine(line);
60
61             // now we can insert all the words, from current line, from
```

```
                current file in map
62              this->addToIndex(words, current_file, ++line_number);
63          }
64      }
65  }
66
67  void Index::writeContent(string out_file) {
68      this->f_util->writeFile(out_file, this->allToString());
69  }
70
71  bool Index::isWordValid(word w) {
72      //cout << "call isWordValid(" << w << ")" << endl;
73
74      int i = 0;
75      for (string::iterator word_it=w.begin(); word_it != w.end();
            word_it++) {
76          // check first char if it is valid
77          if (i == 0) {
78              if (!this->isFirstCharValid(*word_it)) {
79                  //cout << "first char is not valid!" << endl;
80                  //cout << "return false" << endl;
81                  return false;
82              }
83          } else {
84              // all other characters, not at first, check if they're valid
85              if (!this->isCharValid(*word_it)) {
86                  //cout << "some char in word is not valid" << endl;
87                  //cout << "return false" << endl;
88                  return false;
89              }
90          }
91          i++;
92      }
93
94      //cout << "return true" << endl;
95      return true;
96  }
97
98  bool Index::isCharValid(char c) {
99      //cout << "call isCharValid(" << c << ")" << endl;
100     // alternative implementation
101     if ( this->isFirstCharValid(c) || (c >= '0' && c <= '9')  || c == '-')
            {
102         return true;
103     }
104
105     return false;
106 }
107
108 // check the first character
109 bool Index::isFirstCharValid(char c) {
110     if ( (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') || c == '_') {
111         return true;
112     }
113
114     return false;
115 }
116
117 vector<word> Index::extractAllWordsFromLine(string line) {
118     vector<string> words;
119
120     string *word = new string();
121
122     // for each character in line
```

```
123     for (string::iterator char_it=line.begin(); char_it!=line.end();
            char_it++) {
124         //cout << "call for each character in line, char: " << *char_it <<
                endl;
125
126         // if it is a valid character
127         if (this->isCharValid(*char_it)) {
128             #ifdef Lexic_Compare
129                 word->push_back(*char_it);
130             #else
131                 word->push_back(tolower(*char_it));
132             #endif
133         } else {
134
135             // copy word in vector
136             if (word->size()>0) {
137                 words.push_back(*word);
138                 word->clear();
139             }
140             continue;
141         }
142     }
143
144     if (word->size()>0) {
145         words.push_back(*word);
146     }
147
148     delete word;
149
150     vector<string> validWords;
151     // now we have all words from line, but now we have to check if all
            words starts with a valid char
152     for (vector<string>::iterator w_it = words.begin(); w_it !=
            words.end(); w_it++) {
153         string word = *w_it; // get the word
154
155         // here we can decide if we want to remove wrong words, or to
                correct them
156
157         // if the word is not valid, correct it
158         if (!this->isWordValid(word)) {
159             this->correctWord(word);
160
161             // if the word is empty (no valid first character up to the
                    end of the word)
162             if (word.empty()) {
163                 continue; // go on with next word
164             }
165         }
166
167         // now we have an absolut correct word, at to vector.
168         validWords.push_back(word);
169     }
170
171     return validWords;
172 }
173
174 void Index::correctWord(string &word) {
175     // remove all invalid chars at beginning
176     for (string::iterator s_it = word.begin(); s_it != word.end(); s_it++)
            {
177         char c = *s_it;
178
179         // if the character is valid
```

```
180         if (this->isFirstCharValid(c)) {
181             word.assign(s_it, word.end());
182             return;
183         }
184     }
185 
186     // if none character is valid for first sign, so the word can not be
        correct,
187     // beacause there is no other character left
188     word.clear();
189 }
190 
191 void Index::addToIndex(vector<string> words, file f, line_number l) {
192 
193     for (vector<string>::iterator words_it=words.begin() ; words_it !=
        words.end() ; words_it++) {
194         this->addToIndex(*words_it, f, l);
195     }
196 }
197 
198 /**
199  * Adds a word to the map.
200  * If the word already exists, append values.
201  */
202 void Index::addToIndex(word w, file f, line_number l) {
203     //cout << "call addToIndex, word: " << w << ", in file: " << f << ",
        at line: " << l << endl;
204 
205     // the word is in the map and we get an iterator to its position.
206     words::iterator word_it = this->addWord(w);
207 
208     // now we can add the file to that word
209     files::iterator file_it = this->addFile(word_it, f);
210 
211     // now we can add the line to the given file iterator
212     this->addLine(file_it, l);
213 }
214 
215 words::iterator Index::addWord(word w) {
216         // change word to lower case
217         //this->s_util->toLowerCase(w);
218 
219         pair<words::iterator,bool> ret;
220         files *f = new files();
221 
222         /*cout << "  find " << w << " in map -> ";
223         words::iterator w_it = _word_index->find(w);
224         if (w_it != _word_index->end()) {
225             cout << "true" << ", " << w_it->first << " is in map!" << endl;
226         } else {
227             cout << "false" << endl;
228         }*/
229 
230         ret = _word_index->insert(pair<word, files*>(w, f));
231 
232         if (ret.second == false) {
233             //cout << ret.first->first << " is already in map!" << endl;
234             delete f;
235         }
236 
237         return ret.first;
238 }
239 
240 files::iterator Index::addFile(words::iterator word_it, file f) {
```

```
241     // get the value (map_files) from iterator
242     files *m_files = word_it->second;
243 
244     pair<files::iterator,bool> ret;
245     line_numbers *l = new line_numbers();
246     ret = m_files->insert(pair<file, line_numbers* >(f, l));
247 
248     if (ret.second == false) {
249         //cout << "file: '" << f << "' is already in map for the word: "
                << word_it->first << " so no changes!" << endl;
250         delete l;
251     }
252 
253     return ret.first;
254 }
255 
256 void Index::addLine(files::iterator file_it, line_number l) {
257     //cout << "add line number : " << l << " to file : " << file_it->first
            << endl;
258     file_it->second->insert(l);
259 }
260 
261 vector<string>* Index::readAllLines(file f) {
262     // read the given index file
263     // this vectore stores each line in the file
264     vector<string> *lines = new vector<string>();
265     string line;
266 
267     // read from inputfile
268     ifstream in(f.c_str());
269 
270     // test if file exist
271     if (!in) {
272         cout << "Eingabedatei "<< f << " nicht gefunden!" << endl;
273         in.close() ; // Datei schlieÃen
274         return lines;
275     }
276 
277     // read each line of file
278     while (getline(in, line, '\n')) {
279         // put line at the end of vector.
280         lines->push_back(line);
281     }
282 
283     in.close() ; // Datei schlieÃen
284 
285     return lines;
286 }
287 
288 void Index::printIndexFromOutputFile(file f) {
289     cout << "printIndexFromOutputFile : " << f << endl;
290 
291     cout << this->allToString();
292 }
293 
294 string Index::allToString() {
295     string all;
296     for (words::iterator w_it = _word_index->begin(); w_it !=
            _word_index->end(); w_it++) {
297         all.append(this->wordToString(w_it));
298     }
299 
300     return all;
301 }
```

```
302
303 string Index::wordToString(words::iterator w_it) {
304     //words::iterator w_it = _word_index->find(w);
305     string word;
306     word.append(w_it->first);
307
308     files *f_map = w_it->second;
309     for (files::iterator f_it = f_map->begin(); f_it != f_map->end();
            f_it++) {
310         word.append(this->fileToString(f_it));
311     }
312
313     return word;
314 }
315
316 string Index::fileToString(files::iterator f_it) {
317     string files;
318         files.append(" ");
319         files.append(f_it->first);
320
321         line_numbers *l_set = f_it->second;
322         for (line_numbers::iterator l_it = l_set->begin(); l_it !=
                l_set->end(); l_it++) {
323             files.append(this->line_numberToString(l_it));
324         }
325
326         files.append("\n");
327
328     return files;
329 }
330
331 string Index::line_numberToString(line_numbers::iterator l_it) {
332     string lines;
333         lines.append(" ");
334         lines.append(this->s_util->toString(*l_it));
335
336     return lines;
337 }
338
339 // print the Index for one word.
340 // TODO, can be used for other methods
341 void Index::printIndexForWord(string pWord) {
342     // find given word
343     words::iterator w_it = _word_index->find(pWord);
344
345     // if word is in map
346     if (w_it != _word_index->end()) {
347         cout << this->wordToString(w_it);
348     } else {
349         cout << "The word: " << pWord << " is not an element of the
                index!" << endl;
350     }
351 }
352
353 void Index::printWordsMatchesCharactersAtBeginning(string chars) {
354     for (words::iterator w_it = _word_index->begin(); w_it !=
            _word_index->end(); w_it++) {
355         string word = w_it->first;
356         size_t found;
357         found = word.find(chars);
358
359         if (found == 0) {
360             cout << this->wordToString(w_it);
361         }
```

```
362     }
363 }
364
365 void Index::printWordsMatchesCharactersAnywhere(string chars) {
366     for (words::iterator w_it = _word_index->begin(); w_it !=
            _word_index->end(); w_it++) {
367         string word = w_it->first;
368
369         size_t found;
370         found = word.find(chars);
371
372         //cout << "found word: " << chars << " at position: " << found <<
                endl;
373         if (found!=string::npos) {
374             cout << this->wordToString(w_it);
375         }
376     }
377 }
378
379 void Index::printIndexForFile(file f) {
380     for (words::iterator w_it = _word_index->begin(); w_it !=
            _word_index->end(); w_it++) {
381         word w = w_it->first;
382         files *f_map = w_it->second;
383
384         files::iterator f_it = f_map->find(f);
385         if (f_it != f_map->end()) {
386                 cout << w << this->fileToString(f_it);
387             }
388     }
389 }
390
391 void Index::addToIndex(word w, file f, line_numbers lines) {
392     for (line_numbers::iterator l_it = lines.begin(); l_it != lines.end();
            l_it++) {
393         this->addToIndex(w, f, *l_it);
394     }
395 }
```

Listing A.8: „indexparser.hpp“

```
1  /** Class Index
2   *
3   *  This class creates an index of different input files.
4   *
5   *  author  Alexander Miller
6   *  date    11.5.2011
7   *  version 0.1
8   */
9
10 #ifndef _INDEXPARSER_H
11 #define _INDEXPARSER_H
12
13 #include <string>
14 #include <iostream>
15 #include <set>
16
17 #include "index.hpp"
18 #include "stringutil.hpp"
19
20
21 using namespace std;
22
```

```
23 class IndexParser {
24 private:
25
26     Index *index; // this is our reference to the index instance
27
28     /**
29      * Read IndexFile and store each line into vector<string>
30      * @param *lines container to store all text lines
31      * @param index_file IndexFile for parsing
32      */
33     void fileToLines(vector<string> *lines, string index_file);
34
35     /**
36      * Parse given lines
37      * @param *lines Lines for parsing
38      */
39     void parseAllLines(vector<string> *lines);
40
41     /**
42      * Parse current line
43      * @param linie Current line for parsing
44      * @param flag_wort Variable give information about parsing progress
            --> true = Word determined
45      * @param flag_file Variable give information about parsing progress
            --> true = File determined
46      * @param flag_index Variable give information about parsing progress
            --> true = Index determined
47      */
48     string parseLine(string linie, bool flag_wort, bool flag_file, bool
        flag_index, string last_word);
49
50 public:
51     /**
52      * Create an object.
53      * @param *index IndexFile for parsing
54      */
55     IndexParser(Index *index);
56
57     /**
58      * Destroy the object.
59      */
60     ~IndexParser();
61
62     /**
63      * Read each line from given index file.
64      * Parse word, file and index from each line.
65      * Insert parsed data in index map.
66      * @param index_file  index file to be read.
67      */
68     void readIndexFile(string index_file);
69 };
70
71 #endif  /* _INDEXPARSER_H */
```

Listing A.9: „indexparser.cpp“

```
1 using namespace std;
2
3 #include "indexparser.hpp"
4
5 IndexParser::IndexParser(Index *i) {
6     this->index = i;
7 }
```

```
8
9  IndexParser::~IndexParser() {
10 }
11
12 // Store each line from selected index_file into given vector<string>
13 void IndexParser::fileToLines(vector<string> *lines, string index_file){
14     ifstream in(index_file.data()) ; // Open given IndexFile
15     string line;
16
17     // Test if file exist. If exists --> store lines in verctor<string>
18     if (!in){
19         cout << "Index file:  "<< index_file << " not found!" << endl;
20         in.close() ; // Datei schlieÃen
21     }
22     else{
23         // Read each line of given index file
24         while (getline(in, line, '\n')) {
25         lines->push_back(line); // Put line at the end of vector.
26         }
27     }
28     in.close() ; // Close IndexFile
29 }
30
31 string IndexParser::parseLine(string linie, bool flag_wort, bool
       flag_file, bool flag_index, string last_word){
32     string wort, file, ind;
33     stringstream out;
34     int index = 0;
35     line_numbers *index_set = new line_numbers();
36
37     // Parse given line and differ strings between WORD, FILE and INDEX
38     for (string::iterator s_it = linie.begin(); s_it != linie.end();
           s_it++) {
39         out.str("");
40         out << *s_it;
41         string character = out.str();
42         StringUtil s_util;
43
44         // Read rest --> INDEX
45         if (flag_wort == true && flag_file == true && flag_index == false
               && character != " ") {
46             ind += character;
47         }
48         // Insert parsed index into index set
49         if ((flag_wort == true && flag_file == true && flag_index == false
               && character == " " && ind != "") || (s_it == linie.end()-1)) {
50             out.str(ind);
51             s_util.stringTo(out.str(), index);
52             index_set->insert(index);
53             out.str((ind = ""));
54         }
55
56         // Read second string --> FILE
57         if (flag_wort == true && flag_file == false && flag_index == false
               && character != " ") {
58             file += character;
59         }
60         // Set respective flag if file name extracted
61         if (flag_wort == true && flag_file == false && flag_index == false
               && character == " ") {
62             flag_file = true;
63         }
64
65         // Read first string --> WORD
```

```
66          if (flag_wort == false && flag_file == false && flag_index ==
                false && character != " ") {
67              wort += character;
68          }
69          // Set flag if the first word is completely extracted.
70          // Set flag also when the first charakter is BLANK. IT means -->
                word == previous word and first string is a file name
71          if (flag_wort == false && flag_file == false && flag_index ==
                false && character == " " && (s_it != linie.begin()) ) {
72              flag_wort = true;
73          }
74          // If first charakter is BLANK then use previous word and the next
                string will be a file name
75          if (flag_wort == false && flag_file == false && flag_index ==
                false && character == " " && (s_it == linie.begin()) ) {
76              wort = last_word;
77              flag_wort = true;
78          }
79      }
80
81
82      this->index->addToIndex(wort, file, *index_set);
83      /*
84      // TEST FUNKTION ZUR AUSGABE AM TERMINAL
85      cout << wort << " " << file << " ";
86      for (set<int>::iterator line_it = index_set->begin(); line_it !=
            index_set->end(); line_it++) {
87          // write out line number followed by a BLANK
88          stringstream o;
89          o << *line_it;
90          string line = o.str();
91          cout << *line_it << " ";
92      }
93      cout << "\n";
94      // ENDE DER TESTFUNKTION*/
95      delete index_set;
96      return wort;
97  }
98
99  void IndexParser::parseAllLines(vector<string> *lines){
100     string linie;
101     string last_word = "";
102
103     // Read each line of file and parse it
104     for (vector<string>::iterator lines_it = lines->begin(); lines_it !=
            lines->end(); lines_it++) {
105         bool flag_wort = false; // Flag flag_word==true when the word of
                current line is parsed
106         bool flag_file = false; // Flag flag_file==true when file name of
                current line is parsed
107         bool flag_index = false; // Flag flag_index==true when every index
                of current line is parsed
108         stringstream out;
109         out << *lines_it;
110         linie = out.str();
111         // Parse current line and write values into the index; Use flags
                to mark parsing progress
112         last_word = this->parseLine(linie, flag_wort, flag_file,
                flag_index, last_word);
113     }
114 }
115
116 void IndexParser::readIndexFile(string index_file) {
117     vector<string> *lines = new vector<string>();  // vector<string> for
```

```
        text lines
118     this->fileToLines(lines, index_file);   // store each line as a string
        in vector<string
119     this->parseAllLines(lines); // parse every line and put the extracted
        data into index
120     delete lines;
121 }
```

Listing A.10: „lexic.hpp“

```
1  #include <string>
2  #include <iostream>
3
4  using namespace std;
5
6  /** Class Lexic
7   *
8   *  This class is responsilbe for the lexicografical ordering in the index.
9   *
10  *  author  Christian Bunk
11  *  date    19.5.2011
12  *  version 0.1
13  */
14 class Lexic {
15 public:
16     /**
17      * This operator() is called by the map, when a new element is
            inserted to bring it in the right order.
18      * The sort order is lexicografic.
19      * @return bool
20      * @param s1 The first string compare to the second.
21      * @param s2 The second string compare to the first.
22      */
23     bool operator()(string s1, string s2);
24
25 private:
26     /**
27      * A case-insensitive comparison function.
28      * @param c1
29      * @param c2
30      * @return bool
31      */
32     static bool mycomp (char c1, char c2);
33
34     bool isWordEqualCaseInsensitiv(string s1, string s2);
35 } ;
```

Listing A.11: „lexic.cpp“

```
1  #include "lexic.hpp"
2
3  using namespace std;
4
5  #ifdef Lexic_Compare_Standard
6  // Vergleichsfunktion ohne Beruecksichtigung von Gross- und
       Kleinschreibung :
7  bool compare (char c1, char c2)
8  {
9      return tolower(c1) < tolower(c2) ;
10 }
```

```
11
12 bool Lexic::operator()(string s1, string s2)
13 {
14     return lexicographical_compare(s1.begin(), s1.end(), s2.begin(),
           s2.end(), compare) ;
15 }
16 #else
17
18 bool Lexic::operator()(string s1, string s2) {
19     string::iterator s1_it = s1.begin();
20     string::iterator s2_it = s2.begin();
21
22     // check strings of case insensitive equality
23     while (s1_it != s1.end() || s2_it != s2.end()){
24         // s1 is lower than s2
25         if (tolower(*s1_it) < tolower(*s2_it)) {
26             return true;
27         }
28
29         // s1 is greater than s2
30         if (tolower(*s1_it) > tolower(*s2_it)) {
31             return false;
32         }
33
34         s1_it++;
35         s2_it++;
36     };
37
38     // here either both strings are euqal or one is longer and contains
           the other string
39
40     // if both are equal make a check for CASE SENSITIVE difference
41     if (s1.size() == s2.size()) {
42         s1_it = s1.begin();
43         s2_it = s2.begin();
44
45         while(s1_it != s1.end() && s2_it != s2.end()) {
46             // s1 is lower than s2 (CASE SENSITIVE)
47             if (*s1_it < *s2_it) {
48                 return true;
49             }
50
51             // s1 is greater than s2 (CASE SENSITVE)
52             if (*s1_it > *s2_it) {
53                 return false;
54             }
55
56             s1_it++;
57             s2_it++;
58         };
59
60         // if it comes here both strings are even CASE SENSITIVE exactly
               the same
61         return false;
62     } else {
63         return (s1.size() < s2.size()); // true if s1 is lower than s2,
               false otherwise
64     }
65 }
66 #endif
```

Listing A.12: „fileutil.hpp“

```
#ifndef _FILETUIL_H
#define _FILEUTIL_H

#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <iterator>

using namespace std;

/** Class FileUtil
 *
 *  This class provides methods to read and write content to a file.
 *
 *  author  Christian Bunk
 *  date    15.5.2011
 *  version 0.1
 */
class FileUtil {
public:
    /**
     * Read content from a given File.
     * Returns a map with the filename as the key and the content of the
         file as the value.
     * @param files Contains all files to read in.
     * @return map<string, vector<string>*> A map, containing all lines as
         the value for the file as the key.
     */
    map<string, vector<string>* > readFiles(vector<string> files);

    /**
     * Write content to file.
     * @param out_file The file where the content will be written.
     * @param content The content to be written into file.
     */
    void writeFile(string out_file, string content);

private:
    /**
     * Read all lines within a file.
     * @param file The file to read from.
     * @return vector<string>* All lines within the file.
     */
    vector<string>* readAllLinesFromFile(string file);
};

#endif  /* _FILEUTIL_H */
```

Listing A.13: „fileutil.cpp“

```
#include "fileutil.hpp"

map<string, vector<string>* > FileUtil::readFiles(vector<string> files) {
    map<string, vector<string>* > *file_map = new map<string,
        vector<string>* >();

    //cout << "call readFiles()" << endl;
    for (vector<string>::iterator f_it = files.begin(); f_it !=
        files.end(); f_it++) {
        vector<string> *lines = this->readAllLinesFromFile(*f_it);

```

```
10          pair<map<string, vector<string>* >::iterator,bool> ret;
11          ret = file_map->insert(pair<string, vector<string>* >(*f_it,
                lines));
12
13          if (ret.second == false) {
14              delete lines;
15          }
16      }
17
18      return *file_map;
19  }
20
21  // read all lines of a file and returns a vector of string with .
22  vector<string>* FileUtil::readAllLinesFromFile(string f) {
23      // read the given index file
24      // this vectore stores each line in the file
25      vector<string> *lines = new vector<string>();
26      string line;
27
28      ifstream ifs (f.c_str(), ifstream::in);
29
30      if (ifs.fail()) {
31          cout << "Inputfile: "<< f << " not found!" << endl;
32      }
33
34      while (ifs.good()) {
35          getline(ifs, line, '\n');
36           lines->push_back(line);
37      }
38
39      ifs.close();
40
41      return lines;
42  }
43
44  void FileUtil::writeFile(string out_file, string content) {
45      ofstream out(out_file.c_str());
46      if (out.fail()) {
47          cerr << "Outputfile: " << out_file << " could not be opened!" <<
                endl;
48      } else {
49          out << content;
50      }
51
52      out.close();
53  }
```

Listing A.14: „stringutil.hpp“

```
1  #ifndef _STRINGUTIL_H
2  #define _STRINGUTIL_H
3
4  #include <string>
5  //#include <iostream>
6  #include <sstream>
7  #include <algorithm>
8
9  using namespace std;
10
11 /** Class StringUtil
12  *
13  *  This class provides some usefull string methods.
14  *
```

```
15  *  author  Christian Bunk
16  *  date    15.5.2011
17  *  version 0.1
18  */
19 class StringUtil {
20 public:
21
22     /**
23      * Convert a single value into a string.
24      * @param value The value to be convert into a string. Must implement
            operator<<() !
25      * @return string The string representation of the value.
26      */
27     template <typename Type>
28     string toString(Type value) {
29         ostringstream strout;
30         strout << value;
31
32         return (strout.str());
33     }
34
35     /**
36      * Convert a string into a given data type
37      * @param str The string to be converted into a special data type.
38      * @param &value The variable where the representation of the string
            definded by the data type is stored. call-by-reference! Must
            implement operator>>() !
39      */
40     template <typename Type>
41     void stringTo(string str, Type &value) {
42         istringstream strin;
43
44         strin.str(str);
45         strin >> value;
46     }
47
48     /**
49      * Transfomrs a given string to lower case letters.
50      * @param &str The string to be transformed given as an reference.
            call-by-reference!
51      */
52     void toLowerCase(string &str);
53
54 };
55
56 #endif  /* _STRINGUTIL_H */
```

Listing A.15: „stringutil.cpp“

```
1 #include "stringutil.hpp"
2
3 void StringUtil::toLowerCase(string &str) {
4     transform(str.begin(), str.end(), str.begin(), ::tolower);
5 }
```

A.5. Makefile

Listing A.16: „Makefile“

```
1 # Makefile in einem Unterverzeichnis
2 include ../Makefile.rules
3
4 OBJ = main.o cmdline.o index.o lexic.o controller.o indexparser.o
      stringutil.o fileutil.o
5
6 all: $(OBJ)
7
8 clean:
9     rm -f $(OBJ)
```